BASIC Programming Language

Thomas E. Kurtz
Dartmouth College

GLOSSARY

BASIC Name of any of a large number of simple programming languages that are similar and ultimately derived from the original Dartmouth BASIC of 1964.

Keyword Word in a computer language that has a special meaning. (Keywords in BASIC include, for instance, `LET`, `PRINT`, `FOR`, `NEXT`, `TO`, and `STEP`.)

Language In computing, a programming language. Programming languages like human languages, consist of words and symbols together with grammatical rules that govern how they can be put together.

Line Same as a line of text, beginning with a line number in original BASIC.

Line number Integer (whole number) that begins each line of a BASIC program and serves as a kind of "serial number" for that line. Line numbers also serve as "targets" for `GOTO` statements.

List A list of a program is a "printout" on the screen of a computer or on a hard-copy printer, of the text of the program (i.e., its lines).

Program Collection of statements, formed according to the rules of the language and with the purpose of carrying out a particular computing task.

Run Actual carrying out of the instructions of the program by the computer.

Statement Instruction to the computer. In BASIC, statements are virtually synonymous with lines and usually begin with a keyword.

Subroutine Portion of a program, usually set off from the rest of the program, that carries out some specific task. Subroutines are usually invoked by special instructions, such as `GOSUB` and `RETURN` in original BASIC, or `CALL` in modern versions.

Variable Word in a program, usually different from a keyword, that stands for a quantity, just as in ordinary algebra.

BASIC (Beginner's All-Purpose Simplified Instruction Code) began as an interactive computer programming language especially easy to learn. Invented in 1964, it now exists in many widely different versions. Students learning programming for the first time may know BASIC as the simple language found on most personal computers.

Others know BASIC as a development language on personal computers and workstations. This article explores the history and development of BASIC, at least some of the versions of BASIC, and explains why this language has become so diverse and yet so popular.

I. BRIEF DESCRIPTION

Nearly everyone who has heard about computers has heard about the BASIC language. Many of these people can read and write simple programs in BASIC. Note that, in 1964, long before personal computers or display terminals, one entered a program by typing (as now) and the computer responded by typing back onto yellow paper (rather than displaying results on a screen):

```
100 LET X = 3
110 LET Y = 4
120 LET Z = X + Y
130 PRINT Z
140 END
```

Almost anyone who has taken high school algebra will understand what this program does and understand it well enough to make changes in it. (When run, it prints the number 7.)

BASIC was invented for an interactive environment (time-sharing or personal computers). The user could start and stop the program at will and could interact with the running program. For instance, the `INPUT` statement in the following program allowed the user to enter the numbers to be added *after* typing `RUN` (remember, all commands had to be typed; there were no mouses or menus):

```
100 INPUT X, Y
110 LET Z = X + Y
120 PRINT Z
130 END
```

After the user typed `RUN`, the program stopped (temporarily), printed a question mark (?), and waited for the user to respond. The user then typed two numbers, separated by a comma, and followed by hitting the RETURN or ENTER key. The program then commenced, calculated Z (as the sum of the two numbers), and printed the answer. The result might look like this on the yellow paper, or on the screen of an early microcomputer:

```
RUN
? 3.4
7
```

A user who wished to make several additions could arrange for the program to continue indefinitely, as in:

```
100 INPUT X, Y
110 LET Z = X + Y
120 PRINT Z
125 GOTO 100
130 END
```

This time, the result might look like this:

```
RUN
? 3, 4
7
? 1.23, 4.56
5.79
? -17.5, 5.3
-12.2
?
```

The user continued in this fashion until all additional problems had been "solved." The user then stopped the program by some method that varied from machine to machine.

The above examples will be trivial to anyone who knows BASIC but should be understandable even to someone who has not used BASIC. It is not the purpose of this article, however, to teach the language through such simple examples. The purpose is rather to use these and later examples to illustrate an important point: BASIC is not just a single computer language; it is actually a collection of many languages or dialects, perhaps hundreds that have evolved since the mid-1960s. As a result, versions that run on different brands of computers are different. It has even been the case that different models of the same brand have sported different versions of BASIC.

For example, some versions of BASIC allowed the user to omit the word `LET`, to omit the `END` statement, or to employ either uppercase letters or lowercase letters interchangeably. For example, the first program above might be allowed to appear on some computers as:

```
100 x = 3
110 y = 4
120 z = x + y
130 print z
```

One more important way in which versions of BASIC developed is that some allow "structured programming" (discussed later.) Recall an earlier example:

```
100 INPUT X, Y
110 LET Z = X + Y
120 PRINT Z
125 GOTO 100
130 END
```

One of the tenets of structured programming is that GOTO statements (as in line 125 above), used carelessly, are the cause of many programming errors. All modern versions of BASIC allow the user to rewrite the above program without using GOTO statements as:

```
100 DO
110     INPUT x, y
120     LET z = x + y
130     PRINT z
125 LOOP
130 END
```

The collection of lines starting with 100 DO and ending with 125 LOOP is known as a *loop*. The interior lines of the loop are carried out repeatedly, as in the example by using a GOTO statement. Notice, in addition, that the program is written in mixed case (using both upper- and lowercase letters), and the interior lines of the loop are indented. All modern versions of BASIC allow these stylistic improvements.

Eliminating the GOTO statement (line 125) removes the need to reference line numbers in the program. The line numbers themselves, no longer serving a useful purpose, can now be eliminated to get:

```
DO
    INPUT x, y
    LET z = x + y
    PRINT z
LOOP
END
```

We could not have removed the line numbers from the version that used a GOTO statement "GOTO 100" because there would no longer be a line numbered 100 in the program. Some earlier versions of BASIC allowed removing some lines except for those used as GOTO targets, in which case the line numbers became *statement labels*, a concept not present in the original BASIC.

II. EARLY HISTORY OF BASIC

BASIC was invented at Dartmouth College in 1963–1964 by John G. Kemeny and Thomas E. Kurtz, both professors of mathematics, assisted by a group of undergraduate student programmers. Computers then were huge, slow, and expensive; there were no personal computers. Their goal was to bring easy and accessible computing to *all* students, not just science or engineering students. The method they chose called for developing a *time-shared* operating system, which would allow many users simultaneously. (This operating system was developed entirely by Dartmouth undergraduate students.) The new language, BASIC, easy to learn and easy to use, was an essential part of this effort. BASIC was thus developed originally for a large multiple-user, time-shared system and *not* for personal computers, which did not appear widely until the early 1980s.

It has been asked why BASIC was invented. Couldn't an existing language have been used for the purpose? The answer to the second question is no, which also answers the first question. Other computer languages did exist in 1963, although there were not nearly as many as there are today. The principal ones were FORTRAN and Algol; most of the others are long since forgotten. Some of the common languages used today—C, C++, and Java—had not even been conceived. FORTRAN and Algol were each considered briefly. These languages were designed for production use on big machines or for scientific research, using punched cards. But neither was suitable for use by beginners, neither was particularly well suited for a time-shared environment, and neither permitted speedy handling of short programs. Kemeny and Kurtz had experimented with other simple computer languages as early as 1956, but with only modest success. So, in 1963, when they began building a time-shared system for students, they quickly realized that a new language had to be invented—BASIC.

A. Design Goals

The new language had to satisfy these properties:

1. It had to be easy to learn for *all* students.
2. It had to work well in a multiple-user, time-sharing system.
3. It had to allow students to get answers quickly, usually within 5 or 10 sec.

In the years since BASIC was invented the importance of time sharing has been overshadowed by the invention of personal computers: Who needs to time-share a big expensive computer when one can have a big (in power) but inexpensive computer on one's desk top? For a long time, no such dramatic improvement has been made on the programming side. Thus, while the impact of time sharing has been largely forgotten, the importance of BASIC has increased. The ideals that forced the invention of BASIC—simplicity and ease of use—lead many to choose BASIC today.

B. The First Dartmouth BASIC

BASIC came into existence in the early morning of May 1, 1964, when two BASIC programs were run on the Dartmouth time-sharing system at the same time, both giving the correct answer. That early version of BASIC offered 14 different statements:

```
LET     PRINT     END
READ    DATA
GOTO    IF-THEN
FOR     NEXT
GOSUB   RETURN
DIM     DEF       REM
```

LET, PRINT, and END were illustrated in the first example program. READ and DATA were used to supply data to the program other than through LET statements. (It is a strange fact that the first version of BASIC did not have the INPUT statement.) GOTO and IF-THEN provided the ability to transfer to other locations in the program, either unconditionally or conditionally on the result of some comparison. FOR and NEXT were used together and formed a loop. GOSUB and RETURN provided a crude subroutine capability. DIM allowed the user to specify the size of a vector or matrix. DEF allowed the user to define a new function (in addition to the functions such as SQR, SIN, and COS that BASIC included automatically). REM allowed the user to add comments or other explanatory information to programs.

We shall illustrate all 14 statement types in two short programs. The first program uses eight of the statement types and prints a table of the values of the common logarithms (logarithms to the base 10) for a range of arguments and a spacing given in a DATA statement:

```
100 REM PRINTS TABLE OF COMMON LOGS
110 READ A, B, S
120 DATA 1, 2, 0.1
130 DEF FNF(X) = LOG(X)/LOG(10)
140 FOR X = A TO B STEP S
150    PRINT X, FNF(X)
160 NEXT X
170 END
```

(Common logarithms can be computed from "natural" logarithms with the formula shown in line 130. The program, when run, prints a table of values of the common logarithm for arguments 1, 1.1, 1.2, 1.3, etc., up to 2.)

The second program computes, stores in a vector, and prints the Fibonacci numbers up to the first that exceeds 100. (The Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, ... The first two, 1 and 1, are given; each succeeding one is obtained by adding the previous two.)

```
100 REM FIBONACCI NUMBERS
110 DIM F(20)
120 LET F(1) = 1
130 LET F(2) = 1
140 LET N = 2
150 IF F(N) > 100 THEN 999
160    GOSUB 500
170    PRINT F(N),
180 GOTO 150
500 REM SUBROUTINE TO COMPUTE
    NEXT NUMBER
510 LET N = N + 1
520 LET F(N) = F(N - 1) + F(N - 2)
530 RETURN
999 END
```

The DIM statement (line 110) establishes a vector named F having 20 components. Lines 120 and 130 establish the first two numbers of the Fibonacci sequence. The IF-THEN statement (line 150) checks to see if the most recently computed Fibonacci number is greater than 100; if it is, the program jumps to the END statement and stops. If that is not the case, it computes the next number. Lines 150–180 are another example of a loop. The subroutine (lines 500–530) contains the formula for computing the next Fibonacci number. The subroutine is "called on" or "invoked" by the GOSUB statement in line 160, which refers to the subroutine by the line number of its first statement. When the subroutine has finished its work, the RETURN statement (line 530) "returns" control back to the line following the line containing the GOSUB statement.

Even in the earliest days, it was considered good form to use REM statements to explain what the program did and what each of the subroutines in the program did. The use of indentation following the line number to display the extent of the loop also began to appear around the time BASIC was invented. The reason is that people, in addition to computers, have to read computer programs; remarks and indentation help. Two other stylistic features were not common in 1964: lowercase letters (most terminals did not even have them) and completely blank lines.

C. Major Design Decisions

We now consider the major design decisions made in 1964 and why they were made.

1. A Number is a Number is a Number; That is, There is Only One Kind of Number in BASIC

In 1964, as today, most machines could do arithmetic using several kinds of numbers. The two common kinds were, and still are, integer numbers and floating-point numbers. Integer numbers are simply whole numbers such as 0, 17, −239, and 12345678. Floating-point numbers

can be thought of as numbers in "scientific" notation, as in $1.234 \times 10-3$. Although arithmetic using integer numbers was usually faster, Dartmouth BASIC did *all* arithmetic in floating point. While some programs might run more slowly, life was made considerably easier for the beginner.

2. BASIC Should Read and Print Numbers Without Requiring Special "Formats"

FORTRAN, the most widely used language at that time, used a complicated format control statement for all input or output in the program, a requirement too complicated for most students, particularly for those not taking science or engineering.

3. One Should be Able to Create, Change, and Run a BASIC Program from a Typewriter

Video screen terminals were not widely available in 1964. The only alternative was the Teletype™ Model 33. It typed in uppercase only and was very slow—10 characters per second. The line numbers of a BASIC program allowed users to change the program without retyping the entire program; they needed merely to retype the corrected line, including the line number.

4. BASIC Should have no Mysterious Punctuation Rules; Thus, a Statement and a Line are Synonymous

Other languages allowed statements to extend over several lines and allowed several statements to appear on the same line. Many used semicolons to separate statements, regardless of the lines on which the statements appeared, an unnecessarily complicated rule for beginners.

5. All BASIC Statements Should Commence, After the Line Number, with a Keyword

Most languages begin statements with a keyword. A common exception is the assignment statement, wherein variables receive the results of computations. Different languages treat the assignment statement in different ways:

```
FORTRAN  N = N+1
Algol    N := N + 1;
BASIC    LET N = N + 1
```

The FORTRAN method is confusing to the beginner; it looks like an assertion that N is equal to N + 1, which is nonsense. The := symbol of the Algol method is supposed to represent an arrow pointing to the left, but this may also confuse most beginners. It was decided to use the keyword `LET` to make the intention clear:

Three other design features in the original BASIC have not withstood the test of time.

6. BASIC Should not Require Unnecessary Declarations, Suchas Supplying Dimensions for Arrays

The original BASIC allowed the use of single letters as array names (arrays are also called *lists* and *tables* or, in mathematical circles, *vectors* and *matrices*). A single letter with an attached subscript in parentheses represented an array element. The following complete program, which prints the squares of the whole numbers from 1–8, uses a singly subscripted array (i.e., a list or vector):

```
100 FOR I = 1 TO 8
110     LET X(I) = I*I
120 NEXT I
130 FOR I = 1 TO 8
140     PRINT I, X(I)
150 NEXT I
160 END
```

It was not necessary to include a `DIM` statement to establish that X stood for a vector, as in:

```
99 DIM X(8)
```

Such a `DIM` statement could be included, to be sure, but if one were satisfied to work with elements X(1) through X(10), the `DIM` statement would not be required. While supplying sensible default values is still a cornerstone of BASIC, default dimensioning of arrays has given way to multi-character function names.

7. BASIC Should be Blank Insensitive; That is, a User Should be Able to Type in a Program Without Regard to Blank Spaces

This feature was intended to ease life for beginning typists. The idea was that:

```
100 LET N = N + 1
```

could be typed as

```
100LETN=N+1
```

This decision meant that only simple variable names could be used. The allowable variable names consisted of either single letters or single letters followed by single digits.

With this rule, the following program fragment was unambiguous:

```
100FORI=1TON
110LETX1=X1+Y9-I*SQR(N)
120NEXTI
```

It later became abundantly clear that permitting multi-character variable names was far more important than blank insensitivity. The reason that multi-character variable names and blank insensitivity cannot coexist is illustrated in the following example:

```
FOR I = A TO B STEP C
```

might be written, ignoring blanks, as:

```
FOR I = A TO B STEP C
```

If multi-character variable names were allowed, BASIC was in a quandary. It cannot distinguish the first form (the variable I starts at the value of the variable A, finishes at the value of the variable B, and is incremented with a step size given by the value of the variable C) from the second (the variable I starts at the value of the variable A and finishes at the value of the variable BSTEPC, with the step size understood to be 1). Giving up blank insensitivity in favor of multi-character variable names resolves the ambiguity (in favor of the second form).

8. Line Numbers Should Double as Editing Aids and Targets of `GOTO` and `IF-THEN` Statements

For the first 10 years of BASIC, this design decision remained valid, but eventually video terminals replaced Teletypes and soon supported screen editors. (A screen editor permits making changes in the program simply by moving the cursor around; with a screen editor, What You See Is What You Get.) Line numbers were no longer needed as editing aids. This period also saw the birth of structured programming. One of the tenets of structured programming is that the `GOTO` statements are simply not needed, provided that one can use an `IF-THEN-ELSE` structure and a general loop structure. If all old-fashioned `GOTO` and `IF-THEN` statements are eliminated, line numbers are not needed as "targets" for those statements. Line numbers, no longer serving a useful purpose, can quietly disappear.

D. BASIC Starts To Grow

BASIC quickly grew in response to the needs of its users. By 1969, which saw the appearance of the fifth version of BASIC at Dartmouth, features had been added for dealing with strings, arrays as entities, files, and overlays.

1. Strings are Added

The earliest computers dealt solely with numbers, while modern computers deal mostly with text information. By 1965, it had become evident that text processing was as important as numerical processing. The basic ingredient in any text processing is a string of characters. Strings and string variables were quickly added to BASIC. String variable names looked like numerical variable names except that the final character was a dollar sign ($). String constants were strings of characters enclosed in quotation marks:

```
LET A$ = "Hello, out there."
PRINT A$
END
```

The dollar sign was chosen because, of all the characters available on the keyboard, it most suggested the letter *s* in the word *string*. (Note that strings were always a primitive data type and not an *array of characters*, as in the C programming language.)

The early versions of BASIC also allowed string comparisons (e.g., "A" < "B" means that "A" occurs earlier in the ASCII character sequence than "B"). Other string operations were not covered. Instead, early BASIC provided a way to convert a string of characters to a vector of numbers, and vice versa. Called the `CHANGE` statement, it allowed any manipulation of strings whatsoever, since it was easy to carry out the corresponding operation of the numerical values of the characters. As an example, suppose the string N$ contains the name "John Smith" and we want to put the first name ("John") into the string F$:

```
CHANGE N$ TO N
FOR I = 1 TO N(0)
     IF N(1) = 32 THEN GOTO 250
     LET F(1) = N(I)
NEXT I
LET F(O) = I - 1
CHANGE F TO F$
```

The first `CHANGE` statement put the following numbers into the list N:

```
(74, 111, 104, 110, 32, 83, 109,
  105, 116, 104)
```

The numbers correspond to the letters in the name "John Smith;" in particular, the number 32 corresponds to the space. The `FOR-NEXT` loop copies the entries from

the list N to the list F until it encounters a space (32). The 0-th entries, N(0) and F(0), contain the number of characters in the string. While the use of the CHANGE statement was awkward, it did allow a programmer to carry out any conceivable operation or manipulation on strings until more sophisticated string-handling features were added.

2. MAT Statements are Added

Almost immediately after BASIC was invented, operations on arrays (vectors and matrices) as entities were added. For example, suppose the user wished to set each element of a numerical array to the value 17. Without matrix operations, this might be done with:

```
DIM T(10, 10)
FOR i = I to 10
    FOR j = I to 10
        LET T(i, j) = 17
    NEXT j
NEXT i
```

With MAT operations, it might be as simple as:

```
DIM T(10, 10)
MAT T = 17*CON
```

CON stood for a vector or matrix consisting of all ones (1) and of the same size as the vector or matrix being assigned to.

Another operation of importance was "inverting" a matrix. It was too much to expect that most users would be able to program their own matrix inversion routines. So BASIC allowed:

```
MAT T = INV(A)
```

where A and T both stood for square matrices having the same size.

3. Files are Added

When a program is to process only a small number of data, it is reasonable to provide those data in DATA statements included in the program. But when the number of data is large, the program and the data should be separate. By the late 1960s, most versions of BASIC had added the capability for working with data files. The following example was typical (the #1 in lines 120 and 150 refers to the *first* file named in the FILES statements):

```
100 FILES GRADES
110 FOR S = 1 TO 3
120     INPUT #1: N$
130     LET T = 0
140     FOR J = 1 TO 4
150         INPUT #1: G
160         LET T = T + G
170     NEXT J
180     LET A = T/4
190     PRINT N$, A
200 NEXT S
210 END
```

The data file named GRADES might contain:

```
JONES
78
86
61
90
SMITH
66
87
88
91
WHITE
56
77
81
85
```

The purpose of the program was to average the grades of several students. This example illustrates the type of file called the *terminal-format* file, now called a *text* file. These files consist entirely of printable characters. Many versions of BASIC also included random-access files. That term did not mean that the files contained random numbers; it meant that any record in the file could be accessed in roughly the same amount of time. Although the details varied, those versions of BASIC that included files also included the capabilities for erasing them, determining the current position in the file, determining the file's length, and so on.

4. Overlays are Added

The original BASIC allowed subroutines, using the GOSUB and RETURN statements. As early as the late 1960s, however, it was evident that allowing only GOSUB-type subroutines was limiting if one needed to write large programs. One early effort provided an overlay mechanism. While not a true subroutine in the sense we use the term today, it provided a way to get around the limited

memory sizes of the computers of the day. The following trivial example illustrates the method:

```
100 SUB NEG; POS
110 INPUT X
120 LET N = (SGN(X)+3)/2
130 GOSUB #N
140 PRINT Y
150 GOTO 110
160 END
```

The two subroutines in the example are named NEG and POS, and they are numbered 1 and 2, respectively. Which one will be called by the GOSUB statement in line 130 is determined by the value of N as calculated in line 120. If X is negative, N will be 1; if X is positive, N will be 2.

The two subroutines, NEG and POS, look like this:

```
100 LET Y = SQR(-X)
110 RETURN
120 END
```

```
100 LET Y = SQR(X)
110 RETURN
120 END
```

The purpose of this program is to compute the square root of the absolute value of a number. (The program is displayed only to illustrate overlays and is not intended to be a good solution to the square-root problem.) The important points are that the line numbers in the main program and in each of the overlays are private but all variables are shared (similar capabilities in other versions of BASIC were called *chaining with common*).

The overlay technique is one way to fit a large program into a small memory, but it did not address the need to allow subroutines whose variables as well as line numbers were private. Such subroutines could then be treated as black boxes and used without regard to possible conflicts in variable names between the subroutine and the main program. Since this kind of subroutine could not share the variables of the main program, information must be passed to the subroutine, and answers returned, through a set of special variables called *parameters*. Dartmouth BASIC in 1970 was one of the first versions to include external subroutines with parameters. For many years it remained almost the only version to include this capability.

III. GROWTH OF BASIC

BASIC began proliferating in the outside world. While the first versions were clearly based on Dartmouth's BASIC, later versions were not. The profusion of versions of BASIC can be explained by this early history. Dartmouth did not patent or copyright BASIC, nor did it attempt to trademark the name BASIC. People were thus free to modify the language in any way they felt was justified for their purposes and still call it BASIC. In addition, there was no standard, either real or *de facto*. (A real standard called Minimal Basic did appear in 1978, but it has had little influence because the language it defined was too small even for then. A standard, full BASIC appeared in 1987 and is discussed later.) Nor was there an official standard for FORTRAN in the early days. But the *de facto* standard was IBM FORTRAN, because anyone wishing to provide a FORTRAN compiler would almost certainly base it directly on IBM's FORTRAN. Dartmouth enjoyed no similar preeminence with respect to BASIC.

A. Commercial Time Sharing

Outside of Dartmouth, the first provider of BASIC was the commercial time-sharing system operated by the General Electric Corporation. At first, in 1965, GE used Dartmouth BASIC virtually unchanged. It later added features different from the ones added at Dartmouth to meet the needs of their commercial customers. After 1969, GE BASIC and Dartmouth BASIC diverged. Other companies patterned commercial time-sharing services after GE's and almost always included some version of BASIC, but these second-generation versions of BASIC were patterned after GE's rather than Dartmouth's.

Most of the early minicomputer versions of BASIC were also patterned on GE's BASIC. While there were many similarities with Dartmouth's BASIC, these second- and third-generation developers were largely unaware of the original Dartmouth design criteria for BASIC, and wider divergences appeared. The end result was a profusion of versions of BASIC, with almost no possibility of checking the tide of divergence and bringing them together.

B. Personal Computers Appear

The decade of the 1970s was an astounding period in the history of technology. The invention of integrated circuits (as a replacement for individual transistors, which in turn replaced vacuum tubes) brought about truly inexpensive computing, which paved the way for personal microcomputers. By 1980, one could purchase for a few thousand dollars a computer having the power of a million dollar machine 10 years earlier—and no end was in sight.

Makers of these new microcomputers needed a simple language that could be fit into the tiny memories then available. BASIC was invariably chosen, because: (1) it was a small language, (2) it was a simple language, and (3) it already could be found on numerous commercial time-sharing systems and on other microcomputers. The

first microcomputer versions of BASIC were very simple. They were similar in most ways to 1964 Dartmouth BASIC, but since they were not based directly on Dartmouth's BASIC, they were invariably different from it, and from each other. The explosive technological development which spawned personal computers carried with it two other major developments. The first was structured programming, which, in simplest terms, made it possible to write programs without using `GOTO` statements. The second was the sudden availability of graphic displays. Features taking advantage of these two developments were added to almost every version of BASIC, and rarely were the features alike, or even similar, among different versions.

C. Incompatible Versions of BASIC Appear

During the 1970s, BASIC became one of the most widely used computer languages in the world and one of the important languages for applications development on personal computers. People who wished to sell or give away programs for personal computers invariably wrote such programs in BASIC. By 1979, however, the incompatibility between the different versions of BASIC had become such a serious problem that one software organization advised its programmers to use only the simplest features of BASIC. Twenty-one versions of BASIC were studied, only five of which existed on personal computers; they were compared with respect to a number of features.

1. Disadvantages of Tiny Memories

These first microcomputer versions of BASIC were simple, but they were unfortunately also the victims of corner cutting. The first microcomputers had tiny memories, some as small as 4k bytes. Space saving was thus paramount. (A byte consists of 8 bits. In more familiar terms, a byte is roughly the same as a character, such as a letter or a digit. The letter "k" stands for 2 raised to the tenth power, or 1024. Thus, 4k, or 4096, bytes can contain just a bit more information than one single-spaced typed page.)

One commonly used space-saving technique was compression. As each line of a BASIC program was typed in, it was checked for correctness and then converted to a more concise internal format. For instance, the statement:

```
100 IF X < Y THEN 300
```

which contains 21 characters (bytes), could be compressed into about 11 or 12 bytes—two for each line number (if we allow line numbers to be no larger than 65535), two for each variable name, and one for each keyword (`IF`, `THEN`) and relational operator (<). A by-product of such space saving was that blank spaces were ignored, as in the original Dartmouth BASIC. When a program was listed, it was decompressed back to a readable form with spaces. To be more specific, whether the user typed in:

```
100 IF X < Y THEN 300
```

or

```
100 IF X < Y THEN 300
```

listing the program would show:

```
100 IF X < Y THEN 300
```

with exactly one space between the parts. This prevented, for example, the use of indentation to reveal the program's structure.

2. Optional LET

Space can be saved, and typing time reduced, by omitting the keyword `LET` in a `LET` statement, as with:

```
100 X = 3
```

Allowing the omission of the keyword `LET` violated one of the original premises of Dartmouth BASIC (that all statements begin with a keyword so that the assignment statement looks different from an assertion of equality), but this feature is nonetheless quite popular among personal computer users, and most, but not all, versions of BASIC allowed this feature.

3. Multiple Statements on a Line

Another feature motivated partially by the limited memory available was putting several statements on a line, as with:

```
100 LET X = 3: LET Y = 4: LET Z = 5
```

The trouble was that not all versions of BASIC allowed this feature; about half did but the other half did not. Those that did allow multiple statements used different symbols as separators—a colon (:), a solidus (/), or a comma (,). As popular as this feature is, it can become a user trap in at least one version of BASIC that appeared in the late 1970s. (A user trap is a feature whose interpretation is not self-evident and that may induce mistakes.) A concrete example is this: Many versions of BASIC extended the `IF-THEN` statement to allow more than line numbers to appear after the word `THEN`. For example,

```
100 IF X < Y THEN Z = 4: Q = 5
```

meant that, if X were in fact less than Y, the two statements following the THEN were executed. But, a programmer who followed the usual rules of putting several statements on a single line might believe that:

```
100 IF X < Y THEN Z = 4: Q = 5
```

and

```
100 IF X < Y THEN Z = 4
101 Q = 5
```

were equivalent, which they were decidedly not. Contrast this with:

```
100 X = 3: Y = 4: Z = 5
```

and

```
100 X = 3: Y = 4
101 Z = 5
```

which are equivalent.

4. Commenting Conventions

All versions of BASIC have always allowed the REM statement, which allows including remarks or comments in the program. Such comments might include the programmer's name, brief descriptions of how the program works, or a detailed explanation of some particularly tricky section of code. Many versions of BASIC also allowed comments on the same line as other BASIC statements. For instance, in

```
100 LET balance = 0 ! Starting
                      bank balance
```

the comment, which starts with an exclamation point (!), explains the purpose of the LET statement and the variable "balance." Of the 21 versions of BASIC mentioned above, several used the "!" to start the online (on the same line) comment, others usesd an apostrophe, still others used other symbols, and some did not allow online comments at all.

5. Raising to a Power

In BASIC, the symbols "+," "−," "*," and "/" stand for addition, subtraction, multiplication, and division, respectively. The choice of "+" and "−" is obvious. The choice of "/" is less obvious but is natural as it is almost impossible to type built-up fractions on a keyboard. The choice of "*" for multiplication is more difficult to see. One can represent multiplication in several ways in arithmetic and algebra. For example, if *a* and *b* are variables, the product of *a* and *b* could be denoted by:

$$ab,\ a \times b,\ \text{or}\ a \cdot b$$

in algebra. The trouble is that the first and second look like variable names. If *a* or *b* were numbers, as in 2.3, the third option might look like a decimal number rather than a product. That left the "*" to indicate multiplication. This choice is universal, not only with BASIC, but with other languages as well.

That leaves the symbol for exponentiation or "raising to a power." Some use the caret (^) which is now standard. Others used the up arrow (available on the keyboards at that time). Still others used the double asterisk (**) (taken from FORTRAN), while still others allowed more than one.

6. Number Types Reappear

One of the design goals in the original BASIC was to prevent the beginner from having to know the difference between integer and floating-point numbers. Programs were simpler, even though some of them might run more slowly. Many versions of BASIC on minicomputers and personal computers gave up this simplicity and provided integer-valued variables. The purpose was to allow programs that used mostly integer numbers, such as prime number sieve programs, to run faster. The most common approach was to have a "%" be the last symbol of the variable name. Thus,

xyz	stands for a floating-point-valued variable.
xyz%	stands for an integer-valued variable.

Another common approach was to include, somewhere near the beginning of the program, a statement like one of the following:

```
200 DEFINT I−N
200 DECLARE INTEGER I−N
```

where the I–N means that all variables with names that begin with I, J, K, L, M, or N are to be treated as integer variables.

7. Strings Proliferate

Most versions of BASIC added string concatenation, which is simply the joining of two strings, end to end. For example,

```
"Now is the time" & "for all good
  men"
```

results in

```
"Now is the time for all good men"
```

The trouble was that only a few used an ampersand (&) for this operation. The other versions used a plus sign (+) or parallel lines (‖); one used a comma, and two used other methods.

Most versions also provided several string functions, such as `LEN(s$)`, which gave the length of the string (i.e., the number of characters in it). Also provided were functions are cutting a string into pieces. For example, `SEG$(a$, 5, 7)` gave a new string consisting of the three characters of the string a$, starting with character number 5 and ending with character number 7. Other versions of BASIC included such functions as `LEFT$(a$, 7)`, which gave the leftmost seven characters of the string a$; `RIGHT$(a$, 7)`, which gave the rightmost seven characters of the string; and `MID$(a$, 5, 7)`, which gave the middle seven characters of a$, starting with character number 5.

One of the problems with having different versions of BASIC is that, although `MID$` and `SEG$` provide a similar function, the meaning of the third argument differs. For instance, if:

```
a$ = "abcdefghijklmn"
```

then

```
MID$(a$, 5, 7) = "efghijk"
```

while

```
SEG$(a$, 5, 7) = "efg"
```

Inasmuch as `MID$(a$, 1, 5)` does give the same result as `SEG$(a$, 1, 5)`, this caused confusion when users switched from one version of BASIC to another.

Most BASICs also provided ways to locate or find various patterns in a string. For example, if:

```
a$ = "Now is the time for all
     good men"
```

and one wished to locate where the word "the" appeared, one would use:

```
LET p = POS(a$, "the")
```

after which the variable p would have the value 8, which is the character number of the first character of "the" as it appears in the string a$. If the looked-for string was not to be found, `POS` gave the value 0. Most of the versions of BASIC named this function `POS`, some used other names (e.g., `INSTR`, `IDX`, `SCN`, `CNT`, or `INDEX`), while others did not even include the capability.

D. New Influences

As BASIC was growing during the 1970s, three important new ideas in programming theory and practice came into being: *structured programming*, *subroutines*, and *interactive graphics*. Each of the three was to have a profound effect on all computer languages, and particularly BASIC.

1. Structured Programming

One of the major contributions to the theory of programming was the introduction of structured programming during the 1970s. Edsgar Dijkstra, the Dutch computer scientist, is usually credited with having gotten the ball rolling. He pointed out, in the late 1960s, that the undisciplined use of `GOTO` statements seemed to be correlated with programs that contained a greater number of errors or that were difficult to modify. In simplest terms, structured programming involves adding two constructs to programming languages so that programmers can write programs without using `GOTO` statements. (A programming construct is a collection of statements that are treated as a single entity. For instance, the `FOR-NEXT` combination is a loop construct.) Of course, that is not the whole story, because these new constructs can be misused, just as `GOTO` statements can be.

The two required constructs are (1) the general loop, and (2) the `IF-THEN-ELSE`. Neither of these constructs was included in the original BASIC. The `FOR-NEXT` is *not* a general loop because the conditions that determine its completion must be known ahead of time. Thus, with

```
100 FOR I = I TO 10
110 ...
120 NEXT I
```

we know that the insides of the loop (line 110) will be executed exactly 10 times. This construct is not adequate for situations where we want to carry out the loop until, for example, some accuracy requirement has been met. Some programmers used `FOR-NEXT` loops with the `TO` value set sufficiently high and then used a `GOTO` statement to

jump out of the loop when the completion test was met. (This trick is less desirable than using a general loop construct.) Other programmers fiddled with the loop variable (the variable I in the above fragment) to force completion of the loop when the completion test was met; this practice was even less desirable. Furthermore, the IF-THEN statement in the original BASIC, based as it was on jumps to other statements, did not provide the capability found in the IF-THEN-ELSE.

To be sure, both the general loop and the general IF-THEN-ELSE can be "constructed" using GOTO and IF-THEN statements, but that is not the point. Programmers tend to make fewer errors when the language they are using provides the constructs they need to carry out their work.

As an illustration of the weaknesses of the original IF-THEN and GOTO statements, suppose we want to check a student's answer to a question in a drill program (the correct answer is 17):

```
300 IF a <> 17 then 330
310   PRINT "Right"
320   GOTO 340
330 PRINT "Wrong"
340 ...
```

Besides relying on line numbers as targets for the GOTO and IF-THEN statements, this construction is difficult to follow because the test (line 300) checks for a wrong answer rather than a right answer. Contrast this with a more modern way to make this choice:

```
300 IF a = 17 then
310   PRINT "Right"
320 ELSE
330   PRINT "Wrong"
340 END IF
```

(Indentation is used for clarity in both examples and is not essential.)

As the acceptance of structured programming grew, the reputation of BASIC declined. This was a fair assessment of most microcomputer versions of BASIC. Many of them added abbreviated constructs, such as the single-line IF-THEN and IF-THEN-ELSE discussed earlier, but these additions were often made with little thought as to how they would fit with other language features, such as multiple statements on a line.

2. Subroutines

Another limitation of early versions of BASIC was the almost complete dependence of the GOSUB and RETURN statements to provide a subroutine capability. (It must be remembered that even as late as 1980 only a few large-machine versions of BASIC allowed the external subroutines described earlier.) There are three major flaws with the GOSUB-RETURN approach. The beginning of the subroutine must be associated with a line number. Changing the line numbers of a program also changes the starting line numbers of each subroutine. While automatic renumbering programs usually change both the line numbers and the statements that refer to line numbers, programmers have to remember where their subroutines begin.

The second flaw with the GOSUB-RETURN approach is that there is no way to pass arguments to subroutines. For example, a subroutine to add two numbers might be written:

```
1000 REM SUBROUTINE TO ADD TWO
     NUMBERS
1010 LET Z = X + Y 1020 RETURN
```

To add A and B to get C, one must do something like this:

```
200 LET X = A
210 LET Y = B
220 GOSUB 1000
230 LET C = Z
```

How much more convenient it would be to replace these lines with the more modern:

```
CALL ADD (a, b, c)

SUB Add (x, y, z)
    LET z = x + y
END SUB
```

The third flaw with GOSUB-RETURN subroutines is that they are inherently part of the main program. They share line numbers and variables with the main program. A programmer might accidentally choose to use a variable I in the subroutine, forgetting that I was also being used in another part of the program. Such confusions were a frequent source of error, to put it mildly. Some other modern languages allow subroutines to be separate from the rest of the program; if this is the case, such subroutines are called *external*. External subroutines can use variables that have the same names as variables used in the main program without confusion. External subroutines can also be collected into libraries for general use. All one has to know is the subroutine's name and calling sequence, the latter being the number and type of its arguments. The lack of adequate modularization tools contributed to the declining reputation of BASIC in the early 1970s.

3. Graphics

The last big innovation that occurred during BASIC's teenage years was the explosion in graphics. It is undoubtedly true that, while BASIC fell behind other languages in adapting to structured programming, it led all other languages in embracing graphics.

Microcomputers are particularly adept at drawing pictures. We might even assert that they are better at that than they are at printing numbers. Contrast that with big machines, which did better at "number crunching." Drawing a picture with a big machine entailed much more work and much longer periods of waiting than it did with microcomputers. Since BASIC was the only language on most early microcomputers, it was natural to include line-drawing and other graphics commands in the language. If one wanted to draw pictures on a microcomputer, we may safely assert that one would prefer to use BASIC.

Surprisingly, the first interactive graphics versions of BASIC were not on personal microcomputers—they were on big time-sharing systems. Some of the early work was done on the Dartmouth time-sharing system in the late 1960s, before the first personal computers became commercially available. There was one big problem in using graphics with BASIC on most personal computers. Each computer was different and allowed a different number of pixels on the screen. (A pixel is a point on the screen. A commercial television set has about 250,000 pixels, arranged in 500 rows of 500 pixels each.) Drawing a picture on the screen of a different brand of personal computer, or a different model of the same brand, required learning how many pixels there were.

An example should make this clear. Suppose the screen allowed 40 pixels in the horizontal direction and 48 pixels in the vertical direction. Suppose that a version of BASIC had line-drawing commands as follows:

```
HLIN 10, 20 AT 30
VLIN 25, 35 AT 20
```

(These conventions correspond to medium-resolution color graphics in one popular early microcomputer BASIC.) The first would draw a horizontal line 30 pixels below the upper edge of the screen and extending from the 10th pixel from the left edge of the screen to the 20th pixel. The second would draw a vertical line 20 pixels from the left edge of the screen and from the 25th pixel below the top to the 35th pixel.

It is easier to understand these commands in terms of the coordinates of the Cartesian plane (most personal computers, however, turned the vertical axis upside down). The first draws the line given by the two end points (10, 30), (20, 30), while the second draws the line whose end points are (20, 25), (20, 35). To draw a small rectangle in the center of the screen, one might use:

```
HLIN 20, 28 AT 16
VLIN 16, 24 AT 20
HLIN 20, 28 AT 24
VLIN 16, 24 AT 28
```

Although this might seem a bit cryptic to most, it does work. The trouble comes when the user wishes to use a different graphics mode or to move to a machine that has a different number of pixels. The small rectangle will no longer be in the center of the screen (worse yet, these statements might not even be legal). The key idea is that the user should be able to specify what coordinates to use, independently of the number of pixels on a particular machine. To be concrete, suppose the user wants to use coordinates that go from 0 to 1 in the horizontal (x) direction and 0 to 1 in the vertical (y) direction. The user would first specify this using:

```
SET WINDOW 0, 1, 0, 1
```

The rectangle-drawing code might then be reduced to a single statement such as the following:

```
PLOT LINES: .4,.4; .4,.6; .6,.6;
            .6,.4; .4,.4
```

BASIC should figure out which pixels correspond to the four corners of the rectangle, and then draw it. Running the program on a different computer, which might have a different number of pixels, would still result in a small rectangle in the middle of the screen.

E. BASIC Loses Favor

Microcomputers quickly became the dominant force in computing in the late 1970s and early 1980s. BASIC was the predominant language found on these machines. Moreover, BASIC was about the only language available if one wanted to use the graphic capabilities of these machines. Despite all this, BASIC began to lose favor with computer scientists and educators. Why? There are several reasons.

Many of the shortcut features described above, which arose because of the limited power and limited memory of the early microcomputers, were continued in later versions of the language, even when more powerful computers and more spacious memories were available. Another reason was the lack of structured programming constructs. Still another was the primitive `GOSUB-RETURN` subroutine mechanisms still used by most BASICS. Educators began

to wonder how they could teach good program construction and modularization using a language that essentially does not allow either. True, some "better" versions of BASIC were available, but only on certain larger machines such as larger computer-based, time-sharing systems. These were used in secondary schools and colleges, but the total number of students who could be trained using these "better" versions of BASIC was smaller than the number who learned BASIC on a microcomputer. For example, BASIC at Dartmouth College continued to grow until, by the end of the 1970s, it contained many features found only in more sophisticated languages. It allowed external subroutines, for example, to be precompiled (preprocessed to save time) and placed in libraries for general use. But, the work done at such isolated locations did not find its way into general practice, and there was no way to curb the individuality that dictated that different manufacturers have different versions of BASIC or that the same manufacturer might have as many as 5 or 10 different versions of BASIC on its different lines of computers. That is, there was no way until an official standardization activity commenced. We describe that activity in the next section.

IV. STANDARDIZATION AND ITS FAILURE

By 1974, BASIC had become the dominant programming language for time-sharing systems. But, as we have noted, the differences were so great that users of BASIC could move from one machine to another only with difficulty. It was thus evident to many that the computer community should develop a standard for BASIC.

In the early days of computing there were few official standards, especially with programming languages. But there were *de facto* standards. Typically, they were descriptions of a programming language as provided by a dominant manufacturer. Other vendors would provide the "same" language in the hope that some of the customers might switch from the dominant manufacturer to them. This informal approach has been largely replaced in recent years by an increased reliance on official standards.

In the United States, standards are prepared under the auspices of the American National Standards Institute (ANSI). Authority is delegated to voluntary technical committees, which prepare a description of the standard. In the case of the BASIC programming language, standards work began in 1974. A standard for a small version of BASIC was completed in 1976 and published by ANSI in 1978. A standard for "full" BASIC was begun around 1976 and was published in 1987. An addendum was released in 1991. These standards were subsequently adopted by the International Standards Organization (ISO).

A. Standard BASIC

The ANSI standard for BASIC differs from the 1964 original BASIC in a number of ways. The first is that variable names can contain any number of characters up to 31, instead of just one or two. The standard also allows general names for new functions introduced by the DEF statement. In the original BASIC, all such functions had names that started with FN. Anytime one saw something like FNF(x, y), one could be quite certain that it stood for a function with two arguments. Standard BASIC allows more general names, such as cuberoot(x), instead of requiring the name to begin with fn, such as fncuberoot(x).

Standard BASIC includes several loop constructs. The general loop construct can have the exit condition attached to the DO statement, attached to the LOOP statement, or located somewhere in between. The following program fragment illustrates the middle exit from a DO loop:

```
DO
   PRINT "Enter x, n:" : x, n
   If n = int(n) then EXIT DO
   PRINT "n must be an integer;
   please reenter.
LOOP
PRINT "x to the n-th power is"; x^n
```

Alternative ways of coding that fragment without using the EXIT DO statement are either longer or more obscure. Standard BASIC also allows exiting from the middle of a FOR-NEXT loop. Standard BASIC includes a variety of constructs for making choices; for example:

```
IF x < y and x < z then
   PRINT "x is the smallest"
ELSE IF y < z then
   PRINT "y is the smallest"
ELSE
   PRINT "z is the smallest"
END IF
```

and

```
SELECT CASE roll
CASE 7, 11
   PRINT "I win"
CASE 2, 3, 12
   PRINT "I lose"
CASE else
   PRINT "I'm still in"
END SELECT
```

Standard BASIC retains one of the original design features—namely, that there be but one type of number. It

merely specifies that numerical calculations be done with at least ten significant figures of precision, without specifying how this is to be done. Standard BASIC specifies variable-length strings, which are the easiest for the beginner to handle. To extract a piece of a string, Standard BASIC uses the notation:

```
line$ [2:5]
```

which gives the second through fifth characters of the string of characters contained in the string variable, line$. Such special functions as SEG$, LEFT$, RIGHT$, and MID$ are no longer needed.

Standard BASIC includes provision for both new function definitions and named subroutines with arguments. The function definitions can contain multiple lines, as in:

```
FUNCTION answer (response$)
    SELECT CASE ucase$ (response$)
    CASE "YES"
         LET answer = 1
    CASE "NO"
         LET answer = 2
    CASE else
         LET answer = 3
    END SELECT
END FUNCTION
```

Standard BASIC allows both the function definitions and the named subroutines to be either internal or external. Internal defined functions and subroutines are useful in breaking up large programs into more easily managed pieces. External defined functions and subroutines are necessary for building libraries of routines that can be used as "black boxes," without worrying about the details.

External subroutines (and defined functions) are identified by the keyword EXTERNAL and also appear *after* the END statement of the program or within modules:

```
! Main program
DO
   INPUT prompt "Enter two
         numbers": x, y
   CALL sum (x, y, s)
   PRINT "The sum is
   PRINT
LOOP
END
!External Subroutine
EXTERNAL SUB sum (a, b, c)
    LET C = a + b
END SUB
```

Except for the calling sequence, external subroutines are entirely independent of the main program. If a particular computer allows it, they can even be separated from the main program and kept in library files.

Standard BASIC also includes a specification for *modules*, which are a way of packaging several subroutines together with the data upon which they operate. (This capability is the essential requirement of object-oriented programming.) Standard BASIC offers several types of files, the most common of which consists solely of text characters. They are called *text* files and can be printed, just as if they had been created with a screen editor (several other file types are also included in Standard BASIC). Files are opened in a typical fashion:

```
OPEN #2: name "testdata," access
         input
```

will "open" the file whose name is "testdata" for inputting only and will allow the program to refer to it by number (#2). Other options are possible, but they are all provided through English keywords (e.g., "name" and "access") and not through special punctuation, which is easy to forget.

Most versions of BASIC allow detection and recovery from errors through the use of an ON ERROR GOTO 200 type of statement. Standard BASIC provides the equivalent capability, but in a structured form. For example,

```
DO
  INPUT prompt "Filename:fname$
  WHEN EXCEPTION IN
       OPEN #1: name fname$
       EXIT DO
  USE
       PRINT "File"; fname$;
             "not there: retry."
  END WHEN
LOOP
```

This represents a typical need in a program: to allow the user to give the name of a file to be used but to allow the program to continue gracefully if that file happens to not exist. (Modern versions of BASIC use a "file open dialog box," which presents in visual form a list of file names from which to choose.)

Perhaps the single most important contribution of standard BASIC is its provision for graphics. The manner in which graphics is done is based on the GKS (Graphics Kernel System) International Standard, with a few exceptions and additions. The user always works in user coordinates (sometimes called *problem* or *world*

coordinates) regardless of how many pixels the screen contains. An example of how the user specifies these coordinates is

```
SET WINDOW xlow, xhigh, ylow, yhigh
```

The user can then draw lines with:

```
PLOT LINES: x0, y0; x1, y1; x2, y2
```

which obviously draws a line segment from the point (x0, y0) to the point (x1,y1), and then on to the point (x2,y2).

If drawing dots is preferred, the user would use:

```
PLOT POINTS: x0, y0; x1, y1; x2, y2
```

Standard BASIC allows the lines to be drawn in a variety of line styles and the points to be plotted in a variety of point styles.

Another variation:

```
PLOT AREA: x0, y0; x1, y1; x2, y2
```

will draw a triangular region, the interior being filled with the color currently in effect.

B. Impact of the Standard

While not perfect, ANSI standard BASIC defined a clean programming language. Not only would it be as easy to use as the original Dartmouth BASIC, but it also provided the MODULE feature, which would permit a form of object-oriented programming. If the computing world in 1987 and 1991, when the standards appeared, had even remotely resembled the computing world of 1974, when work on the standards commenced, the standard for BASIC might have had a major impact, but it has been largely ignored. Why? Because of the microcomputer revolution.

V. THE MICROCOMPUTER REVOLUTION

The original microcomputers were small and slow. At first they could not compete with larger, time-shared computer for sophistication. But the growth was so rapid that microcomputers, sometimes called personal computers, and work stations have now taken over almost all the computer applications once the province of large machines.

Consider the following approximate comparisons:

	Early microcomputers	**Modern personal computers**
Speed	4 MHz per byte	500 MHz per 8 bytes
Memory (RAM)	4096 bytes	64 megabytes
Disks	Less than a million bytes	More than 16 billion bytes
Monitors	Crude character only, monocolor	Excellent resolution, monocolor
Interaction	Keystrokes only	Keystrokes, mouse, voice-actuated
Modes	Simple computations	Movies, voice, internet
Application size	Several thousand lines of code	Millions of line of code

(The above numbers do not represent any actual computer but are typical. The early microcomputer numbers are typical for machines in the mid-1970s. The modern numbers are typical for machines in the year 2000.) With this almost unbelievable increase in power, it was only natural that applications had to grow as well.

What about BASIC? The versions of BASIC on the small microcomputers of the 1970s were also small and were hardly more powerful than the original Dartmouth BASIC in 1964. But, as microcomputer power rapidly grew, so did these versions of BASIC. Features were added rapidly to allow accessing the specific features of these machines. For example, most versions included the PEEK and POKE commands to get at the underlying details of the crude operating systems.

Except in rare instances, these developments ignored the standard. While most vendors attempted to make their new versions *upward compatible* with their previous versions, there were significant incompatibilities as time went on. The incompatibities were more marked between versions of BASIC from different vendors. Thus, the various versions of BASIC grew to provide programmers of applications access to the increased capabilities of computers.

VI. PRESENT AND FUTURE

In this section, we make general observations about programming language environments existing in the year 2000. Since these are commercial products, we refrain from identifying any of them by name lest we appear to endorse one or more of them, or lest we omit a version that deserves mention.

A. Basic as a Major Application Language

The modern personal computers with their large and colorful screen displays, high speed, and use of mouse or

mouse-like pointing devices dictated far more sophisticated applications than those of a few years earlier. Rather than dropping BASIC, the vendors rapidly added new features to handle the new capabilities. For example, the new applications required sophisticated treatment of objects such as windows and controls such as push buttons. For a time, these features were made available in the usual way, by adding statements or subroutine libraries to the language, but more programmer-friendly ways were devised.

B. Visual Interface Building Tools

One major development was the introduction of graphical tools to build the user interfaces. These interfaces, called graphical user interfaces (GUI), replaced the old typed commands of the original BASIC and the early microcomputer versions. With these tools, application developers can build their application's user interface by clicking and dragging the mouse. The mouse is used to select the control (i.e., push button) and to move it to a new location or resize it. Once the interface has been built, the programmer proceeds to supply substance to the application by filling in the *callback* subroutines. Initially, the callback subroutines are empty, so the programmer must fill them using a traditional programming language—BASIC, in this case.

C. Object-Oriented Programming

Object-oriented programming provides a higher level way for programmers to envision and develop their applications. Without attempting to define the concept, we merely note that one deals with *objects* and *methods*. For example, an object might be a *window*, and a method might be to *display* the window (make it visible.) As applied to BASIC, the concepts of object-oriented programming are partly substantial and partly nomenclature. Dealing with windows, movies, sound strips, internet access, and so on is made simpler, at least conceptually, by thinking of them as objects. At the other end of the spectrum, a BASIC variable, such as `X`, can be thought of as an object, while `PRINT` can be thought of as a method that can be applied to that object. This is not to diminish the importance of object-oriented programming. Its most important aspect is that the detailed coding, which must exist at the lower level, is hidden. This detailed coding can be enormously complicated in itself, but the application developer need not be concerned with it.

The developer must be familiar with the details of the BASIC language available to him, as he must fill in the substance of the callback subroutines. This BASIC language, while a descendant of the original Dartmouth BASIC, would hardly be recognized as such. A few keywords remain, but many have been changed, and many more added. In one popular version, there are over 250 keywords and function names. (Contrast this with the original Dartmouth BASIC that had 14 statements and a small handful of built-in functions.) Another popular version uses so many different keywords that it is hardly recognizable as BASIC. Still, one can trace their genealogy from the Dartmouth BASIC of 1964, and they all retain a measure of the ease of use that was the major goal underlying its invention.

SEE ALSO THE FOLLOWING ARTICLES

C AND C++ PROGRAMMING LANGUAGE • COMPUTER ALGORITHMS • MICROCOMPUTER DESIGN • PROLOG PROGRAMMING LANGUAGE • SOFTWARE ENGINEERING

BIBLIOGRAPHY

American National Standards Institute. (1978). "American National Standard for the Programming Language Minimal BASIC," X3.60-1978, ANSI, New York.

American National Standards Institute. (1987). "American National Standard for the Programming Language BASIC," X3.113-1987, ANSI, New York.

American National Standards Institute. (1991). "American National Standard for the Programming Language BASIC, Addendum," X3.113A-1991, ANSI, New York.

Frederichk, J., ed. (1979). "Conduit Basic Guide," Project: CONDUIT, Universihty of Iowa Press, Iowa City.

Kemeny, J. G., and Kurtz, T. E. (1968). "Dartmouth time sharing," *Science*, **162**, 223–228.

Kemeny, J. G., and Kurtz, T. E. (1985). "Back to Basic," Addison-Wesley, Reading, PA.

Kurtz, T. E. (1980). *In* "History of Programming Languages" (R. L. Wexelblatt, ed.), pp. 515–549. Academic Press, New York.

Sammet, J. (1969). "Programming Languages, History and Fundamentals," Prentice-Hall, Englewood Cliffs, NJ.

C and C++ Programming Language

Kaare Christian
Optical Imaging Inc.

GLOSSARY

ANSI C Version of C standardized by the ANSI (American National Standards Institute) X3J11 committee.

Array Data type that has multiple elements, all of the same type. Individual elements are accessed using a numeric index expression. C array elements are numbered from zero to one less than the number of array elements.

Base class Class from which another can be created by adding and/or redefining elements.

Cast C operation that converts an expression from one type to another.

Class C++ data type whose members (elements) may consist of both procedures and data (information). C++ classes are syntactically based on C structures.

Declaration Description of the name and characteristics of a variable or procedure.

Dereference To access the value that a pointer expression points toward.

Derived class Class that has been created from a base class.

Embedded assignment Assignment statement that appears in a context where many people would expect simple expressions, such as in the control expression of an if or while statement.

Enumeration C data type that allows you to create symbolic names for constants.

Inheritance Capability of a programming Ianguage that allows new data types to be created from existing types by adding and/or redefining capabilities.

Iterate To perform a task repeatedly.

K&R C The first version of C, which was specified in the book *The C Programming Language* by Kernighan and Ritchie.

Multiple inheritance Similar to inheritance, but a capability in which new data types can be created simultaneously from sev

Object An instance of a class. An object occupies a region of storage, is interpreted according to the conventions of the class, and is operated on by class member functions.

Object-oriented programming Style of programming in which classes are used to create software objects

whose use and function corresponds to real world objects.

Operand Value or expression that is acted on by an operator.

Operator Something that can combine and modify expressions, according to conventional rules, such as the + (add) operator that adds its operands, or the * (dereference) operator that can access a value pointed at by a pointer expression.

Pointer Constant or variable whose value is used to access a value or a function. The type of a pointer indicates what item the pointer accesses. The accessed item can itself be a pointer.

Polymorphism Ability of a routine or operator to have various behaviors based on the dynamically determined (runtime) type of the operand.

Standard I/O library (stdio) Set of routines that provides input/output (I/O) facilities that are usable on most systems that support C.

Strongly typed Said of a programming language that only allows operations between variables of the same (or similar) type.

Structure C data type that has multiple elements, each with its own type and name.

Type Characteristic of a value (a variable, a constant, or the return value of a procedure) that specifies what values it can attain and what operations can be performed on it.

Type checking Checking performed by most languages to make sure that operations are only performed between compatible data types.

Union C data type that has multiple elements, each with its own type and name, but all of which are stored at the same location.

Usual arithmetic conversions Conversions that C performs to allow expressions involving different types of operands.

Weakly typed Said of a programming language that allows operations between various types.

WG21 ISO committee that developed a C++ standard.

X3J11 ANSI committee that developed a C standard.

X3J16 ANSI committee that developed a C++ standard.

C IS A FLEXIBLE computer programming language—it has a combination of low- and high-level features that make it a powerful language for a wide range of applications. The low-level aspects of C help programmers access machine-specific features and write efficient programs, while the high-level features promote clear and concise expression of a programmer's ideas. Implementations of C are available on nearly every computer architecture, and it is the only programming language that is available for controlling many computer-like devices, such as graphics processors, signal processors, and other special-purpose programmable machines.

C++ is an extension to C that adds support for object-oriented programming and for generic programming. Its major new feature, the class, is a customizable data type that combines data elements with procedures. Classes allow C++ programmers to create objects, so that the structure of programs can reflect the structure of the original problem. C++ also includes *templates* so that you can write software that generically adapts to different types of data, iostreams, which is a new input/output library, and the standard template library that implements containers and many standard algorithms.

I. THE EVOLUTION OF C AND C++

During the early 1970s, Ken Thompson created the first version of the UNIX' system on an obsolete PDP-7 computer. The system showed promise, but it was written in assembly language, which made it impossible to move it to a more modern computer. In 1973, Dennis Ritchie developed the C language, and then he and Thompson rewrote the UNIX system in C, and moved it to more up-to-date computers. The advantage of C over assembler is that it can operate on many different computer architectures, but like assembler, C gives the programmer fine-grained control of the computer hardware. This key step demonstrated that an operating system could be written in a higher level language, and it also proved that C was a powerful and efficient language that could be used for applications once thought to require arduous assembly language development.

C is part of the FORTRAN and ALGOL family of programming languages. In this family, programs are specified as a series of operations upon data using a notation that resembles algebraic notation. Individual steps in a program can be executed based on a logical condition, executed repetitively, or grouped together and executed as a unit. Most data items are named, and data items have specific types, such as character, whole number, or real number.

Within this general family of programming languages, C owes the most to two specific forerunners, BCPL and B. BCPL is a systems programming language that was developed by Martin Richards, while B is a language developed by Ken Thompson; its most interesting feature is its scarcity of data types.

The first book on C was *The C Programming Language* by Kernighan and Ritchie. It included an appendix that compactly specified the language. The version of C specified in the book is often called K&R C. Nearly all versions

of C, even if they contain additional features, will at minimum contain all of the features of K&R C.

In 1979, AT&T disseminated a paper written by B. R. Rowland that specified a few minor changes to the C language. Some of these features were anticipated in the original K&R definition, others were a result of experience with the language. A more formal description of these changes appeared in the 1984 AT&T Bell Laboratories Technical Journal. Most of these features have been in widespread use since the early 1980s.

Although C was a widely used language by 1980, there was no attempt to forge an official standard until 1984, when the ANSI (American National Standards Institute) committee X3J1I was convened. Thus by the time the committee started work there was already over a decade of experience with the language, and there was a huge existing body of C language software that the committee wanted to preserve. The role of the X3J11 committee was primarily to codify existing practice, and only secondarily to add new features or to fix existing problems.

During the early l980s, while C's position as a leading development language was being consolidated, Bjarne Stroustrup of AT&T Bell Laboratories developed a C language extension called C with Classes. The goal of C with Classes was to create a more productive language by adding higher level abstractions, such as those found in Simula, to C. The major enhancement was the *class*, a user definable data type that combines traditional data elements with procedures to create and manipulate the data elements. Classes enable one to adopt an *object-oriented* programming style, in which programs are composed of software objects whose design and use mirrors that of real world objects.

By 1985, C with Classes had evolved into C++ (pronounced "*C plus plus*"), and it began to be used outside of AT&T Bell Laboratories. The most important reference for C++ at that time was *The Annotated C++ Reference Manual* by Ellis and Stroustrup. In 1987 the International Standards Organization (ISO) formed Working Group 21 (WG21) to investigate standardization of C++, while at about the same time ANSI convened committee X3JI6 to create a standard for the C++ programming language. In late 1998 the standards efforts concluded with the publication of ISO/IEC 14882-1998, a standard for C++.

Building C++ on top of C has worked well for a variety of reasons. First, C's relatively low-level approach provided a reasonable foundation upon which to add higher level features. A language that already had higher level features would have been a far less hospitable host. Second, programmers have found that C++'s compatibility with C has smoothed the transition, making it easier to move to a new programming paradigm. Third, one of Stroustrup's most important goals was to provide advanced features with roughly similar efficiency to that of C. This goal, which has been largely attained, gives C++ major efficiency advantages over most other object-oriented languages.

II. STANDARD C FEATURES

C like most programming languages, is based on a core feature set that is common and uncontroversial. The core features deal with the rules for storing and managing data, and with control statements (control structures) that let a programmer specify a sequence of operations on data.

A. Data Types

C's basic data types include single characters, short and long integers (signed and unsigned), enumerations, and single and double precision floating-point numbers. Typical precisions and ranges for the numeric types on 32-bit computers are shown in Table I. C also has one other basic type, void, which will be discussed in Section III.G.

C has a cavalier attitude toward operations involving different numeric types. It allows you to perform mixed operations involving any of the numeric types, such as adding a character to a floating-point value. There is a standard set of rules, called the *usual arithmetic conversions*, that specifies how operations will be performed when the operands are of different types. Without going into detail, the usual arithmetic conversions typically direct that when two operands have a different precision, the less precise operand is promoted to match the more precise operand, and signed types are (when necessary) converted to unsigned.

C enumerations allow the programmer to create a data type whose values are a set of named numeric constants. Unfortunately, support for enumerations in the popular C compilers has been inconsistent and unreliable, and they

TABLE I Basic C Data Types[a]

Type	Size (bytes)	Range
char	1	−128 ... 127
unsigned char	1	0 ... 255
short	2	−32768 ... 32767
unsigned short	2	0 ... 65535
int	4	−2,147,483,648 ... 2,147,483,647
unsigned int	4	0 ... 4,294,967,295
long	4	−2,147,483,648 ... 2,147,483,647
unsigned long	4	0 ... 4,294,967,295
float	4	-3.4×10^{-38} ... 3.4×10^{38}
double	8	-1.7×10^{-308} ... 1.7×10^{308}

[a] On typical 32-bit computers.

are not widely used. Instead of enumerations, most programmers use the preprocessor *#define* statement to create named constants. This approach lacks some of the merit of enumerations, such as strong type checking, but preprocessor definitions, unlike enumerations, are available on all implementations of C.

In addition to its basic data types, C has four more sophisticated data types: structures, unions, bitfields, and arrays.

A *structure* is a way of grouping data. For example, an employee's record might contain his or her name, address, title, salary, department, telephone number, and hiring date. A programmer could create a structure to store all of this information, thereby making it easier to store, retrieve, and manipulate each employee's data. Each element in a structure has a name. C language structures are analogous to records in a database.

A *union* is a way of storing different types of items in a single place. Of course, only one of those types can be there at any one time. Unions are sometimes used to provide an alternate access to a data type, such as accessing the bytes in a long integer, but the most common use is to save space in a large data set by storing only one of a set of mutually exclusive pieces of information.

A *bitfield* is somewhat like a structure, but each member of a bitfield is one or more bits of a word. Bitfields are a compact way to store small values, and they are also a convenient way to refer to specific bits in computer control registers.

An *array* is a sequence of data items. Each item in an array is the same type as all the other items, though they all may have different values. Each array has a name, but the individual elements of an array do not. Instead, the elements in an array are accessed using an index, which is a numeric value. The first element in a C array is accessed using the index 0, the next using the index 1, and so on up to the highest index (which is always one less than the number of elements in the array).

Whereas there are many operations that can be performed on C's numeric data types, there are only a few operations that are applicable to the four more complex data types. These operations are summarized in Table II.

TABLE II Operations on C's Advanced Data Types

Array	Compute its address
	Access an individual element using index expression
Structure, union, or bitfield	Access it as a whole unit, such as assign one to another, or supply one as a parameter to a procedure
	Access individual elements by name
	Compute its address
	Each type of structure, union, or bitfield is only compatible with others of the same type.

TABLE III Arithmetical and Logical Operators

Arithmetical			
+	Addition (binary)	–	Subtraction (binary)
+	Force order of evaluation (unary)	–	Negation (unary)
++	Increment	––	Decrement
*	Multiplication	/	Division
		%	Remainder
Logical			
==	Equality	!=	Not equal
<	Less than	>	Greater than
<=	Less than or equal	>=	Greater than or equal
&&	Logical AND	||	Logical OR
!	Logical NOT		

B. Operators

C is known as an operator-rich language for good reason. It has many operators, and they can be used in ways that are not allowed in tamer languages, such as Pascal. The basic arithmetical and logical operators, detailed in Table III, are present in most programming languages. The only unusual operators in Table III are increment and decrement. For ordinary numeric variables, increment or decrement is simply addition or subtraction of one. Thus the expression

```
i++;
```

(*i* is a variable; the expression is read aloud as "*i* plus plus") is the same as the expression

```
i = i + 1;
```

(again *i* is a variable; the expression is read aloud as "*i* is assigned the value *i* plus one"). However, the increment and decrement operators can also be used inside a larger expression, which provides a capability not provided by an assignment statement. More on this special capability in Section III.D.

C's bit manipulation operators, shown in Table IV, provide a powerful facility for programming computer hardware. They provide the same capabilities as the traditional logical operators but on individual bits or sets of bits. There are also two shift operators, which allow the bits in an integer or character to be shifted left or right.

TABLE IV Bit Manipulation Operators

&	Bitwise AND	|	Bitwise OR
~	Bitwise complement	^	Bitwise exclusive OR
≪	Left shift	≫	Right shift

TABLE V Assignment Operators

=	Assign	^=	Assign bitwise exclusive OR
&=	Assign bitwise AND	|=	Assign bitwise OR
<<=	Assign left shift	>>=	Assign right shift
+=	Assign sum	−=	Assign difference
*=	Assign product	/=	Assign quotient
		%=	Assign remainder

The assignment operators, shown in Table V, begin to hint at C's operator-rich capabilities. In a traditional assignment statement, an expression is evaluated to produce a result, and then that result is stored in a variable. Assignment operators reflect the fact that the initial value of the result variable is often used in the expression. One simple example is

```
i = i * 2;
```

(*i* is a variable; this statement means that *i* is multiplied by two, and then the result is stored in *i*.)

Using the multiplication assignment operator, this can be written

```
i *= 2;
```

In simple situations assignment operators do not provide much advantage. However, in more complex situations they can be very important. One advantage is that the address calculation (to access the variable) only needs to be performed once. For example, the statement

```
x[2*i] [j/100] [k%10] += 100;
```

adds 100 to one element of a three-dimensional array. The work in this expression is the elaborate indexing calculation on the left side, which only needs to be done once because the addition assignment operator is used. Another advantage is that the assignment operators can provide clarity, by emphasizing the dual role of the variable on the left of the assignment operator.

In a computer, all data is stored in memory locations which are identified by unique numeric addresses. Most programming languages try to present a conceptual framework at a high enough level to avoid the low details, such as where a variable is stored. C, however, makes easy and common use of address information, and it contains a set of operators for working with addresses, which are shown in Table VI. For storing addresses, C contains *pointer* variables, which are variables that contain the addressess of other variables (or occasionally the addresses of functions).

The *indirection* operator (*) takes the address generated by an address-valued expression, and accesses the value stored at that address. For example, if *p* is a variable whose type is pointer to a character, then writing

```
p
```

in a program symbolizes an expression that contains the address of a character, while the expressions

```
*p
```

accesses that character. The expression

```
*(p + 1)
```

accesses the following character in memory, and so on.

The *address-of* operator (&) does the opposite. It takes a reference to a variable, and converts it to an address expression. For example, if *f* is a floating-point variable, the expression

```
f
```

accesses the floating-point variable, while the expression

```
&f
```

is the address of the floating-point variable. Used together, these two have a null effect. Thus, the expression

```
*&f
```

is equivalent to *f* by itself.

The sequential evaluation operator is used to sneak two or more expressions into a place where, syntactically, only one expression is expected. For example, C while loops use a control expression to control the repetition of the loop. While loops execute so long as the control expression is true. A typical while statement is

```
i = 0;
while (i++ < 10)
  processData();
```

TABLE VI Miscellaneous Operators

*	Indirection	&	Address of
,	Sequential evaluation	?:	Conditional (tertiary)
sizeof	Size of type or variable	(*type*)	Type cast
.	Member of	−>	Member pointed toward
[]	Element of array	()	Parentheses (grouping)

In this loop, the repetition will continue as long as the value of the *i* variable is less than 10. The ++ increments the value of *i* after each test is made, and the body of the loop is a call of the function named *processData*. If we want to have the control expression also increment a variable named *k* each time, we can use

```
while (k++, i++ <10)
  processData();
```

The comma following *k*++ is the *sequential evaluation* operator. It specifies that the *k*++ expression should be evaluated, and then the *i*++<10 expression should be evaluated. The value result is always that of the rightmost expression.

C's *conditional* operator, which is also called the *tertiary* operator, lets one use the logic of an if statement inside an expression. The syntax of the conditional operator is difficult. The control expression is followed by a question mark, followed by two variant expressions separated by a colon:

```
controlexpr ? expr1 : expr2
```

If the control expression is true, the result is *expr1*, otherwise, the result is *expr2*. For example, one might want to use a variable called *index* to select a specific element of an array named *stheta*. If *index* is 0, we want to access the zeroth element, if *index* is 1 or −1, we want to access the first element of the array, if it is 2 or −2, we want to access the second element, and so on. This is easily written using a conventional if statement:

```
if (index > 0)
  sx = stheta[index];
else
  sx = stheta[-index];
```

Instead of this four-line if statement, we can use a one-line conditional expression:

```
sx = stheta[index > 0 ? index :
-index];
```

The biggest advantage of the conditional operator is not in simple examples, like that above, but in more complicated settings, in which it allows one to avoid duplication of complicated expressions.

The *sizeof* operator is one of C's most important features for writing portable programs. It helps a program adapt to whatever computer architecture it is running on by producing the size (in bytes) of its operand. The operand may either be a variable or the name of a data type. For example, *sizeof* is often used in conjunction with the standard C memory allocation routine, *malloc()*, which must be passed the number of bytes to allocate. The statement

```
iptr = malloc(1000*sizeof(int));
```

(*iptr* is a pointer to an integer) allocates enough space to store an array of 1000 integers. This statement will work correctly on all machines that support C, even though different machines have different sized integers, and thus need to allocate different amounts of memory for the array.

C, as originally designed, was a very weakly type-checked language. In the earliest versions, pointers and integers were (in many situations) treated equivalently, and pointers to different types of structures could be used interchangeably. Since that time, stronger and stronger type checking has become the norm. By the time of the ANSI C committee (mid-l980s), most implementations of C encouraged programmers to pay much more attention to issues of type compatibility.

One of the most important tools for managing types is the *cast* operator. It allows a programmer to specify that an expression should be converted to a given type. For example, if variables named *tnow* and *tzero* are long integers, the natural type of the expression *tnow-tzero* is also a long integer. If you need another type, you can specify the conversion using a cast operator:

```
(unsigned int)  (tnow−tzero)
```

The parenthesized type name, which is (*unsigned int*) in this case, specifies that the value of the following item should be converted to the given type.

The *member-of* (.) and *member-pointed-toward* (−>) operators are used to access members of structures. For example, if *box* is a structure variable with a member named *TopRight*, then the reference *box.TopRight* (the period in the expression is the member-of operator) accesses the *TopRight* member of the *box* structure. The difference between these two operators is that the member-of operator expects the item to its left to be a structure variable, while the member-pointed-toward operator expects the item to its left to be a pointer to a structure. Both operators expect that the item to the right is the name of a member of the structure.

The member-pointed-toward operator is actually a shortcut. For example, if *pBox* is a pointer to a box structure (with the *TopRight* member mentioned previously), then the expression

```
pBox->TopRight
```

accesses the *TopRight* member of the structure. This is actually a shortcut for writing

```
(*pBox).TopRight
```

which uses the indirection operator (*) to dereference the pointer-to-structure and then uses the member-of operator to access the given member.

The last two operators in Table VI are square brackets (for array subscripting) and parentheses (for grouping). These two operators, together with the assignment operator, are familiar features from other programming languages, but they are not considered operators in most languages. However, in C these elements are considered to be operators to make the expression syntax as regular, powerful, and complete as possible.

C. Control Structures

In contrast to its eclectic set of operators, C has an unremarkable collection of control structures. One possible exception is the C for loop, which is more compact and flexible than its analogs in other languages. The purpose of the C control structures, like those in other languages, is to provide alternatives to strictly sequential program execution. The control structures make it easy to provide alternate branches, multi-way branches, and repeated execution of parts of a program. (Of course, the most profound control structure is the subroutine, which is discussed in Section II.D.)

A *compound statement* is simply a group of statements surrounded by curly braces. It can be used anywhere that a simple statement can be used to make a group of statements into a single entity.

```
for (i=0; i<10; i++) {
   x[i] = 0;
   y[i] = 0;
   z[i] = 0;
}
```

In the example above, the three assignment statements form a compound statement because they are enclosed in the curly braces. (The *for* statement will be discussed later in this section.)

The *goto* statement is the simplest, the oldest, and the most general control statement. Unfortunately, it is also one of the most easily abused statements, and its use is discouraged. It is seldom used by people who write C programs, but it is often used extensively in C programs that are generated by (written by) other programs. In programs written by people, the goto is usually reserved for handling exceptional conditions, such as branching to an error-handling block of code.

The *if* statement is used to provide alternative execution paths in a program. In an if statement, one of two alternate possibilities is taken, based on the true/false value of a test expression. The syntax is the following.

```
if (expr)
   statement1;
else
   statement2;
```

The *else* part is optional, and either *statement* may be a compound statement. (The expression and the statement above are shown in italics, to indicate that they may be any C expression or C statement. The words if and else are keywords, which must appear exactly as shown, which is why they are not shown in italics.) It is very common for the else part of the if statement to contain another if statement, which creates a chain of if-statement tests. This is sometimes called a cascaded if statement:

```
if (code == 10)
   statement1;
else if (code < 0)
   statement2;
else if (code > 100)
   statement3;
else
   statement4;
```

In the series of tests shown here, only one of the four statements will be executed.

An alternative multi-way branch can be created by a C *switch* statement. In a switch statement, one of several alternatives is executed, depending on the value of a test expression. Each of the alternatives is tagged by a constant value. When the test expression matches one of the constant values, then that alternative is executed.

The syntax of the switch statement is somewhat complicated.

```
switch (expr) {
case const1:
   statement1;
   break;
case const2:
   statement2;
   break;
default:
   statement3;
   break;
}
```

In this skeleton switch statement, *expr* is the test expression and *const1* and *const2* symbolize the constants that identify the alternatives. In this example, each alternative is shown terminated by a break statement, which is common but not required. Without these break statements, flow of control would meander from the end of each alternative into the beginning of the following. This behavior is not usually what the programmer wants, but it is one of the possibilities that is present in C's switch statement. The break statement will be discussed further later.

The switch statement is less general than the cascaded if statement, because in a cascaded if statement each alternative can be associated with a complex expression, while in a switch statement each alternative is associated with a constant value (or with several constant values; multiple case labels are allowed).

The switch statement has two advantages over the more flexible cascaded if statement. The first is clarity; when a solution can be expressed by a switch statement, then that solution is probably the clearest solution. The second is efficiency. In a cascaded if statement, each test expression must be evaluated in turn until one of the expressions is true. In a switch statement, it is often possible for the C compiler to generate code that branches directly to the target case.

A *while* loop lets you repeatedly execute a statement (or group of statements) while some condition is true. It is the simplest iterative statement in C, but it is also very general.

```
while (ch != EOF)
   statement;
```

The body of this loop (the statement) will be executed repeatedly until the value of the *ch* variable becomes equal to the value of the standard predefined constant *EOF* (end of file). Presumably, something in the *statement* will alter the value of ch so the loop will end. Also, it is presumed that ch is assigned a value prior to the execution of the while statement. (Note that the *statement* will not be executed if the initial value of ch is EOF.)

It is easy to use a while loop to step through a series of values. For example, the following while loop zeroes the first ten elements of an array named *x*. (An integer variable named *i* is used as the array index and as the loop counter.)

```
i = 0;
while (i < 10) {
   x[i++] = 0;
}
```

A close relative of the while loop is C's *do* loop. It repeats a statement (or a group of statements) while a condition is true, but the condition is tested at the bottom of the loop, not at the top of the loop. This ensures that the body of the loop will always be executed at least once.

```
i = 0;
do
   x[i] = 0;
while (++i < 10);
```

As with a while loop, something in the body of the loop or the control expression presumably changes the value of the test expression, so that the loop will eventually terminate.

C's most elaborate loop is the *for* loop. Here is the general form of the for loop:

```
for (init_expr; test_expr;
  inc_expr)
   statement;
```

As in the previous examples, the *statement* will be repeatedly executed, based on the values of the three control expressions. The *init_expr* is an initialization expression. It is executed just once—before the body of the loop starts to execute. The *test_expr* is executed before each iteration of the loop. If it is true, the next iteration will occur, otherwise the loop will be terminated. The *inc_expr* is executed at the conclusion of every iteration, typically to increment a loop counter.

Here is a typical for loop. It performs the same task as before, setting the first ten elements of an array named *x* to zero, using a loop counter named *i*:

```
for (i=0; i<10; i++)
   x[i] = 0;
```

Note that the while version of this loop was four lines, while the for version of the loop is just two. This for loop is an example of what Kernighan and Ritchie, authors of the seminal reference book on C, call "economy of expression," which is one of C's most distinctive traits.

C has two additional flow of control statements, break and continue, that augment the capabilities of the loops that have just been described. The *break* statement is used to break out of (to immediately terminate) the enclosing do, while, for, or switch statement. Its use is routine in switch statements, to terminate the switch statement at the conclusion of each individual case. In do, while, and for loops, break is often used to terminate the loop prematurely when an error or special condition occurs. Using a break in the body of a loop often simplifies the control expression of the loop, because it allows special case code to be placed elsewhere.

The *continue* statement is used to immediately start the next iteration of the surrounding do, while, or for loop. It is somewhat like a jump to the end of a loop. For example, suppose a program is reading a list of words from a file. Certain processing must occur for most words, but a few words are just ignored. Here is a pseudocode sketch (pseudocode isn't true C; it merely expresses an idea without following strict syntax) of how that could be written using the continue statement.

```
while (readaword ()) {
  if (word is in the ignore list)
        continue;
  process a word;
}
```

The effect of the continue statement is to skip the *process a word* part of the loop body for words that are in the ignore list.

D. Procedures

Procedures are tools for packaging a group of instructions together with a group of local variables to perform a given task. You define a procedure by specifying what information is passed to the procedure each time it is activated, listing its local variables, and writing its statements. As discussed in Section II.B, the statements inside a procedure can access global data that are declared outside the procedure, but it is not possible for other procedures to access the data that are declared within a procedure (unless the procedure exports the address of a local variable). This insularity is the most important feature of procedures. It helps the programmer to create small, easily understandable routines.

Figure 1 contains a procedure to solve the quadratic equation

$$Ax^2 + Bx + C = 0$$

using the well-known quadratic equation:

$$\frac{-B \pm \sqrt{B^2 - 4 \times A \times C}}{2 \times A}$$

For example,

$$2x^2 + 3x + 1 = 0$$

is a quadratic equation (A is 2; B is 3; C is 1) whose solution is

$$\frac{-3 \pm \sqrt{3^2 - 4 \times 2 \times 1}}{2 \times 2},$$

which simplifies to

$$\frac{-3 \pm \sqrt{9 - 8}}{4}$$

which has two solutions, -1 and -0.5.

The first part of *solve* specifies the procedure name, parameter names, and parameter types. The header of the *solve* procedure indicates that it expects three parameters, which it also calls *a*, *b*, and *c*. The body of *solve* calculates the solution's discriminant, which is the expression inside the square root symbol, and then calculates the answers, based on whether the discriminant is positive, zero, or negative. Most of the body of *solve* is a large *if* statement that handles each of the three types of disciriminants.

In a program, you can invoke the *solve* procedure as follows:

```
solve (2.0, 3.0, 1.0);
```

In the example above *solve* is invoked using constant numerical values, but you could also use numerical variables.

```
/*
 * solve the quadratic equation
   ax**2 + b*x + c = 0
 * using the formula x = ( -b
   +/- sqrt (b**2 - 4*a*c))/2*a
 */
void solve (double a, double b,
double c)
{
   double d;   // the discriminant
   double r1, r2;
   d = b * b - 4 * a * c;
   if (d > 0) { // two real roots
     r1 = (-b - sqrt (d))/(2 * a);
     r2 = (-b + sqrt (d))/(2 * a);
     printf("x1 = %g, x2 = %g,
    %g\n", r1, r2);
   } else if (d == 0.0) { // one
     real root
     r1 = -b/(2 * a);
     printf("x1 = x2 = %g\n", r1);
   } else {  // two real/imaginary
     roots
     r1 = -b/(2 * a);
     r2 = sqrt (abs(d))/(2 * a);
     printf("x1 = %g + %gi,
      x2 = %g - %gi\n", r1,
      r2, r1, r2);
   }
}
```

FIGURE 1 A program to solve the quadratic equation.

III. KEY C FEATURES

The features of C that are discussed in the following subsections set C apart from many other languages. Many of these features are found in some form in other languages, but only C combines them within a consistent framework.

A. Separate Compilation

Separate compilation is the ability to develop a program in several independent pieces. Each piece is stored in its own file in the computer, and each can be edited and worked on separately. Each source file is independently translated (compiled) into a machine language file, which is called an *object file*. After the individual parts have been written and compiled, they are combined to form a complete program in a process called *linking*.

C's support for separate compilation has allowed it to provide many vital services in external software libraries. For example, many programming languages have built-in facilities for assigning one text string to another, raising a number to a power (exponentiation), or for performing data input and output. C has all of these capabilities, but they are provided by external I/O libraries. This feature of C has the benefit of making the compiler smaller and easier to maintain, and it also increases the flexibility of C, because it allows programmers to rework some of these features to fit special circumstances. However, it has the disadvantage that some of these vital features are not integrated into the language as closely as in other languages, and it forces the programmer to endure the overhead of subroutine calls more than might otherwise be necessary.

C has several features that facilitate separate compilation. The static and extern storage classes are used to organize programs. The *static* storage class is used to create local data or procedures. This means that things declared static cannot be accessed from outside the current file (unless a procedure in the file broadcasts their address). The *extern* storage class is used to reference data (or procedures) that are declared in other files. By default, you can reference *procedures* from other files simply by writing their name, but for *data* in other files, you must declare it with the extern storage class before it can be referenced.

B. The C Preprocessor

Another feature that facilitates separate compilation is the C preprocessor. It makes it easy for a group of files to all reference a common set of definitions and extern declarations. The *C preprocessor* is an early stage of the C compiler that makes several alterations to program source code files, in preparation for the more traditional compilation phase. The preprocessor makes it easier to create large programs that are stored in several files, because separate files can reference common include files that contain definitions and external declarations. It also performs, just prior to the true compilation phase, some of the chores of a traditional text editor, making it easier to avoid keeping different versions of a program's source code files for different systems or circumstances.

Traditionally, the C preprocessor has provided three major features of the C language: a simple macro facility, file inclusion, and conditional compilation.

File inclusion is controlled by the *#include* C preprocessor mechanism. When a *#include* statement is encountered, the preprocessor replaces it with the contents of the referenced file. For example, the file named *stdio.h* contains definitions and references that are required by programs that are using the standard I/O library. It is included by the following statement.

```
#include <stdio.h>
```

During the preprocessing phase, this statement will be replaced by the contents of the *stdio.h* file, so that the later phases of the compiler will only see the contents of *stdio.h*.

The macro feature of the C preprocessor allows you to replace one item by another throughout a program. This has many uses, such as creating named constants, creating in-line subroutines, hiding complicated constructs, and making minor adjustments to the syntax of C. Macros can have parameters, or they can simply replace one item of text with another. Macros are created using the *#define* mechanism. The first word following *#define* is the name of the macro, and following names are the replacement text.

There are several ubiquitous C macros including *NULL*, an impossible value for pointers; *EOF*, the standard end marker for stdio input streams; and the single character I/O routines, *getc()* and *putc()*. *NULL* and *EOF* are simply named constants. In most versions of C, they are defined as follows:

```
# define EOF (-1)
# define NULL 0
```

Conditional compilation, the third traditional function of the C preprocessor, allows a programmer to specify parts of a program that may or may not be compiled. This feature is used for many purposes, such as writing programs that can be compiled on different computer systems. The conditional compilation lets programmers make small adjustments to the program to adapt to the various computer systems. In addition, conditional compilation is often used to manage debugging features, which should be omitted once the program is finished.

C. Novel Declaration Style

In some languages, the major data types are simple things, such as numbers or characters, and arrays of simple things. C is more complicated because C also has pointers to simple things, pointers to pointers, and functions that can return complicated items, such as a pointer to a function. To declare one of these hybrid data types, you must have a way of describing what you are declaring. Rather than develop a new syntax for describing this large assortment of data types, the approach in C is to make declarations mirror the use of an item. This straightforward idea has not proven to be simple for most people to understand, but it must be understood to work with C.

Declaring simple things is easy. For example, the declaration

```
int a;
```

states that *a* is an integer. The next simplest declaration is to declare an array of something, for example, an array of integers.

```
int b[10];
```

This declaration states that *b* is an array of ten integers. *b[0]* is the first element in the array, *b[1]* is the next, and so on. Notice that the declaration does not contain a keyword stating that *b* is an array. Instead, C's standard array notation, *b[subscript]*, is used in the declaration.

The next simplest declaration creates a pointer to a simple type, such as a pointer to an integer.

```
int *c;
```

This declaration states that **c* is an integer. Remember that * is the C indirection operator, which is used to dereference a pointer. Thus, if **c* is an integer, then *c* itself must be a pointer to an integer.

Another thing that can be declared is the return type of a function. The following declaration states that *d* is a function returning an integer.

```
int d();
```

The ( ) in the declaration indicate that *d* is a function. When *d* is invoked in the program, it can be used in any situation where an integer variable is used. For example, you could write

```
i = 2 * d() + 10;
```

to indicate that the integer variable *i* should be assigned twice the value returned by the *d* procedure plus ten.

The simplest rule for understanding a declaration is to remember that if you use the item just as it is declared it will have the simple type mentioned in the left part of the declaration. The next step is to learn the meanings of the three operators that are used in many declarations: a pair of parentheses indicates a function, square brackets indicate an array, and the asterisk indicates a pointer. Also remember that things to the right of a variable name (parentheses and square brackets) bind more tightly than things to the left.

The following declaration specifies a function named *e* that returns a pointer to an integer

```
int *e();
```

Note that the above declaration does *not* declare a pointer *e* to a function returning an integer, because the parentheses to the right of *e* take precedence over the indirection operator to the left.

When you are verbalizing a declaration, start from the inside and work out, and remember that it is helpful to read ( ) as "function returning," [] as "array of," and * as "pointer to." Thus, this declaration above could be read "*e* is a function returning a pointer to an *int*."

There are a few restrictions on what you can declare in C. For example, you can declare a function, a pointer to a function, or an array of pointers to functions, but you are not allowed to declare an array of functions.

D. Operator-Rich Syntax

C has the usual assortment of numeric operators, plus some additional operators, such as the operators for pointers, the assignment operators, the increment/decrement operators, the comma operator, and the conditional operator. With just this rich set of operators, C could be considered to have an operator-rich syntax.

But C goes one step further. it considers the expression to be a type of statement, which makes it possible to put an expression any place a statement is expected. For example, *c*++ is a complete statement that applies the increment operator (the ++ operator) to the variable named *c*.

C programs take on a very dense appearance when assignment statements are used in the control expressions of loops and if statements. For example, the following snippet of code is extremely common.

```
int ch;
while ((ch = getchar()) != EOF)
  ;
```

The control expression of this while loop calls *getchar* to read in a character, assigns that character to the *ch* variable, and then runs the body of the loop (which in the above example is empty, causing the above code to read in and ignore all of the input). The loop terminates when *getchar* returns the value *EOF* (end of file; a symbolic constant that is defined in the *stdio.h* include file).

Another common technique is to use the pointer increment and decrement operators in a loop control expression. For example, the following loop copies the string pointed to by *p* to the location pointed at by *q* (*p* and *q* are both pointers to characters).

```
while (*q++ = *p++)
   ;
```

Note that the actual body of the loop is empty, the only action is in the control expression of the while statement. When the terminating null of the string is copied, the control expression becomes false, which terminates the loop.

Another aspect of C that makes it possible to construct rich expressions is *short-circuit expression evaluation*. Most C operators have a guaranteed expression evaluation order, which is left to right for most arithmetic and comparison operators. In addition, C guarantees that logical expressions will only be evaluated far enough to determine the outcome. As shown in Table III, the operator || means OR and the operator && means AND. Thus, the expression

```
p && q
```

means *p* AND *q*. According to the rules of Boolean logic, the result will be TRUE only if both *p* and *q* are TRUE. When the program is running, if the *p* part turns out to be FALSE, then the result of the whole expression is immediately known to be FALSE, and in this case the *q* part will not be evaluated.

Similarly, the expression

```
p || q
```

means *p* OR *q*. In this case, according to the rules of Boolean logic, the result will be TRUE if *either* the *p* or *q* part is TRUE. When the program is running, if the *p* part turns out to be TRUE, then the result is immediately known to be TRUE, and in this case the *q* part will not be evaluated, because C uses *short circuit expression evaluation*.

The following code fragment is an example of how short-circuit evaluation is often used. In it, a pointer is compared with the address of the end of an array to make sure that the pointer has not advanced past the end of the array. If the pointer is in bounds, only then is it used to access an item in the array.

```
if ((p < &x[20]) && (*p != 0))
```

Without the short-circuit expression guarantee made by the C language, this expression would have to be written as two expressions, so that the pointer would not be dereferenced when it was pointing outside the bounds of the array. (Dereferencing an out-of-bounds pointer can cause disastrous program failures.)

In all of these examples (and in the examples of the conditional operator and comma operator in Section I.C), C's operator-rich syntax has made it possible to express several things in just one or two lines, a benefit or shortcoming depending upon your viewpoint.

E. Explicit Pointer Usage

In all traditional computer architectures, at the lowest level the machine is constantly working with addresses, because it cannot store or retrieve information without knowing its address. However, most computer languages have tried to manage addresses automatically, to relieve programmers of this burden.

C has taken a best of both worlds approach. If you prefer, you can write programs that avoid working with addresses, which means avoiding the use of pointers as much as possible. Programmers can't completely avoid pointers when working in C, because many of the standard library routines expect pointer arguments.

Many programmers want to work with addresses, because of the control and efficiency that it yields. In addition, programmers writing software that directly accesses computer hardware often are forced to work with addresses. Fortunately, when these needs arise the C language is read with a complete set of features for working with addresses.

One of the areas where C pushes you toward using pointers is with subroutine parameters. In some languages, such as Pascal, you can specify whether subroutine parameters are passed by value or by reference. With Pascal's value parameters, changes inside the subroutine do not change the caller's copy of the parameter, while with reference parameters, there is only one copy of the parameter, and changes made inside a subroutine do alter the caller's value.

In C, all subroutine parameters are (in a strict technical sense) value parameters, but it is extremely common to pass pointers to the actual parameters to subroutines, thereby achieving the effect of reference parameters. For example, the following brief subroutine exchanges its two arguments (the first argument takes on the value of the

second and vice versa). If the two arguments were passed by *value*, this subroutine would have no effect, but instead the arguments are passed by *address* so that the swap can take place.

```
iswap(int *a, int *b) /* swap a
and b (integers) */
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

Inside the *iswap* procedure the * operator is used to access the values that *a* and *b* point toward. The *iswap* procedure is called with two pointers to int (integer), as in the following:

```
int i, j;

i = 50;
j = 20;
iswap(&i, &j);
```

When you call *iswap* you need to put the & (address-of) operator in front of the variables *i* and *j* so that you pass the addresses of the two variables to *iswap*. After *iswap* completes its work, the variable *i* will have the value 20 and the variable *j* will have the value 50.

F. Function Pointers

A function pointer is a pointer variable, but it holds the address of a function, not the address of a data item. The only things you can do with a function pointer are read its value, assign its value, or call the function that it points toward. You cannot increment or decrement the address stored in a function pointer or perform any other arithmetic operations.

Function pointers make it possible to write very general programs. For example, if you have a data structure that contains several different types of items, each item might contain a function pointer that could be used to print, order, or otherwise manipulate the information. Each type of data item would contain a function pointer to the appropriate function. Function pointers provide a very tedious way to build an object, a data structure that combines a set of values with a collection of appropriate behaviors.

Function pointers are declared using the syntax described in Section III.C. In that section, it was mentioned that the declaration

```
int *fn();
```

declares a function named *fn* that returns a pointer to an integer. If we want instead to declare a pointer to a function, we must use parentheses to indicate that what is being declared is primarily a pointer:

```
int (*fnptr)();
```

The parentheses around **fnptr* are necessary; they bind the* (the indirection operator) to the *fnptr*, overriding the normal precedence of the parentheses over the *. This declaration should be read aloud as "*fnptr* is a pointer to a function returning an integer."

G. Void

One of the innovations of ANSI C is the creation of the *void* data type, which is a data type that does not have any values or operations, and that cannot be used in an expression. One important use of *void* is to state that a function does not have a return value. Before ANSI, the best you could do was avoid using procedures in expressions when they did not return a value.

For example, the procedure *iswap()* in Section III.E, does not return a value. It works correctly if used as shown in that section, but it also can be used incorrectly.

```
x = 2 * iswap(&i, &j);
```

In this expression, the value stored in *x* is unpredictable, because *iswap* does not return a value. Most pre-ANSI C compilers will not object to this statement, because by default all procedures were presumed to return an integer. With ANSI C, you can specify that *iswap* has the type void, thereby assuring that erroneously using *iswap* in an arithmetic expression will be flagged as an error.

Another use of *void* is to create a generic pointer. On some machines, different types of pointers have different formats, and on most machines different data types have different alignment requirements, which impose restrictions on legitimate pointer values. Until the ANSI standardization, C lacked a generic pointer type that was guaranteed to meet all of the requirements on any machine, that is, a pointer that would be compatible with all of the other pointer types. This need is met by specifying that something is a pointer to *void*.

IV. KEY C++ FEATURES

From a historical perspective, C++ is a set of features that Bjarne Stroustrup added to C to facilitate *object-oriented programming*, plus an additional set of features added during the ANSI and ISO standardization process. C++

contains all of C, essentially unchanged, plus additional features for working with *objects*. But from the perspective of someone learning to program in C++, it's simply a language, containing a mix of traditional features and object-oriented features.

The major new C++ feature, the class, supports the idea of object-oriented programming. Objects either in the real world or in computer programs have two aspects: physical features, which are represented in computer programs by information storage, and operational features (actions), which are represented in software by procedures. In traditional (non-object-oriented) programming languages much attention has been paid to both data structures and to algorithms, but little attention was paid to combining the two, so that software can model entities found in the world. The general idea of object-oriented programming and the C++ class data type are intended to remedy this omission.

C++ also contains various other improvements, including references, which are related to pointers, and function overloading. These two additions are described in Sections A and B, below, and then the remaining sections detail the class data type, which is the focus of object-oriented programming using C++.

A. References

A reference is a data type that creates a new name for an existing variable. For example, if *x* is an existing integer variable, a reference to *x* can be created with the statement

```
int &rx = x;
```

The ampersand in the above declaration is the syntactical indicator that *rx* is a reference to an int variable, rather than a true int variable. After *rx* has been declared a reference to *x*, it can be used in any situation where *x* itself could be used. The simple-minded form shown above is rarely used, but references are extensively used as procedure parameters and as return values from functions.

One of the simplest practical uses of references is to write a slightly cleaner version of the *iswap* routine. (A pointer version of *iswap* was shown in Section III.E.)

```
// swap  integers  a and  b,
using reference parameters void
iswap(int& a, int& b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

Because this version of *iswap* uses reference-to-int parameters, it is used somewhat differently than the previous version.

```
int i, j;
i = 50;
j = 30;
iswap(i, j);
```

In addition to demonstrating references, this example also shows several aspects of C++, including the positioning of parameter declarations in the procedure header, the more thorough declaration of function return types (void in this example), and the new // syntax (on the first line of the example) to indicate a comment to the end of line.

B. Function Overloading

Programmers often need to create a family of procedures to perform the same task on various data types. For example, the *iswap* procedure shown in Sections III.E and IV.A works only for integers. It would also be useful to have swap procedures for doubles, characters, and so forth. Although each could be given a unique name (e.g., *iswap* for integers, *dswap* for doubles, etc.), unique names quickly become tedious and error prone. Instead, in C++ one can create a family of procedures that have the same name but that accept different parameter types. This lets the *programmer* use a single name, while it gives the *compiler* the job of choosing the correct procedure, based on the parameter types.

The following example shows how one could overload the *swap* procedure, creating versions for characters, integers, and doubles.

```
void swap(char& a, char& b)
{
 char temp = a; a = b; b = temp;
}
 void swap(int& a, int& b)
{
 int temp = a; a = b; b = temp;
}
 void swap(double& a, double& b)
{
 double temp = a; a = b; b =
  temp;
}
```

When the compiler sees the statement *swap(x,y)* it will choose a version of *swap* based on the types of the variables *x* and *y*. For example, if *x* and y are doubles, the compiler will choose the third function shown above.

C. Classes

Classes are the major new feature that C++ adds to C. They are the C++ language feature that facilitates object-oriented programming. The key idea of object-oriented programming is that the fundamental components of programs should be a objects—a data structure that combines data (information storage) and procedures. Software objects are analogous to the raw parts that are used in other creative disciplines. They make it possible to build more complex entities than would otherwise be possible, they may be modified and specialized as necessary, and they allow the programmer to build software whose structure parallels the structure of the problem domain.

It's important to understand the difference between a class and an object. A class is what programmers work with; it's a concept that is expressed in a program. An *object* (also called an *instance*) is the realization of a class when a program is executing. A programmer might write a program that defines a class to represent, say, complex numbers. When a program that uses that class is running, then each complex number in the program is an object. If, for example, the program is using 100 complex numbers, then there are 100 objects, each following the blueprint established in the complex number class that was written by a programmer.

Although their syntax is based on C structures, classes go far beyond the capabilities of ordinary C structures.

- Classes may have both *data* elements and *procedure* elements. The procedure elements, which are called *member functions*, can perform standard operations on the class, such as changing the values of its data elements, or reporting the values of the data elements
- Classes are the basis of *inheritance*, which allows programmers to create class hierarchies, and which reduces code duplication by enabling programmers to modify (specialize) existing classes. When you create a new class from an existing class, the original class is called the *base* class and the new one, which adds additional features or modifies the original behavior, is called the *derived* class.
- Class data elements may be *static*, which means a single copy of the item is shared among all instances of the class, or *nonstatic*, which means one copy of the item exists for each class instance.
- Classes may have *public*, *private*, and *protected* elements, which allow a programmer to control access to the individual parts of the class. Private elements can be accessed only by class member functions, protected elements can be accessed by class member functions and by member functions of derived classes, while public elements are generally available. The public part of a class is often called its *interface* because it defines the set of operations that are used to work with the class.
- Classes may have routines, called *constructors* and *destructors*, that are called automatically when a class instance is created or destroyed. Constructors are responsible for initializing a class, while destructors perform any necessary cleanup. Constructors are also used to convert items of another type to the class type. For example, a class that represents complex numbers might have a constructor that would convert a double into a complex.
- Classes may have *operator functions*, so that objects can be manipulated using algebraic notation.

The following class declaration describes a data type that represents complex numbers, which are numbers defined as having both real and imaginary parts. In algebraic notation the letter *i* indicates the imaginary part of a number, thus 50 + 100*i* represents a complex with real part of 50 and imaginary part of 100. A more realistic complex number class would have many more facilities; the simplifications imposed in this simple example are for clarity.

```
class Complex {
  protected:
       double realpart, imagpart;
  public:
       // constructors
       Complex(void);
       Complex(double r);
       Complex(double r, double i);
       Complex(Complex &c);
       // ADD OP - add a complex
       or a double to a complex
       void operator+=(Complex&
        rhs);
       void operator+=(double d);
       // extract the real and
       imaginary parts of a
       complex
       double getReal() { return
       realpart; }
       double getImag() { return
       imagpart; }
};
```

The class shown above contains two data elements, doubles named *realpart* and *imagpart*, and six operations. The class is divided into two parts, the *protected* part and the *public* part. The public elements of complex are universally accessible, while the elements in the protected part,

which in this case are the data elements, can only be accessed by derived classes.

The first four operations, the constructors, are used to create new complex numbers. The first creates a Complex object that is initialized to zero, the second creates a Complex from a single number (the real part), the third creates a Complex from a pair of numbers (the real and imaginary parts), and the fourth creates a new Complex object from an existing Complex object. In the class declaration shown above, the four construction operations are described but not defined. Here is the definition of the second constructor (the others are similar, hence not shown).

```
// Construct a complex from a
   single number.
Complex::Complex(double r):
  realpart(r), // init the
  realpart data member
  imagpart(0) // init the
  imagpart data member
{
  // The body of the
  constructor,
  // but there is nothing left
  to do.
}
```

Although this member function definition resembles an ordinary C procedure, there are some important differences that should be discussed. The definition starts with the notation *Complex*:: which is a way of reopening the context of the class declaration. The pair of colons, which is called the *scope resolution operator*, is placed after the name of a class to indicate a definition of something in the class. After the constructor header is a pair of initialization expressions. The first initialization expression is *realpart(r)*, which states that the class member named *realpart* should have its initial value taken from the parenthesized expression, `r`. The second initialization expression sets the *imagpart* member to zero. The body of the constructor, like the body of all C procedures, is delimited by a pair of curly braces, but in this particular example there is nothing to do in the constructor body because all the work has been done in the initialization section of the constructor.

The constructors mentioned in the Complex class declaration allow us to create Complex numbers using the following declarations:

```
Complex a;
Complex b(50, 100);
Complex c(b);
```

The Complex named *a* is initialized to zero, the Complex named *b* is initialized to 50 + l00*i*, and the Complex named *c* is initialized to the value of *b*. To understand how the above works, you must remember that the compiler will call the appropriate version of the constructor, based on the arguments. In the example, Complex variables *a*, *b*, and *c* will be constructed using the first, third, and fourth forms, respectively, of the constructors shown previously in the class declaration.

The two procedures named *operator*+ = in the Complex class declaration allow you to use the + = operator (the assign sum operator) to manipulate Complex numbers. (The bodies of these procedures are not shown here.) This capability, which is known as *operator overloading*, is primarily a notational convenience that allows manipulations of class objects to be expressed algebraically. C++ allows nearly all of its rich set of operators to be overloaded. The limitations of C++ operator overloading are that user-defined operator overloading must involve at least one user-defined type (class) and that the standard precedence and associativity may not be altered.

The first *operator*+ = procedure in the Complex class lets you add one complex to another, the second allows you to add a double (a real number) to a complex. For example, the following two expressions automatically invoke the first and second *operator*+ = functions. (Objects *a* and *b* are complex.)

```
a += b;
b += 5;
```

The last two procedures shown in the Complex class declaration are used to extract the real and imaginary parts of a complex number. For example, the following statement assigns the imaginary part of *a*, which is a Complex, to x, which is a double number.

```
x = a.getImag();
```

Note that ordinary C "member of" notation (the dot) is used to access the *getImag* member function.

D. Inheritance

Inheritance is a facility that allows a programmer to create a new class by specializing or extending an existing class. In C++ the original class is called the *base* class and the newly created class is called the *derived* class. Inheritance is also called *class derivation*. Derivation can do two types of things to a base class: new features can be added, and existing features can be redefined. Derivation does not allow for the removal of existing class features.

Derivation is usually used to create specialized classes from more general classes. It often expresses "kind of" relationships. For example, one might derive a *HighVoltageMotor* class from a *Motor* base class. Note that a High Voltage Motor is a *kind of* Motor; the converse is not true.

All motors have a power rating, a maximum speed of rotation, etc.These characteristics can be expressed in C++ as follows. First let us look at a partial declaration of the Motor class.

```
class Motor {
  double power;
  double speed;
  // other Motor characteristics
};
```

Note that only a few of the Motor class's members are sketched above. Next let's look at the class declaration of HighVoltageMotor. The notation in the class header states that a HighVoltageMotor is derived from the Motor base class. The body of HighVoltageMotor only lists things that are added to the base class; the existing elements, such as power, need not be restated:

```
class HighVoltageMotor : public
  Motor {
   double maximumVoltage;
   // other characteristics of
      High Voltage Motors
};
```

Each HighVoltageMotor object will contain all the characteristics (elements) of a Motor, such as power and speed, plus all the additional characteristics that pertain only to a High Voltage Motor, such as the maximumVoltage.

E. Polymorphism

Polymorphism is the ability of something to have various forms. In C++, polymorphism refers to the ability of a class member function to have different forms for related classes. As an example, consider a family of classes that represent shapes that can be displayed (on a screen) or printed. We would probably create a base class called Shape, and then derive specialized classes such as CircleShape, SquareShape, and OblongShape. All of these classes would contain a member function called Draw that would actually draw the given shape. Given an object of one of these shape types, you could call Draw to draw the given shape. This isn't difficult or mysterious when the compiler is able to tell, while analyzing and translating a source program, what type of data object is being used. For example, given an object whose type is CircleShape, calling Draw would obviously draw a circle.

However, the situation is more complex if all that's known during compilation is that the object is a member of the Shape family of objects, In this case, *polymorphism* comes into effect. Given an object whose type is only known to be in the Shape family, calling Draw will still call the correct version for the given object, even if it's not known in advance (i.e, during compilation) what type of shape object exists.

In C++ you engage polymorphism by declaring a member function to be *virtual*. When a class member function is declared to be virtual, then the compiler makes arrangements to call the correct version of the member function based on the type of the object that's present when the program runs. Without virtual, the compiler's job is a bit easier, as it uses the information in the program's source code to determine what member function to call.

Moving beyond the details of what the compiler does, the importance of polymorphism is that it lets a programmer create powerful families of classes, in which each family member behaves differently, yet uniformly. Polymorphism means that you can count on an object, known only as a member of a family, to behave as it should when you call one of its virtual member functions.

F. Exceptions

In practical situations, it's important to write robust software that operates reliably in the face of great difficulty. For example, a user might try to write a file to a floppy disk drive that doesn't contain a diskette, or a calculation might inadvertently try to divide some value by zero. Professional software must be able to handle these and myriad other problems as intelligently and reliably as possible.

The traditional approach to handling errors is to write procedures so that they return a specific value to indicate an error. For example, the procedure to write data to a disk file might return -1 if a failure occurs, and 0 if the operation is a success. There are several difficulties with this simple, traditional approach. The first is that programmers often write software that ignores error return codes. In an ideal world, programmers would not be so careless, but in the real world of deadlines and other pressures, error return values often are unused.

Another problem is that using error codes makes it hard to design and implement an error handling strategy. If every error has to be detected and handled on the spot, the code starts consist of large amounts of hard to debug code managing the problems. Soon, the bulk of code dealing with problems starts to overwhelm and burden the code that is focused on the actual task.

A better way to handle problems is to use exceptions. When a problem occurs, the subroutine that detects the problem then throws an exception, which transfers control

to whatever calling routine has arranged to handle that type of problem. For example, consider a function named *A* that manages the task of writing a document to disk. Naturally *A* will call numerous other routines to do all the chores necessary to accomplish the overall task. Before actually calling its helper routines, *A* will enter a *try/catch* block so that it can catch any input/output exceptions that occur at any point in the operation. Any of the called subroutines that detects a problem can simply throw an input/output exception, knowing that it will be handled elsewhere (in *A* in this example).

Another advantage of handling errors using exceptions is that you can use a hierarchy of exceptions in order to provide a more fine-grained approach to handling errors. For example, in addition to a generic input/output exception that indicates something failed during an I/O operation, you can also derive more specialized exceptions to indicate the precise failure, such as a file open error or a file write error. Catching the generic I/O exception will catch all of the I/O errors, which is probably what function *A* (from our example) would want to do, but the more focused subroutines that *A* calls might want to handle one of the more specialized exceptions locally.

G. Iostream Library

One of the more demanding tasks that must be handled by any software development environment is input and output. There are several difficulties in handling I/O, such as the need to input and output any conceivable type of data, and the fact that different computer systems provide very different low-level primitives for performing I/O. Because of these difficulties, many programming languages provide I/O facilities that are built into the language, tacitly admitting that the language itself isn't powerful enough or flexible enough to meet the needs of I/O operations.

One of the C language's innovations was its Standard I/O Library (stdio), which is a flexible group of subroutines, written in C, that let a programmer perform input and output operations. One problem with the C standard I/O library is that it isn't type safe. For example, you can easily (but erroneously) output or input a floating point number using the format intended for integers. For example, the following snippet of C code does just that, producing a nonsense output:

```
double d = 5.0;
printf("The variable d has the
value, %d\n", d);
```

(The problem with the above is the %d format code, which calls for an integer to output, but which is handed the variable *d*, a double.)

Another problem is that the standard C library can't easily be extended to handle user-defined class types. For example, the C stdio library knows how to work with all the built-in types, such as integers, strings, and floating point values, but it doesn't know how to work with, say, the Complex type that was defined earlier in this article.

In C++ there is an I/O facility called the iostream library. Iostream is type-safe and extensible, which means that it addresses the main deficiencies of the C stdio library. The key idea in iostreams is that input and output is accomplished using functions that are crafted to handle each different type of object. Iostream contains a built-in group of functions to input and output all the standard types, plus you can implement your own functions to handle your own class types.

For example, you could use iostreams to output the value of a double variable as follows:

```
double d = 5.0;
cout << "The variable d has the
value " << d << '\n';
```

As you can see in the above statement, iostreams hijack the << operator to create output expressions. Similarly, it uses the >> operator to form input expressions.

Besides the advantage of being type-safe, iostream is extensible. If you create a function to insert a Complex into an output stream, then you can use Complex objects with the iostream library as conveniently as you can use the built-in types:

```
Complex c(10, 20); // create a
complex initialized to 10+20i
cout << "The Complex c has the
value" << c << '\n';
```

H. Namespaces

One of the original goals of the C language was to be useful for writing large programs, such as operating systems. Large software projects often involve groups of tens to hundreds of programmers, often working for several companies, who then produce programs that contain hundreds of thousands of individual program statements. One problem with large programs is that they contain many thousands of names—object names, procedure names, variable names—that all have to be unique. Even students writing their first program can encounter this problem; students often create a subroutine called read, which is quickly flagged as an error because the standard C library already contains a subroutine called read. Unfortunately, there just aren't enough descriptive names to meet the demand.

The solution adopted in C++ is the namespace, which is a way to create separate spaces for each group of names. If you have a namespace named *A* then it can have a subroutine named *read* that doesn't conflict with a subroutine named *read* that is housed inside the *B* namespace. Using namespaces makes it much easier to create large software projects.

I. Templates

In general usage, a template is a pattern that you use to create things. For example, in a woodworking class you might use a template to guide you when you are sawing a particularly tricky curve in a project. Similarly, in C++ a template is a set of generic instructions for performing some task. For example, the task might be storing a collection of objects. The template would contain generic instructions for storing and retrieving, plus it would include a parameter to indicate what type of object should be managed. Then the C++ compiler would actually create the C++ code to implement the storing operation on the given type of object. This is often referred to as *generic programming* because you are writing software that applies to any type of object.

For example, you might want to create a vector of objects, in which the objects would be accessed by a numeric index. The first step would be to create a template that would incorporate all the details of creating a vector of some type of object. The template itself wouldn't be specific to a given type of object, rather it would be generic, simply a set of instructions that could apply to storing any object type in a vector. Then if the program contained object types *Pt*, *Rect*, and *Sphere*, then you could use the vector template to create a vector of *Pt*, a vector of *Rect*, and a vector of *Sphere*.

In addition to creating containers, templates are often used to implement generic algorithm. An earlier example in this article showed how to create an overloaded family of functions to swap the values held in a pair of variables. Since it would be tedious to create such functions for every type of object in a large project, you could instead use templates to create the function.

```
template<class T> void swap(T& a,
T& b)
{
   T temp = a; a = b; b = temp;
 }
```

The template declaration shown above doesn't actually create a swap function, but the compiler will follow the recipe given in the template if you actually try to use a swap function

```
Complex a(50);
Complex b(10, 20);
swap(a, b);
```

The code shown above first creates a pair of initialized complex variables. When the compiler encounters the call to *swap* with the two *Complex* arguments, it uses the template recipe for *swap* to create a version of *swap* appropriate for Complex arguments, and then it uses that newly minted *swap* function to actually perform the operation.

J. Standard Template Library

The addition of templates to C++ created an opportunity to create new, generic software to address common programming tasks. The most adept solution, which has now become a standard part of C++, is the Standard Template Library (STL), which was primarily the work of Alexander Stepanov at Hewlett Packard. The STL addresses two broad facilities, containers (also called collections) and algorithms that apply to collections, such as finding an element or counting elements. Because it is implemented using templates, all of the operations in the STL can be applied to any type, the built-in types and user-defined types. And because it is based on templates, the compiler generates versions of all the facilities that work specifically with whatever type is used within the program, ensuring that the implementation is as efficient as possible.

Historically, containers such as lists, queues, vectors, and matrices were created as necessary each time the need arose. If you wanted to store, for example, information about Complex numbers in a list, you'd specially create a new list type designed to store Complex objects. With templates, you can simply create a fully featured list, ready to store the Complex objects, with a single line of code:

```
list<Complex> complexNumbers;
```

Given the above declaration, individual Complex objects can be added to the list in many ways, such as adding them to the front of the list:

```
Complex a(50);
Complex b(10, 20);
complexNumbers.push_front(a);
complexNumbers.push_front(b);
```

Other elements of the STL include vectors, stacks, and queues. The emergence of the STL as a standard, key component of C++ has greatly expanded the breadth of tasks that are addressed by the libraries supplied with C++.

SEE ALSO THE FOLLOWING ARTICLES

BASIC PROGRAMMING LANGUAGE • DATABASES • DATA STRUCTURES • MICROCOMPUTER DESIGN • PROLOG PROGRAMMING LANGUAGE • SOFTWARE ENGINEERING

BIBLIOGRAPHY

Ellis, M. A., and Stroustrup, B. (1991). "The Annotated C++ Reference Manual," Addison-Wesley, Reading, Massachusetts.

Kernighan, B. W., and Ritchie, D. M. (1978). "The C Programming Language," Prentice-Hall, Englewood Cliffs, New Jersey.

Kernighan, B. W., and Ritchie, D. M. (1988). "The C Programming Language," 2nd ed., Prentice-Hall, Englewood Cliffs, New Jersey.

Ritchie, D. M., Johnson, S. C., Lesk, M. E., and Kernighan, B. W. (1978). The C Programming Language. Bell System Technical Journal 57(6), Part 2, 1991–2020.

Stepanov, A., and Lee, M. (1994). The Standard Template Library, HP Labs Technical Report HPL-94-34 (R.1).

Stroustrup, B. (1994). "The Design and Evolution of C++," Addison-Wesley, Reading, Mass achusetts.

Stroustrup, B. (2000). "The C++ Programming Language, (Special Edition)," Addison-Wesley, Reading, Massachusetts.

Computer Algorithms

Conor Ryan
University of Limerick

GLOSSARY

Algorithm Sequence of well-defined instructions the execution of which results in the solution of a specific problem. The instructions are unambiguous and each can be performed in a finite amount of time. Furthermore, the execution of all the instructions together takes only a finite amount of time.

Approximation algorithm Algorithm that is guaranteed to produce solutions whose value is within some prespecified amount of the value of an optimal solution.

Asymptotic analysis Analysis of the performance of an algorithm for large problem instances. Typically the time and space requirements are analyzed and provided as a function of parameters that reflect properties of the problem instance to be solved. Asymptotic notation (e.g., big "oh," theta, omega) is used.

Deterministic algorithm Algorithm in which the outcome of each step is well defined and determined by the values of the variables (if any) involved in the step. For example, the value of $x + y$ is determined by the values of x and y.

Heuristic Rule of thumb employed in an algorithm to improve its performance (time and space requirements or quality of solution produced). This rule may be very effective in certain instances and ineffective in others.

Lower bound Defined with respect to a problem. A lower bound on the resources (time or space) needed to solve a specified problem has the property that the problem cannot be solved by any algorithm that uses less resources than the lower bound.

Nondeterministic algorithm Algorithm that may contain some steps whose outcome is determined by selecting from a set of permissible outcomes. There are no rules determining how the selection is to be made. Rather, such an algorithm terminates in one of two modes: success and failure. It is required that, whenever possible, the selection of the outcomes of individual steps be done in such a way that the algorithm terminates successfully.

NP-Complete problem Decision problem (one for which the solution is "yes" or "no") that has the following property: The decision problem can be solved in polynomial deterministic time if all decision problems that can be solved in nondeterministic polynomial time are also solvable in deterministic polynomial time.

Performance Amount of resources (i.e., amount of computer time and memory) required by an algorithm. If the algorithm does not guarantee optimal solutions, the term "performance" is also used to include some measure of the quality of the solutions produced.

Probabilistically good algorithm Algorithm that does not guarantee optimal solutions but generally does provide them.

Simulated annealing Combinatorial optimization technique adapted from statistical mechanics. The technique attempts to find solutions that have value close to optimal. It does so by simulating the physical process of annealing a metal.

Stepwise refinement Program development methods in which the final computer program is arrived at in a sequence of steps. The first step begins close to the problem specification. Each step is a refinement of the preceding one and gets one closer to the final program. This technique simplifies both the programming task and the task of proving the final program correct.

Usually good algorithm Algorithm that generally provides optimal solutions using a small amount of computing resources. At other time, the resources required may be prohibitively large.

IN ORDER to get a computer to solve a problem, it is necessary to provide it with a sequence of instructions that if followed faithfully will result in the desired solution. This sequence of instructions is called a computer algorithm. When a computer algorithm is specified in a language the computer understands (i.e., a programming language), it is called a program. The topic of computer algorithms deals with methods of developing algorithms as well as methods of analyzing algorithms to determine the amount of computer resources (time and memory) required by them to solve a problem and methods of deriving lower bounds on the resources required by any algorithm to solve a specific problem. Finally, for certain problems that are difficult to solve (e.g., when the computer resources required are impractically large), heuristic methods are used.

I. ALGORITHMS AND PROGRAMS

An *algorithm* can take many forms of detail. Often the level of detail required depends on the target of the algorithm. For example, if one were to describe an algorithm on how to make a cup of tea to a human, one could use a relatively coarse (high) level of detail. This is because it is reasonable to assume that the human in question can fill in any gaps in the instructions, and also will be able to carry out certain tasks without further instructions, e.g., if the human is required to get a cup from a cupboard, it would be fair to assume that he/she knows how to do this without elaboration on the task.

On the other hand, a *program* is generally a *computer program*, and consists of a set of instructions at a very fine level of detail. A fine level of detail is required because computer programs are always written in a particular *language*, e.g., Basic, C++, Pascal, etc. Furthermore, every step in a task must be specified, because no background knowledge can be assumed. An often used distinction is that an algorithm specifies *what* a process is doing, while a program specifies *how* the process should be done. The truth is probably somewhere between these two extremes—while an algorithm should be a clear statement of what a process is doing, it is often useful to have some level of specification of functionality in an algorithm.

It is not very natural for humans to describe tasks with the kind of level of detail usually demanded by a programming language. It is often more natural to think in a *top-down* manner, that is, describe the problem in a high level manner, and then rewrite it in more detail, or even in a specific computer language. This can often help the person concerned to get a problem clear in his/her own mind, before committing it to computer. Much of this chapter is concerned with the process of *refinement*. Refinement of algorithms is (usually) an iterative process, where one begins with a very high level—that is, the *what*—and by repeatedly modifying the algorithm by adding more detail (the *how*) brings the algorithm closer and closer to being code, until the final coding of the algorithm becomes a very clear task. Ideally, when one is writing a program, one should not have to figure out any logic problems; all of these should be taken care of in the algorithm.

Algorithms are not just used as an aid for programmers. They are also a very convenient way to describe what a task does, to help people conceptualize it at a high level, without having to go through masses of computer code line by line.

Consider the following problem, which we will state first in English:

Mary intends to open a bank account with an initial deposit of $100. She intends to deposit an additional $100 into this account on the first day of each of the next 19 months for a total of 20 deposits (including the initial deposit). The account pays interest at a rate of 5% per annum compounded monthly. Her initial deposit is also on the first day of the month. Mary would like to know what the balance in her account will be at the end of the 20 months in which she will be making a deposit.

In order to solve this problem, we need to know how much interest is earned each month. Since the annual interest rate 5%, the monthly interest rate is 5/12%. Consequently, the balance of the end of a month is

$$\begin{aligned}&(\texttt{initial balance} + \texttt{interest})\\ &\quad = (\texttt{initial balance}) * (1 + 5/1200)\\ &\quad = 241/240\ (\texttt{initial balance})\end{aligned}$$

Having performed this analysis, we can proceed to compute the balance at the end of each month using the following steps:

1. Let balance denote the current balance. The starting balance is \$100, so set balance = 100.
2. The balance at the end of the month is 241/240 * balance. Update balance.
3. If 20 months have not elapsed, then add 100 to balance to reflect the deposit for the next month. Go to step 2. Otherwise, we are done.

This, then, is an algorithm for calculating the monthly balances. To refine the algorithm further, we must consider what kind of machine we wish to implement our algorithm on. Suppose that we have to compute the monthly balances using a computing device that cannot store the computational steps and associated data. A nonprogrammable calculator is one such device. The above steps a will translate into the following process:

1. Turn the calculator on.
2. Enter the initial balance as the number 100.
3. Multiply by 241 and then divide by 240.
4. Note the result down as a monthly balance.
5. If the number of monthly balances noted down is 20, then stop.
6. Otherwise, add 100 to the previous result.
7. Go to step 3.

If we tried this process on an electronic calculator, we would notice that the total time spent is not determined by the speed of the calculator. Rather, it is determined by how fast we can enter the required numbers and operators (add, multiply, etc.) and how fast we can copy the monthly balances. Even if the calculator could perform a billion computations per second, we would not be able to solve the above problem any faster. When a stored-program computing device is used, the above instructions need be entered into the computer only once. The computer can then sequence through these instructions at its own speed. Since the instructions are entered only once (rather than 20 times), we get almost a 20 fold speed up in the computation. If the balance for 1000 months is required, the speedup is by a factor of almost 1000. We have achieved this speedup without making our computing device any faster. We have simply cut down on the input work required by the slow human!

A different approach would have been to write program for the algorithm. The seven-step computational process stated above translates into the Basic program shown in Program 1.

PROGRAM 1

```
10 balance = 100
20 month = 1
30 balance = 241*balance/240
40 print month, "$";balance
50 if month = 20 then stop 60 month
     = month + 1
70 balance = balance + 100
80 go to 30
```

PROGRAM 2: Pascal Program for Mary's Problem

```
line program account(input,output)
1   {computer the account balance at
      the end of each month}
2   const   InitialBalance = 100;
3           MonthlyDeposit = 100;
4           TotalMonths = 20;
5           AnnualInterestRate = 5;
6   var balance, interest, MonthlyRate:
      real
7           month:integer;
8   begin
9           MonthlyRate := AnnualInterest-
            Rate/1200;
10            balance := InitialBalance;
11            writeln('Month Balance');
12            for month := 1 to TotalMonths
                do
13              begin
14                interest := balance *
                    MonthlyRage;
15                balance := balance
                    + interest;
16                writeln(month:10, '  ',
                     balance:10:2);
17                balance := balance +
                     MonthlyDeposit;
18                end; of for
19                writeln;
18  end; of account
```

In Pascal, this takes the form shown in Program 2. Apart from the fact that these two programs have been written in different languages, they represent different

programming styles. The Pascal program has been written in such a way as to permit one to make changes with ease. The number of months, interest rate, initial balance, and monthly additions are more easily changed into Pascal program.

Each of the three approaches is valid, and the one that should eventually be used will depend on the user. If the task only needs to be carried out occasionally, then a calculator would probably suffice, but if it is to be executed hundreds or thousands of times a day, then clearly one of the computer programs would be more suitable.

II. ALGORITHM DESIGN

There are several design techniques available to the designer of a computer algorithm. Some of the most successful techniques are the following:

- Divide and conquer
- Greedy method
- Dynamic programming
- Branch and bound
- Backtracking

While we do not have the space here to elaborate each of these, we shall develop two algorithms using the divide and conquer technique. The essential idea in divide and conquer is to decompose a large problem instance into several smaller instances, solve the smaller instances, and combine the results (if necessary) to obtain the solution of the original problem instance. The problem we shall investigate is that of sorting a sequence $x[1], x[2], \ldots, s[n]$ of $n, n > 0$ numbers; where n is the size of the instance. We wish to rearrange these numbers so that they are in nondecreasing order (i.e., $x[1] < x[2], \ldots, x[n]$).

For example, if $n = 5$, and $(x[1], \ldots, x[5]) = (10,18, 8,12,19)$, then after the sort, the numbers are in order (8,9,10,12,18). Even before we attempt an algorithm to solve this problem, we can write down and English version of the solution, as in Program 3. The correctness of this version of the algorithm is immediate.

PROGRAM 3: First Version of Sort Algorithm

```
Procedure sort;
Sort x[I], 1 < I < n into nondecreasing
  order;
End;{of sort}
```

Using the divide and conquer methodology, we first decompose the sort instance into several smaller instances. At this point, we must determine the size and number of these smaller instances. Some possibilities are the following:

(a) One of size $n - 1$ and another of size 1
(b) Two of approximately equal size
(c) K of size approximately n/k each, for some integer $k, k > 2$

We shall pursue the first two possibilities. In each of these, we have two smaller instances created. Using the first possibility, we can decompose the instance (10,18,8,12,9) into any of the following pairs of instances:

(a) (10,18,8,12) (9)
(b) (10) (18,8,12,9)
(c) (10,18,8,9) (12)
(d) (10,18,12,9) (8)

and so on. Suppose we choose the first option. Having decomposed the initial instance into two, we must sort the two instances and then combine the two-sorted sequences into one. When (10,18,8,12) is sorted, the result is (8,10,12,18). Since the second sequence is of size 1, it is already in sorted order. To combine the two sequences, the number 9 must be inserted into the first sequence to get the desired five-number sorted sequence. The preceding discussion raises two questions. How is the four-number sequence sorted? How is the one-number sequence inserted into the sorted four-number sequence? The answer to the first is that this, too, can be sorted using the divide and conquer approach. That is, we decompose it into two sequences: one of size 3 and the other of size 1. We then sort the sequence of size 3 and then insert the 1 element sequence. To sort the three-element sequence, we decompose it into two sequences of size 2 and size 1, respectively. To sort the sequence of size 2, we decompose it into two of size 1 each. At this point, we need merely insert one into the other. Before attempting to answer the second question, we refine Program 3, incorporating the above discussion. The result is Program 4.

PROGRAM 4: Refinement of Program 3

```
line procedure sort(n)
1  {sort n numbers into nondecreasing
     order}
2  if n>1 then begin
3        sort(n-1); sort the first
           sequence
4        insert(n-1,x[n]);
5        end; {of if}
6        writeln;
7  end;{of sort}
```

Program 4 is a recursive statement of the sort algorithm being developed. In a recursive statement of an algorithm, the solution for an instance of size n is defined in terms of solutions for instances of smaller size. In Program 4, the sorting of n items, for $n > 1$ items, is defined in terms of the sorting of $n - 1$ items. This just means that to sort n items using procedure sort, we must first use this procedure to sort $n - 1$ items. This in turn means that the procedure must first be used to sort $n - 2$ items, and so on. This use of recursive generally poses no problems, as most contemporary programming languages support recursive programs. To refine Program 4, we must determine how an insert is performed. Let us consider an example. Consider the insertion of 9 into (8,10,12,18). We begin by moving 9 from position 5 of the sequence and then comparing 9 and 18, since 18 is larger, it must be brought to the right of 9. So 18 is moved to position 5. The resulting sequence is as follows ("—" denoted an empty position in the sequence):

8 10 12 — 18

Next, 9 is compared with 12, and 12 is moved to position 4. This results in the following sequence:

8 10 — 12 18

Then 9 is compared with 10 and 10 is moved to position 3. At this time we have the sequence:

8 — 10 12 18

Finally, 9 is compared with 8. Since 9 is not smaller than 8, it is inserted into position 2. This results in the sequence (8,9,10,12,18). With this discussion, we can refine Program 4 to get Program 5. Program 5 is then easily refined to get the Pascal-like code of Program 6. This code uses position 0 of the sequence to handle insertions into position 1. The recursion in this procedure can be eliminated to get Program 7. The algorithm we have just developed for sorting is called insertion sort. This algorithm was obtained using the stepwise refinement process beginning with Program 3. As a result of using this process, we have confidence in the correctness of the resulting algorithm. Formal correctness proofs can be obtained using mathematical induction or other program verification methods.

PROGRAM 5: Refinement of Program 4

```
line procedure sort(n)
1 {sort n numbers into nondecreasing
   order}
2  if n > 1 then begin
3       sort(n - 1); sort the first
          sequence
4       assign t the value x[n];
5       compare t with the xs
          beginning at x[n - 1];
6       move the xs up until the
          correct place for t is
          found
7       insert t into this place
8       end; {of if}
9 end; {of sort}
```

PROGRAM 6: Refinement of Program 5

```
line procedure sort(n)
 1 {sort n numbers into nondecreasing
    order}
 2  if n > 1 then begin
 3       sort(n - 1); sort the first
          sequence
 4       assign t and x[0]the value
          x[n];
 5       assign i the value n - 1;
 6       while t < x[i] do {find correct
          place for t};
 7       begin
 8           move x[i] to x[i + 1];
 9           reduce i by 1;
10       end;{of while}
11       put t into x[i + 1]
12       end;{of if}
13 end;{of sort}
```

PROGRAM 7: Refinement of Program 6

```
line procedure sort(n)
 1 {sort n numbers into nondecreasing
    order}
 2 for j := 2 to n do
 3 begin {insert x[j] into x[1:j - 1]}
 4   assign t and x[0] the value x[j];
 5     assign i the value j - 1;
 6     while t < x[i] do {find correct place
         for t}
 7      begin
 8        move x[i] to x[i + 1];
 9        reduce i by 1;
10      end;{of while}
11      put t into x[i + 1]
12      end;{of if}
13 end;{of sort}
```

Program 7 is quite close to being a Pascal program. One last refinement gives us a correct Pascal procedure to sort. This is given in Program 8. This procedure assumes that the numbers to be sorted are of type integer. In case numbers of a different type are to be sorted, the type of declaration of t should be changed. Another improvement of Program 8 can be made. This involves taking the statement **x**[**0**] := **t** out of the for loop and initializing **x**[**0**] to a very small number before the loop.

PROGRAM 8: Refinement of Program 7

```
line procedure sort (n)
 1 {sort n numbers into nondecreasing
    order}
 2 for j := 2 to n do
 3 begin {insert x[j] into x[1:j - 1]}
 4     assign t and x[0] the value
         x[j];
 5     assign i the value j - 1;
 6     while t < x[i] do {find correct
         place for t}
 7     begin
 8        move x[i] to x[i + 1];
 9        reduce i by 1;
10     end;{of while}
11     put t into x[i + 1]
12 end;{of sort}
```

Let us consider the route our development process would have taken if we had decided to decompose sort instances into two smaller instances of roughly equal size. Let us further suppose that the left half of the sequence is one of the instances created and the right half is the other. For our example we get the instances (10,18) and (8,12,9). These are sorted independently to get the sequence (10,18) and (8.9,12). Next, the two sorted sequence are combined to get the sequence (8,9,10,12,18). This combination process is called merging. The resulting sort algorithm is called merge sort.

PROGRAM 9: Final version of Program 8

```
line procedure sort(n)
 1 {sort n numbers into nondecreasing
    order}
 2 var t, i, j : integer;
 2 begin
 2 for j := 2 to n do
 3 begin {insert x[j] into x[1 : j - 1]}
 4       t := x[j];x[0]:=t;i:=j - 1;
 6       while t < x[i] do {find correct
           place for t}
 7       begin
 8           x[i + 1] := x[i];
 9           i := i - 1;
10       end;{of while}
11       x[i + 1] := t
12 end;{of for}
13 end;{of sort}
```

PROGRAM 9: Merge Sort

```
line procedure MergeSort(X, n)
1 {sort n numbers in X}
2 if n > 1 then
3 begin
4 Divide X into two sequences A and B
    such that A contains ⌊n/2⌋
    numbers, and B the rest
5       MergeSort (A,⌊ n/2⌋)
6       MergeSort (A,n - ⌊ n/2⌋)
7       merge (A, B);
8 end;{of if}
9 end;{of MergeSort }
```

Program 9 is the refinement of Program 3 that results for merge sort. We shall not refine this further here. The reader will find complete programs for this algorithm in several of the references sited later. In Program 9, the notation $\lfloor x \rfloor$ is used. This is called the floor of x and denotes the largest integer less than or equal to x. For example, $\lfloor 2.5 \rfloor = 2$, $\lfloor -6.3 \rfloor = -7$, $\lfloor 5/3 \rfloor = 1$, and $\lfloor n/2 \rfloor$ denotes the largest integer less than or equal to $n/2$.

III. PERFORMANCE ANALYSIS AND MEASUREMENT

In the preceding section, we developed two algorithms for sorting. Which of these should we use? The answer to this depends on the relative performance of the two algorithms. The performance of an algorithm is measured in terms of the space and time needed by the algorithm to complete its task. Let us concentrate on time here. In order to answer the question "How much time does insertion sort take?" we must ask ourselves the following:

1. What is the instance size? The sort time clearly depends on how many numbers are being sorted.
2. What is the initial order? An examination of Program 7 means that it takes less time to sort n numbers that are already in nondecreasing order than when they are not.
3. What computer is the program going to be run on? The time is less on a fast computer and more on a slow one.
4. What programming language and compiler will be used? These influence the quality of the computer code generated for the algorithm.

To resolve the first two question, we ask for the worst case or average time as a function on instance size. The worst case size for any instance size n is defined as

$$T_{\mathrm{W}}(n) = \max\{t(I) \mid I \text{ is an instance of size } n\}.$$

Here $t(I)$ denotes the time required for instance I. The average time is defined as:

$$(T_{\mathrm{A}})m = \frac{1}{N} \sum t(I).$$

where the sum is taken over all instances of size n and N is the number of such instances. In the sorting problem, we can restrict ourselves to the $n!$ different permutations of any n distinct numbers. So $N = n!$.

A. Analysis

We can avoid answering the last two questions by acquiring a rough count of the number of steps executed in the worst or average case rather than an exact time. When this is done, a paper and pencil analysis of the algorithm is performed. This is called performance analysis. Let us carry out a performance analysis on our two sorting algorithms. Assume that we wish to determine the worst-case step count for each. Before we can start we must decide the parameters with respect to which we shall perform the analysis. In our case, we shall obtain times as a function of the number n of numbers to be sorted. First consider insertion sort. Let $t(n)$ denote the worst-case step count of Program 6. If $n < 1$, then only one step is executed (verify that $n < 1$). When $n > 1$, the recursive call to sort $(n-1)$ requires $t(n-1)$ steps in the worst case and the remaining steps count for some linear function of n step executions in the worst case. The worst case is seen to arise when $x[N]$ is to be inserted into position 1. As a result of this analysis, we obtain the following recurrence for insertion sort:

$$t(n) = \begin{cases} a, & n <= 1, \\ t(n-1) + bn + c, & n > 1, \end{cases}$$

where a, b, and c are constants. This recurrence can be solved by standard methods for the solution of recurrences. For merge sort, we see from Program 9 that when $n < 1$, only a constant number of steps are executed. When $N > 1$, two calls to merge sort and one to merge are made. While we have not said much about the division into A and B is to be performed, this can be done in a constant amount of time. The recurrence for Merge Sort is now seen to be

$$t(n) = \begin{cases} a, & n <= 1, \\ t(\lfloor n/2 \rfloor) + t(n - \lfloor n/2 \rfloor) + m(n) + b, & n > 1, \end{cases}$$

where a and b are constants and $m(n)$ denotes the worst-case number of steps needed to merge n numbers. Solving this recurrence is complicated by the presence of the floor function. A solution for the case n is a power of 2 is easily obtained using standard methods. In this case, the floor function can be dropped to get the recurrence:

$$t(n) = \begin{cases} a, & n <= 1, \\ 2t(n/2) + m(n) + b, & n > 1. \end{cases}$$

The notion of a step is still quite imprecise. It denotes any amount of computing that is not a function of the parameters (in our case n). Consequently, a good approximate solution to the recurrences is as meaningful as an exact solution. Since approximate solutions are often easier to obtain than exact ones, we develop a notation for approximate solutions.

Definition [Big "oh"]. $f(n) = O(g(n))$ (read as "f of n is big oh of g of n") iff there exist positive constants c and n_0 such that $f(n) \le cg(n)$ for all n, $n \ge n_0$. Intuitively, $O(g(n))$ represents all functions $f(n)$ whose rate of growth is no more than that of $g(n)$.

Thus, the statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n > n$. It does not say anything about how good this bound is. Notice that $n = O(n^2)$, $n = O(n^{2.5})$, $n = 0\,(n^3) n = O(2^n)$, and so on. In order for the statement $f(n) = 0(g(n))$ to be informative, $g(n)$ should be a small function of n as one can come up with for which $f(n) = O(g(n))$. So while we often say $3n + 3 = O(n2)$, even though the latter statement is correct. From the definition of O, it should be clear that $f(n) = O(g(n))$ is not the same as $0(g(n)) = f(n)$. In fact, it is meaningless to say that $O(g(n)) = f(n)$. The use of the symbol $=$ is unfortunate because it commonly denoted the "equals" relation. Some of the confusion that results from the use of this symbol (which is standard terminology) can be avoided by reading the symbol $=$ as "is" and not as "equals." The recurrence for insertion sort can be solved to obtain

$$t(n) = O(n^2).$$

To solve the recurrence for merge sort, we must use the fact $m(n) = O(n)$. Using this, we obtain

$$t(n) = O(n \log n).$$

It can be shown that the average number of steps executed by insertion sort and merge sort are, respectively, $0(n^2)$ and $0(n \log n)$. Analyses such as those performed above for the worst-case and the average times are called asymptotic analyses. $0(n2)$ and $0(n \log n)$ are, respectively, the worst-case asymptotic time complexities of insertion and merge sort. Both represent the behavior of the algorithms when n is suitably large. From this analysis we learn that the growth rate of the computing time for merge sort is less than that for insertion sort. So even if insertion sort is faster for small n, when n becomes suitably large, merge sort will be faster. While most asymptotic analysis is carried out using the big "oh" notation, analysts have available to them three other notations. These are defined below.

Definition [Omega, Theta, and Little "oh"]. $f(n) = \Omega(g(n))$ read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \ge cg(n)$ for all $n, n \ge n_0$. $f(n)$ is $\Theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$.

$f(n) = o(g(n))$ (read as "f of n is little oh of g of n") iff $\lim_{n\to\infty} f(n)/g(n) = 1$.

Example. $3n + 2 = \Omega(n); 3n + 2 = \Theta(n); 3n + 2 - o(3n); 3n^3 = \Omega(n^2); 2n^2 + 4n = \Theta(n^2);$ and $4n^3 + 3n^2 = o(4n^3)$.

The omega notation is used to provide a lower bound, while the theta notation is used when the obtained bound is both a lower and an upper bound. The little "oh" notation is a very precise notation that does not find much use in the asymptotic analysis of algorithms. With these additional notations available, the solution to the recurrence for insertion and merge sort are, respectively, $\Theta(n^2)$ and $\Theta(n \log n)$. The definitions of O, Ω, Θ, and o are easily extended to include functions of more than one variable. For example, $f(n, m) = O(g(n, m))$ if there exist positive constants c, n_0 and m_0 such that $f(n, m) < cg(n, m)$ for all $n > n_0$ and all $m > m_0$. As in the case of the big "oh" notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. The $g(n)$ is only a lower bound on $f(n)$. The θ notation is more precise that both the big "oh" and omega notations. The following theorem obtains a very useful result about the order of $f(n)$ when $f(n)$ is a polynomial in n.

Theorem 1. Let $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_0$, $a_m \neq 0$.

(a) $f(n) = O(n^m)$
(b) $f(n) = \Omega(n^m)$
(c) $f(n) = \Theta(n^m)$
(d) $f(n) = o(a_m n^m)$

Asymptotic analysis can also be used for space complexity. While asymptotic analysis does not tell us how many seconds an algorithm will run for or how many words of memory it will require, it does characterize the growth rate of the complexity. If an $\Theta(n^2)$ procedure takes 2 sec when $n = 10$, then we expect it to take $\tilde{8}$ sec when $n = 20$ (i.e., each doubling of n will increase the time by a factor of 4). We have seen that the time complexity of an algorithm is generally some function of the instance characteristics. As noted above, this function is very useful in determining how the time requirements vary as the instance characteristics change. The complexity function can also be used to compare two algorithms A and B that perform the same task. Assume that algorithm A has complexity $\Theta(n)$ and algorithm B is t of complexity $\Theta(n^2)$. We can assert that algorithm A is faster than algorithm B for "sufficiently large" n. To see the validity of this assertion, observe that the actual computing time of A is bounded from above by $c * n$ for some constant c and for all $n, n \geq n_2$, while that of B is bounded from below by $d * n^2$ for some constant d and all $n, n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, algorithm A is faster than algorithm B whenever $n \geq \max\{n_1, n_2, c/d\}$.

TABLE I Values of Selected Asymptotic Functions

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

We should always be cautiously aware of the presence of the phrase "sufficiently large" in the assertion of the preceding discussion. When deciding which of the two algorithms to use, we must know whether the n we are dealing with is in fact "sufficiently large." If algorithm A actually runs in $10^6 n$ msec while algorithm B runs in n^2 msec and if we always have $n \leq 10^6$, then algorithm B is the one to use.

Table I and Fig. 1 indicate how various asymptotic functions grow with n. As is evident, the function 2^n grows very rapidly with n. In fact, if a program needs 2^n steps for execution, then when $n = 40$ the number of steps needed is $\sim 1.1 \times 10^{12}$. On a computer performing 1 billion steps per second, this would require $\sim$18.3 min (Table II). If $n = 50$, the same program would run for $\sim$13 days on this computer. When $n = 60$, $\sim$310.56 years will be required to execute the program, and when $n = 100 \sim 4x1013$ years will be needed. So we may conclude that the utility of programs with exponential complexity is limited to small n (typically $n \sim 40$). Programs that have a complexity that is a polynomial of high degree are also of limited utility.

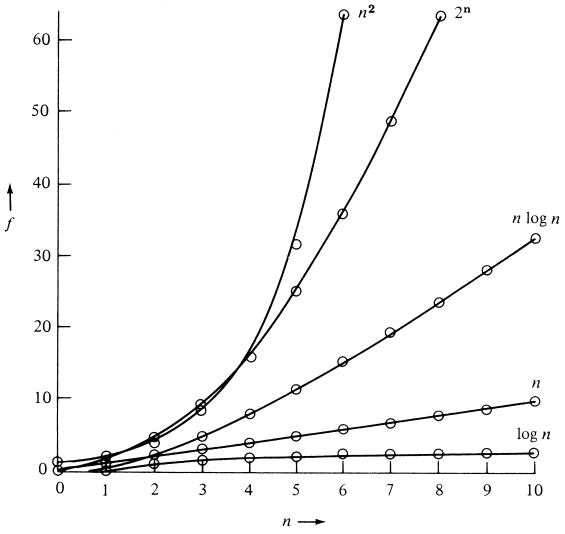


FIGURE 1 Plot of selected asymptotic functions.

TABLE II Times on a 1 Billion Instruction per Second Computer[a]

	Time for $f(n)$ instructions on a 10^9 instruction/sec computer						
n	$f(n)=n$	$f(n)=n\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=n^n$
10	0.01 μsec	0.03 μsec	0.1 μsec	1 μsec	10 μsec	10 sec	1μsec
20	0.02 μsec	0.09 μsec	0.4 μsec	8 μsec	160 μsec	2.84 hr	1 msec
30	0.03 μsec	0.15 μsec	0.9 μsec	27 μsec	810 μsec	6.83 day	1 sec
40	0.04 μsec	0.21 μsec	1.6 μsec	64 μsec	2.56 msec	121.36 day	18.3 min
50	0.05 μsec	0.28 μsec	2.5 μsec	125 μsec	6.25 msec	3.1 yr	13 day
100	0.10 μsec	0.66 μsec	10 μsec	1 msec	100 msec	3171 yr	4×10^3 yr
1,000	1.00 μsec	9.96 μsec	1 msec	1 sec	16.67 min	3.17×10^3 yr	32×10^{283} yr
10,000	10.00 μsec	130.3 μsec	100 msec	16.67 min	115.7 day	3.17×10^{23} yr	—
100,000	100.00 μsec	1.66 msec	10 sec	11.57 day	3171 yr	3.17×10^{33} yr	—
1,000,000	1.00 msec	19.92 msec	16.67 min	31.71 yr	3.17×10^7 yr	3.17×10^{43} yr	—

[a] 1 μsec $=10^{-6}$ sec; 1 msec $=10^{-3}$ sec.

For example, if a program needs n^{10} steps, then using our 1 billion steps per second computer (Table II) we will need 10 sec when $n=10$; 3171 years when $n=100$; and $3.17*10^{13}$ years when $n=1000$. If the program's complexity were n^3 steps instead, we would need 1 sec when $n=1000$; 110.67 min when $n=10{,}000$; and 11.57 days when $n=100{,}000$. From a practical standpoint, it is evident that for reasonably large n (say $n>100$) only programs of small complexity (such as n, $n\log n$, n^2, n^3, etc.) are feasible. Furthermore, this would be the case even if one could build a computer capable of executing 10^12 instructions per second. In this case, the computing times of Table II would decrease by a factor of 1000. Now, when $n=100$, it would take 3.17 years to execute n^{10} instructions, and 4×10^{10} years to execute 2^n instructions.

B. Measurement

In a performance measurement, actual times are obtained. To do this, we must refine our algorithms into computer programs written in a specific programming language and compile these on a specific computer using a specific compiler. When this is done, the two programs can be given worst-case data (if worst-case times are desired) or average-case data (if average times are desired) and the actual time taken to sort measured for different instance sizes. The generation of worst-case and average test data is itself quite a challenge. From the analysis of Program 7, we know that the worst case for insertion sort arises when the number inserted on each iteration of the **for** loop gets into position 1. The initial sequence $(n, n-1, \ldots, 2, 1)$ causes this to happen. This is the worst-case data for Program 7. How about average-case data? This is somewhat harder to arrive at. For the case of merge sort, even the worst-case data are difficult to devise. When it becomes difficult to generate the worst-case or average data, one resorts to simulations. Suppose we wish to measure the average performance of our two sort algorithms using the programming language Pascal and the TURBO Pascal (TURBO is a trademark of Borland International) compiler on an IBM-PC. We must first design the experiment. This design process involves determining the different values of n for which the times are to be measured. In addition, we must generate representative data for each n. Since there are $n!$ different permutations of n distinct numbers, it is impractical to determine the average run time for any n (other than small n's, say $n<9$) by measuring the time for all $n!$ permutations and then computing the average. Hence, we must use a reasonable number of permutations and average over these. The measured average sort times obtained from such experiments are shown in Table III. As predicted by our earlier analysis, merge sort is faster than insertion sort. In fact, on the average, merge sort will sort 1000 numbers in less time than insertion sort will take for 300! Once we have these measured times, we can fit a curve (a quadratic in the case of insertion sort and an $n\log n$ in the case of merge sort) through them and then use the equation of the curve to predict the average times for values of n for which the times have not been measured. The quadratic growth rate of the insertion sort time and the $n\log n$ growth rate of the merge sort times can be seen clearly by plotting these times as in Fig. 2. By performing additional experiments, we can determine the effects of the compiler and computer used on the relative performance of the two sort algorithms. We shall provide some comparative times using the VAX 11780 as the second computer. This popular computer is considerably faster than the IBM-PC and costs ~100 times as much. Our first experiment obtains the average run time of Program 8 (the Pascal program for insertion sort). The times for the V AX llnso were obtained using the combined translator and interpretive executer, pix. These are shown in

TABLE III Average Times for Merge and Insertion Sort[a]

n	Merge	Insert
0	0.027	0.032
10	1.524	0.775
20	3.700	2.253
30	5.587	4.430
40	7.800	7.275
50	9.892	10.892
60	11.947	15.013
70	15.893	20.000
80	18.217	25.450
90	20.417	31.767
100	22.950	38.325
200	48.475	148.300
300	81.600	319.657
400	109.829	567.629
500	138.033	874.600
600	171.167	—
700	199.240	—
800	230.480	—
900	260.100	—
1000	289.450	—

[a] Times are in hundredths of a second.

Table IV. As can be seen, the IBM-PC outperformed the V AX even though the V AX is many times faster. This is because of the interpreter pix. This comparison is perhaps unfair in that in one case a compiler was used and in the other an interpreter. However, the experiment does point out the potentially devastating effects of using a compiler that generates poor code or of using an interpreter. In our second experiment, we used the Pascal compiler, pc, on the

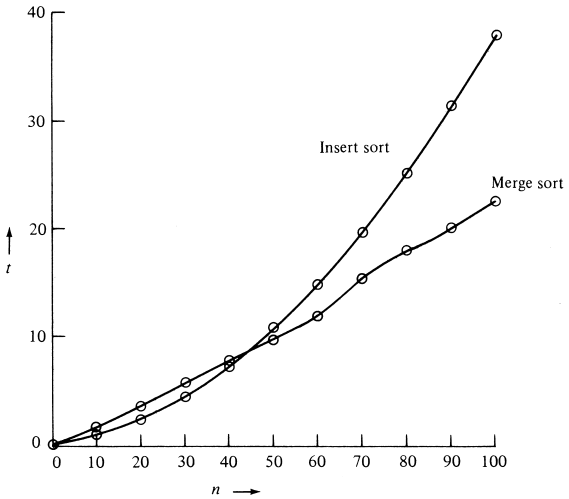


FIGURE 2 Plot of times of Table III.

TABLE IV Average Times for Insertion Sort[a]

n	IBM-PC turbo	VAX pix
50	10.9	22.1
100	38.3	90.47
200	148.3	353.9
300	319.7	805.6
400	567.6	1404.5

[a] Times are in hundredths of a second.

V AX. This time our insertion sort program ran faster on the V AX. However, as expected, when n becomes suitably large, insertion sort on the V AX is slower than merge sort on an IBM-PC. Sample times are given in Table V. This experiment points out the importance of designing good algorithms. No increase in computer speed can make up for a poor algorithm. An asymptotically faster algorithm will outperform a slower one (when the problem size is suitably large); no matter how fast a computer the slower algorithm is run on and no matter how slow a computer the faster algorithm is run on.

IV. LOWER BOUNDS

The search for asymptotically fast algorithms is a challenging aspect of algorithm design. Once we have designed an algorithm for a particular problem, we would like to know if this is the asymptotically best algorithm. If not, we would like to know how close we are to the asymptotically best algorithm. To answer these questions, we must determine a function $f(n)$ with the following property:

PI: Let A be any algorithm that solves the given problem. Let its asymptotic complexity be $O(g(n))$. $f(n)$ is such that $g(n) = \Omega(f(n))$.

That is, $f(n)$ is a lower bound on the complexity of every algorithm for the given problem. If we develop an

TABLE V Comparison between IBM-PC and VAX[a]

n	IBM-PC merge sort turbo	VAX insertion sort pc
400	109.8	64.1
500	138.0	106.1
600	171.2	161.8
700	199.2	217.9
800	230.5	263.5
900	260.1	341.9
1000	289.5	418.8

[a] Times are in hundredths of a second.

algorithm whose complexity is equal to the lower bound for the problem being solved, then the developed algorithm is optimal. The number of input and output data often provides a trivial lower bound on the complexity of many problems. For example, to sort n numbers it is necessary to examine each number at least once. So every sort algorithm must have complexity $\Omega(n)$. This lower bound is not a very good lower bound and can be improved with stronger arguments than the one just used. Some of the methods for obtaining nontrivial lower bounds are the following:

1. Information-theoretic arguments
2. State space arguments
3. Adversary constructions
4. Reducibility constructions

A. Information-Theoretic Arguments

In an information-theoretic argument, one determines the number of different behaviors the algorithm must exhibit in order to work correctly for the given problem. For example, if an algorithm is to sort n numbers, it must be capable of generating $n!$ different permutations of the n input numbers. This is because depending on the particular values of the n numbers to be sorted, any of these $n!$ permutations could represent the right sorted order. The next step is to determine how much time every algorithm that has this many behaviors must spend in the solution of the problem. To determine this quantity, one normally places restrictions on the kinds of computations the algorithm is allowed to perform. For instance, for the sorting problem, we may restrict our attention to algorithms that are permitted to compare the numbers to be sorted but not permitted to perform arithmetic on these numbers. Under these restrictions, it can be shown that $n \log n$ is a lower bound on the average and worst-case complexity of sorting. Since the average and worst-case complexities of merge sort is $\Theta(n \log n)$, we conclude that merge sort is an asymptotically optimal sorting algorithm under both the average and worst-case measures. Note that it is possible for a problem to have several different algorithms that are asymptotically optimal. Some of these may actually run faster than others. For example, under the above restrictions, there may be two optimal sorting algorithms. Both will have asymptotic complexity $\Theta(n \log n)$. However, one may run in $10n \log n$ time and the other in $2On \log n$ time. A lower bound $f(n)$ is a tight lower bound for a certain problem if this problem is, in fact, solvable by an algorithm of complexity $O(f(n))$. The lower bound obtained above for the sorting problem is a tight lower bound for algorithms that are restricted to perform only comparisons among the numbers to be sorted.

B. State Space Arguments

In the case of a state space argument, we define a set of states that any algorithm for a particular problem can be in. For example, suppose we wish to determine the largest of n numbers. Once again, assume we are restricted to algorithms that can perform comparisons among these numbers but cannot perform any arithmetic on them. An algorithm state can be described by a tuple (i, j). An algorithm in this state "knows" that j of the numbers are not candidates for the largest number and that $i = n - j$ of them are. When the algorithm begins, it is in the state $(n, 0)$, and when it terminates, it is in the state $(1, n-1)$. Let A denote the set of numbers that are candidates for the largest and let B denote the set of numbers that are not. When an algorithm is in state (i, j), there are i numbers in A and j numbers in B. The types of comparisons one can perform are $A:A$ ("compare one number in A with another in A"), $A:B$, and $B:B$. The possible state changes are as follows:

$A:A$	This results in a transformation from the state (i, j) to the state $(i-1, j+1)$
$B:B$	The state (i, j) is unchanged as a result of this type of comparison.
$A:B$	Depending on the outcome of the comparison, the state either will be unchanged or will become $(i-1, j+1)$.

Having identified the possible state transitions, we must now find the minimum number of transitions needed to go from the initial state to the final state. This is readily seen to be $n-1$. So every algorithm (that is restricted as above) to find the largest of n numbers must make at least $n-1$ comparisons.

C. Adversary and Reducibility Constructions

In an adversary construction, one obtains a problem instance on which the purported algorithm must do at least a certain amount of work if it is to obtain the right answer. This amount of work becomes the lower bound. A reducibility construction is used to show that, employing an algorithm for one problem (A), one can solve another problem (B). If we have a lower bound for problem B, then a lower bound for problem A can be obtained as a result of the above construction.

V. NP-HARD AND NP-COMPLETE PROBLEMS

Obtaining good lower bounds on the complexity of a problem is a very difficult task. Such bounds are known for a handful of problems only. It is somewhat easier to relate

the complexity of one problem to that of another using the notion of reducibility that we briefly mentioned in the last section. Two very important classes of reducible problems are NP-hard and NP-complete. Informally, all problems in the class NP-complete have the property that, if one can be solved by an algorithm of polynomial complexity, then all of them can. If an NP-hard problem can be solved by an algorithm of polynomial complexity, then all NP-complete problems can be so solved. The importance of these two classes comes from the following facts:

1. No NP-hard or NP-complete problem is known to be polynomially solvable.
2. The two classes contain more than a thousand problems that have significant application.
3. Algorithms that are not of low-order polynomial complexity are of limited value.
4. It is unlikely that any NP-complete or NP-hard problem is polynomially solvable because of the relationship between these classes and the class of decision problems that can be solved in polynomial nondeterministic time.

We shall elaborate the last item in the following subsections.

VI. NONDETERMINISM

According to the common notion of an algorithm, the result of every step is uniquely defined. Algorithms with this property are called deterministic algorithms. From a theoretical framework, we can remove this restriction on the outcome of every operation. We can allow algorithms to contain an operation whose outcome is not uniquely defined but is limited to a specific set of possibilities. A computer that executes these operations are allowed to choose anyone of these outcomes. This leads to the concept of a *nondeterministic algorithm*. To specify such algorithms we introduce three new functions:

- **choice**(S): Arbitrarily choose one of the elements of set S.
- **failure**: Signals an unsuccessful completion.
- **success**: Signals a successful completion.

Thus the assignment statement $x =$ **choice**$(1 : n)$ could result in x being assigned anyone of the integers in the range $[1, n]$. There is no rule specifying how this choice is to be made. The failure and success signals are used to define a computation of the algorithm. The computation of a nondeterministic algorithm proceeds in such a way that, whenever there is a set of choices that leads to a successful completion, one such set of choices is made and the algorithm terminates successfully. A nondeterministic algorithm terminates unsuccessfully if there exists no set of choices leading to a success signal. A computer capable of executing a nondeterministic algorithm in this way is called a nondeterministic computer. (The notion of such a nondeterministic computer is purely theoretical, because no one knows how to build a computer that will execute nondeterministic algorithms in the way just described.)

Consider the problem of searching for an element x in a given set of elements $a[1 \ldots n]$, $n \geq 1$. We are required to determine an index j such that $a[j] = x$. If no such j exists (i.e., x is not one of the a's), then j is to be set to 0. A nondeterministic algorithm for this is the following:

```
j = choice(1:n)
if a[j] = x then print U); success endif
print ("0"); failure.
```

From the way a nondeterministic computation is defined, it follows that the number 0 can be output if there is no j such that $a[j] = x$. The computing times for **choice**, **success**, and **failure** are taken to be $O(1)$. Thus the above algorithm is of nondeterministic complexity $O(1)$. Note that since a is not ordered, every deterministic search algorithm is of complexity $\Omega(n)$. Since many choice sequences may lead to a successful termination of a nondeterministic algorithm, the output of such an algorithm working on a given data set may not be uniquely defined. To overcome this difficulty, one normally considers only decision problems, that is, problems with answer 0 or 1 (or true or false). A successful termination yields the output 1, while an unsuccessful termination yields the output 0. The time required by a nondeterministic algorithm performing on any given input depends on whether there exists a sequence of choices that leads to a successful completion. If such a sequence exists, the time required is the minimum number of steps leading to such a completion. If no choice sequence leads to a successful completion, the algorithm takes $O(1)$ time to make a failure termination. Nondeterminism appears to be a powerful tool. Program 10 is a nondeterministic algorithm for the sum of subsets problem. In this problem, we are given a multiset $w(1 \ldots n)$ of n natural numbers and another natural number M. We are required to determine whether there is a sub multiset of these n natural numbers that sums to M. The complexity of this nondeterministic algorithm is $O(n)$. The fastest deterministic algorithm known for this problem has complexity $O(2^{n/2})$.

PROGRAM 10: Nondeterministic Sum of Subsets

```
line procedure NonDeterministicSumOfSub-
  sets(W,n,M)
2  for i := 1 to n do
```

```
3            x(i) := choice({0,1});
4            endfor
5   if Σ_{i=1}^{n} w(i)x(i) = M then success
6            else failure;
7   endif;
7   end;
```

A. NP-Hard and NP-Complete Problems

The size of a problem instance is the number of digits needed to represent that instance. An instance of the sum of subsets problem is given by $(w(I), w(2), \ldots, w(n), M)$. If each of these numbers is a positive integer, the instance size is

$$\left\lceil \sum_{i=1}^{n} \log_2(w(i)+1) \right\rceil + \lceil \log_2(M+1) \rceil$$

if binary digits are used. An algorithm is of polynomial time complexity if its computing time is $O(p(m))$ for every input of size m and some fixed polynomial $p(-)$.

Let P be the set of all decision problems that can be solved in deterministic polynomial time. Let NP be the set of decision problems solvable in polynomial time by nondeterministic algorithms. Clearly, $P \subset NP$. It is not known whether $P = NP$ or $P \neq NP$. The $P = NP$ problem is important because it is related to the complexity of many interesting problems. There exist many problems that cannot be solved in polynomial time unless $P = NP$. Since, intuitively, one expects that $P \subset NP$, these problems are in "all probability" not solvable in polynomial time. The first problem that was shown to be related to the $P = NP$ problem, in this way, was the problem of determining whether a propositional formula is satisfiable. This problem is referred to as the satisfiability problem.

Theorem 2. Satisfiability is in P iff $P = NP$.

Let A and B be two problems. Problem A is polynomially reducible to problem B (abbreviated A reduces to B, and written as A α B) if the existence of a deterministic polynomial time algorithm for B implies the existence of a deterministic polynomial time algorithm for A. Thus if A α B and B is polynomiaIly solvable, then so also is A. A problem A is NP-hard iff satisfiability α A. An NP-hard problem A is NP-complete if $A \in NP$. Observe that the relation α is transitive (i.e., if A α B and B α C, then A α C). Consequently, if A α B and satisfiabilityα A then B is NP-hard. So, to show that any problem B is NP-hard, we need merely show that A α B, where A is any known NP-hard problem. Some of the known NP-hard problems are as follows:

NP1: Sum of Subsets

Input: Multiset $W = \{w_i \mid 1 \leq i \leq n\}$ of natural numbers and another natural number M.

Output: "Yes" if there is a submultiset of What sums to M; "No" otherwise.

NP2: 0/1-Knapsack

Input: Multisets $P = \{P_i \mid 1 \leq i \leq n\}$ and $W = \{W_i | 1 \leq i \leq n\}$ of natural numbers and another natural number M.

Output: $x_i \in \{0, 1\}$ such that $\sum_i$ `is maximized and` *iWiXi* M.

NP3: Traveling Salesman

Input: A set of n points and distances $d(i, j)$. The $d(i, j)$ is the distance between the points i and j.

Output: A minimum-length tour that goes through each of the n points exactly once and returns to the start of the tour. The length of a tour is the sum of the distances between consecutive points on the tour. For example, the tour $1 \to 3 \to 2 \to 4 \to 1$ has the length $d(1, 3) + d(3, 2) + d(2, 4) + d(4, 1)$.

NP4: Chromatic Number

Input: An undirected graph $G = (V, E)$.

Output: A natural number k such that k is the smallest number of colors needed to color the graph. A coloring of a graph is an assignment of colors to the vertices in such a way that no two vertices that are connected by an edge are assigned the same color.

NP5: Clique

Input: An undirected graph $G = (V, E)$ and a natural number k.

Output: "Yes" if G contains a clique of size k (i.e., a subset $U \subset V$ of size k such that every two vertices in U are connected by an edge in E) or more. "No" otherwise.

NP6: Independent Set

Input: An undirected graph $G = (V, E)$ and a natural number k.

Output: "Yes" if G contains an independent set of size k (i.e., a subset $U \subset V$ of size k such that no two vertices in U are connected by an edge in E) or more. "No" otherwise.

NP7: Hamiltonian Cycle

Input: An undirected graph $G = (V, E)$.
Output: "Yes" if G contains a Hamiltonian cycle (i.e., a path that goes through each vertex of G exactly once and then returns to the start vertex of the path). "No" otherwise.

NP8: Bin Packing

Input: A set of n objects, each of size $s(i)$, $1 \leq i \leq n$ [$s(i)$ is a positive number], and two natural numbers k and C.
Output: "Yes" if the n objects can be packed into at most k bins of size c. "No" otherwise. When packing objects into bins, it is not permissible to split an object over two or more bins.

NP9: Set Packing

Input: A collection S of finite sets and a natural number k.
Output: "Yes" if S contains at least k mutually disjoint sets. "No" otherwise.

NP10: Hitting Set.

Input: A collection S of subsets of a finite set U and a natural number k.
Output: "Yes" if there is a subset V of U such that V has at most k elements and V contains at least one element from each of the subsets in S. "No" otherwise.

The importance of showing that a problem A is NP-hard lies in the $P = NP$ problem. Since we do not expect that $P = NP$, we do not expect. NP-hard problems to be solvable by algorithms with a worst-case complexity that is polynomial in the size of the problem instance. From Table II, it is apparent that, if a problem cannot be solved in polynomial time (in particular, low-order polynomial time), it is intractable, for all practical purposes. If A is NP-complete and if it does turn out that $P = NP$, then A will be polynomially solvable. However, if A is only NP-hard, it is possible for P to equal NP and for A not to be in P.

VII. COPING WITH COMPLEXITY

An optimization problem is a problem in which one wishes to optimize (i.e., maximize or minimize) an optimization function $f(x)$ subject to certain constraints $C(x)$. For example, the NP-hard problem NP2 (0/1-knapsack) is an optimization problem. Here, we wish to optimize (in this case maximize) the function $f(x) = \sum_{i=1}^{n} p_i x_i$ subject to the following constraints:

(a) $x_i \in \{0, 1\}, 1 \leq i \leq n$
(b) $\sum_{i=1}^{n} w_i x_i \leq M$

A feasible solution is any solution that satisfies the constraints $C(x)$. For the 0/1-knapsack problem, any assignment of values to the x_i'S that satisfies constraints (a) and (b) above is a feasible solution. An optimal solution is a feasible solution that results in an optimal (maximum in the case of the 0/1-knapsack problem) value for the optimization function. There are many interesting and important optimization problems for which the fastest algorithms known are impractical. Many of these problems are, in fact, known to be NP-hard. The following are some of the common strategies adopted when one is unable to develop a practically useful algorithm for a given optimization:

1. Arrive at an algorithm that always finds optimal solutions. The complexity of this algorithm is such that it is computationally feasible for "most" of the instances people want to solve. Such an algorithm is called a *usually good algorithm*. The simplex algorithm for linear programming is a good example of a usually good algorithm. Its worst-case complexity is exponential. However, it can solve most of the instances given it in a "reasonable" amount of time (much less than the worst-case time).
2. Obtain a computationally feasible algorithm that "almost" always finds optimal solutions. At other times, the solution found may have a value very distant from the optimal value. An algorithm with this property is called a *probabilistically good algorithm*.
3. Obtain a computationally feasible algorithm that obtains "reasonably" good feasible solutions. Algorithms with this property are called *heuristic algorithms*. If the heuristic algorithm is guaranteed to find feasible solutions that have value within a prespecified amount of the optimal value, the algorithm is called an approximation algorithm. In the remainder of this section, we elaborate on approximation algorithms and other heuristics.

A. Approximation Algorithms

When evaluating an approximation algorithm, one considers two measures: algorithm complexity and the quality of the answer (i.e., how close it is to being optimal). As in the case of complexity, the second measure may refer to the worst case or the average case. There are several categories of approximation algorithms. Let A be an algorithm

that generates a feasible solution to every instance I of a problem P. Let $F^*(I)$ be the value of an optimal solution, and let $F'(I)$ be the value of the solution generated by A.

Definition. A is a k-absolute approximation algorithm for P iff $|F^*(I) - F'(I)| \leq k$ for all instances I. k is a constant. A is an $f(n)$-approximate algorithm for p iff $|F^*(I) - F'(I)|/F^*(I) \leq f(n)$ for all I. The n is the size of I and we assume that $|F^*(I)| > 0$. An $f(n)$-approximate algorithm with $f(n) \leq \varepsilon$ for all n and some constant ε is an ε-approximate algorithm.

Definition. Let $A(\varepsilon)$ be a family of algorithms that obtain a feasible solution for every instance I of P. Let n be the size of I. $A(\varepsilon)$ is an approximation scheme for P if for every $A(\varepsilon) > 0$ and every instance $I, |F^*(I) - F'(I)|F^*(I) \leq \varepsilon$. An approximation scheme whose time complexity is polynomial in n is a polynomial time approximation scheme. A fully polynomial time approximation scheme is an approximation scheme whose time complexity is polynomial in n and $1/\varepsilon$.

For most NP-hard problems, the problem of finding k-absolute approximations is also NP-hard. As an example, consider problem NP2 (01 l-knapsack). From any instance $(P_i, W_i, 1 \leq i \leq n, M)$, we can construct, in linear time, the instance $(k+1)p_i, w_i, \leq i \leq n, M)$. This new instance has the same feasible solutions as the old. However, the values of the feasible solutions to the new instance are multiples of $k+1$. Consequently, every k-absolute approximate solution to the new instance is an optimal solution for both the new and the old instance. Hence, k-absolute approximate solutions to the 0/1-knapsack problem cannot be found any faster than optimal solutions.

For several NP-hard problems, the ε-approximation problem is also known to be NP-hard. For others fast ε-approximation algorithms are known. As an example, we consider the optimization version of the bin-packing problem (NP8). This differs from NP8 in that the number of bins k is not part of the input. Instead, we are to find a packing of the n objects into bins of size C using the fewest number of bins. Some fast heuristics that are also ε-approximation algorithms are the following:

First Fit (*FF*). Objects are considered for packing in the order $1, 2, \ldots, n$. We assume a large number of bins arranged left to right. Object i is packed into the leftmost bin into which it fits.

Best Fit (*BF*). Let $cAvail[j]$ denote the capacity available in bin j. Initially, this is C for all bins. Object i is packed into the bin with the least $cAvail$ that is at least $s(i)$.

First Fit Decreasing (*FFD*). This is the same as FF except that the objects are first reordered so that $s(i) \geq s(i+1)1 \leq ileqn$.

Best Fit Decreasing (*BFD*). This is the same as BF except that the objects are reordered as for FFD. It should be possible to show that none of these methods guarantees optimal packings. All four are intuitively appealing and can be expected to perform well in practice. Let I be any instance of the bin packing problem. Let $b(I)$ be the number of bins used by an optimal packing. It can be shown that the number of bins used by FF and BF never exceeds $(17/10)b(I) + 2$, while that used by FFD and BFD does not exceed $(11/9)b(I) + 4$.

Example. Four objects with $s(1:4) = (3, 5, 2, 4)$ are to be packed in bins of size 7. When FF is used, object 1 goes into bin 1 and object 2 into bin 2. Object 3 fits into the first bin and is placed there. Object 4 does not fit into either of the two bins used so far and a new bin is used. The solution produced utilizes 3 bins and has objects 1 and 3 in bin I, object 2 in bin 2, and object 4 in bin 3.

When BF is used, objects 1 and 2 get into bins 1 and 2, respectively. Object 3 gets into bin 2, since this provides a better fit than bin I. Object 4 now fits into bin I. The packing obtained uses only two bins and has objects 1 and 4 in bin 1 and objects 2 and 3 in bin 2. For FFD and BFD, the objects are packed in the order 2,4, 1,3. In both cases, two-bin packing is obtained. Objects 2 and 3 are in bin 1 and objects 1 and 4 in bin 2. Approximation schemes (in particular fully polynomial time approximation schemes) are also known for several NP-hard problems. We will not provide any examples here.

B. Other Heuristics

PROGRAM 12: *General Form of an Exchange Heuristic*

1. Let j be a random feasible solution [i.e., $C(j)$ is satisfied] to the given problem.
2. Perform perturbations (i.e., exchanges) on i until it is not possible to improve j by such a perturbation.
3. Output i.

Often, the heuristics one is able to devise for a problem are not guaranteed to produce solutions with value close to optimal. The virtue of these heuristics lies in their capacity to produce good solutions most of the time. A general category of heuristics that enjoys this property is the class of exchange heuristics. In an exchange heuristic for an optimization problem, we generally begin with a feasible solution and change parts of it in an attempt to improve its value. This change in the feasible solution is called a perturbation. The initial feasible solution can be obtained using some other heuristic method or may be a randomly generated solution. Suppose that we wish to minimize the

objective function $f(i)$ subject to the constraints C. Here, i denotes a feasible solution (i.e., one that satisfies C). Classical exchange heuristics follow the steps given in Program 12. This assumes that we start with a random feasible solution. We may, at times, start with a solution constructed by some other heuristic. The quality of the solution obtained using Program 12 can be improved by running this program several times. Each time, a different starting solution is used. The best of the solutions produced by the program is used as the final solution.

1. A Monte Carlo Improvement Method

In practice, the quality of the solution produced by an exchange heuristic is enhanced if the heuristic occasionally accepts exchanges that produce a feasible solution with increased $f(\ )$. (Recall that f is the function we wish to minimize.) This is justified on the grounds that a bad exchange now may lead to a better solution later. In order to implement this strategy of occasionally accepting bad exchanges, we need a probability function prob(i, j) that provides the probability with which an exchange that transforms solution i into the inferior solution j is to be accepted. Once we have this probability function, the Monte Carlo improvement method results in exchange heuristics taking the form given in Program 13. This form was proposed by N. Metropolis in 1953. The variables *counter* and n are used to stop the procedure. If n successive attempts to perform an exchange on i are rejected, then an optimum with respect to the exchange heuristic is assumed to have been reached and the algorithm terminates. Several modifications of the basic Metropolis scheme have been proposed. One of these is to use a sequence of different probability functions. The first in this sequence is used initially, then we move to the next function, and so on. The transition from one function to the next can be made whenever sufficient computer time has been spent at one function or when a sufficient number of perturbations have failed to improve the current solution.

PROGRAM 13: *Metropolis Monte Carlo Method*

1. Let i be a random feasible solution to the given problem. Set $counter = 0$.
2. Let j be a feasible solution that is obtained from i as a result of a random perturbation.
3. If $f(j) < f(i)$, then [$i = j$, update best solution found so far in case i is best, $counter = 0$, go to Step 2].
4. If $f(j) \geq f(i)$ If $counter = n$ then output best solution found and stop. Otherwise, $r =$ random number in the range (0, 1).
   If $r <$ prob(i, j), then [$i = j$, $counter = 0$] else [$counter = counter + 1$].
   go to Step 2.

The Metropolis Monte Carlo Method could also be referred to as a *metaheuristic*, that is, a heuristic that is general enough to apply to a broad range of problems. Similar to heuristics, these are not guaranteed to produce an optimal solution, so are often used in situations either where this is not crucial, or a suboptimal solution can be modified.

VIII. THE FUTURE OF ALGORITHMS

Computer algorithm design will always remain a crucial part of computer science. Current research has a number of focuses, from the optimization of existing, classic algorithms, such as the sorting algorithms described here, to the development of more efficient approximation algorithms. The latter is becoming an increasingly important area as computers are applied to more and more difficult problems.

This research itself can be divided into two main areas, the development of approximation algorithms for particular problems, e.g. Traveling salesman problem, and into the area of metaheuristics, which is more concerned with the development of general problem solvers.

IX. SUMMARY

In order to solve difficult problems in a reasonable amount of time, it is necessary to use a good algorithm, a good compiler, and a fast computer. A typical user, generally, does not have much choice regarding the last two of these. The choice is limited to the compilers and computers the user has access to. However, one has considerable flexibility in the design of the algorithm. Several techniques are available for designing good algorithms and determining how good these are. For the latter, one can carry out an asymptotic analysis. One can also obtain actual run times on the target computer.When one is unable to obtain a low-order polynomial time algorithm for a given problem, one can attempt to show that the problem is NP-hard or is related to some other problem that is known to be computationally difficult. Regardless of whether one succeeds in this endeavor, it is necessary to develop a practical algorithm to solve the problem. One of the suggested strategies for coping with complexity can be adopted.

SEE ALSO THE FOLLOWING ARTICLES

BASIC PROGRAMMING LANGUAGE • C AND C++ PROGRAMMING LANGUAGE • COMPUTER ARCHITECTURE •

DISCRETE SYSTEMS MODELING • EVOLUTIONARY ALGORITHMS AND METAHEURISTICS • INFORMATION THEORY • SOFTWARE ENGINEERING • SOFTWARE RELIABILITY • SOFTWARE TESTING

BIBLIOGRAPHY

Aho, A., Hopcroft, J., and Ullman, J. (1974). "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA.

Canny, J. (1990). *J. Symbolic Comput.* **9** (3), 241–250.

Garey, M., and Johnson, D. (1979). "Computers and Intractability," Freeman, San Francisco, CA.

Horowitz, E., and Sahni, S. (1978). "Fundamentals of Computer Algorithms," Computer Science Press, Rockville, MD.

Kirkpatrick, S., Gelatt, C., and Vecchi, M. (1983). *Science* **220,** 671–680.

Knuth, D. (1972). *Commun. ACM* **15** (7), 671–677.

Nahar, S., Sahni, S., and Shragowitz, E. (1985). *ACM/IEEE Des. Autom. Conf.*, 1985, pp. 748–752.

Sahni, S. (1985). "Concepts in Discrete Mathematics," 2nd ed., Camelot, Fridley, MN.

Sahni, S. (1985). "Software Development in Pascal," Camelot, Fridley, MN.

Sedgewick, R. (1983). "Algorithms," Addison-Westley, Reading, MA.

Syslo, M., Deo, N., and Kowalik, J. (1983). "Discrete Optimization Algorithms," Prentice-Hall, Engle-wood Cliffs, NJ.

Computer Viruses

Ernst L. Leiss
University of Houston

GLOSSARY

Data integrity Measure of the ability of a (computer) system to prevent unwanted (unauthorized) changes or destruction of data and software.
Data security Measure of the ability of a (computer) system to prevent unwanted (unauthorized) access to data and software.
Logical bomb Code embedded in software whose execution will cause undesired, possibly damaging, actions.
Subversion Any action that results in the circumvention of violation of security principles.
Worm Self-contained program that is usually not permanently stored as a file and has the capacity of self-replication and of causing damage to data and software.
Virus Logical bomb with the ability of self-replication. It usually is a permanent part of an existing, permanently stored file and has the capability of causing damage to data and software.

I. BACKGROUND AND MOTIVATION

In the years before 1988, a number of incidents suggested the potential for major problems related to the organized and widespread subversion of (networked) computer systems, accompanied by the possibility of massive destruction of data and software. While until then these concerns were considered rather remote, the Internet attack of 1988 shattered this complacency. In the intervening decade, computer viruses have attained significant visibility in the computer-literate population, rivalling the notoriety of Y2K-related problems but with substantially greater staying power.

The reason for the attention attracted by these intruders lies in their potential for destruction of data and software. With the exception of some highly secured systems related to defense and national security, virtually all larger computer systems are connected via computer networks, commonly referred to as the Internet. Personal computers, if they are not permanently linked into these networks, have at least the capability of linking up to them intermittently through a variety of Internet service providers. Networks are systems that allow the transmission of digitally encoded information (data, software, messages, as well as still images, video, and audio) at relatively high speeds and in relatively convenient ways from one system to another. Subverting the function of a network may therefore result in the subversion of the computers linked by it. Consequently, a scenario is very plausible in which a program may be transmitted that is capable of destroying large amounts of data in all the computers in a given network.

The case that such a scenario is plausible has been made for many years, starting with F. Cohen's demonstration of a computer virus in 1983.

In 1988, such a scenario was played out for the first time on a worldwide scale. Since then, numerous incidents have reinforced the public's sense of vulnerability to attacks by insidious code fragments on software and data stored in all kinds of computers. While earlier virus attacks spread via diskettes and later via electronic bulletin boards (in ways that required some user participation through loading infected programs), in recent years, the World Wide Web and more sophisticated e-mail systems have provided transmission channels that facilitated the worldwide spread of the attackers at a unprecedented speed. Moreover, infection which earlier required some explicit action by the victim has become much more stealthy, with the advent of viruses that become activated through the opening (or even previewing) of an apparently innocent attachment to an e-mail document.

The destruction of data and software has obvious economic implications. The resulting malfunctioning of computer systems may also affect safety-critical systems, such as air-traffic control systems or control systems for hydroelectric dams or nuclear power plants. Futhermore, the potential for disruption can be damaging: a bomb threat can conceivably be more paralyzing that the explosion of a small bomb itself. Protection against such treats may be either impossible or unacceptable to usesrs in the necessarily resulting reduction of functionality and ease of use of computer systems. It must be borne in mind that by necessity, the notion of user friendliness of a computer system of communications network is antithetical to the notions of data security and data integrity.

II. VIRUSES, WORMS, AND SO FORTH

From a technical point of view, the most alarming aspect of the attackers under discussion in this article is that they are self-replicating. In other words, the piece of software that performs the subversion has the ability of making copies of itself and transmitting those copies to other programs in the computer or to other computers in the network. Obviously, each of these copies can now wreak havoc where it is and replicate itself as well! Thus, it may be sufficient to set one such program loose in one computer in order to affect all computers in a given network. Since more and more computers, including personal computers, are interconnected, the threat of subversion can assume literally global dimensions. Let us look at this in greater detail. First, we define a few important terms.

A logical bomb is a piece of code, usually embedded in other software, that is only activated (executed) if a certain condition is met. It does not have the capability of self-replication. Activation of the logical bomb may abort a program run or erase data or program files. If the condition for execution is not satisfied at all times, it may be regarded as a logical time bomb. Logical bombs that are activated in every invocation are usually not as harmful as time bombs since their actions can be observed in every execution of the affected software. A typical time bomb is one where a disgruntled employee inserts into complex software that is frequently used (a compiler or a payroll system, for example) code that will abort the execution of the software, for instance, after a certain date, naturally chosen to fall after the date of the employee's resignation or dismissal.

While some programming errors may appear to be time bombs (the infamous Y2k problem certainly being the best known and most costly of these), virtually all intentional logical bombs are malicious.

A computer virus is a logical bomb that is able to self-replicate, to subvert a computer system in some way, and to transmit copies of itself to other hardware and software systems. Each of these copies in turn may self-replicate and affect yet other systems. A computer virus usually attaches itself to an existing program and thereby is permanently stored.

A worm is very similar to a computer virus in that it is self-replicating and subverts a system; however, it usually is a self-contained program that enters a system via regular communication channels in a network and then generates its own commands. Therefore, it is frequently not permanently stored as a file but rather exists only in the main memory of the computer. Note that a logical bomb resident in a piece of software that is explicitly copied to another system may appear as a virus to the users, even though it is not.

Each of the three types of subversion mechanisms, bombs, viruses, and worms, can, but need not, cause damage. Instances are known in which bombs and viruses merely printed out some brief message on the screen and then erased themselves, without destroying data or causing other disruptions. These can be considered as relatively harmless pranks. However, it must be clearly understood that these subversion mechanisms, especially the self-replicating ones, most definitely have enormous potential for damage. This may be due to deliberate and explicit erasure of data and software, or it may be due to far less obvious secondary effects. To give one example, consider a worm that arrives at some system via electronic mail, thereby activating a process that handles the receiving of mail. Typically, this process has a high priority; that is, if there are any other processes executing, they will be suspended until the mail handler is finished. Thus, if the system receives many mail messages, a user may get the impression that the system is greatly slowed down. If

these mail messages are all copies of the same worm, it is clear that the system can easily be saturated and thereby damage can be done, even though no data or programs are erased.

This is what happened in the historic case study cited above. On November 2, 1988, when a worm invaded over 6000 computers linked together by a major U.S. network that was the precursor to the present-day Internet, including Arpanet, Milnet, and NSFnet. Affected were computers running the operating system Berkeley Unix 4.3. The worm took advantage of two different flaws, namely, a debugging device in the mail handler (that most centers left in place even though it was not required any longer after successful installation of the mail handler) and a similar problem in a communications program. The worm exploited these flaws by causing the mail handler to circumvent the usual access controls in a fairly sophisticated way; it also searched users' files for lists of trusted users (who have higher levels of authority) and used them to infiltrate other programs. The worm's means of transmission between computer was the network. Because infiltrated sites could be reinfiltrated arbitrarily often, systems (especially those that were favorite recipients of mail) became saturated and stopped performing useful work. This was how users discovered the infiltration, and this was also the primary damage that the worm caused. (While it did not erase or modify any data, it certainly was capable of doing this had it been so designed.) The secondary damage was caused by the efforts to remove the worm. Because of the large number of sites affected, this cost was estimated to have amounted to many years of work, even though it was relatively easy to eliminate the worm by rebooting each system because the worm was never permanently stored.

One reason this worm made great waves was that it caused the first major infiltration of mainframe computers. Prior to this incident, various computer viruses (causing various degrees of damage) had been reported, but only for personal computers. Personal computers are typically less sophisticated and originally had been designed for personal use only, not for networking; for these reasons they had been considered more susceptible to attacks from viruses. Thus, threats to mainframes from viruses were thought to be far less likely than threats to personal computers. The November 2, 1988, incident destroyed this myth in less than half a day, the time it took to shut down Internet and many computer systems on it.

Since then, a wide variety of attackers have appeared on the scene, substantially aided by the explosive growth of the World Wide Web. Not surprisingly, given the dominance that Microsoft's operating systems have in the market, most recent viruses exist within the context of that company's operating systems. Many of these viruses use the increasingly common use of attachments to be transmitted surreptitiously; in this case, opening an attachment may be all that is required to get infected. In fact, users may not even be aware that an attachment was opened, because it occurred automatically (to support more sophisticated mail functions, such as previewing or mail sorting according to some user-specified criterion). Frequently, the resulting subversion of the mail system facilitates further distribution of the virus, using mailing lists maintained by the system.

III. PREVENTION AND DETECTION

There are two different approaches to defending against viruses and worms. One is aimed at the prevention or detection of the transmission of the infiltrator; the other tries to prevent or detect damage that the infiltrator may cause by erasing or modifying files. The notable 1988 worm incident illustrates, however, that even an infiltrator that does not alter any files can be very disruptive.

Several defense mechanisms have been identified in the literature; below some of the more easily implementable defenses are listed. One should, however, keep in mind that most of them will be implemented by more software, which in turn could be subject to infiltration. First, however, it is necessary to state two fundamental principles:

1. No infection without execution.
2. Detection is undecidable.

The first principle refers to the fact that infection cannot occur unless some type of (infected) software is executed. In other words, merely looking at an infected program will not transmit a virus. Thus, simple-minded e-mail programs that handle only flat ASCII files are safe since no execution takes place. As we pointed out earlier, the execution involved can be virtually hidden from the user (e.g., in the case of previewing attachments), but in every case, the user either enabled the execution or could explicitly turn it off. The second principle has major implications. Essentially, it states that it is probably impossible to design a technique that would examine an arbitrary program and determine whether it contains a virus. This immediately raises the question of how virus detections software functions. Let us make a small excursion first. We claim that any half-way effective virus must have the ability of determining whether a program has already been infected by it. If it did not have this capability,it would reinfect an already infected program. However, since a virus is a code fragment of a certain length, inserting that same code fragment over and over into the same program would result in a program that keeps on growing until it

eventually exceeds any available storage capacity, resulting in immediate detection of the virus. Returning now to our question of how virus detection software works, we can say that it does exactly the same that each virus does. This ofcourse implies trivially that that test can be carried out only if the virus is known. In other words, virus detection software will never be able to find any virus; it will only be able to detect viruses that were known to the authors of the detection software at the time it was written. The upshot is that old virus detection software is virtually worthless since it will not be able to detect any viruses that appeared since the software was written. Consequently, it is crucial to update one's virus detection software frequently and consistently.

While many virus detection programs will attempt to remove a virus once it is detected, removal is significantly trickier and can result in the corruption of programs. Since viruses and worms typically have all the access privileges that the user has, but no more, it is possible to set the permissions for all files so that writing is not permitted, even for the owner of the files. In this way, the virus will not be able to write the files, something that would be required to insert the virus. It is true that the virus could subvert the software that controls the setting of protections, but to date (February 2000), no virus has ever achieved this. (Whenever a user legitimately wants to write a file, the user would have to change the protection first, then write the file, and then change the protection back.) The primary advantage of this method is that it is quite simple and very effective. Its primary disadvantage is that users might find it inconvenient. Other, more complicated approaches include the following:

1. Requirement of separate and explicit approval (from the user) for certain operations: this assumes as interactive environment and is probably far too combersome for most practical use. Technically, this can be implemented either as software that requires the user to enter an approval code or as hardware addendum to the disk drive that prevents any unapproved writes to that disk. Note, however, that a good deal of software, for example, compilers, legitimately write to disk, even though what is written may be already infiltrated.
2. Comparison with protected copy: another way of preventing unauthorized writes is to have a protected copy (encrypted or on a write-once disk, see method 6) of a program and to compare that copy with the conventionally stored program about to be executed.
3. Control key: a control key can be computed for each file. This may be a type of check sum, the length of the file, or some other function of the file. Again, it is important that this control key be stored incorruptible.
4. Time stamping: many operating systems store a date of last modification for each file. If a user separately and incorruptibly stores the date of last modification for important files, discrepancies can indicate possible infiltrations.
5. Encryption: files are stored in encrypted format (that is, as cipher text). Before usage, a file must be decrypted. Any insertion of unencrypted code (as it would be done by a virus trying to infiltrate a program) will give garbage when the resulting file is decrypted.
6. Write-once disks: certain codes (immutable codes, balanced codes) can be used to prevent (physically) the change of any information stored on write once (or laser) disks.

All methods attempt to prevent unauthorized changes in programs or data. Methods 2 (protected copy), 3 (control key), and 6 (write-once disk) work primarily if no changes at all are permitted. Methods 4 (time stamping) and 5 (encryption) work if changes are to be possible.

None of these methods guarantees that attempted infiltrations will be foiled. However, these methods may make it very difficult for a virus to defeat the security defenses of a computer system. Note, however, that all are aimed at permanently stored files. Thus, a worm as in the November 2, 1988, incident may not be affected at all by any of them. As indicated, this worm took advantage of certain flaws in the mail handler and a communications program.

IV. CONCLUSION

Viruses and worms are a threat to data integrity and have the potential for endangering data security; the danger of these attackers is magnified substantially by their ability of self-replication. While the general mechanisms of viruses have not changed significantly over the past decade, the useage patterns of computers have, providing seemingly new ways of attacking data and software. Certain defense mechanisms are available; however, they do not guarantee that all attacks will be repulsed. In fact, technical means in general are insufficient to deal with the threats arising from viruses and worms as they use commonly accepted and convenient means of communication to infiltrate systems.

Using reasonable precautions such as restricting access; preventing users from running unfamiliar, possibly infiltrated software; centralizing software development and maintenance; acquiring software only from reputable vendors; avoiding opening attachments (at least from unknown senders); using virus detection software in a systematic and automated way (e.g., every log-on triggers a virus scan); and most importantly, preparing,

internally disseminating, and strictly adhering to a carefully thought-out disaster recovery plan (which must function even if the usual computer networks are not operational!), it is likely that major damage can be minimized.

SEE ALSO THE FOLLOWING ARTICLE

COMPUTER NETWORKS

BIBLIOGRAPHY

Anti-Virus Emergency Response Team, hosted by Network Associates (http://www.avertlabs.com).

Bontchev, V. (199). "Future Trends in Virus Writing," Virus Test Center, University of Hamburg, Germany (http://www.virusbtn.com/otherpapers/Trends).

Computer Emergency Response Team (CERT). Registered Service mark of Carnegie-Mellon University, Pittsburgh (http://www.cert.org).

Department of Energy Computer Incident Advisory Capability (http://www.ciac.org).

European Institute for Computer Anti-Virus Research (http://www.eicar.com).

Kephart, J. O., Sorkin, G. B., Chess, D. M., and White, S. R. (199). "Fighting Computer Viruses," Sci. Am., New York (http://www.sciam.com/ 1197issue/1197kephart.html).

Leiss, E. L. (1990). "Software Under Siege: Viruses and Worms," Elsevier, Oxford, UK.

Polk, W. T., and Bassham, L. E. (1994). "A Guide to the Selection of Anti-Virus Tools and Techniques," Natl. Inst. Standards and Technology Computer Security Divison (http://csrc.ncsl.nist.gov/nistpubs/select).

Cryptography

Rebecca N. Wright
AT&T Labs—Research

GLOSSARY

Ciphertext All or part of an encrypted message or file.
Computationally infeasible A computation is *computationally infeasible* if it is not practical to compute, for example if it would take millions of years for current computers.
Cryptosystem A *cryptosystem* or *cryptographic system* consists of an encryption algorithm and the corresponding decryption algorithm.
Digital signature Cryptographic means of authenticating the content and sender of a message, like a handwritten signature is to physical documents.
Encryption The process of transforming information to hide its meaning. An encryption algorithm is also called a *cipher* or *code*.
Decryption The process of recovering information that has been encrypted.
Key A parameter to encryption and decryption that controls how the information is transformed.
Plaintext The *plaintext* or *cleartext* is the data to be encrypted.
Public key cryptosystem A two-key cryptosystem in which the encryption key is made public, and the decryption key is kept secret.
Symmetric key cryptosystem A traditional single-key cryptosystem, in which the same key is used for encryption and decryption.

CRYPTOGRAPHY, from the Greek *krýpt-*, meaning hidden or secret, and *gráph-*, meaning to write, is the science of secret communication. Cryptography consists of encryption or enciphering, in which a *plaintext* message is transformed using an encryption key and an encryption algorithm into a *ciphertext* message, and decryption or deciphering, in which the ciphertext is transformed using the decryption key and the decryption algorithm back into the original plaintext. Cryptography protects the privacy and sometimes the authenticity of messages in a hostile environment. For an encryption algorithm to be considered secure, it should be difficult or impossible to determine the plaintext from the ciphertext without knowledge of the decryption key.

Historically, cryptography has been used to safeguard military and diplomatic communications and has therefore been of interest mainly to the government. Now, as the use of computers and computer networks grows, there is an increasing amount of information that is stored electronically, on computers that can be accessed from around the world via computer networks. As this happens, businesses and individuals are finding more of a need for protection of information that is proprietary, sensitive or expensive to obtain.

Traditionally, encryption was used for a *sender* to send a message to a *receiver* in such a way that others could not read or undetectably tamper with the message. Today, encryption protects the privacy and authenticity of data in transit and stored data, prevents unauthorized access to computer resources, and more. Cryptography is commonly used by almost everyone, often unknowingly, as it is increasingly embedded into ubiquitous systems, such as automated bank teller machines, cellular telephones, and World Wide Web browsers.

I. INTRODUCTION

Cryptography probably dates back close to the beginnings of writing. One of the earliest known examples is the Caesar cipher, named for its purported use by Julius Caesar in ancient Rome. The Caesar cipher, which can not be considered secure today, replaced each letter of the alphabet with the letter occurring three positions later or 23 positions earlier in the alphabet: A becomes D, B becomes E, X becomes A, and so forth. A generalized version of the Caesar cipher is an alphabetic substitution cipher. As an example, a simple *substitution cipher* might use the following secret key to provide a substitution between characters of the original message and characters of the encrypted message.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
F R O A H I C W T Z X L U Y N K E B P M V G D S Q J
```

Using this key, a sample encrypted message is:

```
IEQ IBQDXMBQ FX RMBFQW MOWQB IEQ QLT IBQQ.
```

As is evident from even such a short example, this simple method does not disguise patterns in the text such as repeated letters and common combinations of letters. In fact, if the encrypted message is known to be English text, it is usually quite easy to determine the original message, even without the knowledge of the secret key, by using letter frequency analysis, guessing and checking, and maybe a little intuition and luck. Such substitution ciphers are commonly used today as puzzles in newspapers and puzzle books, but are not secure when used alone as cryptosystems. Polyalphabetic substitution ciphers, developed by Len Battista in 1568, improved on regular substitution ciphers by changing the substitution scheme partway through a message. Although substitution is not secure when used alone, it can be useful when used in conjunction with other techniques, and in fact, many cryptosystems used today benefit from substitution when it is carefully used as part of their encryption algorithms.

Another simple technique that is not secure alone, but can be secure when used as part of a cryptosystem, is *transposition* or *permutation*. A simple transposition cipher might rearrange the message

```
WE MEET AT DAWN IN THE MUSEUM,
```

by removing spaces, writing it in columns of letters whose length is determined by the key, and then reading across the columns

```
W E D I E E
E T A N M U
M A W T U M
E T N H S,
```

yielding the ciphertext

```
WEDIEE ETANMU MAWTUM ETNHS.
```

In this simple example, it is easy to see that an attacker can fairly easily determine the column length by seeing which pairs of letters are most likely to be adjacent in English words.

Used together, repeated combinations of substitution and transpositions can make the job of an attacker who is trying to break the system without knowing the key harder by more thoroughly obscuring the relationship between the plaintext and the ciphertext, requiring an attacker to explore many more possibilities. Many of the mechanical and electrical ciphers used in World Wars I and II, such as the Enigma rotor machine, relied on various combinations of substitutions and permutations.

Cryptanalysis is the process of trying to determine the plaintext of a ciphertext message without the decryption key, as well as possibly trying to determine the decryption key itself. Together, cryptography and cryptanalysis comprise the field of cryptology. The job of the cryptographer is to devise systems faster than the cryptanalysts can break them. Until early the 20th century, cryptanalysts generally had a clear upper hand over cryptographers, and most known ciphers or cryptosystems could easily be broken.

The transition to modern cryptography began with the invention of computers, and continued with the

development of ciphers such as the Data Encryption Standard (DES) and the exciting discovery of public key cryptography. The use of computers means more possibilities for ciphers, as sophisticated and lengthy computations that would have error-prone if done by hand and expensive if done by mechanical devices have now become possible.

A. Cryptosystems: A Mathematical Definition

Mathematically, a cryptosystem is defined as three algorithms—a (randomized) key generation algorithm *keyGen*, a (possibly randomized) encryption algorithm *Enc*, and a (usually deterministic) decryption algorithm *Dec*. More specifically, we define the following sets.

$$\mathbf{M} = \text{set of plaintext messages}$$

$$\mathbf{C} = \text{set of encrypted messages}$$

$$\mathbf{K_E} = \text{set of encryption keys}$$

$$\mathbf{K_D} = \text{set of decryption keys}$$

$$\mathbf{R} = \text{set of random values for encryption}$$

$$\mathbf{R}' = \text{set of random values for key generation.}$$

M is called the *message space*, and $\mathbf{K_E} \cup \mathbf{K_D}$ is called the *key space*. In computerized algorithms, the key space and message space are typically sets of all bit strings of particular lengths. As a notational convention, we denote sets by boldface letters and elements of a set by the same letter in italic typeface. The functions *KeyGen*, *Enc*, and *Dec* are defined as follows.

$$\begin{aligned} &\mathit{KeyGen}: \mathbf{R}' \rightarrow \mathbf{K_E} \times \mathbf{K_D} \\ &\mathit{Enc}: \quad \mathbf{M} \times \mathbf{K_E} \times \mathbf{R} \rightarrow \mathbf{C} \\ &\mathit{Dec}: \quad \mathbf{C} \times \mathbf{K_D} \rightarrow \mathbf{M}, \end{aligned}$$

such that for every $r \in \mathbf{R}$ and $r' \in \mathbf{R}'$,

$$\mathit{Dec}(\mathit{Enc}(M, K_E, r), K_D) = M,$$

where $\mathit{KeyGen}(r') = (K_E, K_D)$. We usually suppress explicit mention of the randomization used by *Enc*, and instead write $\mathit{Enc}(M, K_E)$ to donate $\mathit{Enc}(M, K_E, r)$, where r is chosen randomly as specified by an algorithmic description of *Enc*.

Often, $K_E = K_D$; that is, the encryption and decryption keys are identical, and in this case we refer to it simply as the key or the secret key. This is called *symmetric* or *secret key* cryptography. In contrast, in *asymmetric* or *public key* cryptography, the encryption keys and decryption keys are different from each other, and only the decryption key needs to be kept secret. In public key cryptography, the decryption key is also sometimes called the *secret key.*

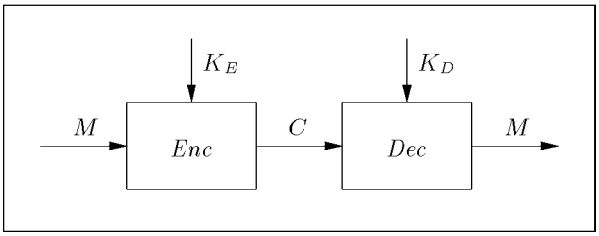


FIGURE 1 Diagram of a cryptosystem.

In order for a cryptosystem to be useful, all three functions *KeyGen*, *Enc*, and *Dec* must be efficiently computable. Since key generation is generally done only infrequently and can often be done in advance, it is acceptable for *KeyGen* to be somewhat less efficient, perhaps even taking many minutes to compute. In contrast, encryption and decryption are usually done more frequently and in real time, so *Enc* and *Dec* should be more efficient, measuring on the order of milliseconds or less.

We would also like additional requirements to capture the security of the cryptosystem—for example, that it is difficult to determine any information about K_D or M from $\mathit{Enc}(M, K_E,)$ alone. However, the specific meaning of this requirement depends the computational power available to an attacker, the abilities of the attacker to learn the encryptions of various messages, and other such factors, so there is not one single definition that can capture security in all settings. A rather strong, and desirable, notion of security is that of *semantic security*: from the ciphertext only, it should be computationally infeasible to learn *anything* about the plaintext except its length. The ciphertext should not reveal, for example, any of the bits of the ciphertext, nor should it suggest that some plaintext messages are more probably than others. Semantic security is much stronger than simply stating that an attacker does not learn the plaintext.

To see the importance of the key generation function in a cryptosystem, consider again the Caesar cipher presented previously. This can be thought of as encryption function that rotates characters in the alphabet according to the key, with a key generation function that always chooses the key 3. It is intuitively easy to see that an attacker who knows that the key is always 3, or infers it by seeing a number of plaintext/ciphertext pairs, clearly has an advantage over an attacker in a system where the key is chosen randomly from 1 to 26. In a system with a large key space, the key generation function can help to formally express the security of the cryptosystem by quantifying the *a priori* uncertainty the attacker has about the decryption key. In some implemented systems, key generation is left to the user, but this can be problematic because users are a bad source of randomness, and therefore this effectively reduces the size of the key space.

B. Goals of Cryptosystems: What Cryptography Can and Cannot Provide

Cryptography can be used to provide a variety of security-related properties. We will use the term "message" to refer either to a message or any other kind of data, either in transit or stored. Cryptography is often used to provide the following important properties.

Confidentiality: protects the contents of data from being read by unauthorized parties.

Authentication: allows the recipient of a message to positively determine the identity of the sender.

Integrity: ensures the recipient that a message has not been altered from its original contents.

Nonrepudiation: allows the recipient of a message to prove to a third party that the sender sent the message.

There are a number of additional security-related properties that cryptography does not directly provide, but for which cryptography can be part of a solution. These include anonymous communication, in which the receiver of a message is prevented from learning the identity of the sender; fair exchange, in which Alice should receive a valid message from Bob if and only if Bob receives a valid message from Alice; privacy from spam (unwanted bulk electronic mail); preventing the recipient of a message from further distributing the message; and protection against message traffic analysis.

Although cryptography is an important tool in securing computer systems, it alone is not sufficient. Even if a strong cryptosystem is used to authenticate users of a computer system before allowing them access, this authentication procedure can be easily subverted if there are other ways for attackers to access the system, whether through mistakenly installed, poorly configured, or just plain buggy software.

II. ATTACKS ON CRYPTOGRAPHIC SYSTEMS

It is generally assumed that an attacker on a cryptographic system knows everything about the specification of the system: that is, the key generation function, the encryption function, and the decryption function. Thus, the security of the system depends only on the fact that the attacker does not know the key, and on the degree to which the ciphertexts produced by the system hide the keys and plaintexts that were used. This is called Kerckhoff's principle, named for Auguste Kerckhoff who advocated it in a book he wrote in 1883. While the attacker may not in fact always know this much information, it is a conservative assumption and avoids "security by obscurity," or basing security on the assumption that the attacker does not know the encryption function. Captured equipment or documentation have frequently played a role in the breaking of military ciphers. In the industrial world, disgruntled employees can often be induced to reveal which ciphers their employers' systems use. If a cryptosystem is implemented in widely distributed software, the algorithm will almost certainly be discovered by reverse engineering, as happened with the RC4 cryptosystem, or leaked, as happened with the A5 cryptosystem. In practice, security by obscurity often turns out not to be security at all because cryptosystems that have not withstood public scrutiny are far more likely to have flaws in their design than those that were heavily scrutinized. Due to these factors, security by obscurity is generally frowned on by the scientific community.

In addition to knowing the specification of the system, an attacker may also know some additional information, such as what kinds of messages will be encrypted. The attacker may also have access to some pairs of plaintext and their corresponding ciphertexts, possibly chosen by the attacker.

A. Types of Attacks and Attackers

There are a number of types of attacks and attackers. One measurement of an attack is how much the attacker is able to learn, described here from weakest to strongest as categorized by Lars Knudsen.

Information deduction: The attacker learns partial information about the plaintext of an intercepted ciphertext or about the secret key.

Instance deduction: The attacker learns the plaintext of an intercepted ciphertext.

Global deduction: The attacker discovers an alternate algorithm for deciphering ciphertexts without the secret key.

Total break: The attacker learns the secret key.

In both global deductions and total breaks, the attacker can then decrypt any ciphertext.

The attacker may have different messages available to analyze in mounting an attack, again described from weakest to strongest.

Ciphertext-only attack: the attacker has access to a number of ciphertexts.

Known-plaintext attack: the attacker has access to a number of plaintext/ciphertext pairs.

Chosen-plaintext attack: the attacker can choose a number of plaintexts and learn their ciphertexts.

Adaptive chosen-plaintext attack: a chosen-plaintext attack in which the attacker can choose which

plaintext message to see the ciphertext of next based on all the messages he has seen so far.

Chosen-ciphertext attack: the attacker can choose a number of ciphertexts and learn their plaintexts.

Adaptive chosen-ciphertext attack: a chosen-ciphertext attack in which the attacker can choose which ciphertext message to see the plaintext of next based on all the messages he or she has seen so far.

In the types of attacks just described, it is usually assumed that all the ciphertexts were generated with the same encryption key. In addition, some cryptosystems are susceptible to related-message and related-key attacks, in which the attacker has access to ciphertexts or plaintext/ciphertext pairs for keys or plaintext messages with certain known relationships.

One measure of the practicality of an attack is the number and type of ciphertexts or plaintext/ciphertext pairs it requires. Other measures include the computational complexity, also called the *work factor*, and the storage requirements of the attack.

In the case that encryption is being used to provide properties other than just secrecy, there are additional types of attacks. For example, if encryption is being used to provide authentication and integrity through digital signatures, an attacker may attempt to forge signatures. As above, successful attacks can range from an existential forgery, in which one signed message is forged, to a total break, and attacks can use any number and type of signed messages.

Cryptanalysis describes attacks that directly attack the cryptosystem itself. The main two classes of cryptanalytic attacks, described below, are brute force attacks and structural attacks. We also describe some non-cryptanalytic attacks.

B. Brute Force Attacks

Brute force attacks are ciphertext-only attacks or known-plaintext attacks in which the decryption algorithm is used as a "black box" to try decrypting a given ciphertext with all possible keys until, in the case of a ciphertext-only attack, a meaningful message is found (if here is a way to determine in the context under attack whether a message is "meaningful"), or in the case of known-plaintext attacks, until the ciphertext decrypts to the corresponding plaintext. On average, a brute force will have to check half the key space before finding the correct key. While such attacks are exponential in the length of the key, they can be successfully carried out if the key space is small enough.

C. Structural Attacks

Brute force attacks simply use then encryption algorithm as a black box. It is often possible to mount more efficient attacks by exploiting the structure of the cipher. For example, in attacking the alphabetic substitution cipher, an efficient attacker makes use of the fact that different occurrences of the same letter is always substituted by the same substitute. More examples of structural attacks are given in the discussion of current cryptosystems.

D. Non-Cryptanalytic Attacks

When strong cryptographic systems with sufficiently long key lengths are used, brute force and other cryptanalytic attacks will not have a high probability of success. However, there a number of attacks that exploit the implementation and deployment of cryptosystems that can be tried instead and indeed are often successful in practice. These attacks usually have the goal of learning a user's secret key, though they also may be carried out only to learn a particular plaintext message. As in the case of cryptanalytic attacks, the attacker may or may not have access to plaintext/ciphertext pairs.

1. Social Attacks

Social attacks describe a broad range of attacks that use social factors to learn a user's secret key. These range from attempting to guess a secret key chosen by a user by using information known about the user to calling a user on the telephone pretending to be a system administrator and asking to be given information that will allow the attacker to gain access to the user's private keys. Alternately, the target of a social attack can be the contents of a particular sensitive message, again by fooling the sender or recipient into divulging information about it. Bribery and coercion are also considered social attacks. The best defense against social attacks is a combination of user education and legal remedies.

2. System Attacks

System attacks are attacks in which an attacker attempts to gain access to stored secret keys or stored unencrypted documents by attacking through non-cryptographic means the computer systems on which they are stored. Common ways that this is done are:

- Exploiting known, publicized holes in common programs such as *sendmail* or World Wide Web browsers.
- Computer viruses (usually distributed through e-mail).
- Trojan horse programs (usually downloaded from the Web by unsuspecting users).

In many cases, there are existing and easily available tools to carry out these types of attacks.

Edward Felten and others have described a number of strong attacks that are partially system attacks and partially social attacks, in which they take advantage of certain features in the way systems such as Web browsers are designed, combined with expected user behavior.

The best defenses against system attacks are prevention, detection, and punishment, achieved by a combination of good system administration, good firewalls, user education, and legal remedies.

3. Timing Attacks

Timing attacks were publicized by Paul Kocher in 1996. They attack the implementation of cryptosystems by measuring observable differences in the timing of the algorithm based on the particular value of the key. They then use statistical methods to determine the bits of key by observing many operations using the same key. Timing attacks typically require a significant number of chosen ciphertexts.

Related attacks can use any measure of differences in the performance of the encryption and decryption functions such as power consumption and heat dissipation.

Timing attacks and related attacks can be protected against to some degree by "blinding" the devices performing encryption and decryption computations so that all computations have the same performance, regardless of the particular key and message being used. However, this can have a substantial performance cost, as it requires all computations to have worst-case performance. Such attacks can also be protected against by designing systems so that they will not act as an "oracle" by decrypting and returning all and any messages that come their way, thereby preventing an attacker from obtaining the necessary data to carry out the attack. However, this is not always possible without interfering with the purpose of the system.

III. DESIGN AND USE OF CRYPTOGRAPHIC SYSTEMS

A good cryptosystem should satisfy several properties. It must be efficient to perform encryption and decryption, the encryption and decryption algorithms should be easy to implement, and the system should be resistant to attacks. Earlier, Kerckhoff's principle was noted, it says that the security of a cryptosystem should depend only on the secrecy of the secret key.

A. Provable Versus Heuristic Security

In some cases, it is possible to actually prove that a cryptosystem is secure, usually relative to certain *hardness assumptions* about the difficulty of breaking some of its components. In other cases, the security of a cryptosystem is only *heuristic*: the system appears to be secure based on resistance to known types of attacks and scrutiny by experts, but no proof of security is known. While provable security is certainly desirable, most of today's cryptosystems are not in fact provably secure.

B. Confusion and Diffusion

Two important properties that can be used to help in guiding the design of a secure cryptosystem, identified by Claude Shannon in 1949, are *confusion* and *diffusion*. Confusion measures the complexity of the relationship between the key and the ciphertext. Diffusion measures the degree to which small changes in the plaintext have large changes in the ciphertext.

For example, substitution creates confusion, while transpositions create diffusion. While confusion alone can be enough for a strong cipher, diffusion and confusion are most effective and efficient when used in conjunction.

C. Modes of Encryption

A *block cipher* encrypts messages of a small, fixed size, such as 128 bits. A *stream cipher* operates on a message stream, encrypting a small unit of data, say a bit or byte, at a time. While stream ciphers can be designed from scratch, it is also possible to use a block cipher as a stream cipher, as we will see below.

To encrypt a large message or data file using a block cipher, it must first be broken into blocks. A mode of encryption describes how the block cipher is applied to different blocks, usually by applying some sort of feedback so that, for example, repeated occurrences of the same block within a message do not encrypt to the same ciphertext. The main modes of block cipher encryption are *electronic codebook mode (ECB)*, *cipher block chaining mode (CBC)*, *cipher feedback mode (CFB)*, *output feedback mode (OFB)*, and *counter mode (CTR).*

1. Electronic Codebook Mode (ECB)

In electronic codebook mode, the block cipher is applied independently (with the same key) to each block of a message. While this is the easiest mode to implement, it does not obscure the existence of repeated blocks of plaintext. Furthermore, if an attacker can learn known plaintext/ciphertext pairs, these will allow him or her to decrypt any additional occurrences of the same block. Additionally, the attacker can replay selected blocks of previously sent ciphertexts, and an attacker may be able to use collected ciphertext blocks to replace blocks of a

new ciphertext message and change its meaning. For example, a lucky attacker might be able to change an electronic funds transfer to a different amount or a different payee.

The other modes discussed as follows avoid this problem by incorporating some feedback that causes different occurrences of the same plaintext block to encrypt to different ciphertexts.

2. Cipher Block Chaining Mode (CBC)

In cipher block chaining mode, the plaintext of a block is combined with the ciphertext of the previous block via an exclusive or (xor) operation, and the result is encrypted. The result is the ciphertext of that block, and will also be used in the encryption of the following block. An initialization vector (IV) acts as the "previous ciphertext block" for the first plaintext block. The initialization vector can be made public (i.e., can be sent in the clear along with the ciphertext), but ideally should not be reused for encryption of different messages to avoid having the same ciphertext prefix for two messages with the same plaintext prefix.

Decryption reverses the process. The first block of ciphertext is decrypted and then xored with the initialization vector; the result is the first plaintext block. Subsequent ciphertext blocks are decrypted and then xored with the ciphertext of the previous block.

One concern in feedback modes is synchronization after transmission errors. Cipher block chaining is *self-synchronizing*: a transmission error in one block will result in an error in that block and the following block, but will not affect subsequent blocks.

Plaintext block chaining is also possible.

3. Cipher Feedback Mode (CFB)

Cipher feedback mode allows a block cipher with block size n bits to be used as a stream cipher with a data encryption unit of m bits, for any $m \leq n$.

In CFB mode, the block cipher operates on a register of n bits. The register is initially filled with an initialization vector. To encrypt m bits of data, the block cipher is used to encrypt the contents of the register, the leftmost m bits of the result are xored with the m bits of data, and the result is m bits of ciphertext. In addition, the register is shifted left by m bits, and those m ciphertext bits are inserted in the right-most m register bits to be used in processing the next m bits of plaintext.

Decryption reverses the process. The register initially contains the initialization vector. To decrypt m bits of ciphertext, the block cipher is used to encrypt the contents of the register, and the resulting leftmost m bits are xored with the m ciphertext bits to recover m plaintext bits. The m ciphertext bits are then shifted left into the register.

Note that the encryption function of the block cipher is used in encryption *and* decryption of CFB mode, and the decryption function of the block cipher is not used at all.

As in CBC mode, an initialization vector is needed to get things started, and can be made public. In CBC mode, however, the initialization vector *must* be unique for each message encrypted with the same key, or else an eavesdropper can recover the xor of the corresponding plaintexts.

A single transmission error in the ciphertext will cause an error in $n/m + 1$ blocks as the affected ciphertext block is shifted through the register, and then the system recovers.

4. Output Feedback Mode (OFB)

Output feedback mode is similar to CFB mode, except that instead of the leftmost m bits of the ciphertext being shifted left into the register, the leftmost m bits of the output of the block cipher are used. As in CBC mode, encryption proceeds by encrypting the contents of the register using the block cipher and xoring the leftmost m bits of the result with the current m plaintext bits. However, OFB mode introduces insecurity unless $m = n$. As with CFB mode, The initialization vector can be made public and must be unique for each message encrypted with the same key.

In OFB mode, the *key stream*—the sequences of m bits that will be xored with the plaintext (by the sender) or the ciphertext (by the receiver)—depend only on the initialization vector, not on the plaintext or the ciphertext. Hence OFB mode has the efficiency advantage that, provided the sender and receiver agree in advance about what initialization vector their next message will use, the key stream can be computed in advance, rather than having to be computed while a message is being encrypted or decrypted. Since xor is a much more efficient operation than most block ciphers, this can be a substantial gain in the time between the receipt of an encrypted message and its decryption.

In OFB mode, the key must be changed before the key stream repeats, on average after $2^m - 1$ bits are encrypted if $m = n$.

5. Counter Mode (CTR)

Counter mode is similar in structure to the feedback modes, CFB and OFB, except that the register contents are determined by a simple counter modulo 2^m, or some other method for generating unique registers for each application of the block cipher. As in OFB mode, the key stream can be computed in advance. Despite the apparent simplicity of CTR mode, it has been shown to be in some sense at least as secure as CBC mode.

IV. SYMMETRIC KEY CRYPTOGRAPHY

Traditionally, all cryptography was symmetric key cryptography. In a symmetric key cryptosystem, the encryption key K_E and the decryption key K_D are the same, denoted simply by K. The key K must be kept secret, and it is also important that an eavesdropper who sees repeated encryptions using the same key can not learn the key. The simple substitution cipher described earlier is an example of a symmetric key cryptosystem.

A. The One-Time Pad

Invented by Gilbert Vernam and Major Joseph Mauborgne in 1917, the one time pad is a provably secure cryptosystem. It is also *perfectly* secure, in the sense that the proof of security does not depend on *any* hardness assumptions. In the one-time pad, the message space M can be the set of all n-bit strings. The space of keys and ciphertexts are also the set of all n-bit strings. The key generation function chooses an n-bit key uniformly at random. Given a message M and a key K, the encryption $Enc(M, K) = M \oplus K$, the xor of M and K. The decryption of a ciphertext C is $Dec(C, K) = C \oplus K$. It follows that $Dec(Enc(M, K), K) = (M \oplus K) \oplus K = M \oplus (K \oplus K) = M$.

A given ciphertext C can correspond to *any* plaintext message M, specifically when the key is $K = M \oplus C$. Hence, since the K is random and is never reused, it is impossible to learn anything about M from C without the secret key. That is, the one-time pad is perfectly secure.

One-time pads are impractical in most settings because the secret key must be as long as the message that is to be sent and cannot be reused. If the key is reused for two different messages M and M', then the corresponding ciphertexts can be xored to learn $M \oplus M'$. If additional information is known about the plaintexts, such as they are English text, or that it is a bit string with a fixed header, this is usually sufficient to reveal M and M', which in turn also reveals the key K. Nonetheless, if it is possible to exchange a sufficiently long random key stream in advance, which can then be used to encrypt messages until it runs out, the one-time pad still can be useful.

B. Pseudorandom Number Generators

A pseudorandom number generator is a function that takes a short random *seed* and outputs a longer bit sequence that "appears random." To be cryptographically secure, the output of a pseudorandom number generator should be computationally indistinguishable from a random string. In particular, given a short prefix of the sequence, it should be computationally infeasible to predict the rest of the sequence without knowing the seed. Many so-called random number generators, such as those based on linear feedback shift registers (LFSR) or linear congruences, are not cryptographically secure, as it is possible to predict the sequence from a short prefix of the sequence. Despite the fact that LFSRs are not secure, a large number of stream ciphers have been developed using them. Most of these have themselves since been shown to be insecure.

The secure Blum–Blum–Shub generator, developed by Lenore Blum, Manuel Blum, and Michael Shub in 1986, is based on the believed computational difficulty of distinguishing quadratic residues modulo n from certain kinds of nonquadratic residues modulo n.

A cryptographically secure pseudorandom number generator can be used to make a stream cipher by using the seed as a key and treating the generator output as a long key for a pseudorandom one-time pad.

C. Data Encryption Standard (DES)

The Data Encryption Standard (DES) was issued by the United States National Bureau of Standards (NBS) in 1977 as a government standard to be used in the encryption of data. The DES algorithm is based on a cryptosystem called Lucifer that was proposed to the NBS by the IBM Corporation. The standard includes the data encryption algorithm itself, as well as instructions for its implementation in hardware and an order for U.S. federal government agencies to use it for protection of sensitive but non-classified information. Since its initial standardization, DES was heavily used for encryption both for government and non-government applications, both in the United States and elsewhere. Although its key length is short enough to now be susceptible to brute force attacks, DES lived a long lifetime and served its function well.

DES is a block cipher with a 56-bit key and 64-bit blocks. DES is an example of a *Feistel* cipher, so named for one of its designers, Horst Feistel. In the Feistel structure, encryption proceeds in rounds on an intermediate 64-bit result. Each round divides the intermediate result into a left half and a right half. The right half then undergoes a substitution followed by a permutation, chosen based on the key and the round. This output is xored with the left half to make the right half of the next intermediate result; the right half of the old intermediate result becomes the left half of the new intermediate result. In DES, there are 16 such rounds. An initial permutation of the plaintext is performed before the first round, and the output from the last round is transformed by the inverse permutation before the last round. This structure has the advantage that the encryption and decryption algorithms are almost identical, an important advantage for efficient hardware implementations.

As approved as a standard, DES can be used in any of ECB, CBC, CFB, and OFB modes of encryption.

1. Stronger Variants of DES

Although the key length of DES is now short enough to be susceptible to brute force attacks, there are variants of DES that effectively have a longer key length. It is important to note that even though these constructions are more resistant to brute force attacks, it is theoretically possible that they are more susceptible to structural attacks. However, no structural attacks are known on the two constructions presented here that are more efficient than structural attacks on DES itself.

a. Triple-DES. Triple-DES uses two DES encryption keys, K_1 and K_2. The triple-DES encryption C of a plaintext block M is

$$C = Enc(Dec(Enc(M, K_1), K_2), K_1),$$

where *Enc* and *Dec* denote regular DES encryption and decryption. Decryption is

$$M = Dec(Enc(Dec(C, K_1), K_2), K_1).$$

The reason for the encrypt/decrypt/encrypt pattern is for compatibility with regular DES: triple-DES with $K_1 = K_2$ is identical to regular DES. With independently chosen keys, triple-DES has an effective key length of 128 bits.

b. DESX. DESX, suggested by Ronald Rivest, has an effective key length of 184 bits, and is much more efficient that triple-DES because it only requires a single DES encryption. In DESX, K is a 56-bit key and K_1 and K_2 are 64-bit keys. The DESX encryption C of a plaintext block M is

$$C = K_2 \oplus Enc(K_1 \oplus M, K),$$

where *Enc* denotes regular DES encryption. Decryption is

$$M = K_1 \oplus Dec(K_2 \oplus C, K).$$

DES compatibility is obtained by taking $K_1 = K_2 = 0$. Joe Kilian and Phillip Rogaway proved that the DESX construction is sound, in that it is in fact more resistant to brute force attacks than DES.

2. Brute Force Attacks on DES

Even at the time of DES's standardization, there was some concern expressed about the relatively short key length, which was shortened from IBM's original proposal. Given a ciphertext/plaintext pair, or from several ciphertexts and a notion of a meaningful message, a brute force attack would on average need to try 2^{55} keys before finding the right key.

Since its introduction, a number of estimates have been given of the cost and speed of doing a brute force DES attack, and a handful of such attacks have actually been performed. Since computers tend to double in speed and halve in price every 18 months, both the cost and the time of these attacks has steadily declined. Furthermore, since DES key search is completely parallelizable, it is possible to find keys twice as fast by spending twice as much money.

When DES was standardized, Whitfield Diffie and Martin Hellman estimated that it would be possible to build a DES-cracking computer for $20 million that would crack a DES key in a day. In 1993, Michael Wiener designed on paper a special purpose brute force DES-cracking computer that he estimated could be built for $1 million and would crack an average DES key in about three and a half hours. In 1997, Wiener updated his analysis based on then-current computers, estimating that a $1 million machine would crack keys in 35 minutes.

In 1998, the Electronic Frontier Foundation (EFF) actually built a DES-cracking computer, at the cost of $200,000, consisting of an ordinary personal computer with a large array of custom-designed chips. It cracks a DES key in an average of four and a half days.

3. Differential and Linear Cryptanalysis of DES

a. Differential cryptanalysis. In 1990, Eli Biham and Adi Shamir introduced *differential cryptanalysis*, a chosen-plaintext attack for cryptanalyzing ciphers based on substitutions and permutations. Applied to DES, the attack is more efficient than brute force, but it is a largely theoretical attack because of the large number of chosen plaintexts required. As compared to brute force, which requires a single known plaintext/ciphertext pair and takes time 2^{55}, differential cryptanalysis requires 2^{36} chosen plaintext/ciphertext pairs and takes time 2^{37}.

Differential cryptanalysis operates by taking many pairs of plaintexts with fixed xor difference, and looking at the differences in the resulting ciphertext pairs. Based on these differences, probabilities are assigned to possible keys. As more pairs are analyzed, the probability concentrates around a smaller number of keys. One can continue until the single correct key emerges as the most probable, or stop early and perform a reduced brute force search on the remaining keys.

Since the publication of differential cryptanalysis, Don Coppersmith, one of DES's designers at IBM, revealed that the DES design team in fact knew about differential cryptanalysis, but had to keep it secret for reasons of national security. He also revealed that they chose the

specific substitution and permutation parameters of DES to provide as much resistance to differential cryptanalysis as possible.

b. Linear cryptanalysis. Linear cryptanalysis was invented by Mitsuru Matsui and Atsuhiro Yamagishi in 1992, and applied by Matsui to DES in 1993. Like differential cryptanalysis, linear cryptanalysis also requires a large number of plaintext/ciphertext pairs. Linear cryptanalysis uses plaintext/ciphertext pairs to generate a linear approximation to each round, that is, a function that approximates the key for each round as an xor of some of the rounds input bits and output bits. An approximation to DES can be obtained by combining the 16 1-round approximations. The more plaintext/ciphertext pairs that are used, the more accurate the approximation will be. With 2^{43} plaintext/ciphertext pairs, linear cryptanalysis requires time 2^{13} and has success probability .85 of recovering the key.

D. Advanced Encryption Standard

In 1997, the United States National Institute of Standards (NIST) began the process of finding a replacement for DES. The new advanced encryption standard (AES) would need to be an unpatented, publicly disclosed, symmetric key block cipher, operating on 128 bit blocks, and supporting key sizes of 128, 192, and 256 bits, large enough to resist brute force attacks well beyond the foreseeable future. Several candidates were submitted, and were considered for security, efficiency, and ease of implementation. Fifteen submitted algorithms from twelve countries were considered in the first round of the selection process, narrowed to five in the second round. On October 2, 2000, NIST announced that it had selected *Rijndael*, a block cipher developed by Belgian cryptographers Joan Daemen and Vincent Rijmen, as the proposed AES algorithm. Rijndael was chosen for its security, performance, efficiency, implementability, and flexibility. Before Rijndael can actually become the standard, it must first undergo a period of public review as a Draft Federal Information Processing Standard (FIPS) and then be officially approved by the United States Secretary of Commerce. This process is expected to be completed by the middle of 2001.

The Rijndael algorithm supports a variable key size and variable block size of 128, 192, or 256 bits, but the standard is expected to allow only block size 128, and key size 128, 192, or 256. Rijndael proceeds in rounds. For a 128-bit block, the total number of rounds performed is 10 if the key length is 128 bits, 12 if the key length is 192 bits, and 14 if the key length is 256 bits.

Unlike the Feistel structure of DES, Rijndael's rounds are divided into three "layers," in each of which each bit of an intermediate result is treated in the same way. (In contrast, the left half and the right half of intermediate results are treated differently in each round of a Fiestel cipher.) The layered structure is designed to resist linear and differential cryptanalysis, and consists of the following.

- Linear mixing layer: adds diffusion.
- non-Linear layer: adds confusion.
- Key addition layer: adds feedback between rounds by xoring a current round key with an intermediate encryption result.

The Rijndael algorithm considers bytes as elements of the finite field GF(2^8), represented as degree 8 polynomials. For example, the byte with decimal representation 105, or binary representation 01101001, is represented as

$$x^6 + x^5 + x^3 + 1.$$

The sum of two such elements is the polynomial obtained by summing the coefficients modulo 2. The product of two such elements is the multiplication of the polynomial modulo the irreducible polynomial

$$m(x) = x^8 + x^4 + x^3 + 1.$$

Let b be the block length in bits. Throughout the encryption process, a matrix of bytes containing containing a partial result is maintained. It is called the State, and has $b/32$ columns and four rows. The State initially consists of a block of plaintext, written "vertically" into the arrays, column by column. At the end, the ciphertext for the block will be taken by reading the State in the same order.

A diagram illustrating the structure of a round is given in Fig. 2. Each round (except the last) consists of four transformations on the State.

ByteSub: This is the non-linear layer. The ByteSub transformation consists of a non-linear byte substitution operating independently on each of the bytes of the State. The substitution is done using an "S-box" determined by taking for each element its inverse element in GF(2^8) followed by some additional algebraic operations. The resulting S-box satisfies several properties including invertibility, minimization of certain kinds of correlation to provide resistance to linear and differential cryptanalysis, and simplicity of description. In an implementation, the entire S-box can either be calculated once and stored as a table, or S-box transformations can be calculated as needed.

ShiftRow: Together, the ShiftRow transformation and the MixColumn operation are the linear mixing layer. The ShiftRow transformation cyclically shifts each rows of the State. For 128-bit blocks, the first row is not shifted, the second row is shifted left by 1 byte,

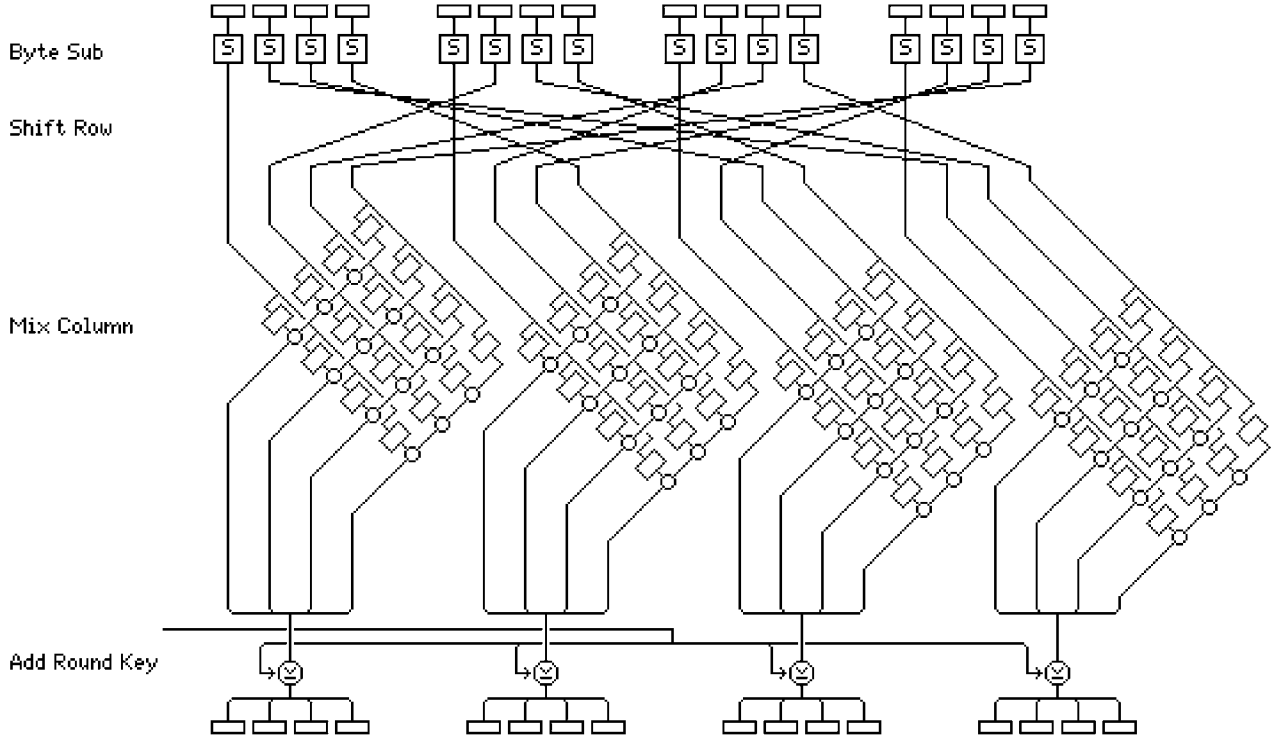


FIGURE 2 A round of Rijndael. (Illustration by John Savard.)

the third row is shifted by 2 bytes, and the last row shifted by 3 bytes.

MixColumn: In the MixColumn transformation, each column of the State is considered as the coefficients of a degree 4 polynomial over GF(2^8), which is then multiplied by the fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$ modulo $x^4 + 1$, where the sum and products of coefficients are as described above.

AddRoundKey: This is the key addition layer. In the AddRoundKey transformation, the State is transformed by being xored with the Round Key, which is obtained as follows. Initially, the encryption key is expanded into a longer key using a Key Expansion transformation. Each AddRoundKey uses the next b bits of the expanded key as its Round Key.

The final round eliminates the MixColumn transformation, consisting only of ByteSub, ShiftRow, and AddRoundKey. This is done to make decryption more structurally similar to encryption, which is desirable for ease of implementation. An additional AddRound-Key is performed before the first round.

Decryption in Rijndael consists of inverting each step. Due to the algebraic properties of Rijndael, the order of steps does not need to be changed for decryption. InvByteSub describes the application of the inverse S-box calculation. InvShiftRow shifts the rows right instead of left, by the same offsets as ShiftRow. InvMixColumn replaces the polynomial by its inverse in GF(2^8). Since xor is its own inverse, AddRoundKey remains the same, except that in decryption the Round keys must be used in the reverse order. Hence, Rijndael decryption starts with an AddRoundKey step, and then operates in rounds consisting of InvByteSub, InvShiftRow, InvMixColumn, and AddRoundKey, followed by a final round of InvByteSub, InvShiftRow, and InvMixColumn.

At the time of this writing, encryption modes for AES are being determined, and will probably consist of ECB, CBC, CFB, OFB, and counter modes, as well as possibly others.

V. PUBLIC KEY CRYPTOGRAPHY

The concept of public key cryptography was originally proposed in 1976 by Whitfield Diffie and Martin Hellman, and independently by Ralph Merkle. In 1997, Britain's Government Communications Headquarters (GCHQ) released previously classified documents revealing three British government employees, James Ellis, Clifford Cocks, and Malcolm Williamson, developed these same ideas several years earlier, but kept them secret for reasons of national security. There is some evidence that the United States' National Security Agency (NSA)

also secretly developed similar ideas as early as the 1960's.

In public key systems, the encryption key K_E, also called the *public key*, and the decryption key K_D, also called the *secret key*, are different from each other. Furthermore, it is computationally infeasible to determine K_D from K_E. Therefore, the encryption key can be made public. This has two advantages for key management. First, instead of each pair of users requiring a different secret key, as in the case of symmetric key cryptography (for a total of $n^2 - n$ encryption keys for n users), each user can have a single encryption key that all the other users can use to send her encrypted messages (for a total of n encryption keys for n users). Second, keys no longer need to be exchanged privately before encrypted messages can be sent.

Although the public keys in a public key cryptosystem need not be communicated secretly, they still must be communicated in an authenticated manner. Otherwise, an attacker Marvin could try convince Bob into accepting Marvin's public key in place of Alice's public key. If Marvin succeeds, then encrypted messages from Bob to Alice will actually be readable by Marvin instead of by Alice. If Marvin has sufficient control over the communication network, he can even prevent detection by Alice by intercepting the messages from Bob and then reencrypting the messages with Alice's real public key and sending them to her. In order to avoid such "man-in-the-middle" attacks, public keys are usually *certified* by being digitally signed by other entities.

Assuming that Alice and Bob already know each other's public keys, they can communicate privately as follows. To send her message M_A to Bob, Alice encrypts it with Bob's public key and sends the resulting ciphertext to Bob. Bob uses his private key to decrypt the ciphertext and obtain M_A. To send his response M_B to Alice, Bob encrypts it with Alice's public key and sends the resulting ciphertext to Alice. Alice uses her private key to decrypt the ciphertext and obtain M_B. An eavesdropper who overhears the ciphertexts does not learn anything about M_A and M_B because she does not have the necessary decryption keys.

The fundamental mathematical idea behind public key cryptosystems are *trapdoor one-way functions*. A function is one-way if it is hard to invert it: that is, given a value y it is computationally infeasible to find x such that $f(x) = y$. A one-way function is said to have the *trapdoor* property if given the trapdoor information, it becomes easy to invert the function. To use a trapdoor one-way function as a public key cryptosystem, the one-way function is used as the encryption algorithm, parametrized by its public key. The trapdoor information is the secret key. Trapdoor one-way functions are conjectured, but not proven, to exist. As such, all known public key cryptosystems are in fact based on widely believed but unproven assumptions. The famous and widely believed conjecture in theoretical computer science, $P \neq NP$, is a necessary but not sufficient condition for most of these assumptions to be true. It is an active area of research in public key cryptography to determine minimal assumptions on which public key cryptosystems can be based.

The earliest proposed public key systems were based on NP-complete problems such as the knapsack problem, but these were quickly found to be insecure. Some variants are still considered secure, but are not efficient enough to be practical. The most widely used public key cryptosystems, the RSA and El Gamal systems, are based on number theoretic and algebraic properties. Some newer systems are based on elliptic curves and lattices. Recently, Ronald Cramer and Victor Shoup developed a public key cryptosystem that is both practical and provably secure against adaptive chosen ciphertext attacks, the strongest kind of attack. The RSA system is described in detail below.

A. Using Public Key Cryptosystems

1. Hybrid Systems

Since public key cryptosystems are considerable less efficient than comparably secure symmetric key systems, public key cryptosystems are almost always used in conjunction with symmetric key systems, called *hybrid systems*. In a hybrid system, when Alice and Bob are initiating a new communication session, they first use a public key cryptosystem to authenticate each other and privately exchange a new symmetric key, called a session key. For the remainder of the session, they use the symmetric session key to encrypt their messages. Hybrid systems enjoy the key management benefits of public key cryptography, most of the efficiency benefits of symmetric key cryptography, as well as gaining additional security from the use of a frequently changing session key.

2. Probabilistic Encryption

It is crucial to the security of a public key cryptosystem that messages are somehow randomized before encryption. To see this, suppose that encryption were deterministic and that an attacker knows that an encrypted message sent to Alice is either "Yes" or "No," but wants to learn which. Since Alice's public key K_E is public, the attacker can simply compute Enc("Yes", K_E) and Enc("No", K_E) to see which one actually corresponds to the ciphertext.

To avoid this problem, Shafi Goldwasser and Silvio Micali introduced *probabilistic encryption*. In probabilistic encryption, the encryption function is randomized

rather than deterministic, and ciphertexts of one message should be computationally indistinguishable from ciphertexts of another message. The same plaintext will encrypt to different ciphertexts if different randomization is used, so an attacker who performs trial encryptions as above will only get the same result as the sender of a message if he also used the same randomness. Provably secure probabilistic encryption schemes have been developed based on the Blum–Blum–Shub pseudorandom number generator.

In practice, randomization is achieved by randomly padding the plaintext before encryption, which is then removed as part of decryption. However, this padding must be done carefully, as there have been attacks that successfully exploit padding.

3. Digital Signatures

Some public key cryptosystems can be used to digitally sign messages. Here, the private key is used to sign the message by applying the *Dec* function. The signature is appended to the plaintext message. Anyone can verify Alice's signature on a document by applying the *Enc* function with her public key to the signature part of the message and checking that the result matches the plaintext. If the public key system is secure, digital signatures provide the following.

- **Authenticity:** only Alice knows her private key and could create the signature.
- **Integrity:** if the plaintext is changed in transit, the signature will not match.
- **Non-repudiation:** a third party who knows Alice's public key can also verify that the signature is Alice's.

Unlike handwritten signatures, digital signatures are a function of the document they sign, so it is not possible to undetectably copy the signature from one document onto another.

If Alice and Bob wish to communicate using both encryption and digital signatures, Alice signs messages with her private signature key then encrypts with Bob's public encryption key. Bob decrypts with his private decryption key and then checks the digital signature using Alice's public signature verification key. If the encryption and decryption functions are commutative, as in the case with RSA, the same key pair could be used for both encryption and signatures. However, this is not recommended as it unnecessarily creates scenarios that provide chosen ciphertext and/or chosen plaintext to a potential attacker.

Since public key operations are expensive, often a hash function—a cryptographic compression function—is applied to a long message or file before it is signed. In this case, the signature consists of plaintext appended to the signature of the hash. The signature is verified by applying the hash to the plaintext and applying the decryption function to the signature, and checking whether these two results are identical.

Some public key systems, such as the Digital Signature Algorithm (DSA), can only be used for digital signatures, and not for encryption.

B. RSA

The first concrete public key system proposed with a security that has withstood the test of time was the RSA system, named for its inventors Ronald Rivest, Adi Shamir, and Leonard Adleman. RSA is based on the computational difficulty of factoring the product of large primes. The public key consists of an n-bit modulus N, which is the product of two primes p and q each of length $n/2$ bits, and an element e of the multiplicative group $\mathbb{Z}_n^*$. N is called the *RSA modulus* and e is called the encryption exponent. The decryption exponent is d such that $ed = 1 \bmod \phi(n)$, where $\phi(N) = (p-1)(q-1)$ is the Euler totient function. The private key is the pair $\langle N, d\rangle$. Once the public and private keys have been generated, p and q are no longer needed. They can be discarded, but should not be disclosed.

Based on the current state of the art for factoring and other attacks on RSA, current security recommendations as of 2001 usually stipulate that n should be 1024 bits, or 309 decimal digits. Although any efficient algorithm for factoring translates into an efficient attack on RSA, the reverse is not known to be true. Indeed, Boneh and Venkatesan have given some evidence that factoring may be harder than breaking RSA.

Suppose we have a message M that we want to encrypt, and further suppose that it has already been padded with appropriate random padding (which is necessary for security reasons) and represented as an element $m \in \mathbb{Z}_n^*$. If M is too large to represent in $\mathbb{Z}_n^*$, it must first be broken into blocks, each of which will be encrypted as described here. To encrypt the resulting message $m \in \mathbb{Z}_n^*$, it is raised to the eth power modulo N, that is, $Enc(m, \langle N, e\rangle) = m^e \bmod N$. Decryption is done by reversing the process: $Dec(c, \langle N, d\rangle) = c^d \bmod N$. Therefore, $Dec(Enc(m, \langle N, e\rangle), \langle N, d\rangle) = m^{ed} \bmod N = m$.

1. An RSA Example

We illustrate the use of RSA by an example. Let $p = 23$ and $q = 31$. While these values are much too small to produce a secure cryptosystem, they suffice to demonstrate the RSA algorithm. Then $n = pq = 713$ and

$$\phi(n) = \phi(713) = 22 \cdot 30 = 660.$$

The encryption exponent e must be chosen relatively prime to 660, say $e = 97$. The decryption exponent is $d = e^{-1} \bmod \phi(n) = 313$. It can be found using the extended euclidean algorithm.

To encrypt the plaintext message $m = 542$, we have

$$c = m^e \bmod 437 = 302.$$

Note that if we attempt to compute m^e as an integer first, and then reduce modulo n, the intermediate result will be quite large, even for such small values of m and e. For this reason, it is important for RSA implementations to use modular exponentiation algorithms that reduce partial results as they go and to use more efficient techniques such as squaring and multiply rather than iterated multiplications. Even with these improvements, modular exponentiation is still somewhat inefficient, particularly for the large moduli that security demands. To speed up encryption, the encryption exponent is often chosen to be of the form $2^k + 1$, to allow for the most efficient use of repeated squarings. To speed up decryption, the Chinese Remainder Theorem can be used provided p and q are remembered as part of the private key.

2. Choosing the RSA Parameters and Attacks Against RSA

Despite a number of interesting attacks on it that have been discovered over the years, it is still generally believed secure today provided certain guidelines are followed in its implementation: the keys are large enough, certain kinds of keys are avoided, and messages are randomly padded prior to encryption in a "safe" way.

The key generation function for RSA specfies how to generate N, e, and d. The usual method is to first choose p and q, compute $N = pq$, choose e, and compute $d = e^{-1} \bmod N$. There are several additional steps left to specify: how are p and q chosen, and how is e chosen. Both steps are influenced by the need to avoid certain attacks, which are described below. A careful choice of the RSA parameters proceeds as follows.

1. Choose the modulus size n large enough, and restrict p and q to be $n/2$ bits (to avoid factoring attacks).
2. Choose p and q to be "safe" primes of length $n/2$ (to avoid re-encryption attacks), and compute $N = pq$.
3. Choose e large enough (to avoid small public exponent attacks), either randomly or according to a specified calculation, and compute $d = e^{-1} \bmod N$.
4. If the resulting d is too small, go back to the previous step and choose a new e and compute a new d (to avoid small private exponent attacks).

a. Breaking RSA by factoring N. If an attacker can factor N to obtain p and q, then he or she can compute $\phi(N)$ and use the extended euclidean algorithm to compute the private exponent $d = e^{-1} mod \phi(N)$. Since it is easy for a brute force search algorithm to find small factors of any integer by trial division, it is clear that p and q should be taken of roughly equal size.

When RSA was first introduced, the continued fraction factoring algorithm could factor numbers up to about 50 digits (around 200 bits). Since that time, spurred by the application of breaking RSA as well as by the inherent mathematical interest, factoring has been a much studied problem. By 1990, the quadratic sieve factoring algorithm could routinley factor numbers around 100 digits, the record being a 116-digit number (385 bits). In 1994, the quadratic sieve algorithm factored a 129-digit number (428 bits), and in 1996, the number field sieve algorithm factored a 130-digit number (431 bits)in less than a quarter of the time the quadratic sieve would have taken. The general number field sieve algorithm is currently the fastest factoring algorithm, with a running time less than $e^{3n^{1/3}\log^{2/3} n}$, where n is the length in bits.

At the time of this writing, security experts usually recommend taking $n = 1024$ (or 309 decimal digits) for general use of RSA. This recommendation usually increases every five to ten years as computing technology improves and factoring algorithm become more sophisticated.

b. Re-encryption attacks and safe primes. Gus Simmons and Robert Morris describe a general attack that can be applied to any deterministic public key cryptosystem, or as a chosen plaintext attack on any deterministic symmetric key cryptosystem. Given a ciphertext C, an attacker should re-encrypt C under the same key, re-encrypt that results, and so forth, until the result is the original ciphertext C. Then the previous result must be the original plaintext M. The success of the attack is determined by the length of such cycles. Although public key systems should not be, and are not, generally used without randomization, it is still desirable to avoid small cycles. Rivest recommends the following procedure for choosing *safe* primes.

1. Select a random $n/2$-bit number. Call it r.
2. Test $2r + 1, 2r + 3, \ldots$ for primality until a prime is found. Call it p''.
3. Test $2p'' + 1, 4p'' + 1, \ldots$ for primality until a prime is found. Call it p''.
4. Test $2p' + 1, 4p' + 1, \ldots$ for primality until a prime is found. This is p.
5. Repeat steps 1–4 to find q.

c. Small public exponent attacks. In order to improve efficiency of encryption, it has been suggested to instead fix $e = 3$. However, certain attacks have been demonstrated when either e is too small. The most powerful of these attacks is based on lattice basis reduction and is due to Don Coppersmith. Coppersmith's attack is not a total break. However, if the public exponent is small enough and certain relationships between messages are known, it allows the attacker to succeed in learning the actual messages. If the encryption key is small enough and some bits of the decryption key are known, it allows the attacker to learn the complete decryption key. To avoid these attacks, it is important that the public exponent is chosen to be sufficiently large. It is still believed secure, and is desirable for efficiency reasons, to choose e to be of the form $2^k + 1$ for some $k \geq 16$.

d. Small private exponent attacks. An attack of Michael Wiener shows that if $d < (1/3)N^{1/4}$, than attacker can efficiently recover the private exponent d from the public key $< N, e)$. The attack is based on continued fraction approximations.

In addition to the attacks just described that relate to how the RSA parameters are chosen, there are also a number of attacks on RSA that relate to how RSA is used. As mentioned earlier, if the message space is small and no randomization is used, an attacker can learn the plaintext of a ciphertext C by encrypting each message in the message space and see which one gives the target ciphertext C. Some additional usage attacks on RSA are described below. RSA is also susceptible to timing attacks, described earlier.

e. Bleichenbacher's padding attack. Daniel Bleichenbacher showed a adaptive chosen-ciphertext attack on RSA as implemented in the PKCS1 standard, which uses the approach of appending random bits to a short message M before encrypting it to make it n bits long. In PKCS1, a padded message looks like this:

02	random pad	00	M

,

which is then encrypted using RSA. The recipient of the message decrypts it, checks that the structure is correct, and strips of the random pad. However, some applications using PKCS1 then responded with an "invalid ciphertext" message if the initial "02" was not present. Given a target ciphertext C, the attacker sends related ciphertexts of unknown plaintexts to the recipient, and waits to see if the response indicates that the plaintexts start with "02" or not. Bleichenbacher showed how this information can be used to learn the target ciphertext C.

This attack demonstrates that the way randomization is added to a message before encryption is very important.

f. Multiplication attacks. When used for signatures, the mathematical properties of exponentiation creates the possibilities for forgery. For example,

$$M_1^d \bmod N \cdot M_2^d \bmod N = (M_1 M_2)^d \bmod N,$$

so an attacker who sees the signature of M_1 and M_2 can compute the signature of $M_1 M^2$. Similarly, if the attacker wants to obtain Alice's signature on a message M that Alice is not willing to sign, he or she can try to "blind" it by producing a message that she would be willing to sign. To do this, the attacker chooses a random r and computes $M' = M \cdot r^e$. If Alice is willing to sign M', its signature is $M^d \cdot r \bmod N$, and the attacker divide by r to obtain the signature for M.

In practice, signatures are generated on hashes of messages, rather than the messages themselves, so this attack is not a problem. Furthermore, it is a useful property for allowing digital signatures where the signer does not learn the contents of a message, which can be useful in desiging systems that require both anonymity of participants and certification by a particular entity, such as anonymous digital cash systems and electronic voting systems.

g. Common modulus attacks. In a system with many users, a system administrator might try to use the same modulus for all users, and give each user their own encryption and decryption exponents. However, Alice can use the Chinese Remainder theorem together with her private key d to factor the modulus. Once she has done that, she can invert other users public exponents to learn their decryption exponents.

VI. KEY DISTRIBUTION AND MANAGEMENT

In order for encryption to protect the privacy of a message, it is crucial that the secret keys remain secret. Similarly, in order for a digital signature to protect the authenticity and integrity of a message, it is important that the signing key remains secret and that the public key is properly identified as the public key of the reputed sender. Therefore is it of paramount importance that the distribution and management of public and private keys be done securely.

Historically, keys were hand delived, either directly or through a trusted courier. When keys had to be changed, replacement keys were also hand delivered. However, this is often difficult or dangerous, for the very same reasons that motivated the need for encryption in the first place. While the initial key may need to be communicated by hand, it is desirable to use encryption to communicate

additional keys, rather than communicating them by hand. Method to do this are called key exchange protocols, and are described below.

With public key cryptography, some of the key management problems are solved. However, in order for Alice's public key to be useful, it is important that others know that it is her key, and not someone else masquerading as her for the purpose of receiving her secret messages. Hence, it is important that the binding between a public key and an identity is authenticated. Wide-scale methods for doing this are called public key infrastructures, and are also described below.

A. Key Exchange Protocols

The simplest key exchange protocol would be to use one secret key for a while, then use it to communicate a new secret key, and switch to that key. However, this is not a satisfactory solution because if one key is compromised (i.e., discovered by an attacker), then all future keys will be compromised as well. Instead, *session keys* are commonly used. A long-term key is exchanged securely (possibly by hand). A session key protocol is used to generate a short-term session key that will be used to encrypt messages for a period of time until the next time the session key protocol is run. Although exposure of the long-term key still results in compromise of all session keys, exposure of one session key does not reveal anything about past or future session keys. Long-term keys can be chosen to optimize security over efficiency, since they are only infrequently used, and long-term keys are less exposed because fewer messages are encrypted with them. Often the long-term key is the public key and private key of a public key cryptosystem, while the session keys are symmetric cryptosystem keys.

B. Diffie–Hellman Key Exchange

Diffie-Hellman key exchange is based on the assumed difficulty of the discrete logarithm problem modulo a prime number—that is, that it is difficult to compute z from $g^z \bmod p$. Diffie–Hellman allows to parties who have not previously exchanged any keys to agree on a secret key. Alice and Bob agree on a prime modulus p and a primitive element g. Alice picks a random number x and sends

$$a = g^x \bmod p$$

to Bob. Similarly, Bob picks a random number y and sends

$$b = g^y \bmod p$$

to Alice. Alice then computes $b^x \bmod p = g^{xy} \bmod p$ and Bob computes $a^y \bmod p = g^{xy} \bmod p$. The computed value $g^{xy} \bmod p$ is then used as a secret key.

Assuming that the discrete logarithm problem is computationally infeasible, an attacker overhearing the conversation between Alice and Bob can not learn $g^{xy} \bmod p$. However, it is subject to the kind of man-in-the-middle attack discussed earlier.

C. Key Distribution Centers

In a key distribution center (KDC) solution, a key distribution center shares a secret key with all participants and is trusted to communicate keys from one user to another. If Alice wants to exchange a key with Bob, she asks the KDC to choose a key for Alice and Bob to use and send it securely to each of them. While it may be possible to have such solutions within a particular business, they do not scale well to large systems or systems that cross administrative boundaries.

D. Public Key Infrastructures

In a public key infrastructure (PKI), any user Alice should be able to determine the public key of any other user Bob, and to be certain that it is really Bob's public key. This is done by having different entities digitally sign the pair: $\langle Bob, K_E \rangle$, consisting of Bob's identity and public key. In practice, a certificate will also contain other information, such as an expiration date, the algorithm used, and the identity of the signer. Now, Bob can present his certificate to Alice, and if she can verify the signature and trusts the signer to tell the truth, she knows K_E is Bob's public key. As with other key exchange solutions, this is simply moving the need for secrecy or authentication from one place to another, but can sometimes be useful.

The two main approaches to building a large-scale PKI are the hierarchical approach and the "web of trust" approach. In either model, a participant authenticates user/key bindings by determining one or more paths of certificates such that the user trusts the first entity in the path, certificates after the first are signed by the previous entity, and the final certificate contains the user/key binding in question. The difference between the two models is in the way trust is conveyed on the path.

In the hierarchical model, a certificate is signed by a certificate authority (CA). Besides a key binding, a CA certificate authorizes a role or privilege for the certified entity, by virtue of its status as an "authority" within its domain. For example, a company can certify its employees' keys because it hired those employees; a commercial certificate authority (CA) can certify its customer's keys because it generated them; a government or commercial CA can certify keys of hierarchically subordinate CAs by its powers of delegation; government agencies can certify keys of government agencies and licensed businesses, as empowered by law; and an international trade bureau can

certify government keys by international agreement. An individual is assumed to know and trust the key of a CA within its domain. A CA is assumed to know and trust the key of the CA who certifies its own keys, and it has a responsibility for accuracy when signing certificates of a principal in its domain. In summary, the hierarchical model conveys trust transitively, but only within a prescribed domain of control and authority.

In the web of trust model, individuals act as *introducers*, by certifying the keys of other individuals whom they have personally authenticated. In order for Alice to determine whether a key K_E belongs to Bob, she considers the signature(s) certifying the binding of K_E to Bob, and must ask whether any of the users who signed Bob certificates are considered trusted to verify and sign someone else's certificate. In other words, trust is not conveyed along the path of certificates, but rather it is awarded by the user of the certificate. Belief in the final certificate is possible only if the user trusts all of the certifying users on a path.

VII. APPLICATIONS OF CRYPTOGRAPHY

Cryptography has found a wide range of applications. Many cryptographic tools use cryptography to create building blocks that provide privacy, authentication, anonymity, and other such properties. In turn, these tools can be used to create secure applications for users. One strong and very general tool, called *secure multiparty computation*, allows a group of parties each holding a private input to jointly compute an output dependent on their private inputs without revealing their private inputs to each other. Secure multiparty computation can be used to solve problems like electronic voting, electronic auctions, and many other such problems.

Cryptography and cryptographic tools are particularly important for providing security in communications networks and on computer systems. Link encryption, which encrypts along a single link of a communication network, and end-to-end encryption, which encrypts all the way from the start to the end of a path in a communication network, are both used to protect the privacy of messages in transit. In computer systems, cryptography can be used to provide access control and prevent unwanted intruders from reading files, changing files, or accessing other resources.

Cryptography can also be used to provide important security properties in electronic commerce. A now familiar example is the use of cryptography to authenticate a merchant and encrypt a credit card number when buying goods over the World Wide Web. Cryptography can also protect the provider of digital content such as music or video by ensuring that recipients cannot widely redistribute the content without being detected. In the future, more advanced applications of cryptography in electronic commerce may be seen, where credit cards are replaced by digital cash and automated agents securely participate in auctions on a user's behalf.

SEE ALSO THE FOLLOWING ARTICLES

COMPUTER ALGORITHMS • COMPUTER NETWORKS • COMPUTER VIRUSES • SOFTWARE RELIABILITY • WWW (WORLD-WIDE WEB)

BIBLIOGRAPHY

Boneh, D. (1999). "Twenty years of attacks on the RSA cryptosystem," *Notices Am. Math. Soc.* (*AMS*), **46**(2), 203–213.

Daemen, J., and Rijmen V. (2000). The Block Cipher Rijndael. In "Proceedings of Smart Card Research and Applications" (CARDIS '98), Louvain-la-Neuve, Belgium, September 14–16, 1998, Lecture Notes in Computer Science, Vol. 1820, Springer, 277–284.

Diffie, W., and Hellman, M. E. (1976). "New directions in cryptography," *IEEE Trans. Inf. Theory*, **IT-22**(6), 644–654.

(1998). "electronic frontier foundation," (ed. John Gilmore) Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design, O'Reilly & Associates.

Goldreich, Oded (1999). "Modern Cryptography, Probabilistic Proofs and Pseudo-randomness," Springer-Verlag.

Kahn, David (1967). "The Codebreakers: The Story of Secret Writing," Macmillan Co., New York, New York.

Menenez, Alfred J., Van Oorschot, Paul C., and Vanstone, Scott A. (1996), "Handbook of Applied Cryptography," CRC Press Series on Discrete Mathematics and Its Applications, CRC Press.

Rivest, R. L., Shamir A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems, *Comm. ACM*, February 1978, 120–126.

Schneier, Bruce (1996). "Applied Cryptography Second Edition: protocols, algorithms, and source code in C," John Wiley and Sons.

Shannon, C. E. (1949). "Communication theory of secrecy systems," *Bell Sys. Tech. J.* **1949,** 656–715.

Stinson, D. E. (1995). "Cryptography Theory and Practice," CRC Press, Boca Raton.

Data Mining and Knowledge Discovery

Sally I. McClean
University of Ulster

GLOSSARY

Association rules link the values of a group of attributes, or variables, with the value of a particular attribute of interest which is not included in the group.

Data mining process takes place in four main stages: Data Pre-processing, Exploratory Data Analysis, Data Selection, and Knowledge Discovery.

Data mining tools are software products; a growing number of such products are becoming commercially available. They may use just one approach (single paradigm), or they may employ a variety of different methods (multi-paradigm).

Deviation detection is carried out in order to discover Interestingness in the data. Deviations may be detected either for categorical or numerical data.

Interestingness is central to Data Mining where we are looking for new knowledge which is nontrivial. It allows the separation of novel and useful patterns from the mass of dull and trivial ones.

Knowledge discovery in databases (KDD) is the main objective in Data Mining. The two terms are often used synonymously, although some authors define Knowledge Discovery as being carried out at a higher level than Data Mining.

DATA MINING is the process by which computer programs are used to repeatedly search huge amounts of data, usually stored in a Database, looking for useful new patterns. The main developments that have led to the emergence of Data Mining have been in the increased volume of data now being collected and stored electronically, and an accompanying maturing of Database Technology. Such developments have meant that traditional Statistical Methods and Machine Learning Technologies have had to be extended to incorporate increased demands for fast and scaleable algorithms.

In recent years, Database Technology has developed increasingly more efficient methods for data processing and

data access. Simultaneously there has been a convergence between Machine Learning Methods and Database Technology to create value-added databases with an increased capability for intelligence. There has also been a convergence between Statistics and Database Technology.

I. DATA MINING AND KNOWLEDGE DISCOVERY

A. Background

The main developments that have led to the emergence of Data Mining as a promising new area for the discovery of knowledge have been in the increased amount of data now available, with an accompanying maturing of Database Technology. In recent years Database Technology has developed efficient methods for data processing and data access such as parallel and distributed computing, improved middleware tools, and Open Database Connectivity (ODBC) to facilitate access to multi-databases.

Various Data Mining products have now been developed and a growing number of such products are becoming commercially available. Increasingly, Data Mining systems are coming onto the market. Such systems ideally should provide an integrated environment for carrying out the whole Data Mining process thereby facilitating end-user Mining, carried out automatically, with an interactive user interface.

B. The Disciplines

Data Mining brings together the three disciplines of Machine Learning, Statistics, and Database Technology. In the Machine Learning field, many complex problems are now being tackled by the development of intelligent systems. These systems may combine Neural Networks, Genetic Algorithms, Fuzzy Logic systems, Case-Based Reasoning, and Expert Systems. Statistical Techniques have become well established as the basis for the study of Uncertainty. Statistics embraces a vast array of methods used to gather, process, and interpret quantitative data. Statistical Techniques may be employed to identify the key features of the data in order to explain phenomena, and to identify subsets of the data that are interesting by virtue of being significantly different from the rest. Statistics can also assist with prediction, by building a model from which some attribute values can be reliably predicted from others in the presence of uncertainty. Probability Theory is concerned with measuring the likelihood of events under uncertainty, and underpins much of Statistics. It may also be applied in new areas such as Bayesian Belief Networks, Evidence Theory, Fuzzy Logic systems and Rough Sets.

Database manipulation and access techniques are essential to efficient Data Mining; these include Data Visualization and Slice and Dice facilities. It is often the case that it is necessary to carry out a very large number of data manipulations of various types. This involves the use of a structured query language (SQL) to perform basic operations such as selecting, updating, deleting, and inserting data items. Data selection frequently involves complex conditions containing Boolean operators and statistical functions, which thus require to be supported by SQL. Also the ability to join two or more databases is a powerful feature that can provide opportunities for Knowledge Discovery.

C. Data Mining Objectives and Outcomes

Data Mining is concerned with the search for new knowledge in data. Such knowledge is usually obtained in the form of rules which were previously unknown to the user and may well prove useful in the future. These rules might take the form of specific rules induced by means of a rule induction algorithm or may be more general statistical rules such as those found in predictive modeling. The derivation of such rules is specified in terms of Data Mining tasks where typical tasks might involve classifying or clustering the data.

A highly desirable feature of Data Mining is that there be some high-level user interface that allows the end-user to specify problems and obtain results in as friendly as matter as possible. Although it is possible, and in fact common, for Data Mining to be carried out by an expert and the results then explained to the user, it is also highly desirable that the user be empowered to carry out his own Data Mining and draw his own conclusions from the new knowledge. An appropriate user interface is therefore of great importance.

Another secondary objective is the use of efficient data access and data processing methods. Since Data Mining is increasingly being applied to large and complex databases, we are rapidly approaching the situation where efficient methods become a *sine qua non*. Such methods include Distributed and Parallel Processing, the employment of Data Warehousing and accompanying technologies, and the use of Open Database Connectivity (ODBC) to facilitate access to multi-databases.

D. The Data Mining Process

The Data Mining process may be regarded as taking place in four main stages (Fig. 1):

- Data Pre-processing
- Exploratory Data analysis
- Data Selection
- Knowledge Discovery

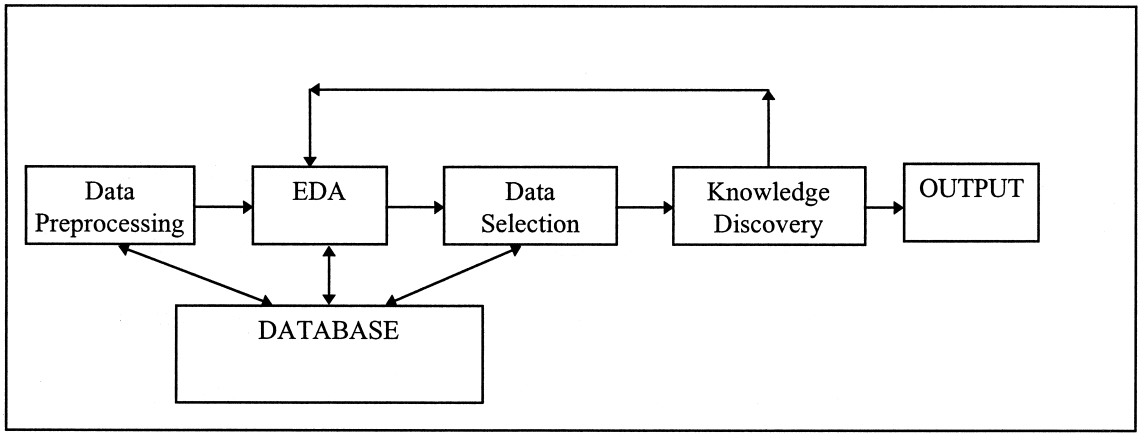


FIGURE 1 The Data Mining Process.

Data Pre-processing is concerned with data cleansing and reformatting, so that the data are now held in a form that is appropriate to the Mining algorithms and facilitates the use of efficient access methods. Reformatting typically involves employing missing value handling and presenting the data in multidimensional views suitable for the multidimensional servers used in Data Warehousing.

In Exploratory Data Analysis (EDA), the miner has a preliminary look at the data to determine which attributes and which technologies should be utilized. Typically, Summarization and Visualization Methods are used at this stage.

For Data Selection, we may choose to focus on certain attributes or groups of attributes since using all attributes at once is likely to be too complex and time consuming. Alternatively, for large amounts of data, we may choose to sample certain tuples, usually chosen at random. We may then carry out Knowledge Discovery using the sample, rather than the complete data, thus speeding up the process enormously. Variable reduction techniques or new variable definition are alternative methods for circumventing the problems caused by such large data sets.

Knowledge Discovery is the main objective in Data Mining and many different technologies have been employed in this context. In the Data Mining Process we frequently need to iterate round the EDA, Data Selection, Knowledge Discovery part of the process, as once we discover some new knowledge, we often then want to go back to the data and look for new or more detailed patterns.

Once new knowledge has been mined from the database, it is then reported to the user either in verbal, tabular or graphical format. Indeed the output from the Mining process might be an Expert System. Whatever form the output takes, it is frequently the case that such information is really the specification for a new system that will use the knowledge gained to best advantage for the user and domain in question. New knowledge may feed into the business process which in turn feeds back into the Data Mining process.

E. Data Mining Tasks

1. Rule Induction

Rule induction uses a number of specific beliefs in the form of database tuples as evidence to support a general belief that is consistent with these specific beliefs. A collection of tuples in the database may form a relation that is defined by the values of particular attributes, and relations in the database form the basis of rules. Evidence from within the database in support of a rule is thus used to induce a rule which may be generally applied.

Rules tend to be based on sets of attribute values, partitioned into an antecedent and a consequent. A typical "if then" rule, of the form "if antecedent = true, then consequent = true," is given by "if a male employee is aged over 50 and is in a management position, then he will hold an additional pension plan." Support for such a rule is based on the proportion of tuples in the database that have the specified attribute values in both the antecedent and the consequent. The degree of confidence in a rule is the proportion of those tuples that have the specified attribute values in the antecedent, which also have the specified attribute values in the consequent.

Rule induction must then be combined with rule selection in terms of interestingness if it is to be of real value in Data Mining. Rule-finding and evaluation typically require only standard database functionality, and they may be carried out using embedded SQL. Often, if a database is very large, it is possible to induce a very large number of rules. Some may merely correspond to well-known domain knowledge, whilst others may simply be of little interest to the user. Data Mining tools must therefore support the selection of **interesting** rules.

2. Classification

A commonly occurring task in Data Mining is that of classifying cases from a dataset into one of a number of well-defined categories. The categories are defined by sets of attribute values, and cases are allocated to categories according to the attribute values that they possess. The selected combinations of attribute values that define the classes represent **features** within the particular context of the classification problem. In the simplest cases, classification could be on a single binary-valued attribute, and the dataset is partitioned into two groups, namely, those cases with a particular property, and those without it. In general it may only be possible to say which class the case is "closest to," or to say how likely it is that the case is in a particular category.

Classification is often carried out by **supervised Machine Learning**, in which a number of training examples (tuples whose classification is known) are presented to the system. The system "learns" from these how to classify other cases in the database which are not in the training set. Such classification may be probabilistic in the sense that it is possible to provide the probability that a case is any one of the predefined categories. **Neural Networks** are one of the main Machine Learning technologies used to carry out classification. A probabilistic approach to classification may be adopted by the use of **discriminant functions**.

3. Clustering

In the previous section, the classification problem was considered to be essentially that of learning how to make decisions about assigning cases to known classes. There are, however, different forms of classification problem, which may be tackled by **unsupervised learning**, or clustering. Unsupervised classification is appropriate when the definitions of the classes, and perhaps even the number of classes, are not known in advance, e.g., market segmentation of customers into similar groups who can then be targeted separately.

One approach to the task of defining the classes is to identify clusters of cases. In general terms, **clusters** are groups of cases which are in some way similar to each other according to some measure of **similarity**. Clustering algorithms are usually iterative in nature, with an initial classification being modified progressively in terms of the class definitions. In this way, some class definitions are discarded, whilst new ones are formed, and others are modified, all with the objective of achieving an overall goal of separating the database tuples into a set of cohesive categories. As these categories are not predetermined, it is clear that clustering has much to offer in the process of Data Mining in terms of discovering **concepts**, possibly within a concept hierarchy.

4. Summarization

Summarization aims to present concise measures of the data both to assist in user comprehension of the underlying structures in the data and to provide the necessary inputs to further analysis. Summarization may take the form of the production of graphical representations such as bar charts, histograms, and plots, all of which facilitate a visual overview of the data, from which sufficient insight might be derived to both inspire and focus appropriate Data Mining activity. As well as assisting the analyst to focus on those areas in a large database that are worthy of detailed analysis, such visualization can be used to help with the analysis itself. Visualization can provide a "drill-down" and "drill-up" capability for repeated transition between summary data levels and detailed data exploration.

5. Pattern Recognition

Pattern recognition aims to classify objects of interest into one of a number of categories or classes. The objects of interest are referred to as **patterns**, and may range from printed characters and shapes in images to electronic waveforms and digital signals, in accordance with the data under consideration. Pattern recognition algorithms are designed to provide automatic identification of patterns, without the need for human intervention. Pattern recognition may be **supervised**, or **unsupervised**.

The relationships between the observations that describe a pattern and the classification of the pattern are used to design **decision rules** to assist the recognition process. The observations are often combined to form **features**, with the aim that the features, which are smaller in number than the observations, will be more reliable than the observations in forming the decision rules. Such **feature extraction** processes may be application dependent, or they may be general and mathematically based.

6. Discovery of Interestingness

The idea of interestingness is central to Data Mining where we are looking for new knowledge that is non-trivial. Since, typically, we may be dealing with very large amounts of data, the potential is enormous but so too is the capacity to be swamped with so many patterns and rules that it is impossible to make any sense out of them. It is the concept of interestingness that provides a framework for separating out the novel and useful patterns from the myriad of dull and trivial ones.

Interestingness may be defined as deviations from the norm for either categorical or numerical data. However, the initial thinking in this area was concerned with categorical data where we are essentially comparing the deviation between the proportion of our target group with

a particular property and the proportion of the whole population with the property. Association rules then determine where particular characteristics are related.

An alternative way of computing interestingness for such data comes from statistical considerations, where we say that a pattern is interesting if there is a statistically significant association between variables. In this case the measure of interestingness in the relationship between two variables A and B is computed as:

$$\text{Probability of } (A \text{ and } B)\text{-Probability of } (A)^{*} \text{ Probability of } (B).$$

Interestingness for continuous attributes is determined in much the same way, by looking at the deviation between summaries.

7. Predictive Modeling

In Predictive Modeling, we are concerning with using some attributes or patterns in the database to predict other attributes or extract rules. Often our concern is with trying to predict behavior at a future time point. Thus, for business applications, for example, we may seek to predict future sales from past experience.

Predictive Modeling is carried out using a variety of technologies, principally Neural Networks, Case-Based Reasoning, Rule Induction, and Statistical Modeling, usually via Regression Analysis. The two main types of predictive modeling are **transparent** (explanatory) and **opaque** (black box). A transparent model can give information to the user about why a particular prediction is being made, while an opaque model cannot explain itself in terms of the relevant attributes. Thus, for example, if we are making predictions using Case-Based Reasoning, we can explain a particular prediction in terms of similar behavior commonly occurring in the past. Similarly, if we are using a statistical model to predict, the forecast is obtained as a combination of known values which have been previously found to be highly relevant to the attribute being predicted. A Neural Network, on the other hand, often produces an opaque prediction which gives an answer to the user but no explanation as to why this value should be an accurate forecast. However, a Neural Network can give extremely accurate predictions and, where it may lack in explanatory power, it more than makes up for this deficit in terms of predictive power.

8. Visualization

Visualization Methods aim to present large and complex data sets using pictorial and other graphical representations. State-of-the-art Visualization techniques can thus assist in achieving Data Mining objectives by simplifying the presentation of information. Such approaches are often concerned with summarizing data in such a way as to facilitate comprehension and interpretation. It is important to have the facility to handle the commonly occurring situation in which it is the case that too much information is available for presentation for any sense to be made of it—the "haystack" view. The information extracted from Visualization may be an end in itself or, as is often the case, may be a precursor to using some of the other technologies commonly forming part of the Data Mining process.

Visual Data Mining allows users to interactively explore data using graphs, charts, or a variety of other interfaces. Proximity charts are now often used for browsing and selecting material; in such a chart, similar topics or related items are displayed as objects close together, so that a user can traverse a topic landscape when browsing or searching for information. These interfaces use colors, filters, and animation, and they allow a user to view data at different levels of detail. The data representations, the levels of detail and the magnification, are controlled by using mouse-clicks and slider-bars.

Recent developments involve the use of "virtual reality," where, for example, statistical objects or cases within a database may be represented by graphical objects on the screen. These objects may be designed to represent people, or products in a store, etc., and by clicking on them the user can find further information relating to that object.

9. Dependency Detection

The idea of dependency is closely related to interestingness and a relationship between two attributes may be thought to be interesting if they can be regarded as dependent, in some sense. Such patterns may take the form of statistical dependency or may manifest themselves as **functional dependency** in the database. With functional dependency, all values of one variable may be determined from another variable. However, statistical dependency is all we can expect from data which is essentially random in nature.

Another type of dependency is that which results from some sort of causal mechanism. Such causality is often represented in Data Mining by using Bayesian Belief Networks which discover and describe. Such causal models allow us to predict consequences, even when circumstances change. If a rule just describes an association, then we cannot be sure how robust or generalizable it will be in the face of changing circumstances.

10. Uncertainty Handling

Since real-world data are often subject to uncertainty of various kinds, we need ways of handling this uncertainty. The most well-known and commonly used way of

handling uncertainty is to use classical, or Bayesian, probability. This allows us to establish the probabilities, or support, for different rules and to rank them accordingly. One well-known example of the use of Bayesian probability is provided by the Bayesian Classifier which uses Bayes' Theorem as the basis of a classification method. The various approaches to handling uncertainty have different strengths and weaknesses that may make them particularly appropriate for particular Mining tasks and particular data sets.

11. Sequence Processing

Sequences of data, which measure values of the same attribute at a sequence of different points, occur commonly. The best-known form of such data arises when we collect information on an attribute at a sequence of time points, e.g., daily, quarterly, annually. However, we may instead have data that are collected at a sequence of different points in space, or at different depths or heights. Statistical data that are collected at a sequence of different points in time are known as time series.

In general, we are concerned with finding ways of describing the important features of a time series, thus allowing Predictive Modeling to be carried out over future time periods. There has also been a substantial amount of work done on describing the relationship between one time series and another with a view to determining if two time series co-vary or if one has a causal effect on the other. Such patterns are common in economic time series, where such variables are referred to as leading indicators. The determination of such leading indicators can provide new knowledge and, as such, is a fertile area for Data Mining.

The methods used for Predictive Modeling for the purpose of sequence processing are similar to those used for any other kind of Predictive Modeling, typically Rule Induction and Statistical Regression. However, there may be particular features of sequences, such as seasonality, which must be incorporated into the model if prediction is to be accurate.

F. Data Mining Approaches

As has already been stated, Data Mining is a multidisciplinary subject with major input from the disciplines of Machine Learning, Database Technology and Statistics but also involving substantial contributions from many other areas, including Information Theory, Pattern Recognition, and Signal Processing. This has led to many different approaches and a myriad of terminology where different communities have developed substantially different terms for essentially the same concepts. Nonetheless, there is much to gain from such an interdisciplinary approach and the synergy that is emerging from recent developments in the subject is one of its major strengths.

G. Advantages of Data Mining

Wherever techniques based on data acquisition, processing, analysis and reporting are of use, there is potential for Data Mining. The collection of consumer data is becoming increasingly automatic at the point of transaction. Automatically collected retail data provide an ideal arena for Data Mining. Highly refined customer profiling becomes possible as an integral part of the retail system, eschewing the need for costly human intervention or supervision. This approach holds the potential for discovering interesting or unusual patterns and trends in consumer behavior, with obvious implications for marketing strategies such as product placement, customized advertising, and rewarding customer loyalty. The banking and insurance industries have also well-developed and specialized data analysis techniques for customer profiling for the purpose of assessing credit worthiness and other risks associated with loans and investments. These include using Data Mining methods to adopt an integrated approach to mining criminal and financial data for fraud detection.

Science, technology, and medicine are all fields that offer exciting possibilities for Data Mining. Increasingly it is the vast arrays of automatically recorded experimental data that provide the material from which may be formed new scientific knowledge and theory. Data Mining can facilitate otherwise impossible Knowledge Discovery, where the amount of data required to be assimilated for the observation of a single significant anomaly would be overwhelming for manual analysis.

Both modern medical diagnosis and industrial process control are based on data provided by automated monitoring systems. In each case, there are potential benefits for efficiency, costs, quality, and consistency. In the Health Care environment, these may lead to enhanced patient care, while application to industrial processes and project management can provide a vital competitive advantage.

Overall, however, the major application area for Data Mining is still Business. For example a recent survey of Data Mining software tools (Fig. 2) showed that over three-quarters (80%) are used in business applications, primarily in areas such as finance, insurance, marketing and market segmentation. Around half of the vendor tools surveyed were suited to Data Mining in medicine and industrial applications, whilst a significant number are most useful in scientific and engineering fields.

II. THE TECHNOLOGIES

A. Machine Learning Technologies

1. Inferencing Rules

Machine Learning, in which the development of Inferencing Rules plays a major part, can be readily applied

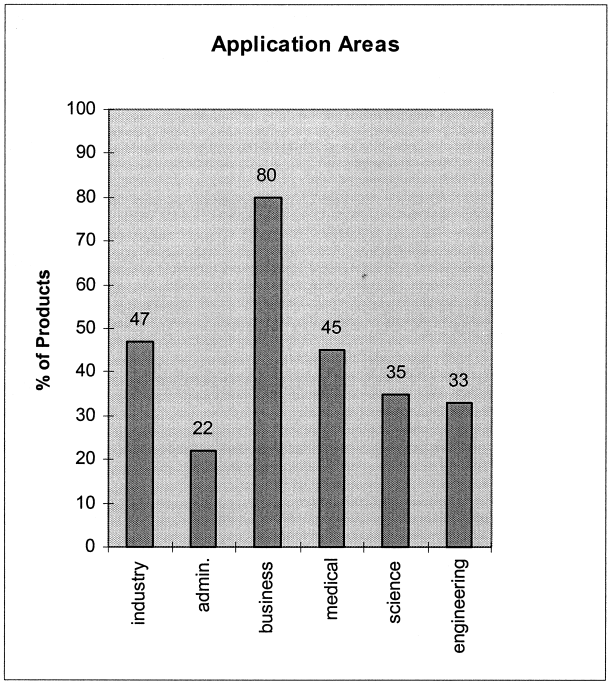


FIGURE 2 Application areas for Data Mining Tools.

to Knowledge Discovery in Databases. Records (tuples) in a database may be regarded as training instances that attribute-value learning systems may then use to discover patterns in a file of database records. Efficient techniques exist which can handle very large training sets, and include ways of dealing with incomplete and noisy data.

Logical reasoning may be automated by the use of a logic programming system, which contains a language for representing knowledge, and an **inference engine** for automated reasoning. The induction of logical definitions of relations has been named **Inductive Logic Programming (ILP)**, and may also be used to compress existing relations into their logical definitions. Inductive learning systems that use ILP construct logical definitions of target relations from examples and background knowledge. These are typically in the form of if–then rules, which are then transformed into clauses within the logic programming system. ILP systems have applications in a wide variety of domains, where Knowledge Discovery is achieved via the learning of relationships. Inference rules may be implemented as **demons**, which are processes running within a program while the program continues with its primary task.

2. Decision Trees

A Decision Tree provides a way of expressing knowledge used for classification. A Decision Tree is constructed by using a **training set** of cases that are described in terms of a collection of attributes. A sequence of tests is carried out on the attributes in order to partition the training set into ever-smaller subsets. The process terminates when each subset contains only cases belonging to a single class. Nodes in the Decision Tree correspond to these tests, and the leaves of the tree represent each subset. New cases (which are not in the training set) may be classified by tracing through the Decision Tree starting at its root and ending up at one of the leaves.

The choice of test at each stage in "growing" the tree is crucial to the tree's predictive capability. It is usual to use a selection criterion based on the gain in classification information and the information yielded by the test. In practice, when "growing" a Decision Tree, a small working set of cases is used initially to construct a tree. This tree is then used to classify the remaining cases in the training set: if all are correctly classified, then the tree is satisfactory. If there are misclassified cases, these are added to the working set, and a new tree constructed using this augmented working set. This process is used iteratively until a satisfactory Decision Tree is obtained. Overfitting of the data and an associated loss of predictive capability may be remedied by **pruning** the tree, a process that involves replacing sub-trees by leaves.

3. Neural Networks

Neural Networks are designed for pattern recognition, and they thus provide a useful class of Data Mining tools. They are primarily used for classification tasks. The Neural Network is first trained before it is used to attempt to identify classes in the data. Hence from the initial dataset a proportion of the data are partitioned into a **training set** which is kept separate from the remainder of the data. A further proportion of the dataset may also be separated off into a **validation set** that is used to test performance during training along with criteria for determining when the training should terminate, thus preventing **overtraining**.

A Neural Network is perhaps the simplest form of parallel computer, consisting of a (usually large) set of simple processing elements called **neurons**. The neurons are connected to one another in a chosen configuration to form a network. The types of connectivities or network architectures available can vary widely, depending on the application for which the Neural Network is to be used. The most straightforward arrangements consist of neurons set out in layers as in the **feedforward network**. Activity feeds from one layer of neurons to the next, starting at an initial input layer.

The **Universal Approximation Theorem** states that a single layer net, with a suitably large number of hidden nodes, can well approximate any suitably smooth function. Hence for a given input, the network output may be compared with the required output. The total mean square error function is then used to measure how close the actual

output is to the required output; this error is reduced by a technique called **back-error propagation**. This approach is a **supervised** method in which the network learns the connection weights as it is taught more examples. A different approach is that of **unsupervised learning**, where the network attempts to learn by finding statistical features of the input training data.

4. Case-Based Reasoning

Case-Based Reasoning (CBR) is used to solve problems by finding similar, past cases and adapting their solutions. By not requiring specialists to encapsulate their expertise in logical rules, CBR is well suited to domain experts who attempt to solve problems by recalling approaches which have been taken to similar situations in the past. This is most appropriate in domains which are not well understood, or where any rules that may have been devised have frequent exceptions. CBR thus offers a useful approach to building applications that support decisions based on past experience.

The quality of performance achieved by a case-based reasoner depends on a number of issues, including its experiences, and its capabilities to adapt, evaluate, and repair situations. First, partially matched cases must be retrieved to facilitate reasoning. The retrieval process consists of two steps: recalling previous cases, and selecting a best subset of them. The problem of retrieving applicable cases is referred to as the **indexing problem**. This comprises the matching or similarity-assessment problem, of recognizing that two cases are similar.

5. Genetic Algorithms

Genetic Algorithms (GA's) are loosely based on the biological principles of genetic variation and natural selection. They mimic the basic ideas of the evolution of life forms as they adapt to their local environments over many generations. Genetic Algorithms are a type of **evolutionary algorithm**, of which other types include Evolutionary Programming and Evolutionary Strategies.

After a new generation is produced, it may be combined with the population that spawned it to yield the new current population. The size of the new population may be curtailed by selection from this combination, or alternatively, the new generation may form the new population. The genetic operators used in the process of generating offspring may be examined by considering the contents of the population as a **gene pool**. Typically an individual may then be thought of in terms of **a binary string** of fixed length, often referred to as a **chromosome**. The genetic operators that define the offspring production process are usually a combination of **crossover** and **mutation** operators. Essentially these operators involve swapping part of the binary string of one parent with the corresponding part for the other parent, with variations depending on the particular part swapped and the position and order in which it is inserted into the remaining binary string of the other parent. Within each child, mutation then takes place.

6. Dynamic Time-Warping

Much of the data from which knowledge is discovered are of a temporal nature. Detecting patterns in sequences of time-dependent data, or time series, is an important aspect of Data Mining, and has applications in areas as diverse as financial analysis and astronomy. Dynamic Time-Warping (DTW) is a technique established in the recognition of patterns in speech, but may be more widely applied to other types of data.

DTW is based on a dynamic programming approach to aligning a selected template with a data time series so that a chosen measure of the distance between them, or error, is minimized. The measure of how well a template matches the time series may be obtained from the table of cumulative distances. A **warping path** is computed through the grid from one boundary point to another, tracing back in time through adjacent points with minimal cumulative distance. This warping path defines how well the template matches the time series, and a measure of the fit can be obtained from the cumulative distances along the path.

B. Statistical and other Uncertainty-Based Methods

1. Statistical Techniques

Statistics is a collection of methods of enquiry used to gather, process, or interpret quantitative data. The two main functions of Statistics are to describe and summarize data and to make inferences about a larger population of which the data are representative. These two areas are referred to as Descriptive and Inferential Statistics, respectively; both areas have an important part to play in Data Mining. Descriptive Statistics provides a toolkit of methods for data summarization while Inferential Statistics is more concerned with data analysis.

Much of Statistics is concerned with statistical analysis that is mainly founded on statistical inference or hypothesis testing. This involves having a Null Hypothesis (H_o): which is a statement of null effect, and an Alternative Hypothesis (H_1): which is a statement of effect. A test of significance allows us to decide which of the two hypotheses (H_o or H_1) we should accept. We say that a result is significant at the 5% level if the probability that the discrepancy between the actual data and what is expected assuming the null hypothesis is true has probability less that 0.05 of

occurring. The significance level therefore tells us where to **threshold** in order to decide if there is an effect or not.

Predictive Modeling is another Data Mining task that is addressed by Statistical methods. The most common type of predictive model used in Statistics is **linear regression**, where we describe one variable as a linear combination of other known variables. A number of other tasks that involve analysis of several variables for various purposes are categorized by statisticians under the umbrella term multivariate analysis.

Sequences of data, which measure values of the same attribute under a sequence of different circumstances, also occur commonly. The best-known form of such data arises when we collect information on an attribute at a sequence of time points, e.g., daily, quarterly, annually. For such time-series data the trend is modeled by fitting a regression line while fluctuations are described by mathematical functions. Irregular variations are difficult to model but worth trying to identify, as they may turn out to be of most interest.

Signal processing is used when there is a continuous measurement of some sort—the signal—usually distorted by noise. The more noise there is, the harder it is to extract the signal. However, by using methods such as filtering which remove all distortions, we may manage to recover much of the original data. Such filtering is often carried out by using **Fourier transforms** to modify the data accordingly. In practice, we may use **Fast Fourier Transforms** to achieve high-performance signal processing. An alternative method is provided by **Wavelets**.

All of Statistics is underpinned by classical or Bayesian probability. Bayesian Methods often form the basis of techniques for the automatic discovery of *classes* in data, known as **clustering** or **unsupervised learning**. In such situations Bayesian Methods may be used in computational techniques to determine the optimal set of classes from a given collection of unclassified instances. The aim is to find the most likely set of classes given the data and a set of prior expectations. A balance must be struck between data fitting and the potential for class membership prediction.

2. Bayesian Belief Networks

Bayesian Belief Networks are graphical models that communicate causal information and provide a framework for describing and evaluating probabilities when we have a network of interrelated variables. We can then use the graphical models to evaluate information about external interventions and hence predict the effect of such interventions. By exploiting the dependencies and interdependencies in the graph we can develop efficient algorithms that calculate probabilities of variables in graphs which are often very large and complex. Such a facility makes this technique suitable for Data Mining, where we are often trying to sift through large amounts of data looking for previously undiscovered relationships.

A key feature of Bayesian Belief Networks is that they discover and describe causality rather than merely identifying associations as is the case in standard Statistics and Database Technology. Such causal relationships are represented by means of **DAGs (Directed Acyclic Graphs)** that are also used to describe conditional independence assumptions. Such conditional independence occurs when two variables are independent, conditional on another variable.

3. Evidence Theory

Evidence Theory, of which Dempster–Shafer theory is a major constituent, is a generalization of traditional probability which allows us to better quantify uncertainty. The framework provides a means of representing data in the form of a **mass function** that quantifies our degree of belief in various propositions. One of the major advantages of Evidence Theory over conventional probability is that it provides a straightforward way of quantifying ignorance and is therefore a suitable framework for handling missing values.

We may use this Dempster–Shafer definition of mass functions to provide a lower and upper bound for the probability we assign to a particular proposition. These bounds are called the **belief** and **plausibility**, respectively. Such an interval representation of probability is thought to be a more intuitive and flexible way of expressing probability, since we may not be able to assign an exact value to it but instead give lower and upper bounds.

The Dempster–Shafer theory also allows us to transform the data by changing to a higher or lower granularity and reallocating the masses. If a rule can be generalized to a higher level of aggregation then it becomes a more powerful statement of how the domain behaves while, on the other hand, the rule may hold only at a lower level of granularity.

Another important advantage of Evidence Theory is that the Dempster–Shafer law of combination (the orthogonal sum) allows us to combine data from different independent sources. Thus, if we have the same frame of discernment for two mass functions which have been derived independently from different data, we may obtain a unified mass assignment.

4. Fuzzy Logic

Fuzzy logic maintains that all things are a matter of degree and challenges traditional two-valued logic which holds

that a proposition is either true or it is not. Fuzzy Logic is defined via a **membership function** that measures the degree to which a particular element is a member of a set. The membership function can take any value between 0 and 1 inclusive.

In common with a number of other Artificial Intelligence methods, fuzzy methods aim to simulate human decision making in uncertain and imprecise environments. We may thus use Fuzzy Logic to express expert opinions that are best described in such an imprecise manner. Fuzzy systems may therefore be specified using natural language which allows the expert to use vague and imprecise terminology. Fuzzy Logic has also seen a wide application to control theory in the last two decades.

An important use of fuzzy methods for Data Mining is for classification. Associations between inputs and outputs are known in fuzzy systems as **fuzzy associative memories** or **FAM**s. A FAM system encodes a collection of compound rules that associate multiple input statements with multiple output statements We combine such multiple statements using logical operators such as conjunction, disjunction and negation.

5. Rough Sets

Rough Sets were introduced by Pawlak in 1982 as a means of investigating structural relationships in data. The technique, which, unlike classical statistical methods, does not make probability assumptions, can provide new insights into data and is particularly suited to situations where we want to reason from qualitative or imprecise information. Rough Sets allow the development of **similarity measures** that take account of semantic as well as syntactic distance. Rough Set theory allows us to eliminate redundant or irrelevant attributes. The theory of Rough Sets has been successfully applied to knowledge acquisition, process control, medical diagnosis, expert systems and Data Mining. The first step in applying the method is to generalize the attributes using domain knowledge to identify the concept hierarchy. After generalization, the next step is to use reduction to generate a minimal subset of all the generalized attributes, called a **reduct**. A set of general rules may then be generated from the reduct that includes all the important patterns in the data. When more than one reduct is obtained, we may select the best according to some criteria. For example, we may choose the reduct that contains the smallest number of attributes.

6. Information Theory

The most important concept in Information Theory is **Shannon's Entropy**, which measures the amount of information held in data. Entropy quantifies to what extent the data are spread out over its possible values. Thus high entropy means that the data are spread out as much as possible while low entropy means that the data are nearly all concentrated on one value. If the entropy is low, therefore, we have high information content and are most likely to come up with a strong rule.

Information Theory has also been used as a measure of interestingness which allows us to take into account how often a rule occurs and how successful it is. This is carried out by using the **J-measure**, which measures the amount of information in a rule using Shannon Entropy and multiplies this by the probability of the rule coming into play. We may therefore rank the rules and only present the most interesting to the user.

C. Database Methods

1. Association Rules

An Association Rule associates the values of a given set of attributes with the value of another attribute from outside that set. In addition, the rule may contain information about the frequency with which the attribute values are associated with each other. For example, such a rule might say that "75% of men, between 50 and 55 years old, in management positions, take out additional pension plans."

Along with the Association Rule we have a **confidence threshold** and a **support threshold**. Confidence measures the ratio of the number of entities in the database with the designated values of the attributes in both A and B to the number with the designated values of the attributes in A. The support for the Association Rule is simply the proportion of entities within the whole database that take the designated values of the attributes in A and B.

Finding Association Rules can be computationally intensive, and essentially involves finding all of the covering attribute sets, A, and then testing whether the rule "A implies B," for some attribute set B separate from A, holds with sufficient confidence. Efficiency gains can be made by a combinatorial analysis of information gained from previous passes to eliminate unnecessary rules from the list of candidate rules. Another highly successful approach is to use **sampling** of the database to estimate whether or not an attribute set is covering. In a large data set it may be necessary to consider which rules are interesting to the user. An approach to this is to use **templates**, to describe the form of interesting rules.

2. Data Manipulation Techniques

For Data Mining purposes it is often necessary to use a large number of data manipulations of various types. When searching for Association Rules, for example, tuples

with certain attribute values are grouped together, a task that may require a sequence of conditional data selection operations. This task is followed by counting operations to determine the cardinality of the selected groups. The nature of the rules themselves may require further data manipulation operations such as summing or averaging of data values if the rules involve comparison of numerical attributes. Frequently knowledge is discovered by combining data from more than one source—knowledge which was not available from any one of the sources alone.

3. Slice and Dice

Slice and Dice refers to techniques specifically designed for examining cross sections of the data. Perhaps the most important aspect of Slice and Dice techniques is the facility to view cross sections of data that are not physically visible. Data may be sliced and diced to provide views in orthogonal planes, or at arbitrarily chosen viewing angles. Such techniques can be vital in facilitating the discovery of knowledge for medical diagnosis without the requirement for invasive surgery.

4. Access Methods

For Data Mining purposes it is often necessary to retrieve very large amounts of data from their stores. It is therefore important that access to data can be achieved rapidly and efficiently, which effectively means with a minimum number of input/output operations (I/Os) involving physical storage devices. Databases are stored on direct access media, referred to generally as **disks**. As disk access times are very much slower than main storage access times, acceptable database performance is achieved by adopting techniques whose objective is to arrange data on the disk in ways which permit stored records to be located in as few I/Os as possible.

It is valuable to identify tuples that are logically related, as these are likely to be frequently requested together. By locating two logically related tuples on the same page, they may both be accessed by a single physical I/O. Locating logically related tuples physically close together is referred to as clustering. Intra-file clustering may be appropriate if sequential access is frequently required to a set of tuples within a file; inter-file clustering may be used if sets of tuples from more than one file are frequently requested together.

D. Enabling Technologies

1. Data Cleansing Techniques

Before commencing Data Mining proper, we must first consider all data that is erroneous, irrelevant or atypical, which Statisticians term **outliers**.

Different types of outliers need to be treated in different ways. Outliers that have occurred as a result of human error may be detected by consistency checks (or integrity constraints). If outliers are a result of human ignorance, this may be handled by including information on changing definitions as **metadata**, which should be consulted when outlier tests are being carried out. Outliers of distribution are usually detected by outlier tests which are based on the deviation between the candidate observation and the average of all the data values.

2. Missing Value Handling

When we carry out Data Mining, we are often working with large, possibly heterogeneous data. It therefore frequently happens that some of the data values are missing because data were not recorded in that case or perhaps was represented in a way that is not compatible with the remainder of the data. Nonetheless, we need to be able to carry out the Data Mining process as best we can. A number of techniques have been developed which can be used in such circumstances, as follows:

- All tuples containing missing data are eliminated from the analysis.
- All missing values are eliminated from the analysis.
- A typical data value is selected at random and imputed to replace the missing value.

3. Advanced Database Technology

The latest breed of databases combines high performance with multidimensional data views and fast, optimized query execution. Traditional databases may be adapted to provide query optimization by utilizing Parallel Processing capabilities. Such **parallel databases** may be implemented on parallel hardware to produce a system that is both powerful and scaleable. Such postrelational Database Management Systems represent data through nested multidimensional tables that allow a more general view of the data of which the relational model is a special case. **Distributed Databases** allow the contributing heterogeneous databases to maintain local autonomy while being managed by a global data manager that presents a single data view to the user. **Multidimensional servers** support the multidimensional data view that represents multidimensional data through nested data. In the three-dimensional case, the data are stored in the form of a **data cube**, or in the case of many dimensions we use the general term **data hypercube**.

The Data Warehousing process involves assembling data from heterogeneous sources systematically by using **middleware** to provide connectivity. The data are

then cleansed to remove inaccuracies and inconsistencies and transformed to give a consistent view. **Metadata** that maintain information concerning the source data are also stored in the warehouse. Data within the warehouse is generally stored in a distributed manner so as to increase efficiency and, in fact, parts of the warehouse may be replicated at local sites, in **data marts**, to provide a facility for departmental decision-making.

4. Visualization Methods

Visualization Methods aim to present complex and voluminous data sets in pictorial and other graphical representations that facilitate understanding and provide insight into the underlying structures in the data. The subject is essentially interdisciplinary, encompassing statistical graphics, computer graphics, image processing, computer vision, interface design and cognitive psychology.

For exploratory Data Mining purposes, we require flexible and interactive visualization tools which allow us to look at the data in different ways and investigate different subsets of the data. We can highlight key features of the display by using color coding to represent particular data values. Charts that show relationships between individuals or objects within the dataset may be color-coded, and thus reveal interesting information about the structure and volume of the relationships. Animation may provide a useful way of exploring sequential data or time series by drawing attention to the changes between time points. **Linked windows**, which present the data in various ways and allow us to trace particular parts of the data from one window to another, may be particularly useful in tracking down interesting or unusual data.

5. Intelligent Agents

The potential of Intelligent Agents is increasingly having an impact on the marketplace. Such agents have the capability to form their own goals, to initiate action without instructions from the user and to offer assistance to the user without being asked. Such software has been likened to an intelligent personal assistant who works out what is needed by the boss and then does it. Intelligent Agents are essentially software tools that interoperate with other software to exchange information and services. They act as an intelligent layer between the user and the data, and facilitate tasks that serve to promote the user's overall goals. Communication with other software is achieved by exchanging messages in an **agent communication language**. Agents may be organized into a federation or agency where a number of agents interact to carry out different specialized tasks.

6. OLAP

The term OLAP (On-line Analytical Processing) originated in 1993 when Dr. E. F. Codd and colleagues developed the idea as a way of extending the relational database paradigm to support business modeling. This development took the form of a number of rules that were designed to facilitate fast and easy access to the relevant data for purposes of management information and decision support. An OLAP Database generally takes the form of a multidimensional server database that makes management information available interactively to the user. Such multidimensional views of the data are ideally suited to an analysis engine since they give maximum flexibility for such database operations as Slice and Dice or drill down which are essential for analytical processing.

7. Parallel Processing

High-performance parallel database systems are displacing traditional systems in very large databases that have complex and time-consuming querying and processing requirements. Relational queries are ideally suited to parallel execution since they often require processing of a number of different relations. In addition to parallelizing the data retrieval required for Data Mining, we may also parallelize the data processing that must be carried out to implement the various algorithms used to achieve the Mining tasks. Such processors may be designed to (1) share memory, (2) share disks, or (3) share nothing. Parallel Processing may be carried out using shared address space, which provides hardware support for efficient communication. The most scaleable paradigm, however, is to share nothing, since this reduces the overheads. In Data Mining, the implicitly parallel nature of most of the Mining tasks allows us to utilize processors which need only interact occasionally, with resulting efficiency in both speed-up and scalability.

8. Distributed Processing

Distributed databases allow local users to manage and access the data in the local databases while providing some sort of global data management which provides global users with a global view of the data. Such global views allow us to combine data from the different sources which may not previously have been integrated, thus providing the potential for new knowledge to be discovered. The constituent local databases may either be homogeneous and form part of a design which seeks to distribute data storage and processing to achieve greater efficiency, or they may be heterogeneous and form part of a legacy system where

the original databases might have been developed using different data models.

E. Relating the Technologies to the Tasks

Data Mining embraces a wealth of methods that are used in parts of the overall process of Knowledge Discovery in Databases. The particular Data Mining methods employed need to be matched to the user's requirements for the overall KDD process.

The tools for the efficient storage of and access to large datasets are provided by the Database Technologies. Recent advances in technologies for data storage have resulted in the availability of inexpensive high-capacity storage devices with very fast access. Other developments have yielded improved database management systems and Data Warehousing technologies. To facilitate all of the Data Mining Tasks, fast access methods can be combined with sophisticated data manipulation and Slice and Dice techniques for analysis of Data Warehouses through OLAP to achieve the intelligent extraction and management of information.

The general tasks of Data Mining are those of description and prediction. Descriptions of the data often require Summarization to provide concise accounts of some parts of the dataset that are of interest. Prediction involves using values of some attributes in the database to predict unknown values of other attributes of interest. Classification, Clustering, and Pattern Recognition are all Data Mining Tasks that can be carried out for the purpose of description, and together with Predictive Modeling and Sequence Processing can be used for the purpose of prediction. All of these descriptive and predictive tasks can be addressed by both Machine Learning Technologies such as Inferencing Rules, Neural Networks, Case-Based Reasoning, and Genetic Algorithms, or by a variety of Uncertainty Methods.

Data Mining methods are used to extract both patterns and models from the data. This involves modeling dependencies in the data. The model must specify both the structure of the dependencies (i.e., which attributes are inter-dependent) and their strengths. The tasks of Dependency Detection and modeling may involve the discovery of empirical laws and the inference of causal models from the data, as well as the use of Database Technologies such as Association Rules. These tasks can be addressed by Machine Learning Technologies such as Inferencing Rules, Neural Networks and Genetic Algorithms, or by Uncertainty Methods, including Statistical Techniques, Bayesian Belief Networks, Evidence Theory, Fuzzy Logic and Rough Sets.

The tasks of Visualization and Summarization play a central role in the successful discovery and analysis of patterns in the data. Both of these are essentially based on Statistical Techniques associated with Exploratory Data Analysis. The overall KDD process also encompasses the task of Uncertainty Handling. Real-world data are often subject to uncertainty of various kinds, including missing values, and a whole range of Uncertainty Methods may be used in different approaches to reasoning under uncertainty.

III. DATA MINING FOR DIFFERENT DATA TYPES

A. Web Mining and Personalization

Developments in Web Mining have been inexorably linked to developments in e-commerce. Such developments have accelerated as the Internet has become more efficient and more widely used. Mining of click streams and session log analysis allows a web server owner to extract new knowledge about users of the service, thus, in the case of e-commerce, facilitating more targeted marketing. Similarly, personalization of web pages as a result of Data Mining can lead to the provision of a more efficient service.

B. Distributed Data Mining

Recent developments have produced a convergence between computation and communication. Organizations that are geographically distributed need a decentralized approach to data storage and decision support. Thus the issues concerning modern organizations are not just the size of the database to be mined, but also its distributed nature. Such developments hold an obvious promise not only for what have become traditional Data Mining areas such as Database Marketing but also for newer areas such as e-Commerce and e-Business.

Trends in DDM are inevitably led by trends in Network Technology. The next generation Internet will connect sites at speeds of the order of 100 times faster than current connectivity. Such powerful connectivity to some extent accommodates the use of current algorithms and techniques. However, in addition, new algorithms, and languages are being developed to facilitate distributed data mining using current and next generation networks.

Rapidly evolving network technology, in conjunction with burgeoning services and information availability on the Internet is rapidly progressing to a situation where a large number of people will have fast, pervasive access to a huge amount of information that is widely accessible. Trends in Network Technology such as bandwidth developments, mobile devices, mobile users, intranets, information overload, and personalization leads to the conclusion that mobile code, and mobile agents, will, in the near future, be a critical part of Internet services. Such developments must be incorporated into Data Mining technology.

C. Text Mining

Text may be considered as sequential data, similar in this respect to data collected by observation systems. It is therefore appropriate for Data Mining techniques that have been developed for use specifically with sequential data to be also applied to the task of Text Mining. Traditionally, text has been analyzed using a variety of information retrieval methods, including natural language processing. Large collections of electronically stored text documents are becoming increasingly available to a variety of end-users, particularly via the World Wide Web. There is great diversity in the requirements of users: some need an overall view of a document collection to see what types of documents are present, what topics the documents are concerned with, and how the documents are related to one another. Other users require specific information or may be interested in the linguistic structures contained in the documents. In very many applications users are initially unsure of exactly what they are seeking, and may engage in browsing and searching activities.

General Data Mining methods are applicable to the tasks required for text analysis. Starting with textual data, the Knowledge Discovery Process provides information on commonly occurring phenomena in the text. For example, we may discover combinations of words or phrases that commonly appear together. Information of this type is presented using **episodes**, which contain such things as the base form of a word, grammatical features, and the position of a word in a sequence. We may measure, for example, the support for an episode by counting the number of occurrences of the episode within a given text sequence.

For Text Mining, a significant amount of pre-processing of the textual data may be required, dependent on the domain and the user's requirements. Some natural language analysis may be used to augment or replace some words by their parts of speech or by their base forms. Post-processing of the results of Text Mining is usually also necessary.

D. Temporal Data Mining

Temporal Data Mining often involves processing time series, typically sequences of data, which measure values of the same attribute at a sequence of different time points. Pattern matching using such data, where we are searching for particular patterns of interest, has attracted considerable interest in recent years. In addition to traditional statistical methods for time series analysis, more recent work on sequence processing has used association rules developed by the database community. In addition Temporal Data Mining may involve exploitation of efficient methods of data storage, fast processing and fast retrieval methods that have been developed for temporal databases.

E. Spatial Data Mining

Spatial Data Mining is inexorably linked to developments in Geographical Information Systems. Such systems store spatially referenced data. They allow the user to extract information on contiguous regions and investigate spatial patterns. Data Mining of such data must take account of spatial variables such as distance and direction. Although methods have been developed for Spatial Statistics, the area of Spatial Data Mining per se is still in its infancy. There is an urgent need for new methods that take spatial dependencies into account and exploit the vast spatial data sources that are accumulating. An example of such data is provided by remotely sensed data of images of the earth collected by satellites.

F. Multimedia Data Mining

Multimedia Data Mining involves processing of data from a variety of sources, principally text, images, sound, and video. Much effort has been devoted to the problems of indexing and retrieving data from such sources, since typically they are voluminous. A major activity in extracting knowledge from time-indexed multimedia data, e.g., sound and video, is the identification of episodes that represent particular types of activity; these may be identified in advance by the domain expert. Likewise domain knowledge in the form of metadata may be used to identify and extract relevant knowledge. Since multimedia contains data of different types, e.g., images along with sound, ways of combining such data must be developed. Such problems of Data Mining from multimedia data are, generally speaking, very difficult and, although some progress has been made, the area is still in its infancy.

G. Security and Privacy Aspects of Data Mining

As we have seen, Data Mining offers much as a means of providing a wealth of new knowledge for a huge range of applications. The knowledge thus obtained from databases may be far in excess of the use to which the data owners originally envisaged for the database. However, such data may include sensitive information about individuals or might involve company confidential information. Care must therefore be taken to ensure that only authorized personnel are permitted to access such databases. However, it may be possible to get around this problem of preserving the security of individual level data by using anonymization techniques and possibly only providing a sample of

the data for Mining purposes. In addition, it is often the case that, for purposes of Data Mining, we do not need to use individual level data but instead can utilize aggregates.

For Database Technology, intrusion detection models must be developed which protect the database against security breaches for the purpose of Data Mining. Such methods look for evidence of users running huge numbers of queries against the database, large volumes of data being downloaded by the user, or users running their own imported software on portions of the database.

H. Metadata Aspects of Data Mining

Currently, most data mining algorithms require bringing all together data to be mined in a single, centralized data warehouse. A fundamental challenge is to develop distributed versions of data mining algorithms so that data mining can be done while leaving some of the data in place. In addition, appropriate protocols, languages, and network services are required for mining distributed data to handle the mappings required for mining distributed data. Such functionality is typically provided via metadata.

XML (eXtensible Markup Language) is fast emerging as a standard for representing data on the World Wide Web. Traditional Database Engines may be used to process semistructured XML documents conforming to Data Type Definitions (DTDs). The XML files may be used to store metadata in a representation to facilitate the mining of multiple heterogeneous databases. **PMML** (predictive Mark-up Language) has been developed by the Data Mining community for the exchange of models between different data sites; typically these will be distributed over the Internet. Such tools support interoperability between heterogeneous databases thus facilitating Distributed Data Mining.

IV. KEY APPLICATION AREAS

A. Industry

Industrial users of databases are increasingly beginning to focus on the potential for embedded artificial intelligence within their development and manufacturing processes. Most industrial processes are now subject to technological control and monitoring, during which vast quantities of manufacturing data are generated. Data Mining techniques are also used extensively in process analysis in order to discover improvements which may be made to the process in terms of time scale and costs.

Classification techniques and rule induction methods are used directly for quality control in manufacturing. Parameter settings for the machinery may be monitored and evaluated so that decisions for automatic correction or intervention can be taken if necessary. Machine Learning technologies also provide the facility for failure diagnosis in the maintenance of industrial machinery.

Industrial safety applications are another area benefiting from the adoption of Data Mining technology. Materials and processes may need to be classified in terms of their industrial and environmental safety. This approach, as opposed to experimentation, is designed to reduce the cost and time scale of safe product development.

B. Administration

There is undoubtedly much scope for using Data Mining to find new knowledge in administrative systems that often contain large amounts of data. However, perhaps because the primary function of administrative systems is routine reporting, there has been less uptake of Data Mining to provide support and new knowledge for administrative purposes that in some other application areas.

Administrative systems that have received attention tend to be those in which new knowledge can be directly translated into saving money. An application of this type is provided by the Inland Revenue that collects vast amounts of data and may potentially save a lot of money by devising ways of discovering tax dodges, similarly for welfare frauds. Another successful application of Data Mining has been to the health care system where again new discoveries about expensive health care options can lead to huge savings. Data Mining is also likely to become an extremely useful tool in criminal investigations, searching for possible links with particular crimes or criminals.

C. Business

As we might expect, the major application area of Data Mining, so far, has been Business, particularly the areas of Marketing, Risk Assessment, and Fraud Detection.

In Marketing, perhaps the best known use of Data Mining is for customer profiling, both in terms of discovering what types of goods customers tend to purchase in the same transaction and groups of customers who all behave in a similar way and may be targeted as a group. Where customers tend to buy (unexpected) items together then goods may be placed on nearby shelves in the supermarket or beside each other in a catalogue. Where customers may be classified into groups, then they may be singled out for customized advertising, mail shots, etc. This is known as micro marketing. There are also cases where customers of one type of supplier unexpectedly turn out to be also customers of another type of supplier and advantage may be gained by pooling resources in some sense. This is known as cross marketing.

Another use of Data Mining for Business has been for Risk Assessment. Such assessment of credit worthiness of potential customers is an important aspect of this use which has found particular application to banking institutions where lending money to potentially risky customers is an important part of the day-to-day business. A related application has been to litigation assessment where a firm may wish to assess how likely and to what extent a bad debtor will pay up and if it is worth their while getting involved in unnecessary legal fees.

Case Study I. A supermarket chain with a large number of stores holds data on the shopping transactions and demographic profile of each customer's transactions in each store. Corporate management wants to use the customer databases to look for global and local shopping patterns.

D. Database Marketing

Database Marketing refers to the use of Data Mining techniques for the purposes of gaining business advantage. These include improving a company's knowledge of its customers in terms of their characteristics and purchasing habits and using this information to classify customers; predicting which products may be most usefully offered to a particular group of customers at a particular time; identifying which customers are most likely to respond to a mail shot about a particular product; identifying customer loyalty and disloyalty and thus improving the effectiveness of intervention to avoid customers moving to a competitor; identifying the product specifications that customers really want in order to improve the match between this and the products actually offered; identifying which products from different domains tend to be bought together in order to improve cross-marketing strategies; and detecting fraudulent activity by customers.

One of the major tasks of Data Mining in a commercial arena is that of market segmentation. Clustering techniques are used in order to partition a customer database into homogeneous segments characterized by customer needs, preferences, and expenditure. Once market segments have been established, classification techniques are used to assign customers and potential customers to particular classes. Based on these, prediction methods may be employed to forecast buying patterns for new customers.

E. Medicine

Potential applications of Data Mining to Medicine provide one of the most exciting developments and hold much promise for the future. The principal medical areas which have been subjected to a Data Mining approach, so far, may be categorized as: diagnosis, treatment, monitoring, and research.

The first step in treating a medical complaint is diagnosis, which usually involves carrying out various tests and observing signs and symptoms that relate to the possible diseases that the patient may be suffering from. This may involve clinical data, data concerning biochemical indicators, radiological data, sociodemographic data including family medical history, and so on. In addition, some of these data may be measured at a sequence of time-points, e.g., temperature, lipid levels. The basic problem of diagnosis may be regarded as one of classification of the patient into one, or more, possible disease classes.

Data Mining has tremendous potential as a tool for assessing various treatment regimes in an environment where there are a large number of attributes which measure the state of health of the patient, allied to many attributes and time sequences of attributes, representing particular treatment regimes. These are so complex and interrelated, e.g., the interactions between various drugs, that it is difficult for an individual to assess the various components particularly when the patient may be presenting with a variety of complaints (multi-pathology) and the treatment for one complaint might mitigate against another.

Perhaps the most exciting possibility for the application of Data Mining to medicine is in the area of medical research. Epidemiological studies often involve large numbers of subjects which have been followed-up over a considerable period of time. The relationship between variables is of considerable interest as a means of investigating possible causes of diseases and general health inequalities in the population.

Case Study II. A drug manufacturing company is studying the risk factors for heart disease. It has data on the results of blood analyses, socioeconomic data, and dietary patterns. The company wants to find out the relationship between the heart disease markers in the blood and the other relevant attributes.

F. Science

In many areas of science, automatic sensing and recording devices are responsible for gathering vast quantities of data. In the case of data collected by remote sensing from satellites in disciplines such as astronomy and geology the amount of data are so great that Data Mining techniques offer the only viable way forward for scientific analysis.

One of the principal application areas of Data Mining is that of space exploration and research. Satellites provide immense quantities of data on a continuous basis via remote sensing devices, for which intelligent, trainable image-analysis tools are being developed. In previous large-scale studies of the sky, only relatively small amount of the data collected have actually been used in manual attempts to classify objects and produce galaxy catalogs.

Not only has the sheer amount of data been overwhelming for human consideration, but also the amount of data required to be assimilated for the observation of a single significant anomaly is a major barrier to purely manual analysis. Thus Machine Learning techniques are essential for the classification of features from satellite pictures, and they have already been used in studies for the discovery of quasars. Other applications include the classification of landscape features, such as the identification of volcanoes on the surface of Venus from radar images. Pattern recognition and rule discovery also have important applications in the chemical and biomedical sciences. Finding patterns in molecular structures can facilitate the development of new compounds, and help to predict their chemical properties. There are currently major projects engaged in collecting data on the human gene pool, and rule-learning has many applications in the biomedical sciences. These include finding rules relating drug structure to activity for diseases such as Alzheimer's disease, learning rules for predicting protein structures, and discovering rules for the use of enzymes in cancer research.

Case Study III. An astronomy cataloguer wants to process telescope images, identify stellar objects of interest and place their descriptions into a database for future use.

G. Engineering

Machine Learning has an increasing role in a number of areas of engineering, ranging from engineering design to project planning. The modern engineering design process is heavily dependent on computer-aided methodologies. Engineering structures are extensively tested during the development stage using computational models to provide information on stress fields, displacement, load-bearing capacity, etc. One of the principal analysis techniques employed by a variety of engineers is the finite element method, and Machine Learning can play an important role in learning rules for finite element mesh design for enhancing both the efficiency and quality of the computed solutions.

Other engineering design applications of Machine Learning occur in the development of systems, such as traffic density forecasting in traffic and highway engineering. Data Mining technologies also have a range of other engineering applications, including fault diagnosis (for example, in aircraft engines or in on-board electronics in intelligent military vehicles), object classification (in oil exploration), and machine or sensor calibration. Classification may, indeed, form part of the mechanism for fault diagnosis.

As well as in the design field, Machine Learning methodologies such as Neural Networks and Case-Based Reasoning are increasingly being used for engineering project management in an arena in which large scale international projects require vast amounts of planning to stay within time scale and budget.

H. Fraud Detection and Compliance

Techniques which are designed to register abnormal transactions or data usage patterns in databases can provide an early alert, and thus protect database owners from fraudulent activity by both a company's own employees and by outside agencies. An approach that promises much for the future is the development of adaptive techniques that can identify particular fraud types, but also be adaptive to variations of the fraud. With the ever-increasing complexity of networks and the proliferation of services available over them, software agent technology may be employed in the future to support interagent communication and message passing for carrying out surveillance on distributed networks.

Both the telecommunications industry and the retail businesses have been quick to realize the advantages of Data Mining for both fraud detection and discovering failures in compliance with company procedures. The illegal use of telephone networks through the abuse of special services and tariffs is a highly organized area of international crime. Data Mining tools, particularly featuring Classification, Clustering, and Visualization techniques have been successfully used to identify patterns in fraudulent behavior among particular groups of telephone service users.

V. FUTURE DEVELOPMENTS

Data Mining, as currently practiced, has emerged as a subarea of Computer Science. This means that initial developments were strongly influenced by ideas from the Machine Learning community with a sound underpinning from Database Technology. However, the statistical community, particularly Bayesians, was quick to realize that they had a lot to contribute to such developments. Data Mining has therefore rapidly grown into the interdisciplinary subject that it is today.

Research in Data Mining has been led by the KDD (Knowledge Discovery in Databases) annual conferences, several of which have led to books on the subject (e.g., Fayyad *et al.*, 1996). These conferences have grown in 10 years from being a small workshop to a large independent conference with, in Boston in 2000, nearly 1000 participants. The proceedings of these conferences are still the major outlet for new developments in Data Mining.

Major research trends in recent years have been:

- The development of scalable algorithms that can operate efficiently using data stored outside main memory

- The development of algorithms that look for local patterns in the data—data partitioning methods have proved to be a promising approach
- The development of Data Mining methods for different types of data such as multimedia and text data
- The developments of methods for different application areas

Much has been achieved in the last 10 years. However, there is still huge potential for Data Mining to develop as computer technology improves in capability and new applications become available.

SEE ALSO THE FOLLOWING ARTICLES

ARTIFICAL NEURAL NETWORKS ● COMPUTER ALGORITHMS ● DATABASES ● FOURIER SERIES ● FUNCTIONAL ANALYSIS ● FUZZY SETS, FUZZY LOGIC, AND FUZZY SYSTEMS ● STATISTICS, BAYESIAN ● WAVELETS

BIBLIOGRAPHY

Adriaans, P., and Zantinge, D. (1996). "Data Mining," Addison-Wesley, MA.

Berry, M., and Linoff, G. (1997). "Data Mining Techniques for Marketing, Sales and Customer Support," Wiley, New York.

Berson, A., and Smith, S. J. (1997). "Data Warehousing, Data Mining, and Olap," McGraw-Hill, New York.

Bigus, J. (1996). "Data Mining With Neural Networks," McGraw-Hill, New York.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). "Classification and Regression Trees," Wadsworth, Belmont.

Cabena, P., Hadjinian, P., Stadler, R., Verhees, J., and Zanasi, A. (1997). "Discovering Data Mining from Concept to Implementation," Prentice-Hall, Upper Saddle River, NJ.

Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R. (1996). "Advances in Knowledge Discovery and Data Mining," AAAI Press/The MIT Press, Menlo Park, CA.

Freitas, A. A., and Lavington, S. H. (1998). "Mining Very Large Databases With Parallel Processing," Kluwer, New York.

Groth, R. (1997). "Data Mining: A Hands on Approach to Information Discovery," Prentice-Hall, Englewood Cliffs, NJ.

Inmon, W. (1996). "Using the Data Warehouse," Wiley, New York.

Kennedy, R. L., Lee, Y., Van Roy, B., and Reed, C. D. (1997). "Solving Data Mining Problems Through Pattern Recognition," Prentice-Hall, Upper Saddle River, NJ.

Lavrac, N., Keravnou, E. T., and Zupan, B. (eds.). (1997). "Intelligent Data Analysis in Medicine and Pharmacology," Kluwer, Boston.

Mattison, R. M. (1997). "Data Warehousing and Data Mining for Telecommunications," Artech House, MA.

Mitchell, T. (1997). "Machine Learning," McGraw-Hill, New York.

Ripley, B. (1995). "Pattern Recognition and Neural Networks," Cambridge University Press, Cambridge.

Stolorz, P., and Musick, R. (eds.). (1997). "Scalable High Performance Computing for Knowledge Discovery and Data Mining," Kluwer, New York.

Weiss, S. M., and Indurkhya, N. (1997). "Predictive Data Mining: A Practical Guide" (with Software), Morgan Kaufmann, San Francisco, CA.

Wu, X. (1995). "Knowledge Acquisition from Databases," Ablex, Greenwich, CT.

Data Structures

Allen Klinger
University of California, Los Angeles

GLOSSARY

Algorithm Regular procedure (like a recipe) that terminates and yields a result when it is presented with input data.

Binary search tree Data structure used in a search.

Binary tree Tree in which each entry has no, one, or two successors; a data structure that is used to store many kinds of other data structures.

Bit Binary digit.

Circular linkage Pointers permitting more than one complete traversal of a data structure.

Data structure An abstract idea concerning the organization of records in computer memory; a way to arrange data in a computer to facilitate computations.

Deque Double-ended queue (inputs and outputs may be at both ends in the most general deques).

Double linkage Pointers in both directions within a data structure.

Field Set of adjoining bits in a memory word; grouped bits treated as an entity.

Graph Set of nodes and links.

Hash (hashing) Process of storing data records in a disorderly manner; the hash function calculates a key and finds an approximately random table location.

Heap Size-ordered tree; all successor nodes are either (consistently) smaller or larger than the start node.

Key Index kept with a record; the variable used in a sort.

Linear list One-dimensional data set in which relative order is important. In mathematical terms, linear list elements are totally ordered.

Link Group of bits that store an address in (primary, fast) memory.

Linked allocation Method for noncontiguous assignment of memory to a data structure; locations or address for the next element stored in a part of the current word.

List Treelike structure useful in representing recursion.

Minimal spanning tree Least sum-of-link path that connects all nodes in a graph with no cycles (closed circuits).

Node Element of a data structure consisting of one or more memory words. Also, an entity in a graph connected to others of its kind by arcs or links.

Pointer See link.

Port Location where data enter or exit a data structure.

Priority queue Queue where each item has a key that governs output; replacing physical position of records in the data structure by their key values.

Quad-tree (also quadtree) A tree where every node has four or fewer successor elements; abbreviated form of quaternary tree.

Queue Linear list with two ports, one for inputs; the other for outputs; data structure supporting algorithms that operate in a first-in–first-out manner.

Recursion Repeated use of a procedure by itself.

Ring Circularly linked data structure.

Search Operation of locating a record or determining its absence from a data structure.

Sequential allocation Assignment of memory by contiguous words.

Sort Permutation of a set of records to put them into a desired order.

Stack One-port linear list that operates in a last-in–first-out manner. This structure is heavily used to implement recursion.

Tag Bit used to signal whether a pointer is as originally intended; if the alternative, the pointer is a thread.

Thread Alternate use for a pointer in a data structure.

Tree Hierarchical data structure.

Trie Tree data structure employing (1) repeated subscripting, and (2) different numbers of node successors. A data structure that is particularly useful for multiway search; structure from information re*trie*val; useful means for search of linguistic data.

DATA STRUCTURES are conceptual tools for planning computer solution of a problem. For a working definition of the term use *the way information is organized in computer memory*. The memory-organization approach taken is closely intertwined with the algorithms that can be designed. That makes *algorithms* (or *analysis of algorithms*) closely related to *data structures*. In practice, computer programs implement algorithms. Data structures have an impact on computer operation. They determine available memory space for other programs and fix the running time of their own program and the ability of other routines to function.

Data structures are places to put useful things inside a computer. The choice of one kind rather than another is a process of arranging key information to enable its use for some purpose. Another way to define data structures is as practical ways to implement computation on digital computers, taking account of characteristics of the hardware, operating system, and programming language.

Data structures are ways of describing relationships among entities. Recognizing a relationship and its properties helps in writing computer programs. Many ways to arrange data are related to, or are like, well-known physical relationships between real objects. Examples include people in business situations or families, books on a shelf, newspaper pages, and cars on a highway. Sometimes data structures are dealt with in combination with all their implementation details. This combination is called an *abstract data type*.

There is a close connection between kinds of data structures and algorithms, one so close that in some cases it is equivalence. An algorithm is *a regular procedure to perform calculations*. This is seen in examples such as a search for a name in a list, as in using phone directories. A directory involves data-pairs (name, telephone number)—usually alphabetically organized by initial letter of the last name. There is a practical usefulness of alphabetization. As with any data structuring, this idea and its implementation enable use of a simpler algorithm. A central data structuring notion is arrangement, particularly in the sense *systematic* or *ordered*. When data are disordered, the algorithm would need to read every record until the one wanted was found. As in this example, data structure selection is an essential choice governing how computers are to be used in practice. Data structures are fundamental to all kinds of computing situations.

Descriptions of the simplest data structure entities and explanations of their nature follow in succeeding sections. Basic data structures are *stack, queue*, and other *linear lists; multiple-dimension arrays*; *(recursive) lists*; and *trees* (including *forests* and *binary trees*). *Pointer or link* simply means computer data constituting a memory location. *Level* indicates position in a structure that is hierarchical. Link, level, and the elementary structures are almost intuitive concepts. They are fairly easily understood by reference to their names or to real-life situations to which they relate. Evolving computer practice has had two effects. First, the impact of the World Wide Web and Internet browsers has acquainted many computer users with two basic ideas: *link (pointer)* and *level*. Second, computer specialists have increased their use of *advanced data structures*. These may be understandable from their names or descriptive properties. Some of these terms are *tries, quad-trees (quadtrees, quaternary trees), leftist-trees, 2–3 trees, binary search trees*, and *heap*. While they are less common data structures and unlikely to be part of a first course in the field, they enable algorithmic procedures in applications such as image transmission, geographic data, and library search.

The basic data structure choice for any computing task involves use of either (1) *reserved contiguous memory*,

or (2) *links, pointers*—locators for related information elements. *Data structuring* concerns deciding among methods that possess value for *some* information arrangements. Often the decision comes down to a mathematical analysis that addresses how frequently one or another kind of data is present in a given problem. The data structure may need to be chosen to deal with any of several issues, for example: (1) things to be computed, (2) characteristics of the programming language, (3) aspects of the operating system, (4) available digital hardware, and (5) ways that results may be used. Thus, *data structures* is a technical term that covers: (1) practical aspects of the computing domain needed to effectively and efficiently use computers, and (2) theoretical aspects involving mathematical analysis issues.

I. INTRODUCTION

Data structures make it easier for a programmer to decide and state how a computing machine will perform a given task. To actually execute even such a simple algorithm as multiplication of two numbers with whole and fractional parts (as $2\frac{1}{4}$ times $3\frac{1}{3}$) using a digital device, a suitable data structure must be chosen since the numeric information must be placed in the machine somehow. How this is done depends on the representation used in the data structure—for example, the number of fields (distinct information entities in a memory word) and their size in binary digits or bits. Finally, the actual program must be written as a step-by-step procedure to be followed involving the actual operations on the machine-represented data. (These operations are similar to those a human would do as a data processor performing the same kind of algorithm.)

The topic of data structures fits into the task of programming a computer as mathematical reasoning leads from word descriptions of problems to algebraic statements. (Think of "*rate* times *time* equals *distance*" problems.) In other words, data structure comparisons and choices resemble variable definitions in algebra. Both must be undertaken in the early stages of solving a problem. Data structures are a computer-oriented analogy of "let x be the unknown" in algebra; their choice leads to the final design of a program system, as the assignment of x leads to the equation representing a problem.

The first three sections of this article introduce basic data structure tools and their manipulation, including the concepts of sequential and linked allocation, nodes and fields, hierarchies or trees, and ordered and nonordered storage in memory. The section on elementary and advanced structures presents stacks, queues, and arrays, as well as inverted organization and multilists. The final sections deal with using data structures in algorithm design. Additional data structure concepts, including binary search trees, parent trees, and heaps, are found in these sections, as is a survey of sorting and searching algorithms.

II. MEMORY ALLOCATION AND ALGORITHMS

A rough data structure definition is a manner for recording information that permits a program to execute a definite method of movement from item to item. An analysis is needed before programming can be done to decide which data structure would be the best choice for the situation (data input, desired result).

Examples of tasks we could want the computer to perform might include:

1. Selection of individuals qualified for a task (typist, chemist, manager)
2. Output of individuals with a common set of attributes (potential carpool members for certain regions, bridge players vested in the company retirement plan and over 50 years in age, widget-twisters with less than two years of seniority)
3. Output of individuals with their organizational relationships (division manager, staff to division manager, department director, branch leader, section chief, group head, member of technical staff)

In data structure terms, all these outputs are lists, but the last item is more. In ordinary terms, it is an *organization chart*. That is a special kind of data structure possessing *hierarchy*. As a data structure, the relationship possesses a hierarchical quality so it is not just a list but also a *tree*. Trees can be represented by a linear list data structure through the use of a special place marker that signifies *change hierarchy level*. In text representation, either a period or a parenthesis is used for that purpose. Whether a data structure is a list or is represented by a list is another of the unusual aspects of this field of knowledge. Nevertheless, these are just two aspects of the problem being dealt with, specifically:

1. The nature of the physical data relationship
2. The way the data may be presented to the computer

Two kinds of storage structures are familiar from everyday life: *stack* and *queue*. Real examples of queues are waiting lines (banks, post office), freeway traffic jams, and

so forth. Physical stacks include piles of playing cards in solitaire and rummy games and unread material on desks.

Data structure methods help programs get more done using less space. Storage space is valuable—there is only a limited amount of *primary computer storage (random access, fast,* or *core* memory; RAM stands for *random access memory*). Programs have been rewritten many times, often to more efficiently use fast memory. Many techniques evolved from memory management programming lore, some from tricks helpful in speeding the execution of particular algorithms. Peripheral memory devices are commonly referred to as *external or secondary storage*. They include compact-disc read-only memory (*CD-ROM*, *disks*, *drums*, and *magnetic tape*). Such devices enable retaining and accessing vast volumes of data. The use of secondary stores is an aspect of advanced data structuring methodology often dealt with under *file management* or *database management*. This article is mainly concerned with fast memory (primary storage).

Improvements in storage use or in execution speed occur when program code is closely matched to machine capabilities. Data structure ideas do this by using special computer hardware and software features. This can involve more full use of *bits* in computer words, *fields* of words in primary storage, and *segments* of files in secondary storage.

Data structure ideas can improve algorithm performance in both the static (space) and dynamic (time) aspects of memory utilization. The static aspects involve the actual memory allocation. The dynamic aspects involve the evolution of this space over time (measured in central processor unit, or CPU, cycles) as programs execute. Note that algorithms act on the data structure. Program steps may simply access or read out a data value or rearrange, modify, or delete stored information. But, they also may combine or eliminate partially allotted sets of memory words. Before-and-after diagrams help in planning dynamic aspects of data structures. Ladder diagrams help in planning static allocations of fast memory. Memory allocation to implement a data structure in available computer storage begins with the fundamental decision whether to use linked or sequential memory. Briefly, linked allocation stores wherever space is available but requires advanced reservation of space in each memory word for pointers to show where to go to get the next datum in the structure. Memory that is sequentially allocated, by contrast, requires reservation of a fixed-sized block of words, even though many of these words may remain unused through much of the time an algorithm is being followed as its program executes.

We now present an example that illustrates several other data structure concepts; the notions of *arrays* (tables) and *graphs*, which we will describe below, are used in handling scheduling problems.

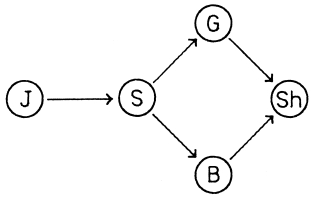
1. Example (Precedence)

A room has several doors. Individuals arrive at the room at different times. This information can be arranged in several ways. Listing individuals in arrival order is one possibility. Another is listing individuals with their arrival times—that is, in a table. A graph is yet another way; see Fig. 1.

A graph can represent the precedence relations via its physical structure (its nodes and its edges), as in Fig. 1. The arrows indicate "successor-of-a-node." J(ones) is no node's successor (was first); Sh(arp) has no successors (was last). G(reen) and B(rown) arrived simultaneously after S(mith) and before Sh(arp).

Computer storage of precedence information via a table leads to a linear list data structure. The alternative, the graph, could be stored in numeric form as node pairs. The arrows in the graph become locators (links, pointers) to data words in computer memory. Ordinal information (first, second, etc.) also could be stored in the data words and could facilitate certain search algorithms. Thus, alternatives exist for presenting facts about time or order of each individual's entry. Decisions to be made about those facts are derived from the data structure contents. One possible data structure is a table giving entry times. Another is a graph showing entry order. Since order can be derived from time data, an algorithm that operates on the data in the table (remove two entries, subtract their time values, etc.) can give information that is explicit in the other data structure via the graph arrows (or locator in computer memory). This is a general property of data structures. That is, data represented can shorten computations. It is also true that computing can reduce storage needs.

Priority of arrival is analogous to scheduling such things as jobs to build a house or courses to complete a degree. Computer algorithms that take advantage of data structure properties are useful tools for planning complex schedules. Efficient scheduling algorithms combine elementary data structures with the fact that graph elements without predecessors can be removed while preserving priorities.



Jones	12:57
Smith	12:58
Green	12:59
Brown	12:59
Sharp	1:00

FIGURE 1 Precedence graph and table of arrival times.

A *node* is the smallest elementary data structure unit. Physical nodes are contiguous words in fast memory. It is useful to think of the realization of a data structure as a network of nodes embedded in memory, possibly connected by links. Parts of nodes and portions of memory words referred to as entities by programs are called *fields*. A one-bit field is called a *flag*, or sometimes a *tag*.

Simple sequential assignment of memory can implement *list structures*. Three elementary data structures are illustrated as follows by list examples. A pile of papers on a desk is a stack. A group of cars in a one-lane traffic bottleneck is a queue. A collection of candies can be a bag. Actually, in computer terms the data objects (papers, cars, and candies) are stored in the named data structure, and all three are generally called *linear lists*. The data type (e.g., real, integer, complex) is another attribute that must be noted in planning the data structure. It can change the space needed by each node (complex values require more bits).

An array is a multidimensional (not linear) data structure. An appointment schedule for the business week, hour by hour and day by day, is an example of an array. A mathematical function can generate data for an array structure. For example, the four-dimensional array shown in Table I (with entries in exponential notation) was obtained by substituting numeric values for x, y, w, and z in the expression:

$$w = x^{3.5} + 2y^z$$

The exponential notation for numerical items displayed here is also called *floating point*. The notation is useful in this table because there is a wide range of values for the w variable for similar x, y, and z.

In sequential allocation of storage, data are grouped together one memory word after another within a given structure. In linked allocation, a field or node part, usually a section of one memory word, is used to store the next location. Reserving more memory than is currently required to hold the data on hand is necessary in sequential allocation since the number of elements stored in a structure may grow during execution of programs. In contrast, with sequential methods, linked allocation does not require reserving extra memory words; instead, it adds bits needed to store pointers as overhead in every node.

TABLE I Four-Dimension Array: $w = x^{3.5} + 2y^z$

x	y	z	w
1.25	50	0.25	7.50
3.75	3	3.75	8.82×10^5
2.35	32	3.75	8.82×10^5
64	21	1.75	2.10×10^6
...			

Although less memory is generally needed with linked allocation than with sequential allocation, the latter facilitates more rapid access to an arbitrary node. This is so since an arbitrary node is located through what may be a lengthy chain of pointers in the former case. Sequential allocation access to the contents of the data structure involves both the start value (a pointer to the abstract structure) and an index to the items it holds.

III. HIERARCHICAL DATA STRUCTURES

Tree data structures enable representation of multidirectional or hierarchic relationships among data elements. There are several related data structures (for example, *trees*, *forests*), and the most general hierarchic structure, the *list*, is usually indicated as a special structure written with capitalized first letter, *List*. Capital L lists represent recursion. A List differs from either a tree or a forest by permitting nodes to point to their ancestors. All these structures are usually represented in memory by yet another one that also is hierarchic, the *binary tree*. This is so because there are straightforward ways to implement *binary tree traversal*.

A *binary tree* is *a hierarchic structure with every element having exactly no, one, or two immediate successors*. When another structure (*tree, forest, List*) is given a binary tree representation, some of the "linked" memory allocation needed to store the data it holds becomes available in a form that is more sequential. This is related to a descriptive term about binary trees, the notion of *completeness*. A binary tree is said to be *complete* when all the nodes that are present at a level are in sequence, beginning at the left side of the structure, with no gaps.

There are many definite ways to move through all the items in a binary tree data structure (several of them correspond to analogous procedures for general trees, forests, and Lists). Some of the most useful are *pre-order* (where a root or ancestor is visited before its descendants; after it is evaluated, first the left successor is visited in preorder, then the right), *in-order* (also called *symmetric order*), and *post-order*. In post-order, the ancestor is seen only after the post-order traversal of the left subtree and then the right. In-order has the in-order traversal of the left subtree preceding the evaluation of the ancestor. Traversal of the two parts occurs before visiting the right sub-tree in the in-order fashion.

Trees may be ordered; that is, all the "children" of any node in the structure are indexed (first, second, etc.). An ordered tree is the most common kind used in computer programs because it is easily represented by a binary tree; within each sibling set is a linear-list-like structure, which is useful for data retrieval; a nonordered tree is much like a

bag or set. A node in either kind of tree has degree equal to the number of its children, that is, the length of its sublist. Another basic tree term is *level*. This signifies the number of links between the first element in the hierarchy, called the *root*, and the datum. As a result, we say that the root is at level zero. A tree node with no successors is called a *tip* or *leaf*.

Strings, linear lists with space markers added, are often used to give text depictions of trees. There are several ways to visually depict the hierarchical relationship expressed by tree structures: nested parentheses, bar indentation, set inclusion, and decimal points—essentially the same device used when library contents are organized, for example, by the Dewey decimal system. All of these methods can be useful. The kind of information that can be stored by a tree data structure is both varied and often needed in practical situations. For example, a tree can describe the chapter outline of a book (Fig. 2a). On personal computers or web browsers, the starting location is called the *root* or *home*. It lists highest level information.

A binary tree is particularly useful because it is easy to represent in computer memory. To begin with, the data words (the contents stored at each node in the structure) can be distinguished from successor links by a one-bit flag. Another one-bit flag indicates a "left/right" characteristic of the two pointers. Thus, at a cost of very little dedicated storage, basic information that is to be kept can be stored with the same memory word field assignments. The binary tree version of Fig. 2a is Fig. 2b.

Binary trees always have tip nodes with unused pointers (link fields). Hence, at the small cost of a tag bit to indicate whether the link field is in use, this space can be used for traversal. The idea is that any pointer field not used to locate other data in the binary tree structure can be used to describe where to go in a traversal (via preorder, post-order, etc.). Hence, these tip fields can be used to speed algorithm execution. These pointer memory location values may be useful in indicating where the algorithm should go next. If a tag bit indicates that the purpose is data structure traversal, the locator is called a *thread*.

IV. ORDER: SIMPLE, MULTIPLE, AND PRIORITY

A. Linear and Indexed Structures

The simplest data structures are lists (also called data strings). Even the ordinary list data structure can be set up to have one or another of several possible input and output properties. There are ways to use such input and output properties to give advantages in different situations. Hence, careful choice of the simple structure can contribute to making a computer program fast and space-efficient.

Two kinds of simple lists are stacks and queues. A *stack* is a data store where both input and output occur at the same location: the general name for such a place is a *port*. Stacks are characterized by last-in–first-out data handling. A *queue* is a two-port store characterized by first-in–first-out data handling. A queue is a more versatile data structure than a stack. A method for storing a queue in a fixed fast memory space uses circular wrapping of the data (modulo storage). There is another kind of linear list—a *deque* or *double-ended queue*, which has two ports. In the most general deque, input and output can take place at both ports.

Linear lists are data structures with entries related only through their one-dimensional relative positions. Because of this, from size and starting location of the structure, both first and last nodes can be located easily.

Data stored in a stack are accessible to the program in the reverse of the input order, a property that is useful in situations involving recursion. Evaluating an algebraic formula involving nested expressions is a recursive process that is facilitated by use of a stack to store left parentheses and symbols (operands) while the program interprets operators.

Stacks, *queues*, and *deques* all make heavy use of linear order among the data elements. However, for some computing purposes, a data structure that pays no attention to order is very useful. One elementary structure that has this nature is called a *bag*. Bags are structures where elements may be repeatedly included. An example of a bag is

DIRECTORY-BAG
M. JONES, 432-6101;
M. JONES, 432-6101; M. JONES, 432-6101;
A. GREEN, 835-7228; A. GREEN, 835-7228;

One use of bags is in looping as a supplementary and more complex index.

Up to this point we have considered only one type of elementary structure, in the sense that all the ones we have dealt with have been one dimensional. In mathematics, a one-dimensional entity is a kind of array known as a *vector*. A vector can be thought of as a set of pairs: an index and a data item. The convention is that data items are all the same type and indices are integers. With the same convention (both indices and data present) allowing an index to be any ordered string of integers yields an elementary data structure called an *array*. If the ordered string of integers has n-many elements, the array is associated with that number of dimensions. In algebra, mathematics applies two-dimensional arrays to simultaneous linear equations, but the data structure idea is of a more general concept. It allows any finite number of dimensions.

A *dense array* is one where data items are uniformly present throughout the structure. In a block array, data

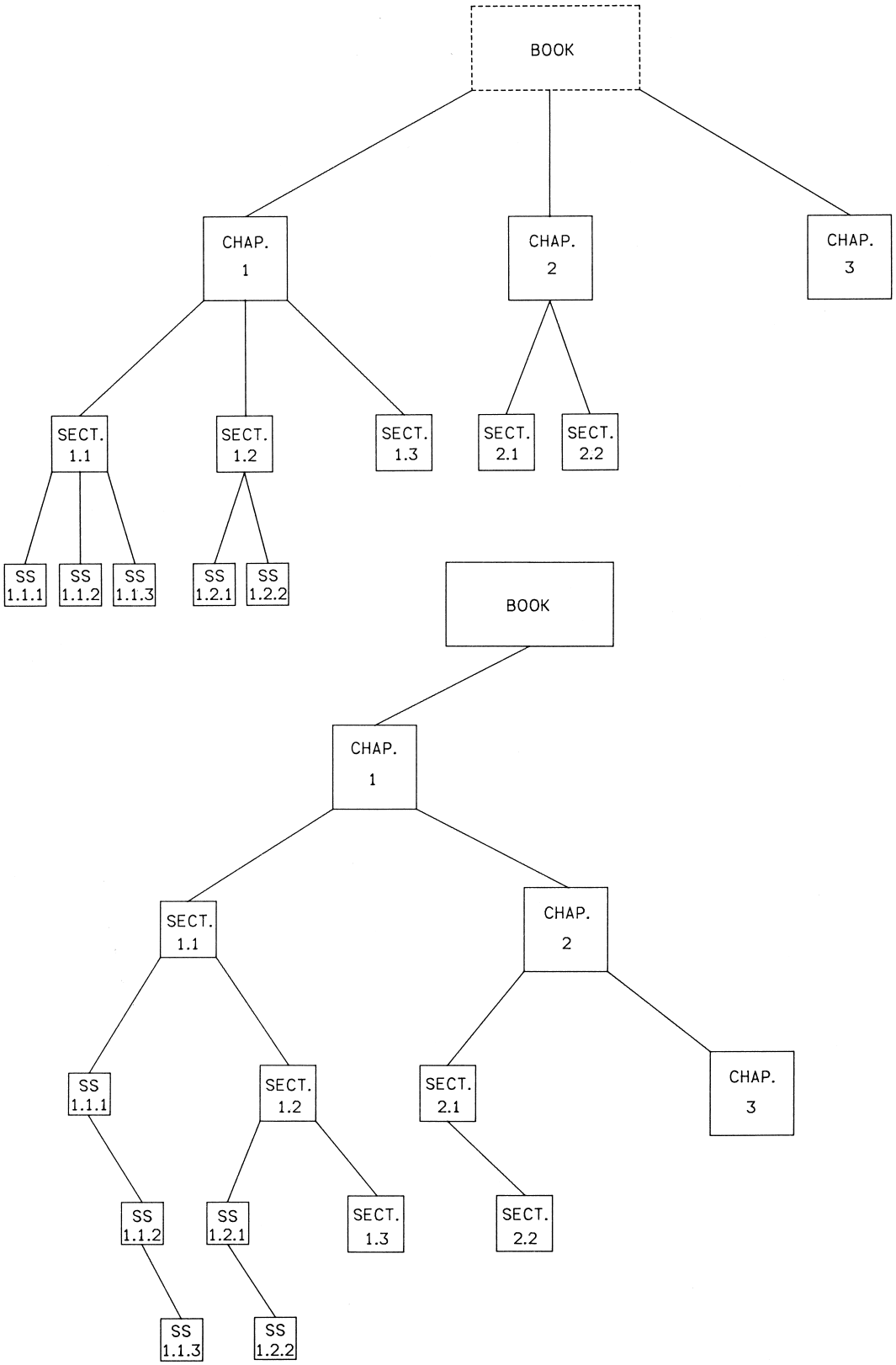


FIGURE 2 Trees and a book chapter outline. (a) General tree representation; (b) binary tree representation. Chap., chapter; Sect., section; SS, subsection.

items may be present only in certain intervals of coordinate directions. Both dense and block arrays are good candidates for sequential storage, in contrast with sparse arrays. A *sparse array* characteristically has a number of data items present much smaller than the product of the maximum coordinate dimensions, and data that are arbitrarily distributed within it. Linked allocation is preferred for sparse arrays. Sparse arrays occur in routing applications. A table using a binary variable to show plants where goods are manufactured and destinations where they are sold would have many *zeros* signifying *plant-not-connected-to-market*. Use of a linked representation for storing this sparse array requires much less storage than by sequential means.

Compressed sequential allocation is another array storage method. It uses the concept of a base location and stores only nonzero data items. It has the advantage that the nonzero data items are stored in sequential order, so access is faster than with linked representation. For r nonzero elements, where r is not very small compared to the product of the array dimensions $m \cdot n$, the storage required for compressed sequential allocation is $2r$. The base locations are essentially list heads (see below), and searching a list of r of them takes, on the average, of the order of $log_2 r$ steps. Arrays are also called *orthogonal lists*, the idea being that the several coordinate directions are perpendicular; rows and columns in a two-dimensional matrix are examples. In an $m \times n$ matrix A with elements $a[i, j]$, a typical element $A[k, l]$ belongs to two orthogonal linear lists (row list $A[k, *]$, column list $A[*, l]$, where the asterisk represents all the values, i.e., $1, \ldots, m$ or $1, \ldots, n$).

B. Linkage

The method of double linkage of all nodes has advantages in speeding algorithm execution. This technique facilitates insertion and deletion in linear lists. Another advanced method has pointers that link the beginning and end of a data structure. The resulting data structures enable simplified traversal algorithms. This technique is usually called *circular linkage* or sometimes simply *rings*; it is a valuable and useful method when a structure will be traversed several times and is entered each time at a starting point other than the starting datum ("top" in a stack, "front" in a queue).

Whenever frequent reversal of traversal direction or insertion or deletion of data elements occurs, a doubly linked structure should be considered. Two situations where this often occurs are in discrete simulation and in algebra. Knuth described a simulation where individuals waiting for an elevator may tire and disappear. Likewise, the case of polynomial addition requires double linkage because sums of like terms with different signs necessitate deletions.

A special type of queue called a *priority queue* replaces the physical order of the data elements by the numerical value of a field in the nodes stored. Data are output from a priority queue in the order given by these numerical values. In order to do this, either the queue must be scanned in its entirety each time a datum is to be output, or the insertion operation must place the new node in its proper location (based on its field value). Whatever the approach, this structure introduces the need for several terms discussed below in greater detail. A *key* is a special field used for ordering a set of records or nodes. The term *key* often signifies "data stored in secondary storage as an entity." The process of locating the exact place to put an item with a particular key is called a *search*. The process of ordering a set of items according to their key values is called a *sort*.

Even though at some time during program execution a data structure may become empty, it could be necessary to locate it. A special memory address (pointer, locator) for the data structure makes it possible for that to be done. This address is called a *list head*. (List heads are traversal aides.)

Multilists are a generalization of the idea of orthogonal lists in arrays. Here, several lists exist simultaneously and use the same data elements. Each list corresponds to some logical association of the data elements. That is, if items are in a list they possess a common attribute.

Two other advanced structures, *inverted lists* and *doubly linked trees*, enable more efficiency in executing certain kinds of algorithms by using more pointers. Inverted organization takes one table, such as Table I, into many. The added pointers make data structure elements correspond to the subtables.

V. SEARCHING AND SORTING TECHNIQUES

The two topics, searching and sorting, are related in the sense that it is simpler to locate a record in a list that has been sorted. Both are common operations that take place many times in executing algorithms. Thus, they constitute part of the basic knowledge of the field of data structures. There are many ways to define special data structures to support searching and sorting. There is also a direct connection between searching itself, as an algorithmic process, and tree data structures.

Sorting tasks appear in so many applications that the efficiency of sort algorithms can have a large effect on overall computer system performance. Consequently, choice of a sort method is a fundamental data structure decision. The ultimate decision depends on the characteristics of the data sets to be sorted and the uses to which the sorted data will be put. The requests for a sorted version of a

list, their nature, and the rate of re-sort on given lists must all be taken into account. Several measures of merit are used to make quantitative comparisons between sorting techniques; memory requirement is one such measure. In essence, a sort yields a permutation of the entries in an input list. However, this operation takes place not on the information contained in the individual entries, but rather on a simpler and often numerical entity known as a *key*.

A. Sorting Algorithms

There are very many different sorting algorithms. Some of them have descriptive names, including *insertion sort*, *distribution sorting*, and *exchange sorting*. Another kind, *bubble sort*, is based on a simple idea. It involves a small key rising through a list of all others. When the list is sorted, that key will be above all larger values. Some sorting methods rely on special data structures. One such case is *heap sort*.

A heap is a size-ordered complete binary tree. The root of the tree is thus either the largest of the key values or the least, depending on the convention adopted. When a heap is built, a new key is inserted at the first free node of the bottom level (just to the right of the last filled node), then exchanges take place (bubbling) until the new value is in the place where it belongs.

Insertion sort places each record in the proper position relative to records already sorted.

Distribution sort (also called *radix sort*) is based on the idea of partitioning the key space into successively finer sets. When the entire set of keys has been examined, all relative positions in the list have been completely determined. (Alphabetizing a set is an example of a radix sort.)

When a large sorted list is out of order in a relatively small area, *exchange sorts* can be useful. This is a kind of strategy for restoring order. The process simply exchanges positions of record pairs found out of order. The list is sorted when no exchanges can take place.

Another sorting strategy takes the most extreme record from an unsorted list, ends a sorted list to it, then continues the process until the unsorted list is empty. This approach is called *sorting by selection*.

Counting sort algorithms determine the position of a particular key in a sorted list by finding how many keys are greater (or less) than that chosen. Once the number is determined, no further relative movement of the key position is found.

Merging two sorted lists requires only one traversal of each list—the key idea in *merg sort*. To sort a list by merging, one begins with many short sorted lists. Often those "runs" of elements in a random list that are already in order form one of them. The process merges them two at a time. The result is a set of fewer long lists. The procedure repeats until a single list remains.

B. Searching: Algorithms and Data Structures

Searching can be thought of as relatively simple in contrast with the many kinds of sorting algorithms. The simple sequential search proceeds stepwise through the data structure. If the item sought is present, it will be found. If it is present, the worst case is when it is last; otherwise (that is, when it is not in the list) the search is both unsuccessful and costly in terms of the number of items examined.

A much better way to search, on the average, in terms of the number of comparisons required either to find an item or to conclude that it is not in the data structure being examined is called a *binary search*. The approach is based on sorting the records into order before beginning the search and then successively comparing the entry sought first with one in the data structure middle. The process continues successively examining midpoints of subsets. At each stage, approximately half the data structure can be discarded. The result is on the average order of magnitude of $log_2 r$ search steps for a file of size r.

Any algorithm for searching an ordered table of length N by comparisons can be represented by a binary tree data structure. That structure is called a *binary search tree* when the nodes are labeled sequentially (left to right by 1 to N in the N-node case). This fact can also be stated as follows: *There is an equivalent algorithm for searching an ordered table of length N and a binary search tree* (i.e., a data structure). Focusing on the recursive definition of a binary search tree, first, all are binary trees; second, binary search trees have their node information fields consists of keys; finally, each key satisfies the conditions that:

1. All keys in the left subtree are less.
2. All keys in the right subtree are greater.
3. Left and right subtrees are binary search trees.

The many special cases of such recursively defined binary search trees (data structure) each correspond to a search algorithm. Movement through a binary search tree is like going to a place in the file. That place has a key, which is to be examined and compared to the one sought.

Two special cases of binary search trees (and, hence, search algorithms) are *binary* and *Fibonaccian*. Each of these algorithms is relatively easy to implement in terms of computations used to create the underlying search tree data structure. By contrast, *interpolation search* is a valuable method but is more complex in terms of computations.

To obtain the search tree that corresponds to binary search of 16 records, assume that the numbering $1, \ldots,$ 16 gives the keys. If we entered the file at the midpoint and

entered each subfile at its midpoint, we would have a binary search tree with key value 8 at the root, left successor 4, and right successor 12. Fibonacci search replaces the division-by-two operations necessary in a binary search by simpler addition and subtraction steps. The Fibonacci series consists of numbers that are each the sum of the immediately preceding two in that sequence. A Fibonacci series begins with two ones, and has values:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

In a Fibonaccian search, the elements of the binary search tree are either Fibonacci numbers or derived from them; the root is a Fibonacci number, as are all nodes reached by only left links. Right links lead to nodes whose values are the ancestor plus the difference between it and its left successor. That is, the difference between the ancestor and left successor is added to the ancestor to get the right successor value. Fibonaccian binary search trees have a total number of elements one less than a Fibonacci number.

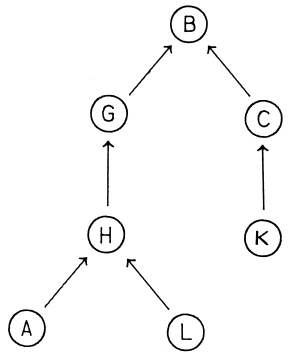
VI. TREE APPLICATIONS

A. Parent Trees and Equivalence Representation

In a *parent tree* data structure, each successor points to its ancestor. Hence, such a structure can be stored in memory as a sequential list of (node, parent-link) pairs, as illustrated by Fig. 3. The parent tree representation facilitates "bottom-up" operations, such as finding the (1) root, (2) depth in the tree, and (3) ancestors (i.e., all nodes in the chain from the selected one to the root). Another advantage is in savings in link overhead: Only one link per node is required, compared to two per node in the conventional (downward-pointer binary tree) representation. The disadvantage of the parent representation is that it is inefficient for problems requiring either enumeration of all nodes in a tree or top-down exploration of tree sections. Further, it is valid only for nonordered trees. Trees where sibling order is not represented are less versatile data structures. For example, search trees cannot be represented as parent trees, since the search is guided by the order on keys stored in the data structure information fields.

Equivalence classes are partitions of a set into subsets whose members can be treated alike. Binary relations can be given explicitly by a list of the objects so paired with the symbol ":" indicating that objects on either side stand in some relation to one another. Sometimes ":" is a symbol for the equivalence relation; another form is "==". These are read "can be replaced by" and "is identically equal to," respectively.

(a)



(b)

NODE	MEMORY ADDRESS
A	1018
B	1182
H	1221
K	1050
G	1129
L	1281
C	1084

(c)

	NODE NAME	NODE LOCATION	PARENT NAME	PARENT-LINK LOCATES
206	A	1018	H	1221
207	B	1182	–	—
208	H	1221	G	1129
209	K	1050	C	1084
210	G	1129	B	1182
211	L	1281	H	1221
212	C	1084	B	1182

FIGURE 3 A parent tree and its memory representation. (a) Parent tree; (b) locations of data found in the parent tree; (c) a sequential table storing the parent tree information.

An equivalence relation of a set of objects is defined by: (1) the list of equivalent pairs, or (2) a rule that permits generation of equivalent pairs (possibly only a subset).

1. Example (Pictures)

The parts of a picture that together compose objects can be members of equivalence classes. A possible rule for generating related pairs is "Two black squares are equivalent if they share an edge." Then, each equivalence class represents a closed pattern.

An algorithm that solves the general equivalence class problem scans a list of input data pairs. The process isolates equivalent pairs and creates parent trees. The idea is that the algorithm creates a node for each of the two elements in a given pairing and one (e.g., always the one

on the left) is made the root. If a new equivalent pair also has an element already in a parent tree, the other item is adjoined to the existing tree (under the relation of this program). Whether two nodes are equivalent can be determined by finding and comparing the roots of their corresponding equivalence trees to determine whether they are in the same partition.

Parent trees are advantageous for algorithms for deciding equivalence, since the most time-consuming step, retrieval of the root of a tested node, is speeded by this data structure. Thus, to organize a set of nodes into equivalence classes, use an algorithm to put them into parent trees.

B. Spanning Trees and Precedence

Spanning trees are illustrated in Fig. 4(a) for a set of four nodes. As the term implies, a spanning tree is a tree that contains all the nodes. Suppose the tree is viewed as a graph so that links are no longer solely pointers (memory addresses of the next datum) but edges with weights. This enables the concept of a *minimal spanning tree*. This idea is simply the spanning tree of least total edge values. The concept is illustrated by the following example and by Fig. 4(b).

1. Example (Minimal Spanning Tree)

. We have four nodes, a, b, c, and d, and seek to build a minimal spanning tree given the edge weights:

$$(a, b) = 6, \quad (b, d) = 12$$
$$(a, c) = 3, \quad (c, d) = 16$$

The tree with the form $(a\,(b\,(d),\ c)$ has total weight 21, which is less than that for the tree with d linked to c instead of b—i.e., $a\,(b, c\,(d))$. Both trees just described appear in Fig. 4 (second row at right).

The minimal spanning tree algorithm uses the following idea. Initially, all the nodes are members of different sets. There are as many sets as nodes. Each node has itself as the only member of its set. As the algorithm proceeds, at each stage it groups more nodes together, just as in equivalence methods. The algorithm stops when all nodes are in one set. The parent tree data structure is the best one to use to implement the minimal spanning tree algorithm.

Yet another algorithm concept and its implications for data structure selection arise from the precedence situation introduced earlier. To create an efficient scheduling procedure, observe that any task that does not require completion of others before it is begun can be started at any time. In data structure terms, this is equivalent to an algorithm using the following fact: *Removal of a graph element without a predecessor does not change the order of priorities stored in the graph.*

VII. RANDOMNESS, ORDER, AND SELECTIVITY

The issue of order is central to data structures, but it is also the opposite of a basic mathematical notion, randomness. Currently, data encryption is a valued use of computation. The basic coding method involves creating a form of disorder approaching randomness to render files unintelligible

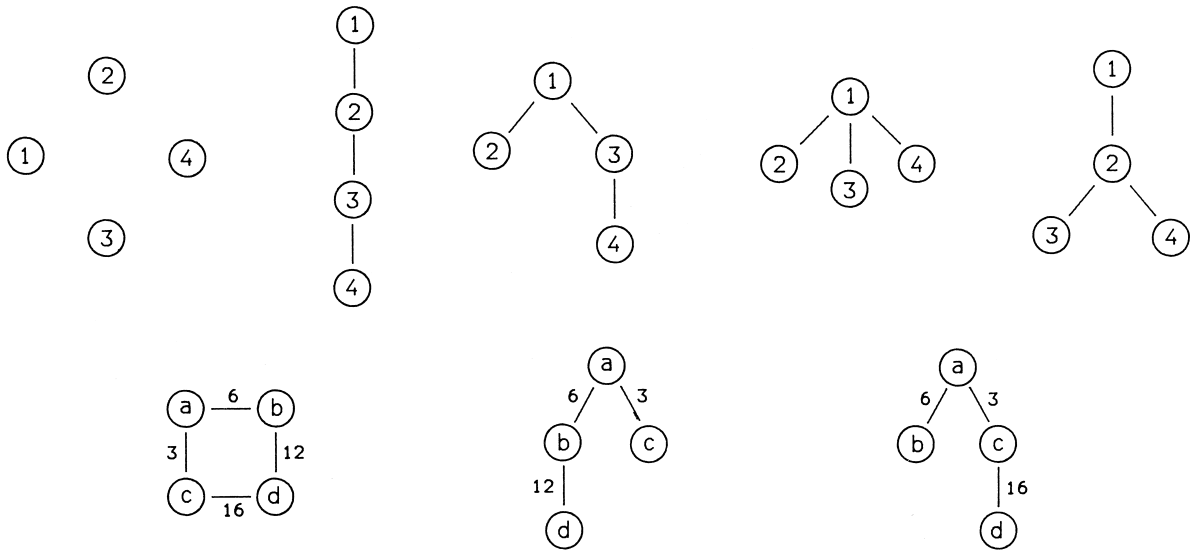


FIGURE 4 Spanning trees. (a) Four nodes with four spanning trees, all with node one at the root; (b) four nodes with edge weights and two different spanning trees.

to an intruder. Factoring a very large integer into a product of prime numbers, finding the greatest common divisor of two integers, and devising new algorithms to calculate fundamental quantities are all central issues in restructuring data that relate to encryption algorithms. This topic is beyond the scope of this article, but is addressed in the second of the three data structure volumes by Knuth.

Regarding disorder versus order, π, the ratio of the circumference of a circle to its diameter, has many known and some esoteric qualities. For example,

1. It is irrational.
2. It is not known whether its successive digits are truly random.
3. Successive digits of π exhibit local nonrandomness (e.g., there are seven successive 3's occurring starting at the 710,100th decimal digit, as noted by Gardner).
4. π is well approximated by several easily computed functions: 22/7, 355/113, $[2143/22]^{1/4}$, this last noted by Ramanujan.

Selecting where to search in the plane is aided by both *quad-trees* and *2–3 trees*. Quad-trees are like binary trees but regulary involve *four or fewer success to every tree node*. As a data structure, quad-trees enable algorithms to process spatial issues, including hidden-line elimination in graphic display. The three-dimensional analog is *octress* (eight or fewer successor nodes). The somewhat similar 2–3 trees require all internal nodes (i.e., nonterminals) to have either two or three successors.

A *trie* is a data structure that uses repeated subscripting and is used to enable multiway search. The term comes from the word re*trie*val. It originated in the field of information retrieval and has developed in part because of machine translation. The benefit of using the trie approach is that it helps reduce the number of ways to search. It does this by eliminating impossible cases. (For example, in English words, a "q" is never followed by anything but a space [Iraq] or the letter "u". That involves a considerable reduction from 26 or 27 other letters or space symbol.) Entries in the nodes of a trie are instructions that describe how to proceed further in a symbol-by-symbol search.

Hashing is a very useful computer technique for locating (storing) data records. The hashing method deliberately introduces disorder. In other words, hashing uses randomness in storing. The rationale is that *on the average* this achieves a significant speed improvement in locating a data record. The index of the item to be stored is called a *key*, and the function that takes the key into a table location is called a *hash function*. Usually, this is done in a way that chops up the numeric key value, introducing a somewhat random element into the actual pattern of data storage. (This is what lead to the term *hashing*.)

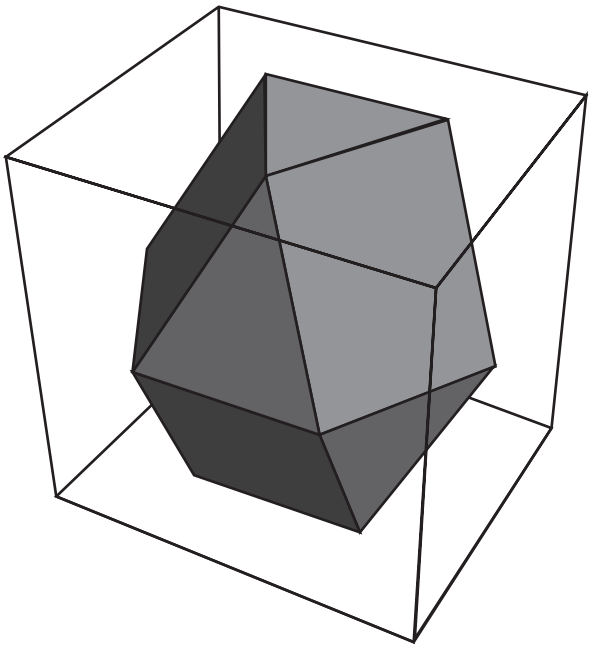


FIGURE 5 Cuboctahedron.

Both fractions and whole numbers led to ideas that are essentially data structures. For fractions, see the universal resource locators in the bibliography. For whole numbers, addition of a long list of large numbers vertically arranged is a much easier matter than all in a row, if they are written in Arabic numerals.

A geometric realization of the value of lists and data structures in general is shown in Fig. 5, a view of a 14-faced solid generally called a *cuboctahedron*. The image was generated by the Mathematica utility from the data structures below. Coordinates of the vertices were calculated by analytic geometry. Once each triple, x, y, z, was known, the graphic routine needed to sequence vertices, generate faces, and then color the result. Two of the lists that were involved in that process follow. (Notice that there are quadrilateral and triangular faces; likewise, each list consists of sublists.) Braces, "{"and"}", initiate and conclude, respectively, each list/sublist. The integers in the "Faces" list range from one to 12. Each one is a specific three-dimensional point or corner of the solid object. The actual coordinates of these points appear in the 1 through 12 order in the "Vertices" list.

1. Faces = {{10, 11, 12}, {7, 8, 9}, {4, 5, 9, 8}, {8, 3, 4}, {2, 3, 8, 7}, {7, 1, 2}, {1, 6, 9, 7}, {9, 5, 6}, {10, 2, 3}, {1, 6, 12}, {11, 4, 5}, {3, 4, 11, 10}, {1, 2, 10, 12}, {11, 12, 6, 5}}

2. Vertices = {{1, 0, 0}, {1/2, $(3^{1/2})/2$, 0}, {−1/2, $(3^{1/2})/2$, 0}, {−1, 0, 0}, {−1/2, $-(3^{1/2})/2$, 0}, {1/2, $-(3^{1/2})/2$, 0}, {1/2, $(3^{1/2})/4$, 1}, {−1/2, $(3^{1/2})/4$, 1}, {0, $-(3^{1/2})/4$, 1}, 0, $(3^{1/2})/4$, −1, {−1/2, $-(3^{1/2})/4$, −1}, {1/2, $-(3^{1/2})/4$, −1}}

VIII. CONCLUSION

The subject of data structures clearly contains a myriad of technical terms. Each of the topics discussed has been briefly mentioned in this article. A basic text on data structures would devote many pages to the fine points of use. Yet the pattern of the subject is now clear. Before programming can begin, planning the algorithms and the data they will operate on must take place. To have efficient computing, a wide range of decisions regarding the organization of the data must be made. Many of those decisions will be based on ideas of how the algorithms should proceed (e.g., "put equivalent nodes into the same tree"). Others will be based on a detailed analysis of alternatives. One example is taking into account the likely range of values and their number. This determines possible size of a data table. Both table size and retrieval of elements within it impact key choices. Two computer data structure considerations are always *memory needed* and *processing time* or *algorithm execution speed*.

Many aspects of data structures and algorithms involve data stored where access is on a secondary device. When that is the situation, procedures deal with search and sorting. Sorting occupies a substantial portion of all the computing time used and contains numerous alternate algorithms. Alternative means exist because data sometimes make them advantageous.

As in the case of sorting, it is always true that actual choice of how an algorithmic task should be implemented can and should be based on planning, analysis, and tailoring of a problem that is to be solved. The data structure also needs to take into account the computer hardware characteristics and operating system.

Explosive development of computer networks, the World Wide Web, and Internet browsers means that many technical terms discussed in this article will join links, pointers, and hierarchy in becoming common terms, not solely the province of computer experts using data structures.

Universal Resource Locators

Mathematics and computer data structures—http://www.cs.ucla.edu/~klinger/inprogress.html
Egyptian fractions—http://www.cs.ucla.edu/~klinger/efractions.html
Baseball arithmetic—http://www.cs.ucla.edu/~klinger/bfractions.html
Thinking about number—http://www.cs.ucla.edu/~klinger/5fractions.html
Cuboctahedron image—http://www.cs.ucla.edu/~klinger/tet1.jpg

SEE ALSO THE FOLLOWING ARTICLES

C AND C++ PROGRAMMING LANGUAGE • COMPUTER ALGORITHMS • COMPUTER ARCHITECTURE • DATABASES • DATA MINING • EVOLUTIONARY ALGORITHMS AND METAHEURISTICS • QUEUEING THEORY

BIBLIOGRAPHY

Aho, A., Hopcroft, J., and Ullman, J. (1983). "Data Structures and Algorithms," Addison-Wesley, Reading, MA.
Dehne, F., Tamassia, R., and Sack, J., eds. (1999). "Algorithms and Data Structures," (Lecture Notes in Computer Science), Proc. 6th Int. Workshop, Vancouver, Canada, August 11–14, Springer-Verlag, New York.
Graham, R., Knuth, D., and Patashnik, O. (1988). "Concrete Mathematics," Addison-Wesley, Reading, MA.
Knuth, D. (1968). "The Art of Computer Programming," Vol. I, "Fundamental Algorithms," Addison-Wesley, Reading, MA.
Knuth, D. (1973). "The Art of Computer Programming," Vol. III, "Sorting and Searching," Addison-Wesley, Reading, MA.
Knuth, D. (1981). "The Art of Computer Programming," Vol. II, "Seminumerical Algorithms," Addison-Wesely, Reading, MA.
Meinel, C. (1998), "Algorithmen und Datenstrukturen im VLSI-Design [Algorithms and Data Structures in VLSI Design]," Springer-Verlag, New York.
Sedgewick, R. (1990). "Algorithms in C," Addison-Wesley, Reading, MA.
Sedgewick, R. (1999). "Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching," 3rd ed., Addison-Wesley, Reading, MA.
Waite, M., and Lafore, R. (1998). "Data Structures and Algorithms in Java," Waite Group.

Databases

Alvaro A. A. Fernandes
Norman W. Paton
University of Manchester

GLOSSARY

Application A topic or subject for which information systems are needed, e.g., genomics.

Conceptual model A data model that describes application concepts at an abstract level. A conceptual model for an application may be amenable to implementation using different database management systems.

Concurrency control Mechanisms that ensure that each individual accessing the database can interact with the database as if they were the only user of the database, with guarantees as to the behavior of the system when many users seek to read from or write to the same data item at the same time.

Database A collection of data managed by a *database management system*.

Database management system A collection of services that together give comprehensive support to applications requiring storage of large amounts of data that are to be shared by many users.

DBMS *See* **database management system**.

Data model A collection of data types which are made available to application developers by a DBMS.

Data type A set of instances of an application concept with an associated set of legal operations in which instances of the concept can participate. Some data types are *primitive* (e.g., integer, string) insofar as they are supported by the DBMS directly (and, hence, are not application specific). Some are *application specific* (e.g., gene). Some data types are *atomic* (e.g., integer) and some are *complex* (e.g., address, comprising street name, state, etc.). Some data types are *scalar* (e.g., integer) and some are *bulk*, or *collection* (e.g., sets, lists).

Persistence The long-term storage of an item of data in the form supported by a data model.

Referential integrity The requirement that when a reference is made to a data item, that data item must exist.

Secondary storage management The ability to store data on magnetic disks or other long-term storage devices, so that data outlive the program run in which they were created.

Security Mechanisms that guarantee that no users see or modify information that they are not entitled to see or modify.

A DATABASE is a collection of data managed by a *database management system* (DBMS). A DBMS provides facilities for describing the data that are to be stored, and is engineered to support the long-term, reliable storage of large amounts of data (Atzeni *et al*., 1999). DBMSs also provide query language and/or programing language interfaces for retrieving and manipulating database data. Many organizations are dependent in a significant way upon the reliability and efficiency of the DBMSs they deploy.

I. DATABASE MANAGEMENT SYSTEMS

A **database management system** (DBMS) provides a collection of services that together give comprehensive support to applications requiring storage of large amounts of data that are to be shared by many users. Among the most important services provided are:

1. Secondary storage management. The ability to store data on magnetic disks or other long-term storage devices, so that data outlive the program run in which they were created. Sophisticated secondary storage management systems are required to support effective storage and access to large amounts of data. Storage managers typically include facilities for providing rapid access to specific data items using *indexes*, minimize the number of distinct accesses required to the secondary store by *clustering* related data items together, and seek wherever possible to make effective use of *buffers* that retain recently accessed data items in the memory of the computer on which the database system or the application is running.
2. Concurrency control. Large databases are valuable to the organizations that hold them, and often have to be accessed or updated by multiple users at the same time. Uncontrolled updates to a shared store can lead to the store being corrupted as a result of programs interfering with each other in unanticipated ways. As a result, DBMSs incorporate mechanisms that ensure that each individual accessing the database can interact with the database as if they were the only user of the database, with guarantees as to the behavior of the system when many users seek to read from or write to the same data item at the same time.
3. Recovery. Many database applications are active 24 hr a day, often with many different programs making use of the database at the same time, where these programs are often running on different computers. As a result, it is inevitable that programs accessing a database will fail while using the database, and even that the computer running the DBMS can go down while it is in use. The DBMS must be as resilient as possible in the face of software or hardware failures, with a view to minimizing the chances of a database being corrupted or information lost. As a result, DBMSs include recovery facilities that often involve some temporary *replication* of data so that enough information is available to allow automatic recovery from different kinds of failure.
4. Security. Most large database systems contain data that should not be visible to or updateable by at least some of the users of the database. As a result, comprehensive mechanisms are required to guarantee that no users see information that they are not entitled to see, and more important still, that the users who are able to modify the contents of the database are carefully controlled—staff should not be able to modify their own salaries, etc. As a result, database systems provide mechanisms that allow the *administrator* of a database to control who can do what with the database.

DBMSs can thus be seen as having many responsibilities. Like a bank, it is good if a DBMS provides a wide range of facilities in an efficient way. However, in the same way as it is of overriding importance that a bank does not lose your money, it is crucial that a DBMS must keep track of the data that are entrusted to it, minimizing errors or failures. Thus the provision of a database management facility is essentially a serious matter, and substantial database failures would be very damaging to many organizations. However, the details of how the aforementioned services are supported are quite technical, and the remainder of this chapter focuses on the facilities provided to users for describing, accessing, and modifying database data. Details on how databases are implemented can be found in Elmasri and Navathe (2000) and Garcia-Molina *et al*. (2000).

II. DATA MODELS

In the context of database technology, by **data model** is meant a collection of data types which are made available to application developers by a DBMS. By **application** is meant a topic or subject for which information systems are needed, e.g., genomics. The purpose of a data model is to provide the formal framework for modeling application requirements regarding the persistent storage, and later retrieval and maintenance, of data about the entities in the conceptual model of the application and the relationships in which entities participate.

Each data type made available by the data model becomes a building block with which developers model

application requirements for persistent data. The permissible operations associated with each data type become the building blocks with which developers model application requirements for interactions with persistent data. Data models, therefore, differ primarily in the collection of data types that they make available. The differences can relate to structural or behavioral characteristics. Structural characteristics determine what states the database may be in (or, equivalently, are legal under the data model). Behavioral characteriztics determine how database states can be scrutinized and what transitions between database states are possible under the data model.

A DBMS is said to implement a data model in the sense that:

1. It ensures that every state of every database managed by it only contains instances a data type that is well-defined under the implemented data model.
2. It ensures that every retrieval of data and every state transition in every database managed by it are the result of applying an operation that is (and only involves types that are) well defined under the implemented data model.

From the point of view of application developers then, the different structural or behavioral characteriztics associated with different data models give rise to the pragmatic requirement of ensuring that the data model chosen to model application requirements for persistent data does not unreasonably distort the conceptual view of the application that occurs most naturally to users. The main data models in widespread use at the time of this writing meet this pragmatic requirement in different degrees for different kinds of application.

A. The Relational Data Model

The relational data model makes available one single data type, referred to as a *relation*. Informally, a relation can be thought of as a table, with each row corresponding to an instance of the application concept modeled by that relation and each column corresponding to the values of a property that describes the instances. Rows are referred to as *tuples* and column headers as *attributes*. More formal definitions now follow.

Let a **domain** be a set of atomic values, where a value is said to be *atomic* if no further structure need be discerned in it. For example, integer values are atomic, and so, quite often, are strings. In practice, domains are specified by choosing one data type, from the range of primitive data types available in the software and hardware environment, whose values become elements of the domain being specified.

A **relation schema** $R(A_1, \ldots, A_n)$ describes a relation R of **degree** n by enumerating the *n-attributes* $A_1, \ldots, A_n$ that characterize R. An **attribute** names a role played by some domain D in describing the entity modeled by R. The domain D associated with an attribute A is called the **domain** of A and is denoted by $dom(A) = D$. A **relational database schema** is a set of relation schemas plus the following **relational integrity constraints**:

1. *Domain Constraint*: Every value of every attribute is atomic.
2. *Key Constraint*: Every relation has a designated attribute (or a concatenation thereof) which acts as the **primary key** for the relation in the sense that the value of its primary key identifies the tuple uniquely.
3. *Entity Integrity Constraint*: No primary key value can be the distinguished `NULL` value.
4. *Referential Integrity Constraint*: If a relation has a designated attribute (or a concatenation thereof) which acts as a **foreign key** with which the relation refers to another, then the value of the foreign key in the referring relation must be a primary key value in the referred relation.

For example, a very simple relational database storing nucleotide sequences might draw on the primitive data type `string` to specify all its domains. Two relation names might be `DNA-sequence` and `organism`. Their schemas can be represented as in Fig. 1. In this simple example, a `DNA_sequence` has an identifying attribute `sequence_id`, one or more `accession nos` by which the DNA_sequence is identified in other databases, the identifying attribute `protein_id` which the DNA_sequence codes for and the `organism_ids` of the organisms where the DNA_sequence has been identified. An `organism` has an identifying attribute `organism_id`, perhaps a `common_name` and the `up_node` of the organism in the adopted taxonomy.

A **relation instance** $r(R)$ of a relation schema $R(A_1, \ldots, A_n)$ is a subset of the Cartesian product of the domains of the attributes that characterize R. Thus, a relation instance of degree n is a set, each element of which is an n-tuple of the form $\langle v_1, \ldots, v_n \rangle$ such that

DNA_sequence			
sequence_id	*accession_no*	*protein_id*	*organism_id*

organism		
organism_id	*common_name*	*up_node*

FIGURE 1 Relation schemas for `DNA_sequence` and organism.

DNA_sequence			
sequence_id	accession_no	protein_id	organism_id
TRBG361	X56734	AA40058	Trif. repens
BNBGL	X82577	CAA57913	Bras. napus
...	...	...	...

organism		
organism_id	common_name	up_node
Bras. napus	rape	Brassicaceae
Trif. repens	white clover	Papilionoideae
...	...	...

FIGURE 2 Relation instances under the schemas in Fig. 1.

either $v_i \in dom(A_i)$, $1 \leq i \leq n$, or else v_i is the distinguished NULL value. A **relational database state** is a set of relation instances. Legal relation instances for DNA_sequence and organism from Fig. 1 might be as depicted in Fig. 2.

Notice, in Fig. 2, how each tuple in DNA_sequence is an element of the Cartesian product:

$$dom(\texttt{sequence\_id}) \times dom(\texttt{accession\_no}) \times dom(\texttt{protein\_id}) \times dom(\texttt{organism\_id})$$

and correspondingly in the case of organism. Notice also, in Figs. 1 and 2, that all the relational integrity constraints are satisfied, as follows. The domain constraint is satisfied due to the fact that the domain of every attribute is the primitive data type string, whose instances have been assumed to be atomic. The designated primary keys for DNA_sequence and organism are, respectively, sequence_id and organism_id, and since they are intended as unique identifiers of the entities modeled, the key constraint is satisfied. The entity integrity constraint is satisfied assuming that tuples that are not shown also do not contain NULL as values for the primary keys. There is a designated foreign key, organism_id, from DNA_sequence to organism, and since the values of organism_id appear as primary key values in organism, the referential integrity constraint is satisfied.

The relational data model has several distinctive features as a consequence of its simple mathematical characterization:

- Since a relation instance is a set of tuples, no order can be assumed in which tuples occur when retrieved or otherwise are operated on.
- Since there are no duplicate elements in a set, each tuple in a relation instance is unique. In other words, the concatenation of all the attributes is always a candidate key for the relation.
- Since values are atomic, the relational model cannot ascribe to an attribute a domain whose elements are collections (e.g., sets, or lists). For example, in Fig. 2, if a DNA_sequence has more than one accession_no, this cannot be captured with sets of strings as attribute values.

The languages with which data in relation instances can be retrieved and manipulated (thereby effecting state transitions) are discussed in Section III.A.

The relational model was proposed in a form very close to that described in this section in Codd (1970). An excellent formal treatment can be found in Abiteboul *et al.* (1995).

B. Object-Oriented Data Models

Rather than a single data type, as is the case with the relational data model, an object-oriented data model makes available to application developers a collection of type constructors with which complex, application-specific types can be built. The immediate effect of this approach, in contrast with the relational case, is that more complex application data can be modeled more directly in the database. Instead of having to break down one application concept into many relations, the complex structure needed to model the concept is often directly represented by a combination of type constructors provided by the data model.

The most basic notion in object-oriented data models is that of an *object*, whose purpose is to model application entities. An **object** is characterized by having an *identity*, by being in a certain *state* (which conforms to a prescribed *structure*) and by being capable of interaction via a collection of *methods* (whose specifications characterize the *behavior* of the object).

The **identity** of an object is unique. It is assigned to the object by the DBMS when the object is created, and deleted when the object is destroyed. In contrast with a relational primary key, between these two events the identity of the object cannot be changed, neither by the DBMS nor by application systems. Another contrast is that the identity of an object is distinct and independent from both the state of the object and the methods available for interacting with it.

Objects that are structurally and behaviorally similar are organized into *classes*. A **class** has an associated extent which is a set of the identifiers of objects whose structural and behavioral similarity the class reflects. An **object-oriented schema** is a collection of classes plus integrity constraints.

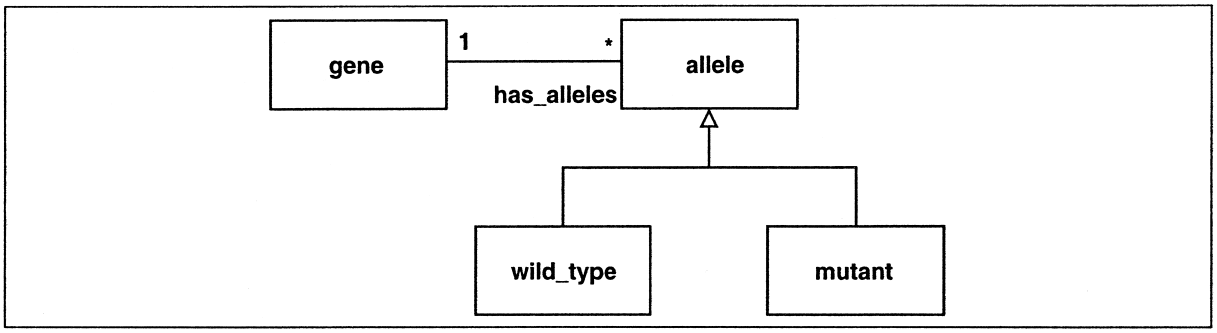


FIGURE 3 A fragment of an object-oriented schema diagram.

In contrast to the relational case, there is no widespread agreement on which integrity constraints are an intrinsic part of an object-oriented model, but one of the most often used stipulates how many objects of each of the classes related by some relationship type can participate in relationship instances. For binary relationships (which are the most commonly used), there are three possibilities, viz., *one-to-one*, *one-to-many*, and *many-to-many*.

Figure 3 depicts in diagrammatic form a small fragment of an object-oriented database schema with four classes, viz., `gene`, `allele`, `wild_type`, and `mutant`. The latter two are subclasses of `allele` as indicated by the arrow head and explained later. Note also the relationship type from `gene` to `allele` named `has_alleles`. The annotations 1 and * specify a one-to-many constraint to the effect that an object of class `gene` may be related to many objects of class `allele`, but one allele is only related to one gene.

The **structure** of an object is characterized by a collection of *properties* and *relationships*. An object is in a certain **state** when all the properties and relationships defined by its structure are assigned legal values. Both **properties** and **relationships** resemble attributes in the relational model, with the following main differences:

1. The domain of a property need not be atomic insofar as the type of its values can be structures or collections by the application of the available **type constructors** (typically, `tuple_of`, `set_of`, `list_of`, and `bag_of`).
2. The domain of a relationship is a set of object identities (or power set thereof).

Figure 4 shows how the structure of the `gene` class from Fig. 3 might be specified in an object-oriented data model. Notice the use of type constructors in assigning domains to the property `citation` and the relationship `has_alleles`. Notice the nonatomic domain `codon_structure`. Notice also that the identity of individual `gene` objects is not dependent on any of the specifications in Fig. 4: It is the DBMS that assigns identifiers, manages their use (e.g., in establishing a relationship between `gene` objects and `allele` objects), and annuls them if objects are destroyed. Notice, finally, that

```
tuple_of codon_structure{
        initiation_codon      :  string;
        codon_sequence        :  list_of(string);
        termination_codon     :  string; }
class gene{
        properties
                standard_name :  string;
                sequence      :  codon_structure;
                function      :  string;
                citation      :  list_of(string);
                protein_id    :  protein;
                organism_id   :  organism;
        relationships
                has_alleles   :  set_of(allele);
        methods
                ... }
```

FIGURE 4 A structure for the `gene` class from Fig. 3.

```
class gene{
        properties
                ...
        relationships
                ...
        methods
                specification
                        compute_G+C_density   ()            -> float;
                        retrieve_region_mRNA  (int, int)    -> list_of(mRNA);
                        ...
                definition
                        ...
        ... }
```

FIGURE 5 (Partial) behavior of the `gene` class from Fig. 4.

while an object-oriented database allows relationships to be expressed as simple properties (e.g., `protein_id` and `organism_id` establish relationships between `gene` and `protein` and `organism`, respectively), it is only by explicitly expressing them as relationships that an object-oriented DBMS would have the information needed to enforce the kind of referential integrity that relational systems enforce for attributes designated as foreign keys.

The **behavior** of an object is characterized by a collection of *method specifications*. A **method specification** is an application-defined declaration of the name and parameter types for a method that can be invoked on the object. By **method invocation** on an object o with identity i is meant, roughly, that the method call refers to i (thereby identifying o as the object to which the method is addressed) and is meant to be executed in the context of the current state of o using o's assigned behavior (where that execution may itself invoke methods defined on objects other than o). A **method definition** is an executable specification of a function or procedure.

Typically, **accessor methods** (i.e., those that return the current value of a property or relationship) and **update methods** (i.e., those that modify to a given value the current value of a property or relationship) are automatically generated at schema compilation time. A popular approach to naming such methods is to prefix the property or relationship name by `get_` and `set_`, respectively. If type constructors are involved then more methods are made available, accordingly. For example, with reference to Fig. 4, methods would be generated that implement (implicit) specifications such as `get_citation() -> list_of(string)`, or `insert_citation(string)`. The former invoked on a `gene` object o would return a list of strings currently assigned as the value of the `citation` property of o, while the latter would cause a state transition in o by adding the string passed as argument to the list of strings currently assigned as the value of the `citation` property of o.

Figure 5 shows how behavior for the `gene` class from Fig. 4 might be specified (but method definitions are omitted). Thus, a `gene` object will respond to a request for its density of guanine and cytosine pairs with a float value. Also, given a region inside it (as defined by a start and an end position in its nucleotide sequence) a `gene` object will respond to a request for mRNA found inside the region with a list of identifiers of `mRNA` objects (assuming a class `mRNA` to have been declared in the schema).

The requirement that one must always invoke methods to interact with objects gives rise to a distinctive feature of object-oriented approaches, viz., **encapsulation**, by which is meant that the state of an object is protected from direct manipulation or scrutiny. This is beneficial in the sense that it allows application developers to ignore the structural detail of the state of objects: all they need to know is the behavior of the object, i.e., which methods can be invoked on it. This, in turn, allows the detailed structure of objects to change without necessarily requiring changes in the application programs that use them.

The approach of organizing objects into classes gives rise to another distinctive feature of object-oriented approaches, viz., **inheritance**, by which is meant a subsumption relationship between the extents of two classes C and C'. Thus, a class C is said to **extend** (or **specialize**, or **inherit from**) a class C' if all objects in the extent of C have the same properties, relationships, and compatible method specifications as (but possibly more than) the objects in the extent of C'. In this case, C is said to be a **subclass** of C'. Thus, inheritance in object-oriented data models is both structural and behavioral (but the compatibility constraints for the latter are beyond the scope of this discussion). There are many motivations for inheritance mechanisms as described, including a stimulus to reuse, a more direct correspondence with how applications are conceptualized, etc.

In the presence of inheritance, not only can a method name be **overloaded** (i.e., occur in more than one method

specification), but it can also be **overridden** (i.e., occur in more than one method definition). In this case, method invocations can be attended to by more than one method definition and the decision as to which one to use usually depends on the type of the object that the method was invoked on. This selection process which binds a method definition to a method invocation can only be carried out at run time, therefore this distinctive object-oriented feature is known as **late** (or **dynamic**) **binding**.

To summarize, an object-oriented data model has the following main distinctive features:

- Objects have not only structure but also behavior.
- Objects have a unique identity that is independent of their state and behavior.
- Values can be drawn from complex domains resulting from the application of type constructors such as `tuple_of`, `set_of`, `list_of`, and `bag_of`.
- The state of objects is encapsulated.
- Objects are organized into classes by structural and behavioral similarity.
- Structural and behavioral inheritance is supported.
- Overloading, overriding, and late binding are supported.

Because an object-oriented model brings together structure and behavior, much of the functionality required by applications to interact with application objects can be modeled inside the database itself. Nevertheless, most object-oriented database systems provide application program interfaces that allow a programming language to retrieve and manipulate objects.

Unlike the relational model, object-oriented data models cannot all be traced back to a single major proposal. The closest thing to this notion is a manifesto (found, e.g., in Atkinson *et al.* (1990)) signed by prominent members of the research community whose recommendations are covered by and large in the treatment given in this section. The ODMG industry consortium of object database vendors has proposed an object-oriented data model that is widely seen as the *de facto* standard (Cattell *et al.*, 2000).

C. Object-Relational Data Models

While object-oriented data models typically relegate relations to the status of one among many types that can be built using type constructors (in this case, e.g., `tuple_of`), object-relational data models retain the central role relations have in the relational data model.

However, they relax the domain constraint of the relational data model (thereby allowing attribute values to be drawn from complex domains), they incorporate some of the distinctive features of the object-oriented model such as inheritance and assigned behavior with encapsulated states, and they allow for database-wide functionality to be specified by means of *rules* (commonly referred to *as triggers*, as described in Section IV.B) that react to specific interactions with application entities by carrying out some appropriate action.

As a consequence of the central role that relations retain in object-relational data models, one crucial difference with respect to the object-oriented case is that the role played by object identity is relaxed to an optional, rather than mandatory, feature. Thus, an object-relational DBMS stands in an evolutionary path regarding relational ones, whereas object-oriented ones represent a complete break with the relational approach. In this context, notice that while a `tuple_of` type constructor may allow a relation type to be supported, each tuple will have an identity, and attribute names will be explicitly needed to retrieve and interact with values.

Such differences at the data model level lead to pragmatic consequences of some significance at the level of the languages used to interact with application entities. In particular, while object-oriented data models naturally induce a navigational approach to accessing values, this leads to chains of reference of indefinite length that need to be traversed, or navigated.

In contrast, object-relational data models retain the associative approach introduced by the relational data model. Roughly speaking, this approach is based on viewing the sharing of values between attributes in different relations as inherently establishing associations between relation instances. These associations can then be exploited to access values across different relations without specifically and explicitly choosing one particular chain of references. These issues are exemplified in Sections III.B and III.C

Since object-relational models are basically hybrids, their notions of schema and instance combine features of both relational and object-oriented schemas and instances. The object-relational approach to modeling the *gene* entity type is the same as the object-oriented one depicted in Figs. 4 and 5 but for a few differences, e.g., relationships are usually not explicitly supported as such by object-relational data models.

As in the object-oriented case, object-relational data models were first advocated in a concerted manner by a manifesto (Stonebraker *et al.*, 1990) signed by prominent members of the research community. A book-length treatment is available in Stonebraker and Brown (1999). Any undergraduate textbook on Database Systems (e.g., Atzeni *et al.*, 1999; Elmasri and Navathe, 2000) can be used to complement the treatment of this and the section that follows.

```
CREATE SCHEMA genomics AUTHORIZATION DBA;
CREATE TABLE  genomics.DNA_sequence(
       sequence_id          VARCHAR(10)   NOT NULL
       accession_no         VARCHAR(10)   NOT NULL,
       protein_id           VARCHAR(10)   NOT NULL,
       organism_id          VARCHAR(10)   NOT NULL
       PRIMARY KEY (sequence_id),
       FOREIGN KEY (organism_id)
              REFERENCES genomics.organism(organism_id)
              ON DELETE CASCADE    ON UPDATE CASCADE );
CREATE TABLE  genomics.organism(
       organism_id          VARCHAR(10)   NOT NULL,
       common_name          VARCHAR(50),
       up_node              VARCHAR(30)
       PRIMARY KEY (organism_id) );
```

FIGURE 6 SQL declarations for the schemas in Fig. 1.

III. DATABASE LANGUAGES

A DBMS must support one or more languages with which application developers can set up, populate, scrutinize, and maintain data. In principle, different purposes are best served by different database languages, but in practice, DBMSs tend to use one language to cater for more than one purpose.

The kinds of **database languages** that must in principle be supported in DBMSs are:

1. **Data definition languages**, which are used to declare schemas (perhaps including application-specific integrity constraints)
2. **Data manipulation languages**, which are used to retrieve and manipulate the stored data

Data manipulation languages can be further categorized as follows:

1. **Query languages**, which most often are high-level, declarative, and computationally incomplete (i.e., not capable of expressing certain computations, typically due to the lack of support for updates or for recursion or iteration)
2. **Procedural languages**, which most often are low-level, imperative, and computationally complete

Besides these, most DBMSs provide interfacing mechanisms with which developers can implement applications in a general-purpose language (referred to in this context as a **host language**) and use the latter to invoke whatever operations are supported over the stored data (because they are part of the behavior of either the data types made available by the data model or the application types declared in the application schema).

A. Relational Database Languages

SQL is the ISO/ANSI standard for a relational database language. SQL is both a data definition and a data manipulation language. It is also both a query language and capable of expressing updates. However, SQL is not computationally complete, since it offers no support for either recursion or iteration. As a consequence, when it comes to the development of applications, SQL is often embedded in a host language, either one that is specific to the DBMS being used or a general purpose language for which a query language interface is provided.

Figure 6 shows how the schema illustrated in Fig. 1 can be declared in SQL. Notice that SQL includes constructs to declare integrity constraints (e.g., a referential integrity on from `DNA_sequence` to `organism`) and even an action to be performed if it is violated (e.g., cascading deletions, by deleting every referring tuple when a referenced tuple is deleted).

SQL can also express insertions, deletions and updates, as indicated in Fig. 7. Note in Fig. 7 that on insertion it is possible to omit null values by listing only the attributes for which values are being supplied. Note also that the order of insertion in Fig. 7 matters, since referential integrity constraints would otherwise be violated. Finally, note that, because of cascading deletions, the final statement will also delete all tuples in `DNA_sequence` that refer to the primary key of the tuple being explicitly deleted in `organism`.

After the operations in Fig. 7 the state depicted in Fig. 2 will have changed to that shown in Fig. 8.

As a relational query language, SQL always returns results that are themselves relation instances. Thus, the basic constructs in SQL cooperate in specifying aspects of the schema as well as the instantiation of a query result. Roughly, the `SELECT` clause specifies the names of attributes to appear in the result, the `FROM` clause specifies

```
INSERT INTO genomics.organism(organism_id, up_node)
VALUES      ('Poly. tinctorium', 'Polygonaceae');

INSERT INTO genomics.DNA_sequence
VALUES      ('AB003089', 'AB003089', 'BAA78708', 'Poly. tinctorium');

DELETE FROM genomics.organism
WHERE       organism_id = 'Bras. napus';
```

FIGURE 7 Using SQL to effect state transitions in the relation instances from Fig. 2.

the names of relations contributing data to the result, and the WHERE clause specifies, in terms of the relations (and their attributes) mentioned in the FROM clause, the conditions which each tuple in the result must satisfy. SQL queries tend to be structured around this combination of SELECT, FROM and WHERE clauses. Figure 9 shows example SQL queries.

Query RQ1 in Fig. 9 returns a unary table, each tuple of which records the organism_id of organisms that share the common_name of "white_clover." Query RQ2 returns a binary table relating each common_name found in the organism table with the protein_ids produced by their identified genes. Figure 10 shows the relations instances returned by RQ1 and RQ2.

SQL also supports aggregations (e.g., COUNT and AVG, which, respectively, count the number of tuples in a result and compute the average value of a numeric attribute in the result), groupings, and sorting.

Embedding SQL into a host language is another approach to retrieving and manipulating data from relational databases. Vendors typically provide a host language for SQL of which the fragment in Fig. 11 is an illustration. The fragment in Fig. 11 uses a CURSOR construct to scan the organism relation instance for organisms with no common name. When one is found, rather than leaving the value unassigned, the program uses the UPDATE construct to set the common_name attribute to the string None.

SQL is legally defined by ISO/ANSI standards which are available from those organizations. For comprehensive treatment, a good source is Melton and Simon, 1993. A detailed treatment of the relational algebra and calculi which underpin SQL can be found in Abiteboul *et al.* (1995).

B. Object-Oriented Database Languages

In the object-oriented case, the separation between the languages used for data definition, querying and procedural manipulation is more explicit than in the relational case. This is because in object-oriented databases, the syntactic style of SQL is circumscribed largely to the query part of the data manipulation language.

Also, rather than make use of vendor-specific host languages, object-oriented DBMSs either provide interfaces to general-purpose languages or else the DBMS itself supports a persistent programming language strategy to application development (i.e., one in which a distinction between the data space of the program and the database is deliberately not made, which leads to applications that need not explicitly intervene to transfer data from persistent to transient store and back again).

The *de facto* standard for object-oriented databases is the proposal by the ODMG consortium of vendors. The ODMG standard languages are ODL, for definition, and OQL (which extends the query part of SQL), for querying. The standard also defines interfaces for a few widely used general-purpose languages. Figure 4 could be declared in ODL as shown in Fig. 12.

Note how ODL allows inverse relationships to be named, as a consequence of which referential integrity is enforced in both directions.

Two OQL queries over the gene class in Fig. 12 are given in Fig. 13. Query OQ1 returns a set of complex values, i.e., name-cited pairs, where the first element is the standard_name of an instance of the gene class and the second is the list of strings stored as the value of the citation attribute for that instance. Query OQ2 returns the common_name of organisms associated with genes that have alleles. Note the use of the COUNT aggregation function over a collection value. Note, finally, the denotation g.organism_id.common_name (known as a *path expression*). Path expressions allow a navigational style of access.

For ODMG-compliant DBMSs, the authoritative reference on ODL and OQL is (Cattell *et al.*, 2000). A more

DNA_sequence

sequence_id	accession_no	protein_id	organism_id
TRBG361	X56734	AA40058	Trif. repens
AB003089	AB003089	BAA78708	Poly. tinctorium
...	...	...	...

organism

organism_id	common_name	up_node
Trif. repens	white clover	Papilionoideae
Poly. tinctorium	NULL	Polygonaceae
...	...	...

FIGURE 8 Updated relation instances after the commands in Fig. 7.

```
RQ1:   SELECT  organism_id
       FROM    organism
       WHERE   common_name = 'white clover';

RQ2:   SELECT  o.common_name, s.protein_id
       FROM    organism AS o, DNA_sequence AS s
       WHERE   o.organism_id = s.organism_id;
```

FIGURE 9 Using SQL to query the database state in Fig. 9.

formal treatment of some of the issues arising in object-oriented languages can be found in Abiteboul *et al.* (1995).

C. Object-Relational Database Languages

The proposed standard for object-relational database languages is SQL-99. Figure 14 shows how Fig. 4 could be specified in SQL-99. Note the use of `ROW TYPE` to specify a complex domain, the use of `REF` to denote tuple identifiers and the use of type constructors such as `SET` and `LIST`. Note also that, unlike ODL (cf. Fig. 12), in SQL-99 inverse relationships are not declared. Note, finally, how `gene` is modeled as including operations, as indicated by the keyword `FUNCTION` introducing the behavioral part of the specification of `gene`.

Two SQL-99 queries over the `gene` type in Fig. 14 are given in Fig. 15. Query `ORQ1` returns a binary table relating the `standard_name` of each gene with the `common_name` of `organisms` where the gene is found. Query `ORQ2` returns the `common_name` of `organisms` associated with genes that have `alleles`. Note that in SQL-99 path expressions use the symbol `->` to dereference identifiers and (not shown in Fig. 15) the symbol '..' to denote attributes in row types.

SQL-99 is legally defined by ISO/ANSI standards which are available from those organizations.

IV. ADVANCED MODELS AND LANGUAGES

The previous sections have described mainstream database models and languages, presenting facilities that are in widespread use today. This section introduces several extensions to the mainstream data models and languages with facilities that allow the database to capture certain application requirements more directly.

RQ1
organism_id
Trif. repens

RQ2	
common_name	protein_id
white clover	AA40058
rape	CAA57913
...	...

FIGURE 10 Results of the SQL queries in Fig. 9.

A. Deductive Databases

Deductive database systems can be seen as bringing together mainstream data models with logic programming languages for querying and analyzing database data. Although there has been research on the use of deductive databases with object-oriented data models, this section illustrates deductive databases in the relational setting, in particular making use of Datalog as a straightforward deductive database language (Ceri *et al.*, 1990).

A **deductive (relational) database** is a Datalog program. A **Datalog program** consists of an *extensional database* and an *intensional database*. An **extensional database** contains a set of facts of the form:

$$p(c_1, \ldots, c_m)$$

where p is a predicate symbol and $c_1, \ldots, c_m$ are constants. Each predicate symbol with a given arity in the extensional database can be seen as analogous to a relational table. For example, the table `organism` in Fig. 2 can be represented in Datalog as:

```
organism('Bras. napus', 'rape',
  'Brassicaceae',).
organism ('Trif. repens',
  'white clover', 'Papilionoideae',).
```

Traditionally, constant symbols start with a lower case letter, although quotes can be used to delimit other constants.

An **intensional database** contains a set of rules of the form:

$$p(x_1, \ldots, x_m) :- q_1(x_{11}, \ldots, x_{1k}), \ldots, q_j(x_{j1}, \ldots, x_{jp})$$

where p and q_i are predicate symbols, and all argument positions are occupied by variables or constants.

Some additional terminology is required before examples can be given of Datalog queries and rules. A **term** is either a constant or a variable—variables are traditionally written with an initial capital letter. An atom $p(t_1, \ldots, t_m)$ consists of a predicate symbol and a list of arguments, each of which is a term. A **literal** is an atom or a negated atom $\neg p(t_1, \ldots, t_m)$. A **Datalog query** is a conjunction of literals.

```
DECLARE  o_id            CHAR(10);
         o_common_name   CHAR(50);
         CURSOR organism_cursor
         IS     SELECT organism_id, common_name
                FROM organism;
BEGIN    OPEN   organism_cursor;
         LOOP   FETCH    organism_cursor
                INTO     o_id, o_common_name;
                EXIT WHEN organism_cursor%NOTFOUND;
                IF       o_common_name IS NULL
                THEN     UPDATE  organism
                         SET     common_name = 'None'
                         WHERE   organism_id = o_id;
         END LOOP;
         CLOSE  organism_cursor;
END;
```

FIGURE 11 Embedding SQL in a host language to effect state transitions.

```
class gene{
    (extent       Genes)
    attribute     string        standard_name ;
    attribute     struct        codon_structure {
                                string initiation_codon,
                                list<string> codon_sequence,
                                string termination_codon } sequence;
    attribute     string        function;
    attribute     list<string>  citation;
    attribute     protein       protein_id;
    attribute     organism      organism_id;
    relationship  set<alleles>  has_alleles
                  inverse       allele ::  is_allele_of;
          ... }
```

FIGURE 12 ODL to specify the gene class in Fig. 4.

```
OQ1:   SELECT  struct(name   :  g.standard_name
                      cited  :  g.citation)
       FROM    g IN gene;

OQ2:   SELECT  g.organism_id.common_name
       FROM    g AS gene
       WHERE   COUNT(g.has_alleles) > 0;
```

FIGURE 13 Using OQL to query the gene class in Fig. 12.

```
CREATE ROW TYPE  codon_structure(
                 initiation_codon     CHAR(3),
                 codon_sequence       LIST(CHAR(3)),
                 termination_codon    CHAR(3));
CREATE TABLE     codon_sequence       OF TYPE      codon_structure;

CREATE TYPE      gene(  standard_name VARCHAR(50),
                 sequence      REF (codon_sequence),
                 function      VARCHAR(30),
                 citation      LIST(VARCHAR(30)),
                 protein_id    REF(protein_tuple) REFERENCES (protein),
                 organism_id   REF(organism_tuple) REFERENCES (organism),
                 has_alleles   SET(REF(allele_tuple)) REFERENCES (allele),
                 FUNCTION ...
                 ... );
```

FIGURE 14 SQL-99 to specify the gene entity in Fig. 3.

```
ORQ1:  SELECT  standard_name, SET(organism_id -> common_name)
       FROM    gene;

ORQ2:  SELECT  g.organism_id -> common_name
       FROM    gene AS g
       WHERE   (SELECT COUNT(a) FROM a IN g.has_alleles)) > 0;
```

FIGURE 15 Using SQL-99 to query the gene type in Fig. 14.

Figure 16 shows some queries and rules expressed in Datalog. The queries DQ1 and DQ2 are expressed using a set comprehension notation, in which the values to the left of the | are the result of the Datalog query to the right of the |.

Query DQ1 retrieves the organism identifier of 'white clover'. Informally, the query is evaluated by unifying the query literal with the facts stored in the extensional database. Query DQ2 retrieves the identifiers of the proteins in 'white clover'. However, the predicate species_protein representing the relationship between the common name of a species and the identifiers of its proteins is not part of the extensional database, but rather is a rule defined in the intensional database. This rule, DR1 in Fig. 16, is essentially equivalent to the SQL query RQ2 in Fig. 9, in that it retrieves pairs of species common names and protein identifiers that satisfy the requirement that the proteins are found in the species. In DR1, the requirement that a dna_Sequence be found in an organism is represented by the atoms for dna_Sequence and organism sharing the variable OID, i.e., the organism identifier of the organism must be the same as the organism identifier of the dna_Sequence.

Deductive databases can be seen as providing an alternative paradigm for querying and programming database applications. A significant amount of research has been conducted on the efficient evaluation of recursive queries in deductive languages (Ceri *et al.*, (1990), an area in which deductive languages have tended to be more powerful than other declarative query languages. Recursive queries are particularly useful for searching through tree or graph structures, for example in a database representing a road network or a circuit design. A further feature of deductive languages is that they are intrinsically declarative, so programs are amenable to evaluation in different ways—responsibility for finding efficient ways of evaluating deductive programs rests as much with the query optimizer of the DBMS as with the programmer.

Although there have been several commercial deductive database systems, these have yet to have a significant commercial impact. However, some of the features of deductive languages, for example, relating to recursive query processing, are beginning to be adopted by relational vendors, and are part of the SQL-99 standard.

B. Active Databases

Traditional DBMSs are **passive** in the sense that commands are executed by the database (e.g., query, update, delete) as and when requested by the user or application program. This is fine for many tasks, but where a DBMS simply executes commands without paying heed to their consequences, this places responsibilities on individual programmers that they may struggle to meet (Paton and Diaz, 1990).

For example, imagine that it is the policy of an organization managing a repository of DNA sequence data that whenever a new sequence is provided with a given sequence_id, the previous sequence is recorded, along with the date when the change takes place and the name of the user making the update. This straightforward policy may, however, be difficult to enforce in practice, as there may be many different ways in which a new replacement can be provided—for example, through a program that accepts new information from a WWW interface, or through an interactive SQL interface. The policy should be enforced in a consistent manner, no matter how the sequence has come to be updated. In an **active database**, the database system is able to respond automatically to

```
DQ1:   { OID | organism(OID, 'white clover', _) }

DR1:   species_protein(CommonName, ProteinId) :-
              organism(OID, CommonName, _),
              dna_Sequence(_, _, ProteinId, OID).

DQ2:   { ProteinId | species_protein('white clover', ProteinId) }
```

FIGURE 16 Using datalog to query the database state in Fig. 8.

events of relevance to the database, such as the updating of a table, and thus the database can provide centralized support for policies such as those described earlier.

Active behavior is described using **event-condition-action** or **ECA** rules. These rules have three components:

1. **Event:** a description of the happening to which the rule is to respond. In the example above, this could be the updating of the relation storing the sequence data.
2. **Condition:** a check on the context within which the event has taken place. The condition is either a check on the parameters of the event or a query over the database.
3. **Action:** a program block that indicates the reaction that the system should take to the event in a context where the condition is true.

Active rules will now be illustrated using the example described above, based around a table `sequence` with attributes `sequence_id` and `sequence`. A record will be kept of modified sequences in the table `modified_sequence`, with attributes `sequence_id`, `old_sequence`, `date_of_change`, `user` and `update_type`.

The active rule in Fig. 17 is written using the SQL-99 syntax. In the SQL standard, active rules are known as **triggers**. The event the trigger is monitoring is `after update of sequence on sequence`, which is raised after any change to the `sequence` attribute of the table `sequence`. The tuple prior to the update taking place is referred to within the trigger as `oldseq`. The example is of a `row` trigger, which means that the trigger responds to updates to individual tuples, so if a single update statement modifies 10 rows in the `sequence` table, the trigger will be executed 10 times. In this example there is no condition—the action is executed regardless of the state of the database, etc.

Writing correct triggers is not altogether straightforward. For the example application, additional triggers would be required, for example, to log the deletion of an existing sequence entry, so that a modification of a sequence as a delete followed by an insert is not omitted from the log. Considerable skill is required in the development and maintenance of large rule bases. However, it is clear that active rules can be used for many different purposes—supporting business rules, enforcing integrity constraints, providing fine tuned authorization mechanisms, etc. As a result, the principal relational database products support active mechanisms, and the SQL-99 standard should encourage consistency across vendors in due course.

V. DISTRIBUTION

A conventional DBMS generally involves a single **server** communicating with a number of user **clients**, as illustrated in Fig. 18. In such a model, the services provided by the database are principally supported by the central server, which includes a secondary storage manager, concurrency control facilities, etc., as described in Section I.

In relational database systems, clients generally communicate with the database by sending SQL query or update statements to the server as strings. The server then compiles and optimizes the SQL, and returns any result or error reports to the client.

Clearly there is a sense in which a client-server database is distributed, in that application programs run on many clients that are located in different parts of the network. However, in a client-server context there is a single server managing a single database described using a single data model. The remainder of this section introduces some of the issues raised by the relaxation of some of these restrictions.

A. Distributed Databases

In a distributed database, data is stored in more than one place, as well as being accessed from multiple places, as

```
create trigger logUpdate
after update of sequence on sequence
referencing old as oldseq
for each row
begin
       insert into modified_sequence values (
              oldseq.sequence_id,
              oldseq.sequence,
              SYSDATE,
              USER,
              'delete');
end
```

FIGURE 17 Using an active rule to log changes to the `sequence` table.

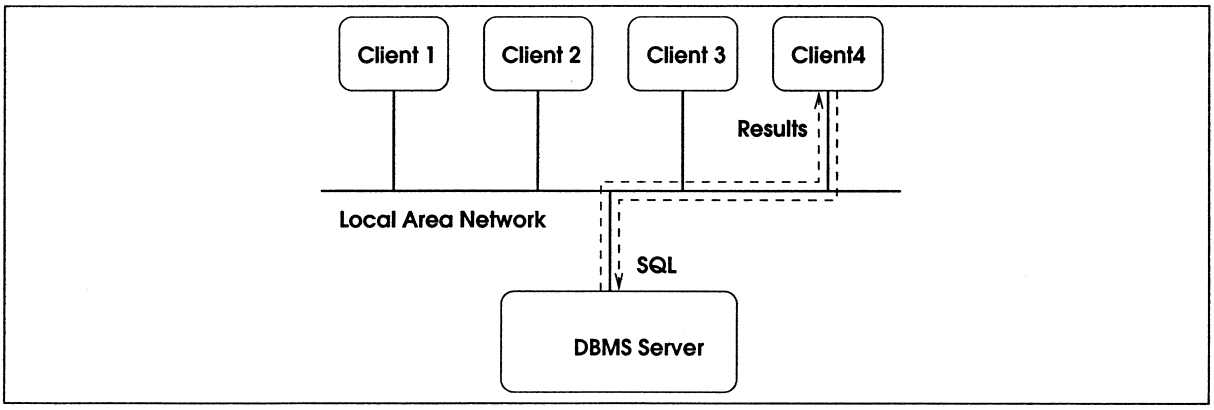


FIGURE 18 A client-server database architecture.

in the client-server model. A typical distributed DBMS architecture is illustrated in Fig. 19, in which there are two servers, each of which is accessed by multiple clients.

Distributed databases can come into being for different reasons. For example, a multinational organization may have distinct databases for each of its national subsidiaries, but some tasks may require access to more than one of these databases (e.g., an item that is out of stock in one country may be available in another). Another situation in which multiple databases may have to be used together is where organizations merge, and it becomes necessary to support some measure of interoperation across their independently developed information systems.

The context within which a distributed database comes into being significantly influences the complexity of the resulting system. The nature of a distributed database can be characterized by the following features:

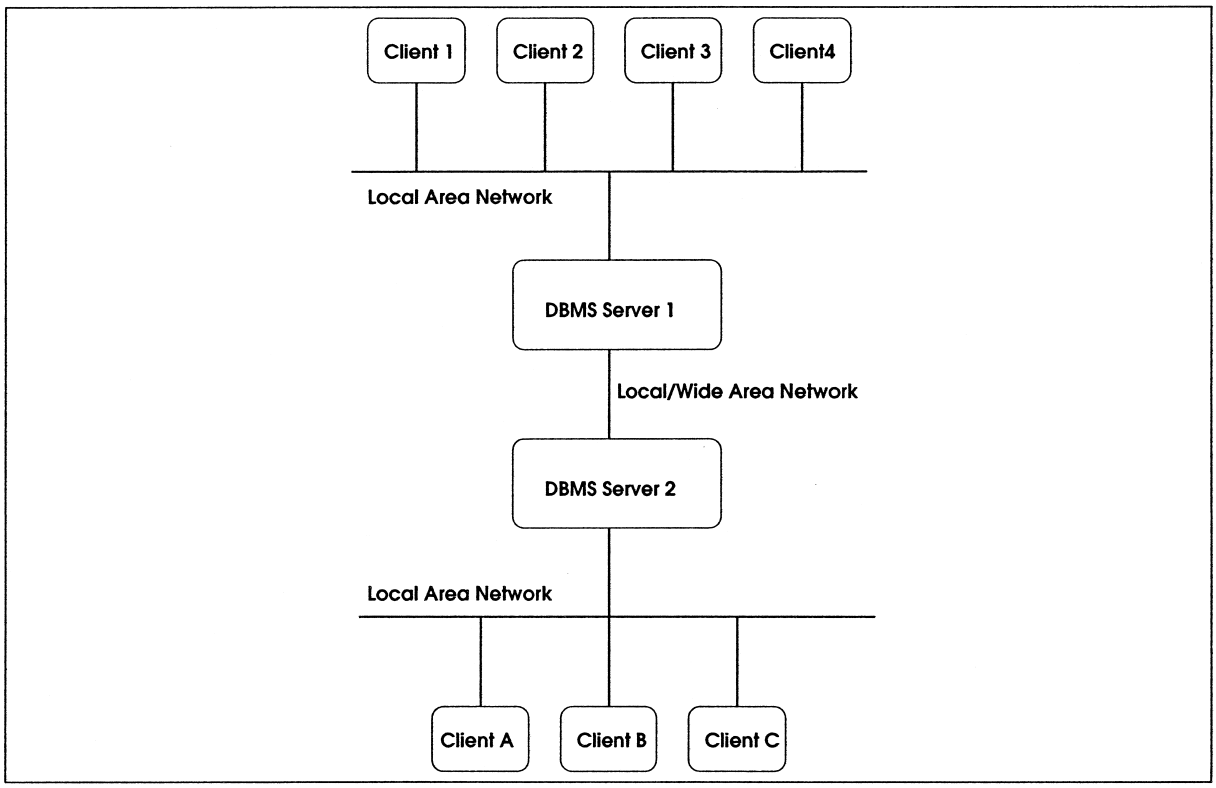


FIGURE 19 A distributed database architecture.

1. **Fragmentation.** The fragmentation of data in a distributed database describes how individual data items are partitioned across the database servers. For example, in the genomic database of Fig. 1, one form of fragmentation could lead to all the sequences from an `organism` being stored on the same server. This might happen, for example, if the sequencing of an organism is being carried out at a single site. This form of fragmentation is known as **horizontal fragmentation**, as the tuples in the tables `organism` and `DNA_sequence` are partitioned on the basis of their `organism_id` values. In **vertical fragmentation**, some of the attributes of a table are placed on one server, and other attributes on another.

   In many applications, it is difficult to arrange a straightforward fragmentation policy—for example, if distributed databases are being used where two organizations have recently merged, it is very unlikely that the existing databases of the merging organizations will have made use of similar fragmentation schemes.
2. **Homogeneity.** In a **homogeneous** distributed database system, all the servers support the same kind of database, e.g., where all the servers are relational, and in particular where all the servers have been provided by the same vendor. It is common, however, for more than one database paradigm to be represented in a distributed environment, giving rise to a **heterogeneous** distributed database.
3. **Transparency.** In Fig. 19, a user at `Client 1` can issue a request that must access or update data from either of the servers in the distributed environment. The level of transparency supported indicates the extent to which users must be aware of the distributed environment in which they are operating (Atzeni *et al.*, 1999). Where there is **fragmentation transparency**, requests made by the user need not take account of where the data is located. Where there is **replication transparency**, requests made by the user need not take account of where data has been replicated in different servers, for example, to improve efficiency. Where there is **language transparency**, the user need not be concerned if different servers support different query languages. The greater the level of transparency supported, the more infrastructure is required to support transparent access. Distributed query processing and concurrency control present particular challenges to database systems developers (Garcia-Molina *et al.*, 2000).

B. Data Warehouses

Traditional database systems have tended to be designed with an emphasis on **on line transaction processing** (OLTP), in which there are large numbers of short transactions running over the database. In our biological example, a typical transaction might add a new sequence entry into the database. Such transactions typically read quite modest amounts of data from the database, and update one or a few tuples.

However, the data stored in an organization's databases are often important for management tasks, such as decision support, for which typical transactions are very different from those used in OLTP. The phrase **on line**

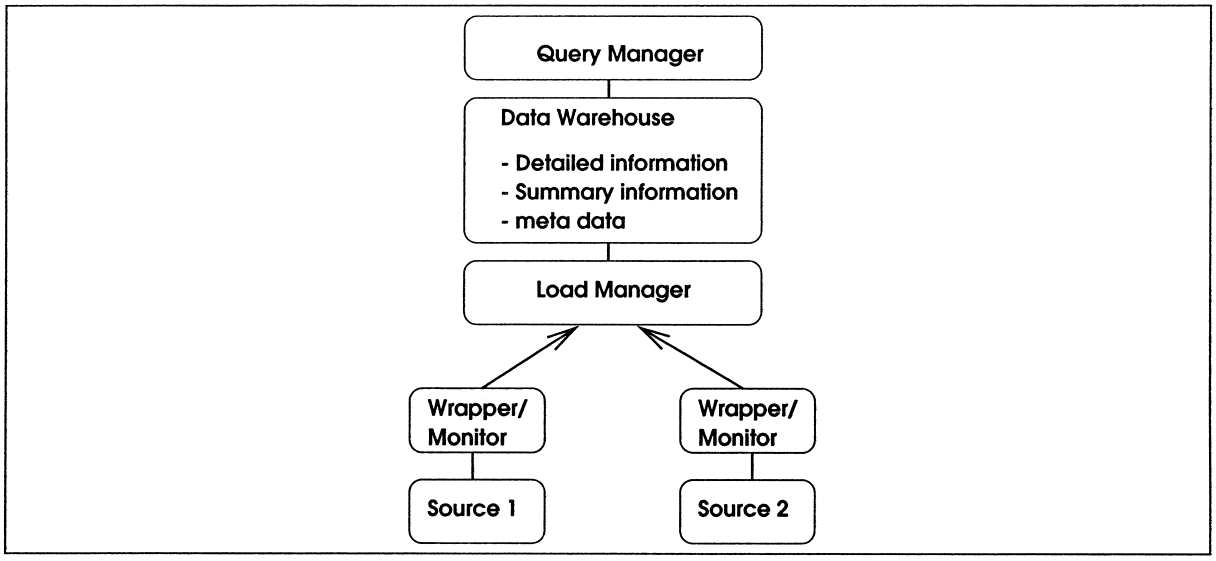


FIGURE 20 A data warehouse architecture.

analytical processing (OLAP) has been used to characterize this different kind of access to databases, in which typical transactions read rather than update the database, and may perform complex *aggregation* operations on the database. As the data required for OLAP often turn out to be stored in several OLTP databases, it often proves to be necessary to replicate data in a system designed specifically to support OLAP—such a replicated store is known as a **data warehouse**.

A data warehouse architecture is illustrated in Fig. 20 (Anahory and Murray, 1997). A data warehouse contains three principal categories of information: detailed information (for example, in the genomic context, this may be the raw sequence data or other data on the function of individual protein); summary information (for example, in the genome context, this may record the results of aggregations such as the numbers of genes associated with a given function in each species); and meta data (which describes the information in the warehouse and where it came from).

An important prerequisite for the conducting of effective analyses is that the warehouse contains appropriate data, of adequate quality, which is sufficiently up-to-date. This means that substantial effort has to be put into loading the data warehouse in the first place, and then keeping the warehouse up-to-date as the sources change. In Fig. 20, data essentially flow from the bottom of the figure to the top. Sources of data for the warehouse, which are often databases themselves, are commonly **wrapped** so that they are syntactically consistent, and **monitored** for changes that may be relevant to the warehouse (e.g., using active rules). The **load manager** must then merge together information from different sources, and discard information that doesn't satisfy relevant integrity constraints. The **query manager** is then responsible for providing comprehensive interactive analysis and presentation facilities for the data in the warehouse.

VI. CONCLUSIONS

This chapter has provided an introduction to databases, and in particular to the models, languages, and systems that allow large and complex data-intensive applications to be developed in a systematic manner. DBMSs are now quite a mature technology, and almost every organization of any size uses at least one such system. The DBMS will probably be among the most sophisticated software systems deployed by an organization.

This is not to say, however, that research in database management is slowing down. Many data-intensive applications do not use DBMSs, and developments continue, with a view to making database systems amenable for use in more and more different kinds of application. For example, spatial, temporal, multimedia, and semistructured data are difficult to represent effectively in many database systems, and many issues relating to distributed information management continue to challenge researchers.

SEE ALSO THE FOLLOWING ARTICLES

DATA MINING AND KNOWLEDGE DISCOVERY • DATA STRUCTURES • DESIGNS AND ERROR-CORRECTING CODES • INFORMATION THEORY • OPERATING SYSTEMS • PROJECT MANAGEMENT SOFTWARE • REQUIREMENTS ENGINEERING (SOFTWARE) • SOFTWARE ENGINEERING • SOFTWARE MAINTENANCE AND EVOLUTION • WWW (WORLD-WIDE WEB)

BIBLIOGRAPHY

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*, Addison-Wesley. ISBN 0-201-53771-0.

Anahory, S., and Murray, D. (1997). *Data Warehousing in the Real World*, Addison-Wesley. ISBN 0-201-17519-3.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. B. (1990). "The Object-Oriented Database System Manifesto," *In* [(Kim *et al.*, 1990)], pp. 223–240.

Atzeni, P., Ceri, S., Paraboschi, S., and Torlone, R. (1999). *Database Systems: Concepts, Languages and Architectures*, McGraw-Hill.

Cattell, R. G. G., Barry, D. K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., and Velez, F. (2000). *The Object Data Standard*: *ODMG 3.0*, Morgan Kaufman, ISBN 1-55860-647-5.

Ceri, S., Gottlob, G., and Tanca, L. (1990). *Logic Programming and Databases*, Springer-Verlag, Berlin. ISBN 0-387-51728-6.

Codd, E. F. "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* **13**(6): 377–387, June 1970; Also in CACM **26**(1) January 1983 pp. 64–69.

Elmasri, R., and Navathe, S. B. (2000). *Fundamentals of Database Systems*, Addison-Wesley, Reading, MA, USA, 3rd. edition, ISBN 0-201-54263-3.

Garcia-Molina, H., Ullman, J., and Widom, J. (2000). *Database System Implementation*, Prentice Hall. ISBN 0-13-040264-8.

Kim, W., Nicolas, J.-M., and Nishio, S. (eds.). (1990). *Deductive and Object-Oriented Databases* (*First International Conference DOOD'89*, *Kyoto*), Amsterdam, The Netherlands, Elsevier Science Press (North-Holland), ISBN 0-444-88433-5.

Melton, J., and Simon, A. R. (1993). *Understanding the New SQL: A Complete Guide*, Morgan Kaufman, ISBN 1-55860-245-3.

Paton, N., and Diaz, O. (1999). "Active Database Systems," *ACM Computing Surveys* **1**(31), 63–103.

Stonebraker, M., and Brown, P. (1999). *Object-Relational DBMS*: *Tracking the Next Great Wave*, Morgan Kaufman, ISBN 1-55860-452-9.

Stonebraker, M., Rowe, L. A., Lindsay, B. G., Gray, J., Carey, M. J., Brodie, M. L., Bernstein, P. A., and Beech, D. "Third-Generation Database System Manifesto—The Committee for Advanced DBMS Function," *SIGMOD Record* **19**(3): 31–44, September 1990.

Evolutionary Algorithms and Metaheuristics

Conor Ryan
University of Limerick

GLOSSARY

Chromosome A strand of DNA in the cell nucleus that carries the genes. The number of genes per chromosome and number of chromosomes per individual depend on the species concerned. Often, evolutionary algorithms use just 1 chromosome; humans, for example, have 46.

Cossover Exchange of genetic material from two parents to produce one or more offspring.

Genotype The collective genetic makeup of an organism. Often in evolutionary algorithms, the genotype and the chromosome are identical, but this is not always the case.

Hill climber A metaheuristic that starts with a potential solution at a random place in the solution landscape. Random changes are made to the potential solution which, in general, are accepted if they improve its performance. Hill climbers tend to to find the best solution in their immediate neighborhood.

Fitness The measure of how suitable an individual or point in the solution landscape is at maximizing (or minimizing) a function or performing a task.

Local optima An point in a solution landscape which is higher, or fitter, than those in its immediate neighborhood.

Mutation A small, random change to an individual's genotype.

Neighborhood The area immediately around a point in the solution landscape.

Phenotype The physical expression of the genotype, resulting from the decoding and translation of the genes. Examples of phenotypes are eye color and hair colour.

Population A group of potential solutions searching a solution landscape in parallel. Typically, a population of individuals is manipulated by an evolutionary algorithm.

Recombination See crossover.

Reproduction Copying of an individual unchanged into

the next generation. The probability that an individual will reproduce (and thus have a longer lifetime) is directly proportional to its fitness.

Selection In evolutionary algorithms, individuals in a new generation are created from parents in the previous generation. The probability that individuals from the previous generation will be used is directly proportional to their fitness, in a similar manner to Darwinian survival of the fittest.

Solution landscape A map of the fitness of every possible set of inputs for a problem. Like a natural landscape, a solution landscape is characterized by peaks, valleys, and plateaus, each reflecting how fit its corresponding inputs are.

Weak artificial intelligence An artificial intelligence (AI) method that does not employ any information about the problem it is trying to solve. Weak AI methods do not always find optimal solutions, but can be applied to a broad range of problems.

METAHEURISTICS are search techniques that explore the solution landscape of a problem or function in an attempt to find an optimal solution or maximum/minimum value. Typically, a metaheuristic starts from a randomly chosen point and explores the landscape from there by modifying the initial solution. In general, if the modified solution is better than its predecessor, it is retained and the original one is discarded. This is not always the case, however, as sometimes slight deterioration of the initial solution is accepted, in the hope that it might lead to some long-term gain. This process is then repeated, often thousands or more times, until a satisfactory solution has been generated. Some metaheuristic methods are referred to as *hill climbers* because of the way they randomly start with a point on the solution landscape and proceed to find the best solution (the highest peak) in the neighborhood.

Evolutionary algorithms are a special case of metaheuristics which maintain a *population* of potential solutions and use a variety of biologically inspired mechanisms to produce new populations of solutions which inherit traits from the initial population. Like the hill-climbing metheuristics mentioned above, the process is repeated many times until a satisfactory solution has been discovered.

The advantage of using metaheuristics over other search methods in some problems is that one often need know very little about the task at hand. Furthermore, these search methods often produce counterintuitive results that a human would be unlikely to have derived.

I. METAHEURISTICS VERSUS ALGORITHMS

A. Introduction

An algorithm is a fixed series of instructions for solving a problem. Heuristics, on the other hand, are more of a "rule of thumb" used in mathematical programming and usually mean a procedure for seeking a solution, but without any guarantee of success. Often, heuristics generate a reasonably satisfactory solution, which tends to be in the neighborhood of an optimal one, if it exists.

Heuristics tend use domain-specific knowledge to explore landscapes, which is usually given by an expert in the area. This renders them less than useful when applied to other problem areas, as one cannot expect them to be general enough to be applied across a wide range of problems. Metaheuristics, on the other hand, operate at a higher level than heuristics, and tend to employ little, if any, domain-specific knowledge.

The absence of this knowledge qualifies metaheuristics as *weak* methods in classic artificial intelligence parlance. Although they have little knowledge which they can apply to the problem area, they are general enough to be applicable across a broad range of problems. Metaheuristics can be broadly divided into two groups: the first continuously modifies a single potential solution until it reaches a certain performance threshold; the second employs a *population* of candidate solutions which is effectively *evolved* over time until one of the candidates performs satisfactorily.

B. Solution Landscapes

Often, when discussing metaheuristics, or indeed, any search method, one refers to the *solution landscape*. Simply put, the solution landscape for a problem is a mapping of every set of inputs to their corresponding output. Consider the graph in Fig. 1, which shows the solution landscape for the function $X - X^2$. Clearly, the maximum value for the function is at $X = 0$, and any search method should quickly locate the area in which this value lies and then slowly home in on it.

Not all solution landscapes have such a simple shape, however. Figure 2 gives the landscape for the function

$$F^2(x) = \exp\left[-2\log(2) * \left(\frac{x - 0.1}{0.8}\right)^2\right] * \sin^6(5\pi x).$$

This function contains five peaks, each at different heights. Landscapes such as these can cause difficulties for searches because if one of the lower peaks is mistakenly identified as the optimal value, it is unlikely that the search

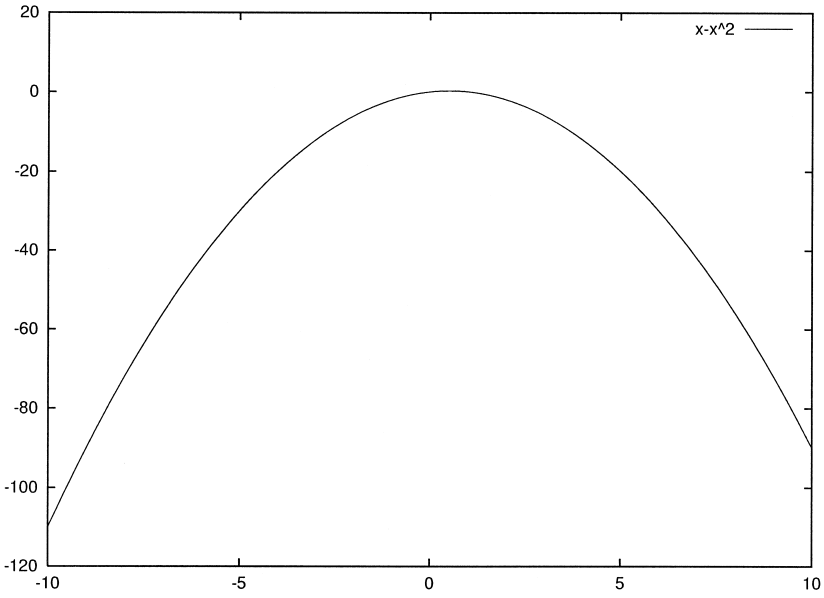


FIGURE 1 Solution landscape for $X - X^2$.

will be able to jump to the correct peak. These lower peaks are known as *local optima* because, although they are higher than their immediate neighbors, they are still lower than the global optimum. This underlines the crucial difference between an algorithm and a heuristic. One would never expect such uncertainties when employing an algorithm, while such vagueness is part and parcel when employing heuristics. Finally, Fig. 3 shows a still more complex landscape. In this case we have two input variables, leading to a three-dimensional representation of the landscape. This landscape was generated by the function

$$f(x, y) = 200 - (x^2 + y - 11)^2 - (x + y^2 - 7)^2$$

and contains four equal peaks. This presents yet another complication for a search method, as it will often be desirable to discover all peaks in a landscape, and there is often no simple way to discourage a search method from continuously searching the same area.

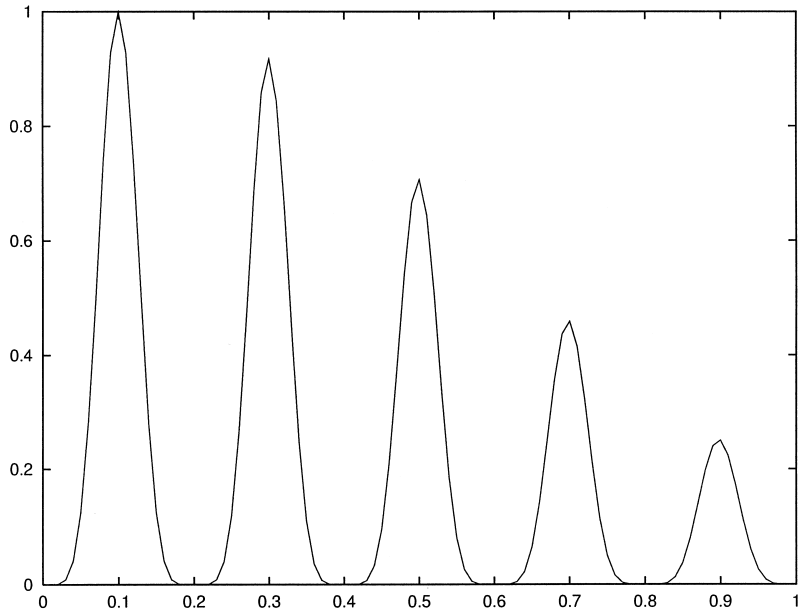


FIGURE 2 A solution landscape with several local optima.

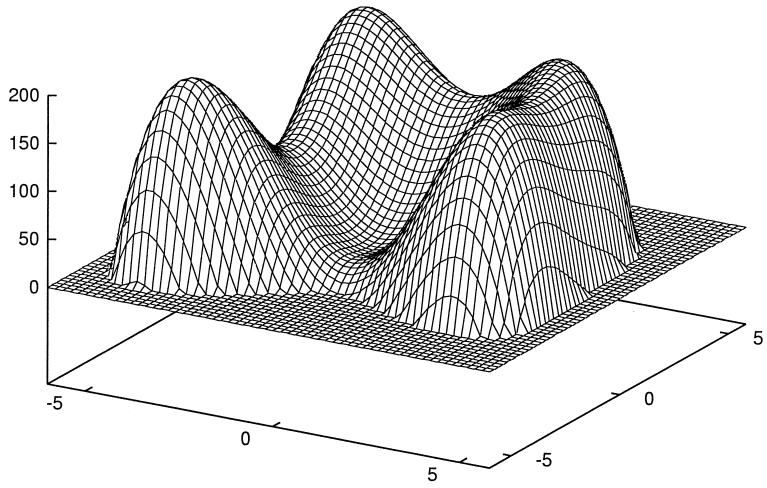


FIGURE 3 A more complex solution landscape.

We will see in the following sections that the task of the search function is to identify the areas in the landscape which are most likely to contain global optima. Once these areas have been identified, the search must then narrow to find the highest peak in the neighborhood.

II. EVOLUTIONARY ALGORITHMS

Evolutionary algorithms are metaheuristics that typically maintain populations of potential solutions. These populations simulate the evolution of individual strutures through the processes of selection, reproduction, recombination, and mutation. Each individual structure, more commonly referred to simply as an individual, in a population represents one possible solution to the problem at hand. Generally, each individual is considered to have a *chromosome* which consists of a number of genes that encode the individual's behavior. What the genes encode is entirely dependent on the problem at hand, and we will see later that different evolutionary algorithms use different encoding schemes and are often suited to very different problem domains. Thus, when trying to maximize or minimize a function, it would be appropriate to represent a real number with an individual's chromosome, while if one were trying to generate control routines for a robot, it would be more appropriate to encode a program.

The genes of an individual are collectively known as its *genotype*, while their physical expression, or result of decoding, is referred to as the *phenotype*. Some EAs exploit the mapping process from genotype to phenotype, while others rely on the fact that the phenotype is directly encoded, i.e., the genotype and the phenotype are the same.

Based on the Darwinian process of "survival of the fittest," those individuals that perform best at the problem survive longer and thus have more of an opportunity to contribute to new offspring. While the exact manner in which individuals are combined to produce new individuals depends on both the particular evolutionary algorithm being employed and the type of structures being manipulated, most evolutionary algorithms follow a series of steps similar to that in Fig. 4.

The origin of life in genetic algorithms happens in a somewhat less romantic fashion than the sudden spark which gave rise to life from the primordial ooze on earth. In a manner not unlike that suggested by the theory of "directed panspermia,"[1] the implementor of a genetic algorithm seeds the initial population with an appropriate

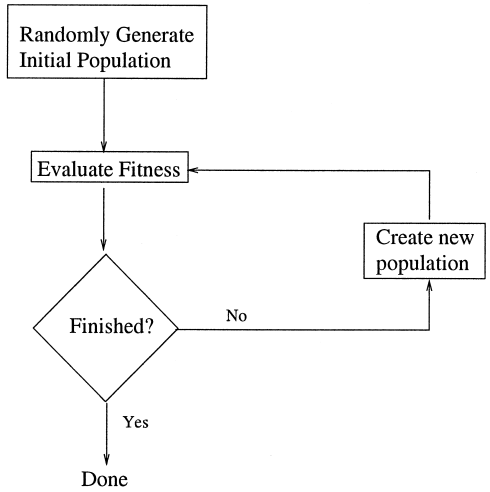


FIGURE 4 Flowchart for a simple evolutionary algorithm.

[1] *Panspermia* is the belief that the life on earth derives from "seeds" of extraterrestial origin. According to the notion of *directed panspermia*, these "seeds" were deliberately sent out by intelligent beings.

diversity of genetic material on which evolution operates. This initial seeding generally involves the encoding of every possible phenotypic trait into binary form, which then serves as the genotype.

Testing involves determining how suitable each individual is for solving the problem. Each individual is assigned a *fitness* measure that reflects how well adapted it is to its environment, i.e., how good it is at solving the problem. The more fit an individual is, the more likely it is to be *selected* to contribute to the new population, often referred to as the next *generation*. There are a number of different approaches to the selection scheme, but, in general, the overall effect is that the more fit an individual is, the more likely it is to be selected.

Once selected, there are a number of operators that can be applied to an individual. One such operator is *reproduction*, which involves copying the individual unchanged into the next generation; thus a more fit individual will have a greater probability of a long lifespan than less fit individuals. Another operator is the *mutation* operator. Mutation involves randomly changing one part of an individual's chromosome and then copying the resulting new individual into the next generation. The exact implementation of mutation depends on the representation scheme being employed, and is detailed below.

The final operator, which is also highly implementation-specific, is the *crossover* or *recombination* operator, and (usually) involves two individuals. Crossover is analogous to sexual reproduction, and involves two individuals exchanging genetic material to produce one or more offspring. Although the exact nature of the operator depends on the representation, the overall effect is similar to that in Fig. 5. In this case, two parents exchange different amounts of genetic material to produce two children. Two children are often produced to ensure that no genetic material is lost. However, more often than not, nature takes no such steps, and any extraneous genetic material is simply discarded.

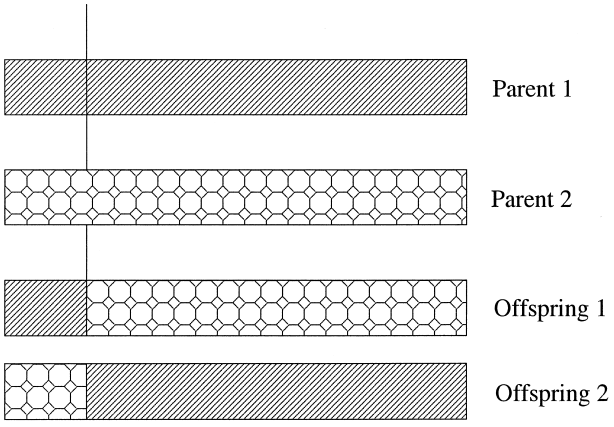


FIGURE 5 Flowchart for a simple evolutionary algorithm.

A common view is that mutation is an *explorative* operator, in that a random change in an individual's genes can cause it to jump to a previously unexplored region in the solution landscape. It then follows that crossover is an *exploitative* operator, as it effectively concentrates the search in an area already occupied by individuals. However, it is a matter of some debate as to which of the operators is the more important, and there is considerable disagreement in particular on the role played by mutation. Some of the evolutionary algorithms discussed below have very different views on the importance of each. For example, genetic programming generally does not employ any mutation, while evolutionary programming does not employ any crossover. It is perhaps indicative of the complexity of the evolutionary processes that no consensus has been reached.

What all proponents of evolutionary algorithms would agree on, however, is that evolutionary search is *not* random search. Although the initial population is random, further operations are only performed on the more fit individuals, which results in a process that is distinctly nonrandom and produces structures that far outperform any generated by a strictly random process.

A. Genetic Algorithms

Genetic algorithms are possibly the simplest evolutionary algorithms and deviate the least from the basic description given above. Genetic algorithms tend to use fixed-length binary strings to represent their chromosomes, which are analogous to the base-4 chromosomes employed in DNA. However, there is no particular reason why chromosome length should be fixed, nor why a representation scheme other than binary could not be used. Figure 6 shows an example of chromosomes of different lengths being crossed over.

Genetic algorithms are most often used in function maximization (or minimization) or object selection problems. In a function maximization problem, the algorithm must find a value for x that maximizes the output of some function $f(x)$. The chromosomes of individuals are thus simple binary (or possibly gray-coded) numbers, and the fitness for each individual is the value the function generates with it as the input. Consider the sample individuals in Table I, which are trying to maximize the problem $f(x) = X^3 - X^2 + X$. In this case, we are using five-bit numbers, the first-bit of which is used as a sign.

In this extremely small population, the individual 11011 is clearly the best performer, but the rule for selection is simply that the greater the fitness of an individual, the more likely it is to be selected. This suggests that even the relatively poor individual 10011 should have some chance, too. The simplest selection scheme is "roulette wheel" selection. In this scheme, each individual is assigned a

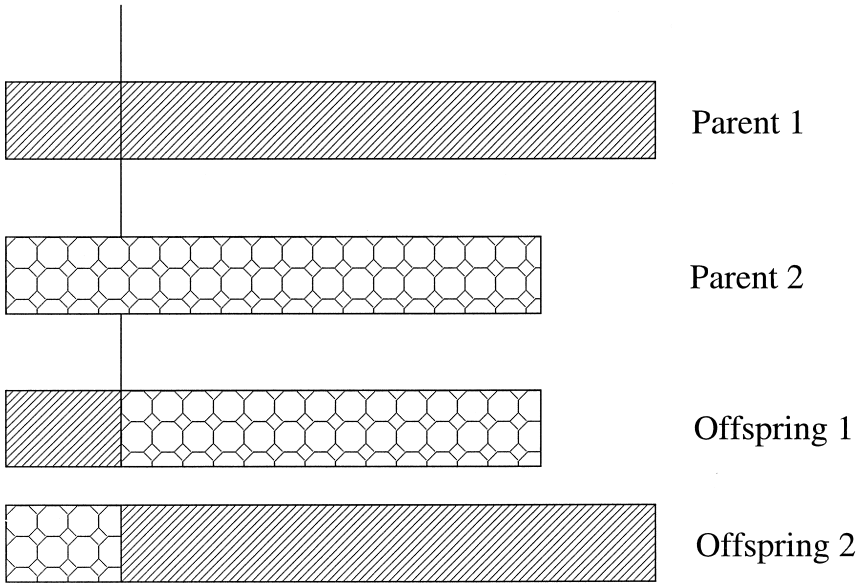


FIGURE 6 Variable-length crossover.

"slice" of a roulette wheel—the more fit the individual, the larger the slice. The size of this slice, which is effectively the probability that an individual will be selected, is calculated using the formula

$$P(x) = \frac{f_x}{\sum f_i}. \quad (1)$$

That is, the probability that an individual will be selected is equal to the individual's fitness divided by the total fitness.

The total fitness is 30,977, so the probability that the fittest individual will be selected is 14,509/30,977, or .468, while the probability that the least fit individual will be selected is just 69/30977, or .002. This ensures that although the less fit individuals have some chance of contributing to the next generation, it is far more likely to be the more fit individuals who contribute their genes. This is analogous to individuals in a population competing for resources, where the more fit individuals attain more.

In an experiment such as this, the flowchart in Fig. 4 is repeated until there is no longer change in the population, which means that no more evolution is taking place.

TABLE I Example Individuals from a Simple Genetic Algorithm Population

Individual	Binary	Decimal	Fitness
1	01010	10	9,910
2	11011	−11	14,509
3	01001	9	6,489
4	10011	−3	69

Types of Problems

Genetic algorithms are typically used on function optimization problems, where they are used to calculate which input yields the maximum (or minimum) value. However, they have also been put to use on a wide variety of problems. These range from game-playing, where the genotype encodes a series of moves, to compiler optimization, in which case each gene is an optimization to be applied to a piece of code.

B. Genetic Programming

Genetic programming (GP) is another well-known evolutionary algorithm, and differs from the genetic algorithm described above in that the individuals it evolves are *parse trees* instead of binary strings. The benefit of using parse trees is that individuals can represent programs, rather than simply represent numbers.

A simple parse tree is illustrated in Fig. 7. The nodes of a tree contain functions, such as + or −, while the leaves contain the arguments to that function. Notice that, because all functions in a parse tree return a value, it is possible to pass the result of a function as the argument to another function. For example, the expression **a** + **b** ∗ **c** would be represented as in Fig. 7.

Parse trees also have a convenient linear representation, identical to that used in the language Lisp. For example, the left tree in Fig. 7 would be (+ **1 2**), i.e., the function followed by its arguments, while the right tree in Fig. 7 would be described by (+ **a** (∗ **b c**)). Because of the close relationship between Lisp and parse trees and the

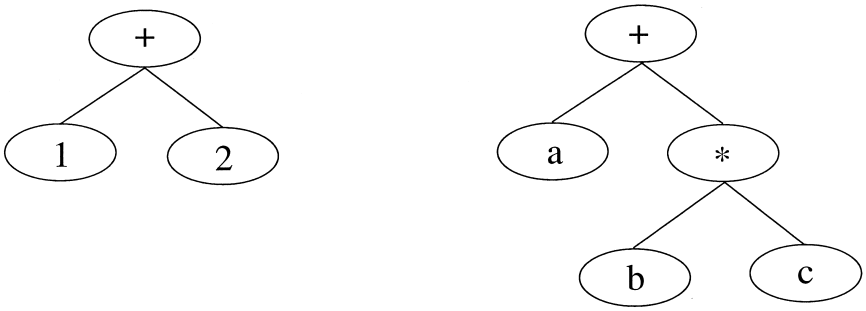


FIGURE 7 Two simple parse trees.

convenient manner in which the latter can be represented in Lisp, many users of GP implement their system in Lisp.

1. Crossover in Genetic Programming

The tree-based representation scheme employed by GP requires special operators for crossover. In this case, a subtree is selected from each parent, and the two subtrees are then swapped, as in Fig. 8. Individuals in GP enjoy a property known as *closure*, that is, any terminal, integer or variable, can be passed to any function. Similarly, the return type of any function can be legally passed as an argument to any other function. This property ensures that, regardless of the way in which individuals are combined, the offspring will be a legal tree. The offspring in Fig. 8 are of a different length from their parents, and this is an important trait of GP. It is highly unlikely that one will know the length of a solution in advance, and this property permits individuals to grow and contract as necessary.

2. Initial Population

In common with most evolutionary algorithms, the initial population in GP is randomly generated. However, due to the variable sizes of the individuals, there are a number of methods for producing this initial population. One such method, the *grow* method, randomly chooses which functions and terminals to put in. This can result in a variety of shapes and sizes of trees, as the choice of a function will cause an individual to keep growing. Usually, there is an upper limit set on the depth of individuals, which, once encountered, forces the inclusion of a terminal. Each

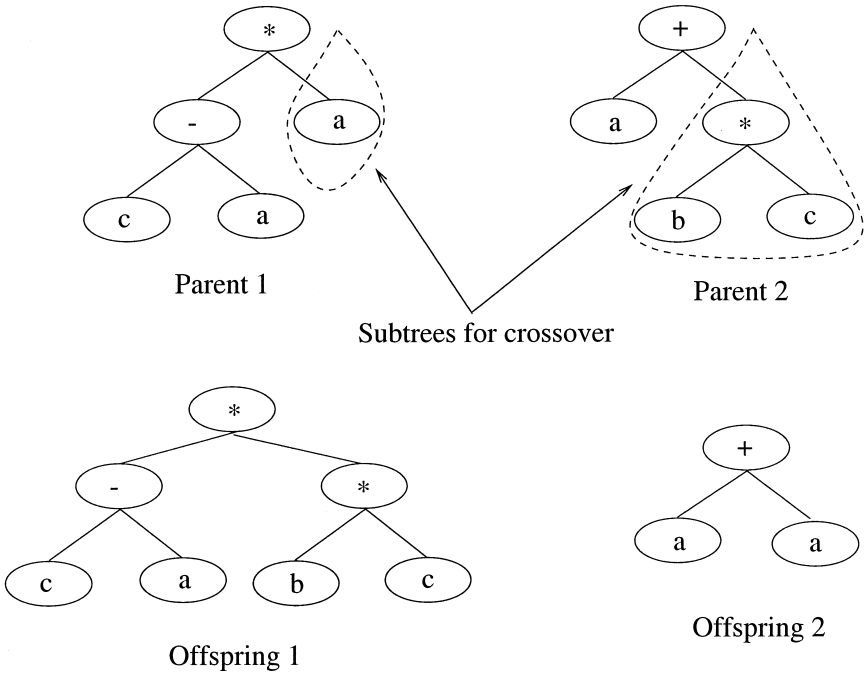


FIGURE 8 Crossover in genetic programming.

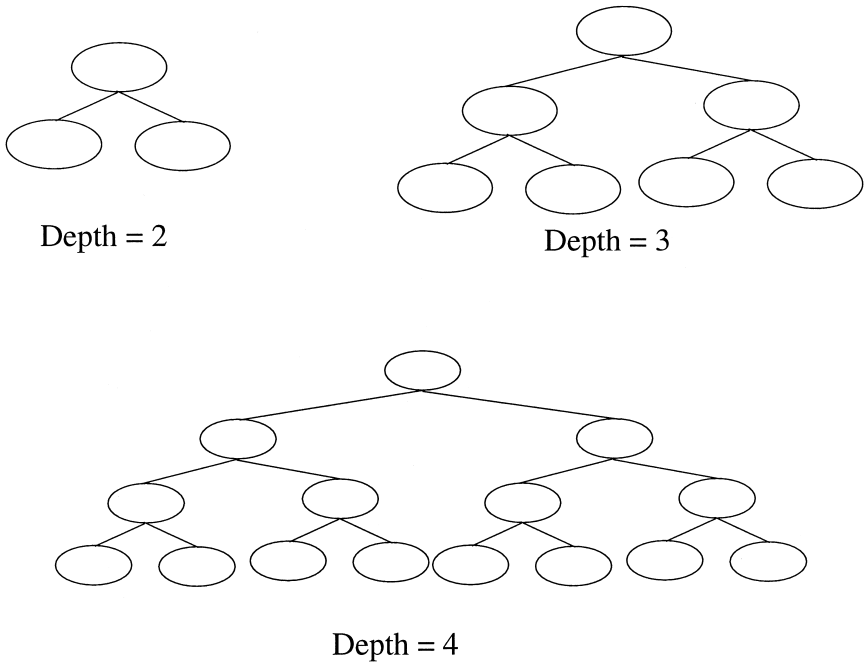


FIGURE 9 A selection of trees generated using the full method.

of the individuals in Fig. 7 could have been generated using the grow method. Trees generated in this way are usually characterized by having different-sized branches, although it is possible that they may be the same.

Another method is referred to as the *full* method. This only chooses function nodes until the maximum depth has been reached. At this level, only terminals are chosen. This method gives much richer trees than the grow method, as, in general, individuals are much bigger, but the population tends to be quite uniform, especially with regard to size. Most evolutionary algorithms thrive on a diverse population, and GP is no exception, so this method is rarely used on its own. Figure 9 shows some examples of full trees.

One final popular method is the *ramped half-and-half* method, which is essentially a compromise between the above two methods. For ramped half-and-half, one first decides on a maximum depth N, and the population is then divided into groups, each of which has a maximum going from 2 to N. Half of the individuals in each group are then generated using the grow method and the other half are generated using the ramped method.

3. Types of Problems

The flexible representation employed by GP has given rise to a wide variety of applications. One area where GP has enjoyed much success is in that of *symbolic regression*, also refered to as *sequence problems*. Symbolic regression problems involve determining a function that maps a set of input data to a set of output data. For example, if one had a set of stock prices for a particular commodity at the end of each of the last 7 days, it would be extremely useful to have a function that could forecast the following day's price. Other examples include speech recognition, where various sounds are matched to a particular word, DNA pattern recognition, and weather prediction.

Available data are divided into two sets, the *training* set and the *test* set. GP is then used to generate individuals which are functions that map the inputs—previous prices, wave forms, etc.—onto one of a set of outputs, which, depending on the problem task, could be anything from the next day's price to a word.

C. Evolutionary Programming

Evolutionary programming (EP) is distinct from most other types of evolutionary algorithms in that it operates at a higher level, that is, instead of groups of individuals, it maintains groups of *species* which are evolving toward a particular goal.

The main steps in EP are similar to that of other GAs, with the exception of crossover. Crossover does not take place between individuals in EP because the finest grain in the population is a species, and crossover tends not to take place between different species in nature. Instead of crossover, the driving force behind EP is mutation, the idea of which is to reinforce the notion that, in general, new generations of a species tend to be somewhat similar to

the previous ones. Despite the notion of acting at a species level, the same terminology is used in EP as in other evolutionary algorithms i.e., one still deals with populations of individuals.

One of the strengths of EP is that it is metaheuristic in the truest sense of the word because its operation is totally independent of the representation scheme employed by its individuals. This is partly due the absence of a crossover operator, as only certain structures can be randomly crossed over without running the risk of producing illegal offspring.

As with all other evolutionary algorithms, the initial population is generated at random. The exact manner in which this happens, however, is entirely dependent on the problem domain. EP can handle any structure, whether as simple as the binary strings used by GAs, sets of real numbers, or even highly complex structures like neural networks.

Once the population has been generated, each individual is tested, again in the same manner as other evolutionary algorithms. Another difference occurs when generating the next generation. Each individual in the population is looked upon as a species, and if that individual is chosen to survive into the next generation, the species should evolve, in that, although it is somewhat different from its predecessor, there should still be some clear behavioral similarities.

This inheritance of traits is achieved by a simple mutation of the parent individual. The manner in which the parent is mutated depends on the representation scheme. For example, if one were manipulating binary strings, one could simply change a bit value, while if one were concerned with evolving neural networks, mutation could be achieved by a simple modification of one or more weights.

We have not yet discussed selection with regard to EP. This is because it is possible to avoid selection entirely with EP. If each individual produces one offspring per generation, one can simply replace the parent generation in its entirety. However, there is no reason why the population size should be held constant, or why each individual should only produce one offspring. An element of competition can be introduced by conducting *tournaments* during selection. Effectively, for each new individual in the next generation, a tournament is held whereby a (small) number of individuals are selected from the parent population at random. The fittest individual in this group is then used to produce the next offspring.

Types of Problems

Due to the absence of any representation-dependent operators, EP can literally be applied to any optimization problem. Indeed, there is no particular reason why one could not conduct experiments with a population of just one. However, in practice, the population is usually >1, although there is no clear indication of at what point increasing the population simply slows down search. Because there is no crossover, the effectiveness of the algorithm is not as sensitive to the population size as in most other evolutionary algorithms.

D. Evolutionary Strategies

While most evolutionary algorithms operate at a genotype level, evolutionary strategies (ES) operate at a phenotype or behavioral level. There is no mapping from an individual's genes to its physical expression, as the physical expression is coded directly. The reason for this approach is to encourage a strong causality, that is, a small change in the coding leads to a small change in the individual. Similarly, a large change in the coding has associated with it a large change.

Due to this concentration on the phenotype, individuals often directly encode whatever is being evolved, e.g., it is not unusual for an individual to be a real number or, indeed, a set of real numbers. Items encoded in this way are usually referred to as the *objective variables*. Unlike most evolutionary algorithms, individuals in an ES population control several of their own parameters, in particular, mutation. These parameters, known as *strategy variables*, control the degree of mutation that can occur in each of the objective variables. The higher the value of a strategy variable, the greater the degree of variation that can occur.

Population sizes in ES are extremely small compared to other evolutionary algorithms. At the extreme, it is possible to have a population size of just one. This is known as a two-membered, or $(1+1)$, ES. There are considered to be two members because there is always a parent and a child. The $(1+1)$ refers to the fact that there is one parent, which produces one offspring in each generation. In this case, there is no crossover, just mutation. When the child is created, it is compared to the parent and, if it has a greater fitness, replaces the parent.

This algorithm has been enhanced to the so-called $(m+1)$ ES, which refers to the fact that there are m parents, each of which produces one offspring. Sometimes crossover is employed in this case, to permit the sharing of useful information. Once all the offspring are created, the best m individuals from the combined parent and offspring group are selected to become the next generation. This strategy is referred to as the $+$ strategy, and is in contrast to the alternative *comma*, or $(m, 1)$, ES, in which the parent group is replaced entirely by the offspring group, regardless of any improvements or degradations in performance.

Crossover in ES is intended to produce children that are behaviorally similar to their parents, and there are three different approaches. The first, *discrete* recombination, is

Parent 1 : 1 0 0 1 1 1

Parent 2 : 0 1 1 0 0 0

Mask: 1 0 0 1 1 0

Child 1 : 0 0 0 0 0 1

Child 2: 1 1 1 1 1 0

FIGURE 10 Uniform crossover.

similar to a method often used in GAs, *uniform* crossover. Uniform crossover involves creating a *crossover mask*, a binary string the same length as the parents. A 0 in the mask results in the relevant gene being selected from the first parent, while a 1 results in the second parent donating the gene. The crossover mask is a random string, and generally ensures that each parent contributes equally to the child. An example is shown in Fig. 10.

The other two methods exploit the fact that the genes are real-valued. The first of these, the *intermediate* recombination operator, determines the value of the child's genes by averaging the two values in the parents' genes. The second method, the *random intermediate* recombinator, probabilistically determines the evenness of the contribution of each parent for each parameter.

E. Grammatical Evolution

Grammatical evolution (GE) is effectively a cross between genetic programming and genetic algorithms. It is an automatic programming system that can generate computer programs from variable-length binary strings.

An evolutionary algorithm is used to perform the search upon a population of these binary strings, each of which represents a program. The output program, the phenotype, is generated through a genotype-to-phenotype mapping process, where the genotype refers to the binary string. It is this mapping process that distinguishes GE from other linear GP-type systems, and bestows several advantages on the system. Figure 11 compares the mapping process in grammatical evolution to that found in nature.

This mapping separates the search and solution spaces, and permits GE to enjoy many of the operators designed for GA; it does away with the need to design special mutation and crossover operators to handle the program syntax. Individuals in GE map a genotype onto a program using a Backus Naur form (BNF) grammar definition that can be tailored to the particular problem in terms of complexity and the language of the output code.

The mapping process takes place by reading N-bit codons (N typically takes the value 8, but depends on the grammar) on the binary string, converting a codon into its corresponding integer value. The integer value is then used to select an appropriate production rule from the BNF definition to apply to the current nonterminal,

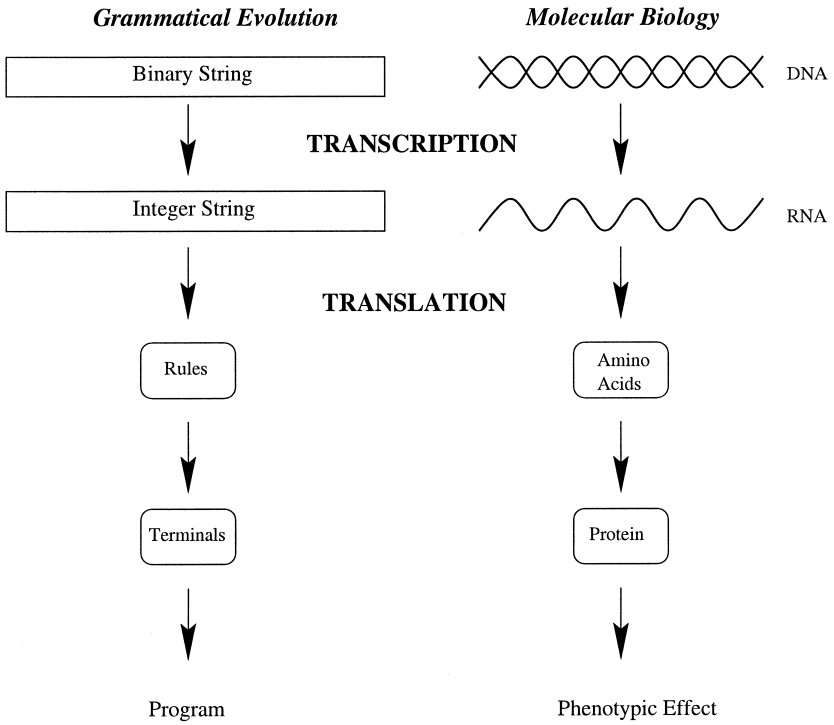


FIGURE 11 Mapping process for grammatical evolution.

e.g., in the first instance, the start symbol. This is achieved by determining how many production rules can be applied to the current nonterminal and then getting the codon integer value modulus the number of possible production rules. For example, given a nonterminal ⟨expr⟩ with the production rules

```
⟨expr⟩ ::= ⟨expr⟩⟨op⟩⟨expr⟩      (0)
         |(⟨expr⟩⟨op⟩⟨expr⟩)    (1)
         |⟨pre-op⟩(⟨expr⟩)      (2)
         |⟨var⟩                 (3)
```

In this case there are four possible rules which can be applied to ⟨expr⟩. If the current codon integer value was 2, this would give 2 MOD 4, which equals 0. In this case, then, the zeroth production rule is applied to the nonterminal resulting in the replacement of ⟨expr⟩ with ⟨expr⟩⟨op⟩⟨expr⟩; the mapping continues by reading codon after codon using the generated number to select appropriate rules for each nonterminal until a completed program is achieved.

It is the modulus function to select production rules that gives the system a degenerate genetic code. That is, many different codon integer values can represent the same production rule. A consequence of this is that a mutation at a codon while changing the integer value will not necessarily change the production rule applied at that instance, the result being a functionally equivalent phenotype. A mutation of this type has been termed a neutral mutation, and it is proposed that these neutral mutations are responsible for maintaining genetic diversity in biological populations; a similar effect has been observed in GE.

III. SIMULATED ANNEALING

Simulated annealing (SA) attempts to model the manner in which a metal cools and solidifies, and uses this model as a metaheuristic. When metal is heated, its atoms become increasingly free to move about. The more heat that is applied, the greater is the degree of movement. When the metal is subsequently permitted to cool, the atoms slow, and eventually stop. If the atoms are moved during their excited state, they will stay in the new position once the metal solidifies. Furthermore, if the metal is cooled too quickly, it will become brittle.

The connection between annealing and simulated annealing is that a potential solution is analogous to the metal being manipulated. The process starts with a randomly generated solution, to which random changes are made. The degree of change is sensitive to the "temperature" of the system, that is, a high temperature permits a large change and a lower temperature only permits a smaller change. Each change should eventually lead to the solution becoming better.

An important question in SA is what kinds of changes should be permitted? If one only ever accepts improvements, then the algorithm will only ever search a single peak in the solution landscape. To permit a more global search, the algorithm must occasionaly accept decreases in performance, in the hope that they may lead to better increases later on. SA acknowledges this, and accepts decreases in performance with a certain probability, and this probability itself decreases over time. This permits SA to do something of a global search early on in a run and then to focus on a more local search as time progresses.

The all-important control parameters for SA are the initial temperature, the final temperature, and the cooling rate, for these directly affect the number of perturbations that can be made to a structure. One final parameter is the number of perturbations that can be made at each temperature. Again, to take the analogy of cooling metal, at any given temperature, one can make several modifications before permitting the temperature to drop. For the process to work at its peak, a particular temperature should be kept constant until all the useful modifications have been made. The exact implementation of this varies, but many practitioners keep making perturbations until a certain amount have occurred without any increase in performance.

An algorithm, then, for simulated annealing could be:

1. Get an initial solution s and temperature T.
2. While at this temperature perform the following:
   (a) Generate a solution s' from s.
   (b) Let $\delta = \text{fitness}(s) - \text{fitness}(s')$.
   (c) If $\delta \leq 0$ (increase in fitness), set $s = s'$.
   (d) If $\delta > 0$ (decrease in fitness), set $s = s'$ with some probability.
3. Reduce temperature T according to the cooling rate.

The probability mentioned in step 2 above is generally

$$\text{Prob(accept)} = e^{-\delta/T}.$$

That is, the probability varies inversely with the current temperature, so that as time goes on, decreases in fitness are less likely to be accepted.

The decrease in temperature is set according to the cooling rate r, which is a number between 0 and 0.99, although in practice it is usually around 0.8–0.99. This gives us the following equation for calculating T', the new temperature value:

$$T' = rT.$$

The process continues until either the minimum temperature is reached or a satisfactory solution is reached. Unfortunately, setting the parameter values is something of a black art. If the initial temperature is too high, then there may be too many changes at the early stages to do

any useful exploration. Similarly, if it is too low, the search may be unnecessarily constrained in one area. In the same manner, if the cooling rate is too quick, the system may converge on a suboptimal solution.

Representation

Like evolutionary programming, discussion of representation in SA has been left until last. This is because its operation is totally independant of representation, and any structure that can be randomly modified can be processed using simulated annealing. Thus, any structure from a simple binary number to something as complex as a neural network may be used.

IV. TABU SEARCH

All of the metaheuristic search methods described thus far are *blind* search methods, in that they are not aware of what part of the solution landscape they are searching. This can lead to inefficiencies if the same part of the landscape is repeatedly searched or, as can happen in EP or SA, a move is made that brings the potential solution back into a region that has already been searched.

Consider Fig. 12, where point A has already been searched and the current point is B. It is more attractive to return to point A than to move to C in the short term, even though C may eventually lead to a higher reward, as it could direct the search toward D. Tabu search attempts to inhibit this kind of behavior by maintaining a "tabu list," which is simply a list of previously made moves, or areas of the landscape which have already been searched, and therefore should not be revisited (are considered taboo).

This is often a problem for hill-climbing algorithms. In many instances, they can get stuck in this situation, even if they did travel toward C, as they are likely to be tempted back up toward A.

When moving from one solution s to another solution s', tabu search generates a set of possible moves known as the *neighborhood* of s. The move that gives the greatest reward is taken and then entered into the tabu list to ensure that the same move will not be made again. The exact form of the tabu list varies. Sometimes the actual solutions themselves are stored, to prevent them from being revisited, while other implementations store the *inverse* of the move, to prevent the search from going back. Before a move can be made, however, the tabu list is examined to see if it is there; if so, the next best move is chosen, and so on.

Tabu search can be looked upon as a search with a short-term memory, as the last number of moves are remembered. There is a limit to the number remembered, as it is generally not practical to store every move. Using this implementation, however, tabu search can be too restrictive; quite often there may only be a relatively small number of moves leading away from a particular position. Once all of these are in the tabu list, it is possible for the search to become trapped.

To prevent this from happenning, an *aspiration* function is often employed. An aspiration function permits a move that is in the tabu list if this move gives a better score than all others found. This is sometimes necessary because some level of backtracking is often needed. The probability of the aspiration function permitting a move from the tabu list to be made often varies with time.

While tabu search can be very efficient, its utility can often depend on the problem being tackled. If a relatively small number of moves can be made, the list is often too small to be of any use. Conversely, if the number of moves possible from each solution, i.e., the neighborhood, is very large, the system becomes extremely inefficient, as it requires testing each item in the neighborhood.

V. MEMETIC ALGORITHMS

While the population-based evolutionary algorithms may appear to be quite different from the single-entity-based algorithms of simulated annealing and tabu search, there is no reason why they cannot work together. After all, our own evolution is characterized by both the evolution of genes and the evolution of ideas. An individual often benefits from the experience of its parents, and during its own lifetime, strives to become as successful as possible—in EA terms, to become as fit as it can be.

Memetic algorithms, sometimes refered to as *hybrid genetic algorithms*, attempt to capture this behavior. At a simple level, they are similar to evolutionary algorithms,

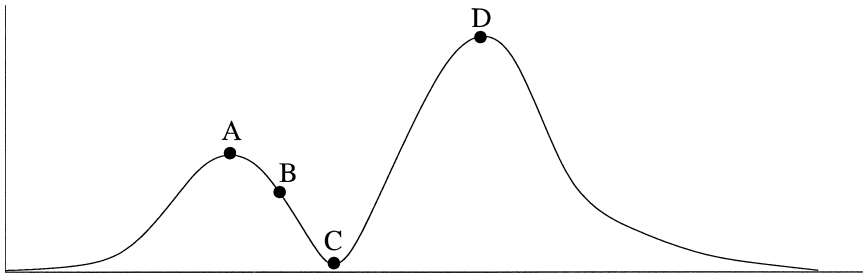


FIGURE 12 Escaping from a local maximum.

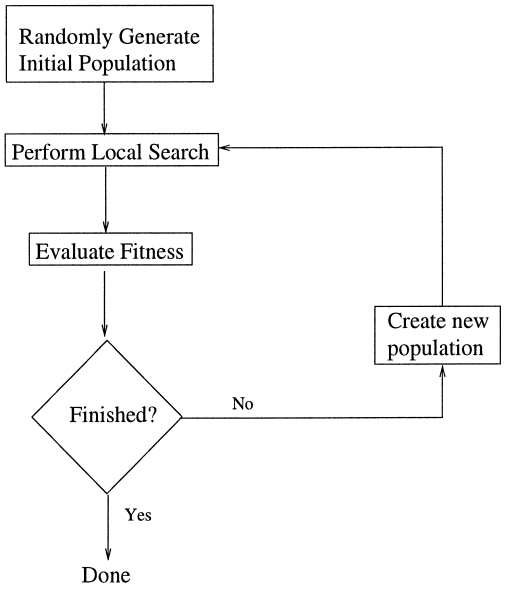


FIGURE 13 Flowchart for a simple memetic algorithm.

but each individual also undergoes some kind of hill climbing, possibly simulated annealing or tabu search. The fitness of that individual becomes whatever the fitness is after the hill-climbing. Figure 13 shows a flowchart for a memetic algorithm. As can be seen, the only difference between this and the flowchart for evolutionary algorithms in Fig. 4 is the extra "Perform Local Search" step.

In true metaheuristic fashion, there is no detail about either the method used to produce each generation or the local search method employed. This means that any representation scheme may be used. Thus, it is possible to combine GP and SA, or GA and tabu search; indeed, any combination is possible.

VI. SUMMARY

In order to solve difficult or poorly understood problems, it is often useful to employ a metaheuristic. Metaheuristics are adept at discovering good solutions without necessarily requiring much information about the problem to which they are being applied.

There are two broad categories of metaheuristics, often termed evolutionary algorithms and hill climbers. However, it is often difficult to decide which one is appropriate for a problem without at least some degree of knowledge about the problem domain.

Hill climbers continuously modify a single structure, ideally with each change bringing some kind of improvement, but often accepting degradations, in the hope that they may eventually lead to a long-term gain.

Evolutionary algorithms, on the other hand, are population-based techniques. These search a problem space in parallel and try to share information with each other using a number of biologically inspired mechanisms.

In general, metaheuristics tend not to have hard and fast rules about their application. Knowing the best way to apply them is often an acquired skill, and even then something of a black art. Nothing can replace experience, and the interested reader is encouraged to experiment with the various algorithms.

SEE ALSO THE FOLLOWING ARTICLES

ARTIFICIAL INTELLIGENCE • CHROMATIN STRUCTURE AND MODIFICATION • COMPUTER ALGORITHMS • DATA MINING, STATISTICS • GENE EXPRESSION, REGULATION OF • METAL FORMING • NONLINEAR PROGRAMMING • STOCHASTIC PROCESSES

BIBLIOGRAPHY

Axelrod, R. (1984). "The Evolution of Cooperation," Basic Books, New York.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). "Genetic Programming—An Introduction," Morgan Kaufmann, San Francisco.

Darwin, C. (1859). "The Origin of Species," John Murray, London.

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). "Artificial Intelligence through Simulated Evolution," Wiley, New York.

Goldberg, D. (1989). "Genetic Algorithms in Search, Optimization, and Machine Learning," Addison-Wesley, Reading, MA.

Holland, J. H. (1975). "Adaptation in Natural and Artificial Systems," University of Michigan Press, Ann Arbor, MI.

Koza, J. R. (1992). "Genetic Programming: On the Programming of Computers by Means of Natural Selection," MIT Press, Cambridge, MA.

Ryan, C. (1999). "Automatic Re-engineering of Software Using Genetic Programming," Kluwer, Amsterdam.

Image Processing

Rama Chellappa
Azriel Rosenfeld
University of Maryland

GLOSSARY

Compression Reduction of the amount of data used to represent an image, by compact encoding or by approximation.
Description Information about image parts and their properties and relations.
Digitization Conversion of an image into a discrete array of numbers for computer processing.
Enhancement Processing of an image to improve its appearance.
Matching Comparison of images for purposes of pattern recognition, registration, stereopsis, or motion analysis.
Recognition Recognition of objects in an image by comparing image descriptions with object models.
Reconstruction Computation of cross sections of an image or volume, given a set of its projections.
Recovery Estimation of the orientation of a surface from the shading, texture, or shape of the corresponding region of an image.
Representation Mathematical description of image data.
Restoration Processing of an image to correct for the effects of degradations.
Segmentation Partitioning of an image into homogeneous regions; extraction of features such as edges or curves from an image.

COMPUTERS are used to process images for many purposes. Image processing is the computer manipulation of images to produce more useful images; subfields of image processing include image compression or coding, image enhancement and restoration, and image reconstruction from projections; examples of applications include compression of DVD movies, restoration of Hubble telescope images, and medical imaging. The goal of image analysis is to produce a description of the scene that gave rise to the

image; examples of applications are reading documents (character recognition), counting blood cells on a microscope slide, detecting tumors in chest X rays, producing land use maps from satellite images, detecting defects in printed circuits, and guiding robots in navigating or in manipulating objects. In addition to processing images in the visible spectrum, acquisition, processing and analysis of infrared, synthetic aperture radar, medical, hyperspectral, and range images have become important over the last 20 years due to the emergence of new sensors and applications.

I. INTRODUCTION

This chapter deals with the manipulation and analysis of images by computer. In image processing, both the input and the output are images, the output being, for example, an approximated or improved version of the input. In image analysis (also known by such names as pictorial pattern recognition, image understanding, and computer vision), the input is an image and the output is (typically) a description of the scene that gave rise to the image. Computer graphics, which is not covered in this chapter, is the inverse of image analysis: The input is a scene description, and the output is an image of the scene as it would appear from a given viewpoint.

An image is defined by specifying how its value (brightness, color, etc.) varies from point to point—in other words, by a function of two variables defined over an "image plane." Before an image can be processed and analyzed by (digital) computer, it must be converted into a discrete array of numbers each of which represents the value at a given point. This process of conversion is called digitization (Section II).

A digitized image can be viewed as a matrix of gray-level values. To understand/analyze the structure of this matrix, image models and image transforms have been used. Image models attempt to describe the image data quantatively, while image transforms enable the analysis of the image data in the transform domain for various applications such as compression, restoration, and filtering.Image models and representations are discussed in Section III.

To represent the input image with sufficient accuracy, the array of numbers must usually be quite large—for example, about 500×500 in the case of a television image. Image compression (or coding) deals with methods of reducing this large quantity of data without sacrificing important information about the image (Section IV).

One of the central goals of image processing is to improve the appearance of the image—for example, by increasing contrast, reducing blur, or removing noise. Image enhancement (Section V) deals with methods of improving the appearance of an image. More specifically, image restoration (Section VI) is concerned with estimating image degradations and attempting to correct them.

Another important branch of image processing is image reconstruction from projections (Section VII). Here we are given a set of images (e.g., X rays) representing projections of a given volume, and the task is to compute and display images representing cross sections of that volume.

Comparison or matching of images is an important tool in both image processing and analysis. Section VIII discusses image matching and registration and depth measurement by comparison of images taken from different positions (stereomapping).

Section IX summarizes methods for the analysis of image sequences. Techniques for motion compensation, detection and tracking of moving objects, and recovery of scene structure from motion using optic flow and discrete features are discussed.

The brightness of an image at a point depends on many factors, including the illumination, reflectivity, and surface orientation of the corresponding surface point in the scene. Section X discusses methods of recovering these "intrinsic" scene characteristics from an image by analyzing shading, texture, or shapes in the image.

Image analysis usually begins with feature detection or segmentation—the extraction of parts of an image, such as edges, curves, or regions, that are relevant to its description. Techniques for singling out significant parts of an image are reviewed in Section XI. Methods of compactly representing image parts for computer manipulation, as well as methods of decomposing image parts based on geometric criteria and of computing geometric properties of image parts, are treated in Section XII.

Section XIII deals with image description, with an emphasis on the problem of recognizing objects in an image. It reviews properties and relations, relational structures, models, and knowledge-based image analysis.

A chapter such as this would not be complete without some discussion of architectures designed for efficient processing of images. The eighties witnessed an explosion of parallel algorithms and architectures for image processing and analysis; especially noteworthy were hypercube-connected machines. In the early nineties attention was focused on special processors such as pyramid machines. Recently, emphasis is being given to embedded processors and field-programmable gate arrays. Section XIV presents a summary of these developments.

The treatment in this chapter is concept-oriented; applications are not discussed, and the use of mathematics has been minimized, although some understanding of Fourier transforms, stochastic processes, estimation theory, and linear algebra is occasionally assumed. The Bibliography

lists basic textbooks, as well as a recent survey article that celebrated 50 years of progress in image processing and analysis.

Since an earlier version of this chapter appeared in 1986, image processing and analysis have grown so dramatically in depth and breadth that it is nearly impossible to cover all their subareas in detail. To adhere to our page limitations, we have made a conscientious selection of topics for discussion. We avoid discussion of topics such as scanners, display devices, and hardware and software issues. On the other hand, since the mideighties, much to our delight, the field has been supported by a strong underlying analytical framework based on mathematics, statistics, and physics. We point out the influence of these fields on image processing, analysis, understanding, and computer vision.

II. DIGITIZATION

Digitization is the process of converting an image into a discrete array of numbers. The array is called a digital image, its elements are called pixels (short for "picture elements"), and their values are called gray levels. (In a digital color image, each pixel has a set of values representing color components.)

Digitization involves two processes: sampling the image value at a discrete grid of points and quantizing the value at each of these points to make it one of a discrete set of gray levels. In this section we briefly discuss sampling and quantization.

A. Sampling

In general, any process of converting a picture into a discrete set of numbers can be regarded as "sampling," but we assume here that the numbers represent the image values at a grid of points (or, more precisely, average values over small neighborhoods of these points). Traditionally, rectangular uniform sampling lattices have been used, but depending on the shape of the image spectrum, hexagonal, circular, or nonuniform sampling lattices are sometimes more efficient.

The grid spacing must be fine enough to capture all the detail of interest in the image; if it is too coarse, information may be lost or misrepresented. According to the sampling theorem, if the grid spacing is d, we can exactly reconstruct all periodic (sinusoidal) components of the image that have period $2d$ or greater (or, equivalently, "spatial frequency" $1/2d$ or fewer cycles per unit length). However, if patterns having periods smaller than $2d$ are present, the sampled image may appear to contain spurious patterns having longer periods; this phenomenon is called aliasing. Moiré patterns are an everyday example of aliasing, usually arising when a scene containing short-period patterns is viewed through a grating or mesh (which acts as a sampling grid).

Recent developments in the design of digital cameras enable us to acquire digital images instantly. In addition, many image processing operations such as contrast enhancement and dynamic range improvement are being transferred to the acquisition stage.

B. Quantization

Let z be the value of a given image sample, representing the brightness of the scene at a given point. Since the values lie in a bounded range, and values that differ by sufficiently little are indistinguishable to the eye, it suffices to use a discrete set of values. It is standard practice to use 256 values and represent each value by an 8-bit integer $(0, 1, \ldots, 255)$. Using too few discrete values give rise to "false contours," which are especially conspicuous in regions where the gray level varies slowly. The range of values used is called the grayscale.

Let the discrete values ("quantization levels") be $z_l, \ldots, z_k$. To quantize z, we replace it by the z_j that lies closest to it (resolving ties arbitrarily). The absolute difference $|z - z_j|$ is called the quantization error at z.

Ordinarily, the z's are taken to be equally spaced over the range of possible brightnesses. However, if the brightnesses in the scene do not occur equally often, we can reduce the average quantization error by spacing the z's unequally. In fact, in heavily populated parts of the brightness range, we should space the z's closely together, since this will result in small quantization errors. As a result, in sparsely populated parts of the range, the z's will have to be farther apart, resulting in larger quantization errors; but only a few pixels will have these large errors, whereas most pixels will have small errors, so that the average error will be small. This technique is sometimes called tapered quantization.

Quantization can be optimized for certain distributions of brightness levels, such as Gaussian and double-exponential. Another class of techniques, known as moment-preserving quantizers, does not make specific assumptions about distributions. Instead, the quantizers are designed so that low-order moments of the brightness before and after quantization are required to be equal. These scalar quantizers do not exploit the correlation between adjacent pixels. Vector quantization (VQ), in which a small array of pixels is represented by one of several standard patterns, has become a popular method of image compression on its own merits, as well as in combination with techniques such as subband or wavelet decomposition. A major difficulty with VQ techniques is their computational complexity.

III. REPRESENTATION

Two dominant representations for images are two-dimensional (2-D) discrete transforms and various types of image models. In the former category, linear, orthogonal, separable transforms have been popular due to their energy-preserving nature and computational simplicity. When images are represented using discrete transforms, the coefficients of the expansion can be used for synthesis, compression, and classification. The two most often used discrete transforms are the discrete Fourier transform (due to its FFT implementation) and discrete cosine transform (due to its FFT-based implementation and its adoption in the JPEG image compression standard). If minimizing the mean squared error between the original image and its expansion using a linear orthogonal transform is the goal, it can be shown that the Karhunen–Loeve transform (KLT) is the optimal transform, but the KLT is not practical, since it requires eigencomputations of large matrices. Under a circulant covariance structure, the DFT is identical to the KLT.

All the 2-D discrete transforms mentioned above analyze the data at one scale or resolution only. Over the last 20 years, new transforms that split the image into multiple frequency bands have given rise to schemes that analyze images at multiple scales. These transforms, known as wavelet transforms, have long been known to mathematicians. The implementation of wavelet transforms using filter banks has enabled the development of many new transforms.

Since the early eighties, significant progress has been made in developing stochastic models for images. The most important reason is the abstraction that such models provide of the large amounts of data contained in the images. Using analytical representations for images, one can develop systematic algorithms for accomplishing a particular image-related task. As an example, model-based optimal estimation-theoretic principles can be applied to find edges in textured images or remove blur and noise from degraded images. Another advantage of using image models is that one can develop techniques to validate a given model for a given image. On the basis of such a validation, the performance of algorithms can be compared.

Most statistical models for image processing and analysis treat images as 2-D data, i.e., no attempt is made to relate the 3-D world to its 2-D projection on the image. There exists a class of models, known as image-formation models, which explicitly relate the 3-D information to the 2-D brightness array through a nonlinear reflectance map by making appropriate assumptions about the surface being imaged. Such models have been the basis for computer graphics applications, can be customized for particular sensors, and have been used for inferring shape from shading and other related applications. More recently, accurate prediction of object and clutter models (trees, foliage, urban scenes) has been recognized as a key component in model-based object recognition as well as change detection. Another class of models known as fractals, originally proposed by Mandelbrot, is useful for representing images of natural scenes such as mountains and terrain. Fractal models have been successfully applied in the areas of image synthesis, compression, and analysis.

One of the earliest model-based approaches was the multivariate Gaussian model used in restoration. Regression models in the form of facets have been used for deriving hypothesis tests for edge detection as well as deriving the probability of detection. Deterministic 2-D sinusoidal models and polynomial models for object recognition have also been effective.

Given that the pixels in a local neighborhood are correlated, researchers have proposed 2-D extensions of time-series models for images in particular, and for spatial data in general, since the early 1950s. Generalizations have included 2-D causal, nonsymmetric half-plane (NSHP), and noncausal models.

One of the desirable features in modeling is the ability to model nonstationary images. A two-stage approach, regarding the given image as composed of stationary patches, has attempted to deal with nonstationarities in the image. A significant contribution to modeling nonstationary images used the concept of a dual lattice process, in which the intensity array is modeled as a multilevel Markov random field (MRF) and the discontinuities are modeled as line processes at the dual lattice sites interposed between the regular lattice sites. This work has led to several novel image processing and analysis algorithms.

Over the last 5 years, image modeling has taken on a more physics-based flavor. This has become necessary because of sensors such as infrared, laser radar, SAR, and foliage-penetrating SAR becoming more common in applications such as target recognition and image exploitation. Signatures predicted using electromagnetic scattering theory are used for model-based recognition and elimination of changes in the image due to factors such as weather.

IV. COMPRESSION

The aim of image compression (or image coding) is to reduce the amount of information needed to specify an image, or an acceptable approximation to an image. Compression makes it possible to store images using less memory or transmit them in less time (or at a lower bandwidth).

A. Exact Encoding

Because images are not totally random, it is often possible to encode them so that the coded image is more compact than the original, while still permitting exact reconstruction of the original. Such encoding methods are generally known as "lossless coding."

As a simple illustration of this idea, suppose that the image has 256 gray levels but they do not occur equally often. Rather than representing each gray level by an 8-bit number, we can use short "codes" for the frequently occurring levels and long codes for the rare levels. Evidently this implies that the average code length is relatively short, since the frequent levels outnumber the rare ones. If the frequent levels are sufficiently frequent, this can result in an average code length of less than eight bits. A general method of constructing codes of this type is called Shannon–Fano–Huffman coding.

As another example, suppose that the image has only a few gray levels, say two (i.e., it is a black-and-white, or binary, image: two levels might be sufficient for digitizing documents, e.g., if we can distinguish ink from paper reliably enough). Suppose, further, that the patterns of black and white in the image are relatively simple—for example, that it consists of black blobs on a white background. Let us divide the image into small blocks, say 3×3. Theoretically, there are $2^9 = 512$ such blocks (each of the nine pixels in a block can have gray level either 0 or 1), but not all of these combinations occur equally often. For example, combinations like

$$\begin{matrix}1&1&1\\1&1&1\\1&1&1\end{matrix} \quad \text{or} \quad \begin{matrix}0&1&1\\0&1&1\\0&0&1\end{matrix} \quad \text{or} \quad \begin{matrix}1&1&0\\0&0&0\\0&0&0\end{matrix} \quad \text{or} \quad \begin{matrix}0&0&0\\0&0&0\\0&0&0\end{matrix}$$

should occur frequently, but combinations like

$$\begin{matrix}1&0&1\\0&1&0\\1&0&0\end{matrix}$$

should occur rarely or not at all. By using short codes to represent the frequent blocks and longer codes for the rare ones, we can reduce (to much less than 9) the average number of bits required to encode a block. This approach is sometimes called area character coding or block coding. Other methods of compactly encoding such images, including run length coding and contour coding, are discussed in Section 12 when we consider methods of compactly representing image parts.

B. Approximation

The degree of compression obtainable by exact encoding is usually small, perhaps of the order of 2:1. Much higher degrees of compression, of 20:1 or greater, can be obtained if we are willing to accept a close approximation to the image (which can, in fact, be virtually indistinguishable from the original).

The fineness of sampling and quantization ordinarily used to represent images are designed to handle worst cases. Adequate approximations can often be obtained using a coarser sampling grid or fewer quantization levels. In particular, sampling can be coarse in image regions where the gray level varies slowly, and quantization can be coarse in regions where the gray level varies rapidly. This idea can be applied to image compression by using "adaptive" sampling and quantization whose fineness varies from place to place in the image. Alternatively, we can use Fourier analysis to break the image up into low-frequency and high-frequency components. The low-frequency component can be sampled coarsely, the high-frequency component can be quantized coarsely, and the two can then be recombined to obtain a good approximation to the image. Still another idea is to add a pseudorandom "noise" pattern, of amplitude about one quantization step, to the image before quantizing it and then subtract the same pattern from the image before displaying it; the resulting "dithering" of the gray levels tends to break up false contours, so that quantization to fewer levels yields an acceptable image.

C. Difference Coding and Transform Coding

In this section we discuss methods of transforming an image so as to take greater advantage of both the exact encoding and approximation approaches.

Suppose we scan the image (say) row by row and use the gray levels of a set of preceding pixels to predict the gray level of the current pixel—by linear extrapolation, for example. Let $\hat{z}_i$ be the predicted gray level of the ith pixel and z_i its actual gray level. If we are given the actual gray levels of the first few pixels (so we can initiate the prediction process) and the differences $z_i - \hat{z}_i$ between the actual and the predicted values for the rest of the pixels, we can reconstruct the image exactly. The differences, of course, are not a compact encoding of the image; in fact, they can be as large as the largest gray level and can be either positive or negative, so that an extra bit (a sign bit) is needed to represent them. However, the differences do provide a basis for compact encoding, for two reasons.

1. The differences occur very unequally; small differences are much more common than large ones, since large, unpredictable jumps in gray level are rare in most types of images. Thus exact encoding

techniques, as described in Section IV.A, can be used to greater advantage if we apply them to the differences rather than to the original gray levels.
2. When large differences do occur, the gray level is fluctuating rapidly. Thus large differences can be quantized coarsely, so that fewer quantization levels are required to cover the range of differences.

The main disadvantage of this difference coding approach is the relatively low degree of compression that it typically achieves.

More generally, suppose that we apply an invertible transformation to an image—for example, we take its discrete Fourier transform. We can then encode or approximate the transformed image, and when we want to reconstruct the original image, we apply the inverse transformation to the approximated transform. Evidently, the usefulness of this transform coding approach depends on the transformed image being highly "compressible."

If we use the Fourier transform, it turns out that for most classes of images, the magnitudes of the low-frequency Fourier coefficients are very high, whereas those of the high-frequency coefficients are low. Thus we can use a different quantization scheme for each coefficient—fine quantization at low frequencies, coarse quantization at high frequencies. (For sufficiently high frequencies, the magnitudes are so small that they can be discarded.) In fact, the quantization at high frequencies can be very coarse because they represent parts of the image where the gray level is fluctuating rapidly. When the image is reconstructed using the inverse transform, errors in the Fourier coefficients are distributed over the entire image and so tend to be less conspicuous (unless they are very large, in which case they show up as periodic patterns). An example of transform coding using the discrete cosine transform is shown in Fig. 1.

Difference and transform coding techniques can be combined, leading to hybrid methods. For example, one can use a 1-D transform within each row of the image, as well as differencing to predict the transform coefficients for each successive row by extrapolating from the coefficients for the preceding row(s). Difference coding is also very useful in the compression of time sequences of images (e.g., sequences of television frames), since it can be used to predict each successive image by extrapolating from the preceding image(s).

D. Recent Trends

The difference and transform compression schemes described above attempt to decorrelate the image data at a single resolution only. With the advent of multiresolution representations (pyramids, wavelet transforms, subband decompositions, and quadtrees), hierarchical compression schemes have become the dominant approach. Subband and wavelet compression schemes provide better distributions of bits across the frequency bands than are possible in single-resolution schemes. Also, much higher compression factors with improved peak to signal-to-noise ratios are being achieved using these schemes. Additional features such as progressive compression schemes also make these methods more attractive. An example of subband coding is also shown in Fig. 1.

During the early years of image compression research, although compressed images were transmitted over noisy channels, source and channel coding aspects were treated independently. Due to the emergence of wired and wireless networks, the problems of compression and transmission are now solved using a joint source/channel framework, with the goal of obtaining the best possible compression results for the given channel. Depending on whether the state of the channel is known a priori or is estimated on-line, different strategies are possible. These developments, in addition to reenergizing image compression research, have also led to new coding techniques such as turbo coding.

Compression of image sequences has been investigated for more than 20 years for applications ranging from video telephony to DVD movies. One of the ways to achieve the very high compression factors required in these applications is to exploit temporal redundancy, in addition to the spatial redundancy that is present in still images. Temporal redundancy is removed by using pixel- or object-based motion compensation schemes, skipping frames, or conditional replenishment. During the early years, pel-recursive displacement estimates were used to assign motion vectors to blocks of pixels. Recently, object-based schemes are being introduced. An example of moving object detection in a video frame is shown in Fig. 2.

One of the clear signs that technology has contributed to a commercial product is that standards exist for its implementation. Image compression, because of its usefulness in a large number of applications, is an area that has seen standardization of its technology. For still image compression, standards such as those developed by the Joint Photographic Experts Group (JPEG) and the Joint Bi-level Photographic Experts Group (JBIG) are in daily use. A new standard (JPEG-2000), which includes wavelet transform compression, will be available soon. For compressing image sequences, many standards are available, including (but not limited to) H. 261, H. 263, MPEG-1, MPEG-2, and MPEG-4, and new standards such as MPEG-7 are being developed. These new and improved standards for still and sequence compression reflect the widespread acceptance of research results and technological advances obtained in universities, research laboratories, and industry.

(a)

FIGURE 1 Image compression. (a) Original image; (b) compressed image, using the discrete cosine transform; (c) compressed image, using a subband coding scheme.

V. ENHANCEMENT

The general goal of image enhancement is to improve the appearance of an image so that it can be easier to use. The appropriate method of enhancement depends on the application, but certain general-purpose methods are applicable in many situations. In this section we describe some basic methods of grayscale modification (or contrast stretching), blur reduction, shading reduction, and noise cleaning.

A. Grayscale Modification

When the illumination of a scene is too high or too low or the objects in a scene have reflectivities close to that of the background, the gray levels in the resulting image will occupy only a portion of the grayscale. One can improve the appearance of such an image by spreading its gray levels apart to occupy a wider range. This process of contrast stretching does not introduce any new information, but it may make fine detail or low-contrast objects more clearly

(b)

FIGURE 1 *(continued)*

visible, since spreading apart indistinguishable (e.g., adjacent) gray levels makes them distinguishable.

Even if an image occupies the entire grayscale, one can spread apart the gray levels in one part of the grayscale at the cost of packing them closer together (i.e., combining adjacent levels) in other parts. This is advantageous if the information of interest is represented primarily by gray levels in the stretched range.

If some gray levels occur more frequently than others (e.g., the gray levels at the ends of a grayscale are usually relatively uncommon), one can improve the overall contrast of the image by spreading apart the frequently occurring gray levels while packing the rarer ones more closely together; note that this stretches the contrast for most of the image. (Compare the concept of tapered quantization in Section II.B.) The same effect is achieved by requantizing the image so that each gray level occurs approximately equally often; this breaks up each frequently occuring gray level into several levels, while compressing sets of consecutive rarely occuring levels into a single level. Given an

(c)

FIGURE 1 *(continued)*

image, we can plot a graph showing how often each gray level occurs in the image; this graph is called the image's histogram. The method of requantization just described is called histogram flattening. It is sometimes preferable to requantize an image so as to give it a histogram of some other standard shape.

Since humans can distinguish many more colors than shades of gray, another useful contrast stretching technique is to map the gray levels into colors; this technique is called pseudo-color enhancement.

B. Blur Reduction

When an image is blurred, the ideal gray level of each pixel is replaced by a weighted average of the gray levels in a neighborhood of that pixel. The effects of blurring can be reversed, to a first approximation, by subtracting from the gray level of each pixel in the blurred image a weighted average of the gray levels of the neighboring pixels. This method of "sharpening" or deblurring an image is sometimes referred to as Laplacian processing, because the

(a)

(b)

FIGURE 2 Moving object detection in a video frame. (a) Original frame; (b) detected object.

difference between a pixel's gray level and the average of its neighbors' gray levels is a digital approximation to the Laplacian (sum of second partial derivatives) of the image.

Blurring weakens the high-frequency Fourier components of an image more than it does the low ones. Hence high-emphasis frequency filtering, in which the high-frequency components are strengthened relative to the low ones, has a deblurring effect. Such filtering will also enhance noise, since noise tends to have relativity strong high-frequency components; one should avoid strengthening frequencies at which the noise is stronger than the information-bearing image detail.

C. Shading Reduction

The illumination across a scene usually varies slowly, while the reflectivity may vary rapidly from point to point. Thus illumination variations, or shading, give rise primarily to low-frequency Fourier components in an image, while reflectivity variations also give rise to high-frequency components. Thus one might attempt to reduce shading effects in an image by high-emphasis frequency filtering, weakening the low-frequency components in the image's Fourier transform relative to the high-frequency components. Unfortunately, this simple approach does not work, because the illumination and reflectivity information is combined multiplicatively rather than additively; the brightness of the scene (and hence of the image) at a point is the product of illumination and reflectivity, not their sum. Better results can be obtained using a technique called homomorphic filtering. If we take the logarithm of the gray level at each point of the image, the result is the sum of the logarithms of the illumination and reflectivity; in other words, the multiplicative combination has been transformed into an additive one. We can thus take the Fourier transform of the log-scaled image and apply high-emphasis filtering to it. Taking the inverse Fourier transform and the antilog at each point then gives us an enhanced image in which the effects of shading have been reduced.

D. Noise Cleaning

Noise that is distinguishable from the image detail is relatively easy to remove from an image. For example, if the image is composed of large objects and the noise consists of high-contrast specks ("salt and pepper"), we can detect the specks as pixels that are very different in gray level from (nearly) all of their neighbors and remove them by replacing each such pixel by the average of its neighbors. As another example, if the noise is a periodic pattern, in the Fourier transform of the image it gives rise to a small set of isolated high values (i.e., specks); these can be detected and removed as just described, and the inverse Fourier transform can be applied to reconstruct an image from which the periodic pattern has been deleted. (This process is sometimes called notch filtering.)

Image noise can also be reduced by averaging operations. If we have several copies of an image that are identical except for the noise (e.g., several photographs or television frames of the same scene), averaging the copies reduces the amplitude of the noise while preserving the image detail. In a single image, local averaging (of each pixel with its neighbors) will reduce noise in "flat" regions of the image, but it will also blur the edges or boundaries of regions. To avoid blurring, one can first attempt to detect edges and then average each pixel that lies near an edge only with those of its neighbors that lie on the same side of the edge. One need not actually decide whether an edge is present but can simply average each pixel only with those of its neighbors whose gray levels are closest to its own, since these neighbors are likely to lie on the same side of the edge (if any) as the pixel. [Better results are obtained by selecting, from each symmetric pair of neighbors, the one whose gray level is closer to that of the pixel. If the image's histogram has distinctive peaks (see Section XI.A), averaging each pixel with those of its neighbors whose gray levels belong to the same peak also has a strong smoothing effect.] Another approach is to examine a set of wedge-shaped neighborhoods extending out from the pixel in different directions and to use the average of that neighborhood whose gray levels are least variable, since such a neighborhood is likely to lie on one side of an edge. A more general class of approaches uses local surface fitting (to the gray levels of the neighbors) rather than local averaging; this too requires modifications to avoid blurring edges.

Another class of noise-cleaning methods is based on rank ordering, rather than averaging, the gray levels of the neighbors of each pixel. The following are two examples of this approach.

1. *Min–max filtering.* Suppose that the noise consists of small specks that are lighter than their surroundings. If we replace each pixel's gray level by the minimum of its neighbors' gray levels, these specks disappear, but light objects also shrink in size. We now replace each pixel by the maximum of its neighbors; this reexpands the light objects, but the specks do not reappear. Specks that are darker than their surroundings can be removed by an analogous process of taking a local maximum followed by a local minimum.
2. *Median filtering.* The median gray level in a neighborhood is the level such that half the pixels in the neighborhood are lighter than it and half are darker. In a "flat" region of an image, the median is usually close to the mean; thus replacing each pixel by the median of its neighbors has much the same effect as local averaging. For a pixel near a (relatively straight) edge, on the other hand, the median of the neighbors will usually be one of the neighbors on the same side of the edge as the pixel, since these neighbors are in the majority; hence replacing the pixel by the median of the neighbors will not blur the edge. Note that both min–max and median filtering destroy thin features such as lines, curves, or sharp corners; if they are used on an image that contains such features, one should first attempt to detect them so they can be preserved.

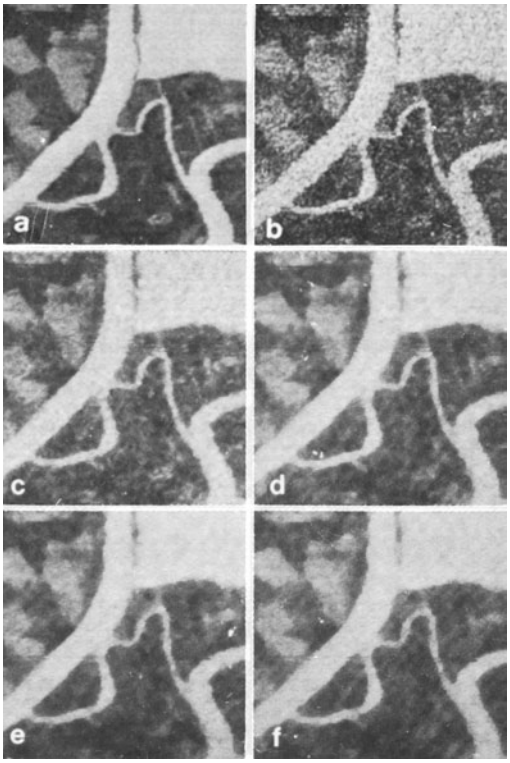


FIGURE 3 Image smoothing by iterated median filtering. [Figure 7 in the previous edition.]

Median filters belong to a class of filters known as order-statistic or rank order-based filters. These filters can effectively handle contamination due to heavy-tailed, non-Gaussian noise, while at the same time not blurring edges or other sharp features, as linear filters almost always do. An example of iterated median filtering is shown in Fig. 3.

VI. RESTORATION

The goal of image restoration is to undo the effects of given or estimated image degradations. In this section we describe some basic methods of restoration, including photometric correction, geometric correction, deconvolution, and estimation (of the image gray levels in the presence of noise).

A. Photometric Correction

Ideally, the gray levels in a digital image should represent the brightnesses at the corresponding points of the scene in a consistent way. In practice, however, the mapping from brightness to gray level may vary from point to point—for example, because the response of the sensor is not uniform or because the sensor collects more light from scene points in the center of its field of view than from points in the periphery ("vignetting").

We can estimate the nonuniformity of the sensor by using images of known test objects. A nonnoisy image of a uniformly bright surface should have a constant gray level; thus variations in its gray level must be due to sensor nonuniformity. We can measure the variation at each point and use it to compute a correction for the gray level of an arbitrary image at that point. For example, if we regard the nonuniformity as an attenuation by the factor $a(x, y)$ at each point, we can compensate for it by multiplying the gray level at (x, y) by $1/a(x, y)$.

B. Geometric Correction

The image obtained by a sensor may be geometrically distorted, by optical aberrations, for example. We can estimate the distortion by using images of known test objects such as regular grids. We can then compute a geometric transformation that will correct the distortion in any image.

A geometric transformation is defined by a pair of functions $x' = \phi(x, y)$, $y' = \psi(x, y)$ that map the old coordinates (x, y) into new ones (x', y'). When we apply such a transformation to a digital image, the input points (x, y) are regularly spaced sample points, say with integer coordinates, but the output points (x', y') can be arbitrary points of the plane, depending on the nature of the transformation. To obtain a digital image as output, we can map the output points into the nearest integer-coordinate points. Unfortunately, this mapping is not one-to-one; some integer-coordinate points in the output image may have no input points mapped into them, whereas others may have more than one.

To circumvent this problem, we use the inverse transformation $x = \phi(x', y')$, $y = \psi(x', y')$ to map each integer-coordinate point of the output image back into the input image plane. This point is then assigned a gray level derived from the levels of the nearby input image points—for example, the gray level of the nearest point or a weighted average of the gray levels of the surrounding points.

C. Deconvolution

Suppose that an image has been blurred by a known process of local weighted averaging. Mathematically, such a blurring process is described by the convolution $(g = h * f)$ of the image f with the pattern of weights h. (The value of $h * f$ for a given shift of h relative to f is obtained by pointwise multiplying them and summing the results.)

By the convolution theorem for Fourier transforms, we have $G = HF$, where F, G, H are the Fourier transforms

of f, g, h, respectively. Thus in principle we can restore the unblurred f by computing $F = G/H$ and inverse Fourier transforming. (At frequencies where H has zeros, G/H is undefined, but we can take $F = G = 0$.) This process is called inverse filtering. If h is not given, we can estimate it from the blurred images of known test objects such as points, lines, or step edges.

The simple deconvolution process just described ignores the effects of noise. A more realistic model for a degraded image is $g = h * f + n$, where n represents the noise. If we try to apply inverse filtering in this situation we have $G = HF + N$, so that $G/H = F + N/H$. Thus at high spatial frequencies, where the noise is stronger than the information-bearing image detail, the results of inverse filtering will be dominated by the noise. To avoid this, the division G/H should be performed only at relatively low spatial frequencies, whereas at higher frequencies G should be left intact.

A more general process known as least-squares filtering or Wiener filtering can be used when noise is present, provided the statistical properties of the noise are known. In this approach, g is deblurred by convolving it with a filter m, chosen to minimize the expected squared difference between f and $m * g$. It can be shown that the Fourier transform M of m is of the form $(1/H)[1/(1 + S)]$, where S is related to the spectral density of the noise; note that in the absence of noise this reduces to the inverse filter: $M = 1/H$. A number of other restoration criteria lead to similar filter designs.

Other methods of deblurring can be defined that assume some knowledge about the statistical properties of the noise. A typical approach is to find an estimate $\hat{f}$ of f such that the "residual" $g - h * \hat{f}$ has the same statistical properties as n. This problem usually has many solutions, and we can impose additional constraints on $\hat{f}$ (that it be nonnegative, "smooth," etc.). Deblurring techniques have also been developed in which the choice of filter dependson the local characteristics of the image or in which the blur itself is assumed to vary in different parts of the image.

D. Estimation

Suppose that an image has been corrupted by the addition of noise, $g = f + n$, where the statistics of the noise are known. We want to find the optimum estimate $\hat{f}$ of the image gray level at each point. If the image has no structure, our only basis for this estimate is the observed gray level g at the given point; but for real-world images, neighboring points are not independent of one another, so that the estimates at neighboring points can also be used in computing the estimate at a given point.

One approach to image estimation, known as Kalman filtering, scans the image row by row and computes the estimate at each pixel as a linear combination of the estimates at preceding, nearby pixels. The coefficients of the estimate that minimizes the expected squared error can be computed from the autocorrelation of the (ideal) image.

Recursive Kalman filters for image restoration are computationally intensive. Approximations such as reduced-update Kalman filters give very good results at a much lower computational complexity. The estimation-theoretic formulation of the image restoration problem using the Kalman filter has enabled investigation of more general cases, including blind restoration (where the unknown blur function is modeled and estimated along with the original image) and nonstationary image restoration (piecewise stationary regions are restored, with the filters appropriate for the regions being selected using a Markov chain).

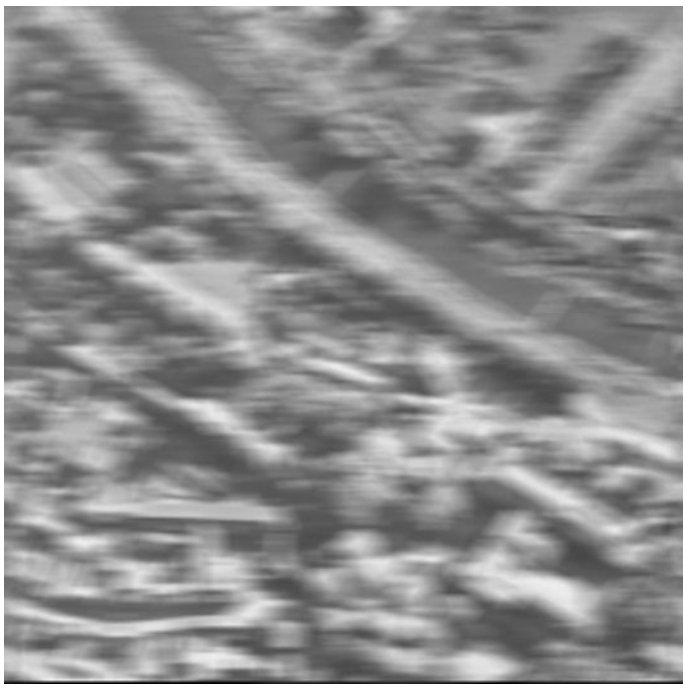
In addition to recursive filters, other model-based estimation-theoretic approaches have been developed. For example, in the Wiener filter described above, one can use random field models (see Section III) to estimate the power spectra needed. Alternatively, one can use MRF models to characterize the degraded images and develop deterministic or stochastic estimation techniques that maximize the posterior probability density function.

A seminal approach that models the original image uisng a composite model, where stationary regions are represented using MRF models, and the discontinuities that separate the stationary regions are represented using "line processes," has yielded a new unified framework for handling a wide variety of problems in image estimation, restoration, surface reconstruction, and texture segmentation. The composite model, when used in conjunction with the MAP criterion, leads to nonconvex optimization problems. A class of stochastic search methods known as simulated annealing and its variants has enabled the solution of such optimization problems. Although these methods are computationally intensive, parallel hardware implementations of the annealing algorithms have taken the bite out of their computational complexity. An image restoration example is shown in Fig. 4.

VII. RECONSTRUCTION

A projection of an image is obtained by summing its gray levels along a family of parallel lines. In this section we show how an image can be (approximately) reconstructed from a sufficiently large set of its projections. This process of reconstruction from projections has important applications in reconstructing images of cross sections of a volume, given a set of X-ray images of the volume taken from different directions. In an X-ray image of an object, the ray striking the film at a given point has been attenuated by passing through the object; thus its strength is

(a)



(b)

FIGURE 4 Image restoration. (a) Original image; (b) blurred image; (c) restored image. [These images were provided by Prof. A. Katsaggelos of Northwestern University.]

proportional to the product of the attenuations produced by unit volumes of the object along its path. On a logarithmic scale, the product becomes a sum, so that an X-ray image (produced by a parallel beam of X rays) can be regarded as a projection of the object, and each row of the image can be regarded as a projection of a planar cross section of the object. Other methods of obtaining projections of a volume have been developed using radioactive tracers or ultrasound. An example of a cross section of the body reconstructed from a set of X rays is shown in Fig. 5.

(c)

FIGURE 4 *(continued)*

By the projection theorem for Fourier transforms, the 1-D transform of a projection of an image is equal to a cross section of the 2-D transform of the image. Specifically, let f_θ be the projection obtained by summing the gray levels of the image f along the family of lines in direction θ. Let F be the Fourier transform of f, and let $F^{\theta'}$ be the cross section of F along the line through the origin in direction $\theta' = \theta + \pi/2$. Then the Fourier transform of f_θ is just $F^{\theta'}$. This implies that if we have projections of f in many directions and we take their Fourier transforms, we obtain cross sections of the Fourier transform F along many lines through the origin. We can approximate F by interpolating from these cross sections and then reconstruct f by inverse Fourier transforming. Note that the

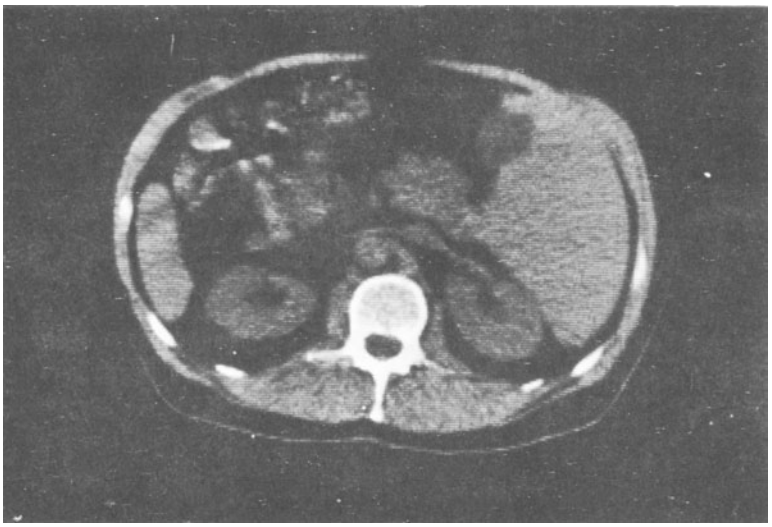


FIGURE 5 Reconstruction of a cross section of the human body from a set of X rays.

high-frequency components of f will not be reconstructed accurately using this method, since the cross-section data far from the origin are sparse.

Another approach to reconstruction from projections is based on "back-projection." The value v of a projection at a point is the sum of the values of the original image along a given line; to back-project, we divide v into many equal parts and distribute them along that line. If we do this for every projection, we obtain a highly blurred version of the original image. (To see this, suppose that the image contains a bright point P on a dark background; then P will give rise to a high value in each projection. When we back-project, these values will be spread along a set of lines that all intersect at P, giving rise to a high value at P and lower values along the lines radiating from it.) To obtain a more accurate reconstruction of the image, we combine back-projection with deblurring; we first filter the projections to precompensate for the blurring and then back-project them. This method of reconstruction is the one most commonly used in practice.

Each projection of an image f gives us a set of linear equations in the gray levels of f, since the value of a projection at a point is the sum of the gray levels along a line. Thus if we have a large enough set of projections of f, we can, in principle, solve a large set of linear equations to determine the original gray levels of f. Many variations of this algebraic approach to reconstruction have been formulated. (In principle, algebraic techniques can also be used in image deblurring; the gray level of a pixel in the blurred image is a weighted average of neighboring gray levels in the ideal image, so that the ideal gray levels can be found by solving a set of linear equations.)

Over the last 15 years, the emphasis in image reconstruction has been on introducing probabilistic or statistical principles in modeling noise and imaging mechanisms and deriving mathematically sound algorithms. As multimodal images (X ray, CT, MRI) are becoming increasingly available, registration of these images to each other and positioning of these images to an atlas have also become critical technologies. Visualization of reconstructed images and objects is also gaining attention.

VIII. MATCHING

There are many situations in which we want to "match" or compare two images with one another. The following are some common examples.

1. We can detect occurrences of a given pattern in an image by comparing the image with a "template" of the pattern. This concept has applications in pattern recognition, where we can use template matching to recognize known patterns, and also in navigation, where we can use landmark matching as an aid in determining the location from which an image was obtained.
2. Given two images of a scene taken from different positions, if we can identify points in the two images that correspond to a given scene point, we can determine the 3-D location of the scene point by triangulation. This process is known as stereomapping. The identification of corresponding points involves local matching of pieces of the two images.
3. Given two images of a scene taken (from the same position) at different times, we can determine the points at which they differ and analyze the changes that have taken place in the scene.

This section discusses template matching and how to measure the match or mismatch between two images, image registration, and stereomapping and range sensing. The analysis of time sequences of images is discussed in Section IX.

A. Template Matching

A standard measure of the "mismatch" or discrepancy between two images f and g is the sum of the absolute (or squared) differences between their gray levels at each point—for example, $\sum\sum(f-g)^2$. A standard measure of match is the correlation coefficient $\sum\sum fg/\sqrt{(\sum\sum f^2)(\sum\sum g^2)}$; by the Cauchy–Schwarz inequality, this is always ≤ 1 and is equal to 1 if f and g differ by a multiplicative constant. Thus to find places where the image f matches the template g, we can cross-correlate g with f (i.e., compute $\sum\sum fg$ for all shifts of g relative to f) and look for peaks in the correlation value. Note that by the convolution theorem for Fourier transforms, we can compute the cross-correlation by pointwise multiplying the Fourier transforms F and G^* (where the asterisk denotes the complex conjugate) and then inverse transforming.

The matched filter theorem states that if we want to detect matches between f and g by cross-correlating a filter h with f, and the criterion for detection is the ratio of signal power to expected noise power, then the best filter to use is the template g itself. Depending on the nature of f and the detection criterion, however, other filters may be better; for example, if f is relatively "smooth," better results are obtained by correlating the derivative of f with the derivative of g, or f with the second derivative of g.

Matching of derivatives generally yields sharper match peaks, but it is more sensitive to geometric distortion. To handle distortion, a good approach is first to match the

image with smaller (sub)templates, since these matches will be less sensitive to distortion, and then to look for combinations of these matches in approximately the correct relative positions.

Matching by cross-correlation is a computationally expensive process. We can reduce its cost by using inexpensive tests to determine positions in which a match is likely to be found, so that we need not test for a match in every possible position. One possibility is to match at a low resolution (i.e., at coarsely spaced points) and search for match peaks only in the vicinity of those points where the coarse match is good. An alternative is to match with a subtemplate and check the rest of the template only at those points where the subtemplate matches well. Note that if we use a very low resolution, or a very small subtemplate, there will be many false alarms, but if we use a relatively high resolution, or a relatively large subtemplate, the saving will not be significant.

B. Image Registration

Before we can match two or more images, they may have to be aligned to compensate for different acquisition viewpoints or times. For example, if we wish to combine information from CT, X-ray and MRI images, they have to be registered to a common coordinate frame before fusion or change detection can be attempted. Similar situations arise in image exploitation and automatic target recognition applications. There are also situations where instead of registering multiple images to each other, one needs to register multiple images, acquired at different times, to a 3-D model. These methods are referred to as model-supported positioning methods. Most traditional methods of image registration based on area correlation or feature matching can handle only minor geometric and photometric variations (typically, in images collected by the same sensor). In a multisensor context, however, the images to be registered may be of widely different types, obtained by disparate sensors with different resolutions, noise levels, and imaging geometries. The common or "mutual" information, which is the basis of automatic image registration, may manifest itself in a very different way in each image. This is because different sensors record different physical phenomena in the scene. For instance, an infrared sensor responds to the temperature distribution of the scene, whereas a radar responds to material properties such as dielectric constant, electrical conductivity, and surface roughness.

Since the underlying scene giving rise to the shared information is the same is all images of the scene, certain qualitative statements can be made about the manner in which information is preserved across multisensor data. Although the pixels corresponding to the same scene region may have different values depending on the sensor, pixel similarity and pixel dissimilarity are usually preserved. In other words, a region that appears homogeneous to one sensor is likely to appear homogeneous to another, local textural variations apart. Regions that can be clearly distinguished from one another in one image are likely to be distinguishable from one another in other images, irrespective of the sensor used. Although this is not true in all cases, it is generally valid for most types of sensors and scenes. Man-made objects such as buildings and roads in aerial imagery, and implants, prostheses, and metallic probes in medical imagery, also give rise to features that are likely to be preserved in multisensor images. Feature-based methods that exploit the information contained in region boundaries and in man-made structures are therefore useful for multisensor registration.

Feature-based methods traditionally rely on establishing feature correspondences between the two images. Such correspondence-based methods first employ feature matching techniques to determine corresponding feature pairs in the two images and then compute the geometric transformation relating them, typically using a least-squares approach. Their primary advantage is that the transformation parameters can be computed in a single step and are accurate if the feature matching is reliable. Their drawback is that they require feature matching, which is difficult to accomplish in a multisensor context and is computationally expensive, unless the two images are already approximately registered or the number of features is small.

Some correspondence-less registration methods based on moments of image features have been proposed, but these techniques, although mathematically elegant, work only if the two images contain exactly the same set of features. This requirement is rarely met in real images. Another proposed class of methods is based on the generalized Hough transform (GHT). These methods map the feature space into a parameter space, by allowing each feature pair to vote for a subspace of the parameter space. Clusters of votes in the parameter space are then used to estimate parameter values. These methods, although far more robust and practical than moment-based methods, have some limitations. Methods based on the GHT tend to produce large numbers of false positives. They also tend to be computationally expensive, since the dimensionality of the problem is equal to the number of transformation parameters.

Recently, methods similar in spirit to GHT-style methods, but employing a different search strategy to eliminate the problems associated with them, have been proposed. These methods first decompose the original transformation into a sequence of elementary stages. At each stage, the value of one transformation parameter

is estimated by a feature consensus mechanism in which each feature pair is allowed to estimate the value of the parameter that is consistent with it. The value of the parameter that is consistent with the most feature pairs is considered to be its best estimate. Using the concepts of parameter observability and parameter separability, it is possible in most cases to avoid the need for feature pairings; instead, aggregate properties of features determined from each image separately are used.

The global registration achieved by feature consensus should be sufficient for many applications such as those employing registration for determining a focus of attention. If more accurate global registration is needed, as in medical applications, the feature consensus result may be used as an initial condition for more elaborate schemes that use feature correspondence or multidimensional search, which require a good initial guess about the transformation parameters. Methods like deformable template matching can also be invoked for local refinement of the registration. An example of image registration is shown in Fig. 6.

C. Stereomapping and Range Sensing

Let P be a point in a scene and let P_1 and P_2 be the points corresponding to P in two images obtained from two known sensor positions. Let the "lens centers" in these two positions be L_1 and L_2, respectively. By the geometry of optical imaging, we know that P must lie on the line P_1L_1 (in space) and also on the line P_2L_2; thus its position in space is completely determined.

The difficulty in automating this process of stereomapping is that it is difficult to determine which pairs of image points correspond to the same scene point. (In fact, some scene points may be visible in one image but hidden in the other.) Given a point P_1 in one image, we can attempt to find the corresponding point P_2 in the other image by matching a neighborhood of P_1 with the second image. (We need not compare it with the entire image; since the camera displacement is known, P_2 must lie on a known line in the second image.) If the neighborhood used is too large, geometric distortion may make it impossible to find a good match; but if it is too small, there will be many false matches, and in a featureless region of the image, it will be impossible to find unambiguous (sharp) matches. Thus the matching approach will yield at best a sparse set of reasonably good, reasonably sharp matches. We can verify the consistency of these matches by checking that the resulting positions of the scene points in space lie on a smooth surface or a set of such surfaces. In particular, if the matches come from points that lie along an edge in an image, we can check that the resulting spatial positions lie on a smooth space curve.

(a)

FIGURE 6 Image registration. (a, b) The two images to be registered; (c) the registration result ("checkerboard" squares show alternating blocks of the two registered images.)

(b)

FIGURE 6 *(continued)*

(c)

FIGURE 6 *(continued)*

A strong theoretical basis for this approach evolved around the midseventies. Since then, advances have been made in interpolating the depth estimates obtained at positions of matched image features using surface interpolation techniques and hierarchical feature-based matching schemes, and dense estimates have been obtained using gray-level matching guided by simulated annealing. Although these approaches have contributed to a greater understanding of the problem of depth recovery using two cameras, much more tangible benefits have been reaped using a larger number of cameras. By arranging them in an array pattern, simple sum of squared difference-based schemes are able to produce dense depth estimates in real time. Using large numbers of cameras (in excess of 50), new applications in virtual reality, 3-D modeling, and computer-assisted surgery have become feasible.

Consistent with developments in multiscale analysis, stereo mapping has benefited from multiscale feature-based matching techniques. Also, simulated annealing and neural networks have been used for depth estimation using two or more images.

Another approach to determining the spatial positions of the points in a scene is to use patterned illumination. For example, suppose that we illuminate the scene with a plane of light Π, so that only those scene points that lie in Π are illuminated, and the rest are dark. In an image of the scene, any visible scene point P (giving rise to image point P_1) must lie on the line P_1L_1; since P must also lie in Π, it must be at the intersection of P_1L_1 and Π, so that its position in space is completely determined. We can obtain complete 3-D information about the scene by moving Π through a set of positions so as to illuminate every visible scene point, or we can use coded illumination in which the rays in each plane are distinctive (e.g., by their colors). A variety of "range sensing" techniques based on patterned illumination has been developed. Still another approach to range sensing is to illuminate the scene, one point at a time, with a pulse of light and to measure the time interval (e.g., the phase shift) between the transmitted and the reflected pulses, thus obtaining the range to that scene point directly, as in radar.

IX. IMAGE SEQUENCE ANALYSIS

Suppose that we are given a sequence of images (or frames) of the same scene taken at different times from the same position. By comparing the frames, we can detect changes that have taken place in the scene between one frame and the next. For example, if the frames are closely spaced in time and the changes are due to the motions of discrete objects in the scene, we can attempt to track the objects from frame to frame and so determine their motions. One way to do this is to match pieces of consecutive frames that contain images of a given object to estimate the displacement of that object. However, if the object is moving in three dimensions, the size and shape of its image may change from frame to frame, so that it may be difficult to find good matches.

During the early eighties, much effort was focused on estimating the motions and depths of features (points, lines, planar or quadric patches) using two or three frames. This problem was broken into two stages: establishing correspondences of features between frames and estimating motion and depth from the corresponding features using linear or nonlinear algorithms. However, the performance of these algorithms on real image sequences was not satisfactory.

To exploit the large numbers of frames in video image sequences, model-based recursive filtering approaches for estimating motion and depth were suggested in the mideighties. This approach led to applications of extended Kalman filters and their variants to problems in image sequence analysis. In general, model-based approaches enabled the use of multiple cameras, auxilary information such as inertial data, and statistical analysis. Successful applications of recursive filters to automobile navigation have been demonstrated. Figure 7 illustrates the use of recursive filters for feature tracking.

If the motion from frame to frame is not greater than the pixel spacing, then by comparing the space and time derivatives of the gray-level at a given point, in principle we can estimate the component of the image motion at that point in the direction of the gray-level gradient, but the component in the orthogonal (tangential) direction is ambiguous. Equivalently, when we look at a moving (straight) edge, we can tell how fast it is moving in the direction perpendicular to itself but not how fast it is sliding along itself. Unambiguous velocity estimates can be obtained at corners where two edges meet. Such estimates can be used as boundary conditions to find a smooth velocity field that agrees with the observed velocities at corners and whose

FIGURE 7 Feature tracking using recursive filtering.

components in the gradient direction agree with the observed components perpendicular to edges.

When a sequence of frames is taken by a moving sensor, there will be changes nearly everywhere, with the magnitude of the change at a given image point depending on the velocity of the sensor and on its distance from the corresponding scene point. The array of motion vectors at all points of the image is known as the optical flow field. An example of such a field is shown in Fig. 8. Different types of sensor motion (ignoring, for the moment, the motions of objects in the scene) give rise to different types of flow fields. Translational sensor motion perpendicular to the optical axis of the sensor simply causes each point of the image to shift, in the opposite direction, by an amount proportional to its distance from the sensor, so that the resulting image motion vectors are all parallel. Translation in other directions, on the other hand, causes the image to expand or shrink, depending on whether the sensor is approaching or receding from the scene; here the motion vectors all pass through a common point, the "focus of expansion" (or contraction). Thus if we know the translational sensor motion, we can compute the relative distance from the sensor to each point of the scene. (The distances are only relative because changes in absolute distance are indistinguishable from changes in the speed of the senser.) The effects of rotational sensor motion are more complex, but they can be treated independently of the translation effects.

Motion of a rigid body relative to the sensor gives rise to a flow field that depends on the motion and on the shape of the body. In principle, given enough (accurate) measurements of the flow in a neighborhood, we can determine both the local shape of the body and its motion.

Parallel to, and often independent of, the correspondence-based approach, optical flow-based structure and motion estimation algorithms have flourished for more than two decades. Although robust dense optical flow estimates are still elusive, the field has matured to the extent that systematic characterization and evaluation of optical flow estimates are now possible. Methods that use robust statistics, generalized motion models, and filters have all shown great promise. Significant work that uses directly observable flow ("normal flow") provides additional insight into the limitations of traditional approaches. An example of depth estimation using optical flow is shown in Fig. 8.

Segmentation of independently moving objects and dense scene structure estimation from computed flow have become mature research areas. New developments such as fast computation of depth from optical flow using fast Fourier transforms have opened up the possibility of real-time 3-D modeling. Another interesting accomplishment has been the development of algorithms

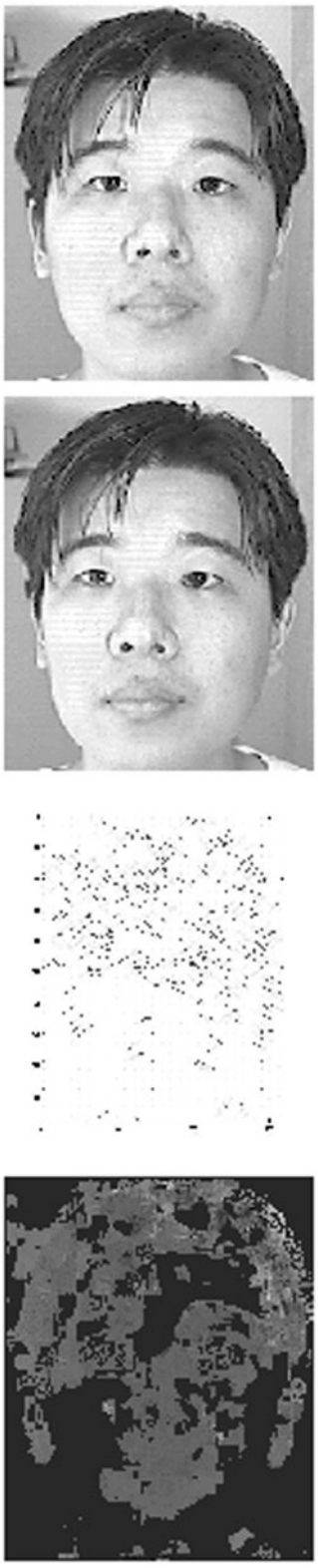


FIGURE 8 Depth estimation from optical flow. (a, b) Two frames of a video sequence; (c) computed flow field; (d) depth map; (e, f) views synthesized from the depth map.

for sensor motion stabilization and panoramic view generation.

Detection of moving objects, structure and motion estimation, and tracking of 2-D and 3-D object motion using contours and other features have important applications in surveillance, traffic monitoring, and automatic target recognition. Using methods ranging from recursive Kalman filters to the recently popular Monte Carlo Markov chain algorithms (known as CONDENSATION algorithms), extensive research has been done in this area. Earlier attempts were concerned only with generic tracking, but more recently, "attributed tracking," where one incorporates the color, identity, or shape of the object as part of the tracking algorithm, is gaining importance. Also, due to the impressive computing power that is now available, real-time tracking algorithms have been demonstrated.

X. RECOVERY

The gray level of an image at a given point P_1 is proportional to the brightness of the corresponding scene point P as seen by the sensor; P is the (usually unique) point on the surface of an object in the scene that lies along the line P_1L_1 (see Section VIII.C). The brightness of P depends in turn on several factors: the intensity of the illumination at P, the reflective properties of the surface S on which P lies, and the spatial orientation of S at P. Typically, if a light ray is incident on S at P from direction i, then the fraction r of the ray that emerges from S in a given direction e is a function of the angles θ_i and θ_e that i and e, respectively, make with the normal n to S at P. For example, in perfect specular reflection we have $r = 1$, if i and e are both coplanar with n and $\theta_i = \theta_e$, and $r = 0$ otherwise. In perfectly diffuse or Lambertian reflection, on the other hand, r depends only on θ, and not on θ_e; in fact, we have $r = p \cos\theta_i$, where p is a constant between 0 and 1.

If we could separate the effects of illumination, reflectivity, and surface orientation, we could derive 3-D information about the visible surfaces in the scene; in fact, the surface orientation tells us the rate at which the distance to the surface is changing, so that we can obtain distance information (up to a constant of integration) by integrating the orientation. The process of inferring scene illumination, reflectivity, and surface orientation from an image is called recovery (more fully, recovery of intrinsic scene characteristics from an image). Ideally, recovery gives us a set of digital image arrays in which the value of a pixel represents the value of one of these factors at the corresponding scene point; these arrays are sometimes called intrinsic images.

In this section we briefly describe several methods of inferring intrinsic scene characteristics from a single image. These methods are known as "shape from . . ." techniques, since they provide information about the 3-D shapes of the visible surfaces in the scene.

A. Shape from Shading

Suppose that a uniformly reflective surface is illuminated by a distant, small light source in a known position and is viewed by a distant sensor. Then the directions to the light source and to the sensor are essentially the same for two nearby surface points P and P'. Thus the change in surface brightness (and hence in image gray level) as we move from P to P' is due primarily to the change in direction of the surface normal. In other words, from the "shading" (i.e., the changes in gray level) in a region of the image, we can compute constraints on the changes in orientation of the corresponding surface in the scene. These constraints do not determine the orientation completely, but we can sometimes estimate the orientation of a smooth surface from shading information with the aid of boundary conditions (e.g., contours along which the surface orientation is known).

During the past two decades, significant strides have been made in recovering shape from shading. Using optimal control-theoretic principles, deeper understanding of the existence and uniqueness of solutions has become possible. Computational schemes using calculus of variations, multigrid methods, and linearization of the reflectance map have enabled practical implementations. Robust methods for simultaneously estimating the illuminant source direction and the surface shape have also been developed. Extensions to other imaging modalities, such as synthetic aperture radar and fusion of steropsis with shape from shading, have expanded the domain of applications.

If we can illuminate a scene successively from two (or more) directions, we can completely determine the surface orientation, for any surface that is visible in both images, by combining the constraints obtained from the shading in the two images. This technique is known as photometric stereo; an example is shown in Fig. 9.

An abrupt change in image gray level (i.e., an "edge" in an image) can result from several possible causes in the scene. It might be due to an abrupt change in illumination intensity; that is, it might be the edge of a shadow. It might also be due to an abrupt change in surface orientation (e.g., at an edge of a polyhedron), or it might be due to a change in reflectivity (e.g., an occluding edge where one surface partially hides another). Ideally, it should be possible to distinguish among these types of edges by careful analysis of the gray-level variations in their vicinities. For example, a shadow edge might not be very sharp, a convex orientation edge might have a highlight on it, and so on.

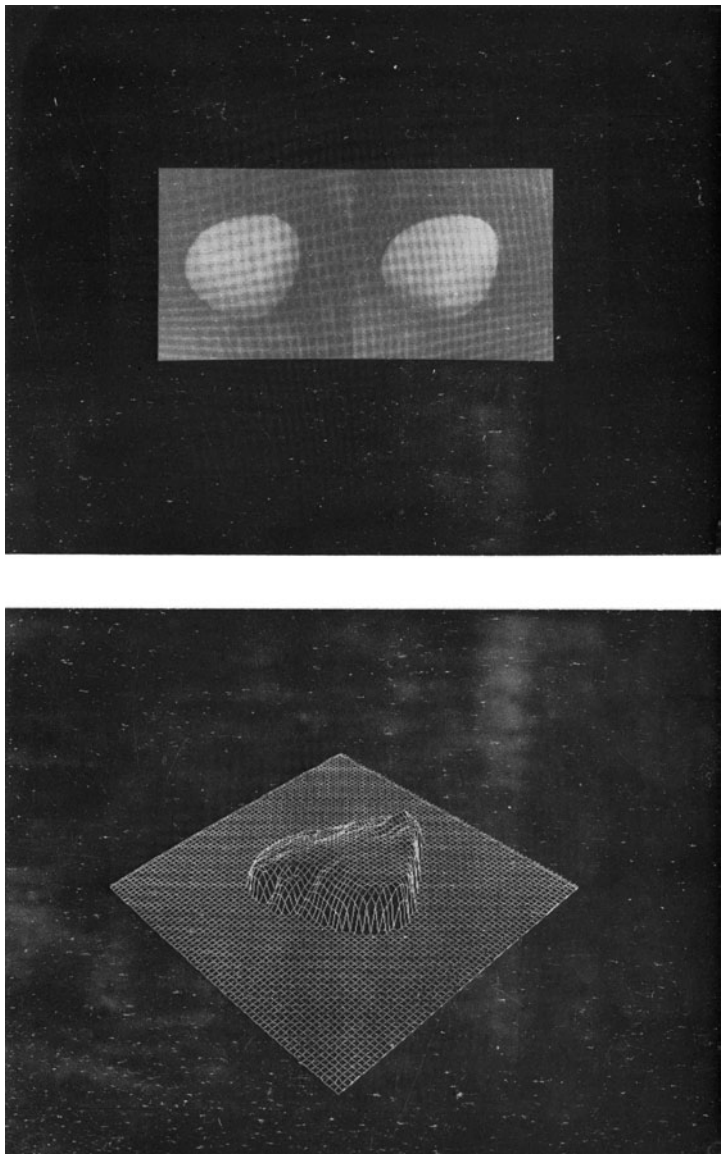


FIGURE 9 Photometric stereo.

B. Shape from Texture

Suppose that a surface is uniformly covered with an isotropic pattern or "texture." If the surface is flat and faces directly toward the sensor, its image will also be uniformly textured. On the other hand, if the surface is slanted, the patterns in the image will be foreshortened in the direction of the slant due to perspective distortion. Moreover, the scale of the image will change along the direction of the slant, since the distance from the sensor to the surface is changing; hence the sizes and spacings of the patterns will vary as we move across the image.

The slant of a uniformly textured surface can be inferred from anisotropies in the texture of the corresponding image region. For example, in an isotropic texture, the distribution of gray-level gradient directions should be uniform, but if the textured surface is slanted, the distribution of directions should have a peak in the direction of slant, due to the foreshortening effect, and the sharpness of the peak gives an indication of the steepness of the slant.

C. Shape from Shape

Various global assumptions can be used to infer surface orientation from the 2-D shapes of regions in an image. If the regions arise from objects of known shapes, it is straightforward to deduce the orientations of the objects from the shapes of their images, or their ranges from their image sizes. Even if the object shapes are unknown, certain inferences can be regarded as plausible. For example, if

an observed shape could be the projected image of a more symmetric or more compact slanted shape (e.g., an ellipse could be the projection of a slanted circle), one might assume that this is actually the case; in fact, human observers frequently make such assumptions. They also tend to assume that a continuous curve in an image arises from a continuous curve in space, parallel curves arise from parallel curves, straight lines arise from straight lines, and so on. Similarly, if two shapes in the image could arise from two congruent objects at different ranges or in different orientations, one tends to conclude that this is true.

A useful assumption about a curve in an image is that it arises from a space curve that is as planar as possible and as uniformly curved as possible. One might also assume that the surface bounded by this space curve has the least possible surface curvature (it is a "soap bubble" surface) or that the curve is a line of curvature of the surface. Families of curves in an image can be used to suggest the shape of a surface very compellingly; we take advantage of this when we plot perspective views of 3-D surfaces.

XI. SEGMENTATION

Images are usually described as being composed of parts (regions, objects, etc.) that have certain properties and that are related in certain ways. Thus an important step in the process of image description is segmentation, that is, the extraction of parts that are expected to be relevant to the desired description. This section reviews a variety of image segmentation techniques.

A. Pixel Classification

If a scene is composed of surfaces each of which has a constant orientation and uniform reflectivity, its image will be composed of regions that each have an approximately constant gray level. The histogram of such an image (see Section V.A) will have peaks at the gray levels of the regions, indicating that pixels having these levels occur frequently in the image, whereas other gray levels occur rarely. Thus the image can be segmented into regions by dividing the gray scale into intervals each containing a single peak. This method of segmentation is called thresholding.

Thresholding belongs to a general class of segmentation techniques in which pixels are characterized by a set of properties. For example, in a color image, a pixel can be characterized by its coordinates in color space—e.g., its red, green, and blue color components. If we plot each pixel as a point in color space, we obtain clusters of points corresponding to the colors of the surfaces. We can thus segment the image by partitioning the color space into regions each containing a single cluster. This general method of segmentation is called pixel classification.

Even in a black-and-white image, properties other than gray level can be used to classify pixels. To detect edges in an image (Section XI.B), we classify each pixel as to whether or not it lies on an edge by thresholding the rate of change of gray level in the neighborhood of that pixel; that is, we classify the pixel as edge or nonedge, depending on whether the rate of change is high or low. Similarly, we detect other features, such as lines or curves, by thresholding the degree to which the neighborhood of each pixel matches a given pattern or template.

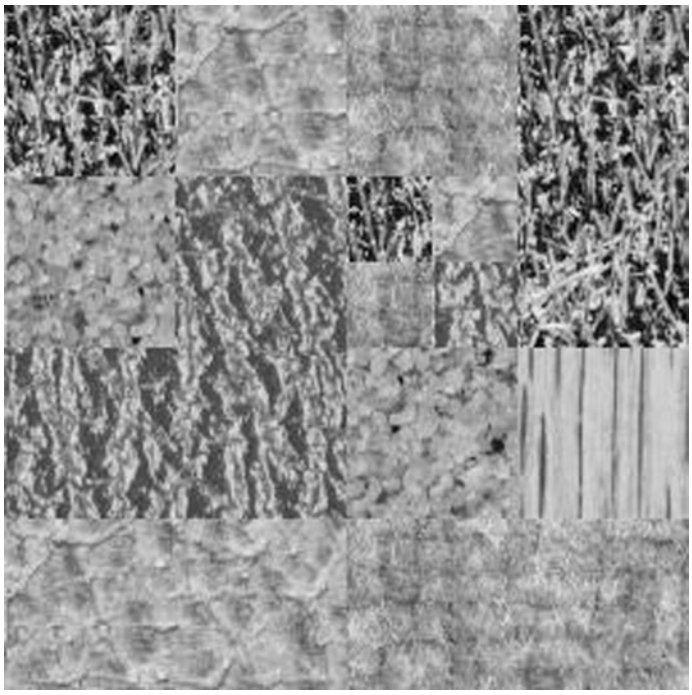
Properties derived from the neighborhoods of pixels—in brief, local properties—can also be used to segment an image into regions. For example, in a "busy" region the rate of change of gray level is often high, whereas in a "smooth" region it is always low. Suppose that we compute the rate of change at each pixel and then locally average it over a neighborhood of each pixel; then the average will be high for pixels in busy regions and low for pixels in smooth regions, so we can segment the image into such regions by thresholding the local average "busyness." We can use this method to segment an image into various types of differently "textured" regions by computing a set of local properties at each pixel and locally averaging their values; pixels belonging to a given type of textured region will give rise to a cluster in this "local property space."

Early pixel-based classification schemes made the assumption that adjacent pixels are uncorrelated. Since the mideighties, pixel classifications methods have tended to exploit the local correlation of pixels in single- or multiband images (often called "context" in the remote sensing literature). The use of MRFs and Gibbs distributions to characterize the local spatial/spectral correlation, combined with the use of estimation-theoretic concepts, has enabled the development of powerful pixel classification methods. An example of region segmentation is shown in Fig. 10.

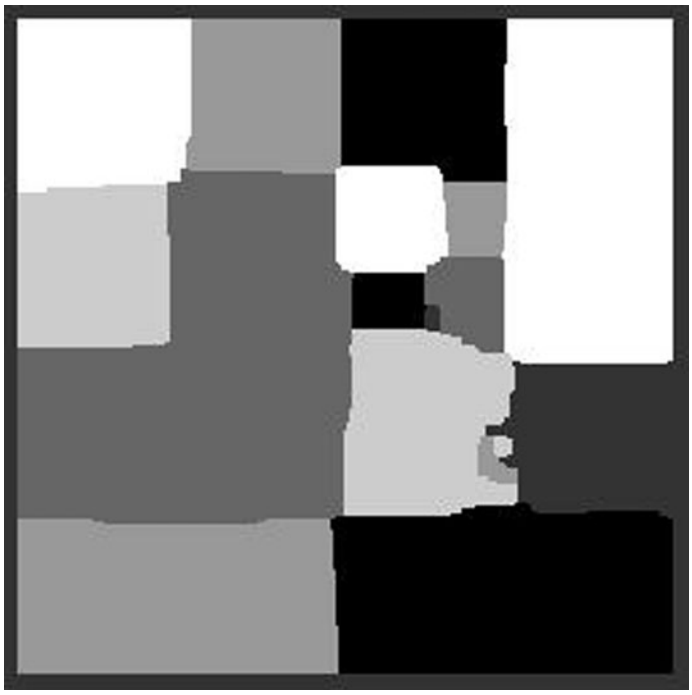
We have assumed up to now that the image consists of regions each of which has an approximately constant gray level or texture. If the scene contains curved surfaces, they will give rise to regions in the image across which the gray level varies; similarly, slanted textured surfaces in the scene will give rise to regions across which the texture varies. If we can estimate the orientations of the surfaces in the scene, as in Section X, we can correct for the effects of orientation.

B. Feature Detection

The borders of regions in an image are usually characterized by a high rate of change of gray level [or color or local property value(s)]. Thus we can detect region borders, or edges, in an image by measuring this rate of change (and thresholding it; see Section XI.A). For simplicity, we discuss only edges defined by gray-level changes, but similar



(a)



(b)

FIGURE 10 Texture segmentation. (a) Original texture mosaic; (b) segmented result.

remarks apply to other types of edges. The following are some standard methods of detecting edges.

1. Estimate the rate of change of gray level at each pixel P in two perpendicular directions using first-difference operators; let the estimates be f_1 and f_2. The gray-level gradient (i.e., greatest rate of change) at P is the vector whose magnitude is $\sqrt{f_1^2 + f_2^2}$ and whose direction is $\tan^{-1}(f_2/f_1)$.
2. Fit a polynomial surface to the gray levels in the neighborhood of P and use the gradient of the surface as an estimate of the gray level gradient.

3. Match a set of templates, representing the second derivatives of gray-level "steps" in various orientations, to the neighborhood of P (see Section VIII.A) and pick the orientation for which the match is best. (It turns out that convolving such a template with the image amounts to applying a first-difference operator at each pixel; thus this method amounts to computing first differences in many directions and picking the direction in which the difference is greatest.) Alternatively, fit a step function to the gray levels in the neighborhood of P and use the orientation and height of this step to define the orientation and contrast of the edge at P.
4. Estimate the Laplacian of the gray level at each pixel P. Since the Laplacian is a second-difference operator, it is positive on one side of an edge and negative on the other side; thus its zero-crossings define the locations of edges. (First differences should also be computed to estimate the steepnesses of these edges.)

Although the use of the Laplacian operator for edge detection has long been known, the idea of applying the Laplacian operator to a Gaussian smoothed image, using filters of varying sizes, stimulated theoretical developments in edge detection. In the mideighties, edge detectors that jointly optimize detection probability and localization accuracy were formulated and were approximated by directional derivatives of Gaussian-smoothed images. Fig. 11 shows an example of edge detection using this approach.

All these methods of edge detection respond to noise as well as to region borders. It may be necessary to smooth the image before attempting to detect edges; alternatively, one can use difference operators based on averages of blocks of gray levels rather than on single-pixel gray levels, so that the operator itself incorporates some smoothing. Various types of statistical tests can also be used to detect edges, based, for example, on whether the set of gray levels in two adjacent blocks comes from a single population or from two populations. Note that edges in an image can arise from many types of abrupt changes in the scene, including changes in illumination, reflectivity, or surface orientation; refinements of the methods described in this section would be needed to distinguish among these types.

To detect local features such as lines (or curves or corners) in an image, the standard approach is to match the image with a set of templates representing the second derivatives of lines (etc.) in various orientations. It turns out that convolving such a template with the image amounts to applying a second-difference operator at each pixel; thus this method responds not only to lines, but also to edges or to noise. Its response can be made more specific by applying the templates only when appropriate logical conditions are satisfied; for example, to detect a dark vertical line (on a light background) at the pixel P, we can require that P and its vertical neighbors each be darker than their horizontal neighbors. (Similar conditions can be incorporated when using template matching to detect edges.) To detect "coarse" features such as thick lines, similar templates and conditions can be used, but scaled up in size, that is, based on blocks of pixels rather than on single pixels.

It is usually impractical to use templates to detect global features such as long straight edges or lines, since too many templates would be needed to handle all possible positions and orientations. An alternative idea, called the Hough transform approach, is to map the image into a parameter space such that any feature of the given type gives rise to a peak in the space. For example, any straight line is characterized by its slope θ and its (shortest) distance ρ from the origin. To detect long straight lines in an image, we match it with local templates as described above; each time a match is found, we estimate its θ and ρ and plot them as a point in (θ, ρ) space. When all matches have been processed in this way, any long straight lines in the image should have given rise to clusters of points in (θ, ρ) space. Note that this technique detects sets of collinear line segments, whether or not they form a continuous line. A similar approach can be used to find edges or curves of any specific shape. Perceptual grouping of local features, using the Gestalt laws of grouping, provides an alternative approach when explicit shape information is not available.

C. Region Extraction

The segmentation techniques described in Sections XI.A and XI.B classify each pixel independently, based on its local properties; they are oblivious to whether the resulting set of pixels forms connected regions or continuous edges (etc.), and they therefore often give rise to noisy regions or broken edges. In this subsection we briefly discuss methods of segmentation that take continuity into account.

One approach is to require that the pixel classifications be locally consistent. If the classes represent region types and most of the neighbors of pixel P are of a given type, P should also be of that type. In edge (or curve) detection, if P lies on an edge in a given orientation, its neighbors in that direction should lie on edges in similar orientations. Once we have defined such consistency constraints, we can formulate the pixel classification task as one of optimization: Classify the pixels so as to maximize the similarity between each pixel and its class, while at the same time maximizing the consistencies between the classifications of neighboring pixels. Standard iterative methods can be

(a)

FIGURE 11 Edge detection. (a) Original image; (b) detected edges.

used to solve this optimization problem. If we are more concerned with consistency than with class similarity, we can simply reclassify pixels so as to make the results more consistent, thus reducing the "noise" in classified regions (or deleting isolated edge pixels), filling gaps in edges, and so on. The criteria of region consistency and edge consistency can, in principle, be combined to extract regions that are homogeneous and are surrounded by strong edges.

Another way of enforcing local continuity is to "grow" the regions by starting with (sets of) pixels whose classifications are clear-cut and extending them by repeatedly adding adjacent pixels that resemble those already classified. Similarly, to obtain continuous edges (or curves) we can start with strong-edge pixels and extend or "track" them in the appropriate directions. More generally, if we are given a set of region fragments or edge fragments, we can merge them into larger fragments based on similarity

(b)

FIGURE 11 *(continued)*

or consistency criteria. The results of these approaches will generally depend on the order in which the pixels or fragments are examined for possible incorporation or merging; it may be desirable to use search techniques (lookahead, backtracking) as an aid in making good choices.

We can also use global consistency criteria in segmenting an image; for example, we can require that the gray levels in a region be approximately constant (or, more generally, be a good fit to a planar or higher-order surface) or that a set of edge or line elements be a good fit to a straight line (or higher-order curve). To impose such a condition, we can start with the entire image, measure the fit, and split the image (e.g., into quadrants) if the fit is not good enough. The process can then be repeated for each quadrant until we reach a stage where no further splitting is needed; we have then partitioned the image into blocks on each of which the fit is good. Conversely, given such a partition, we may be able to merge some pairs of adjacent

parts and still have good fits to the merged parts. By using both splitting and merging, we can arrive at a partition such that the fit on each part is acceptable, but no two parts can be merged and still yield an acceptable fit. Note that by fitting surfaces to the gray levels we can, in principle, handle regions that arise from curved or shaded surfaces in the scene.

In many situations we have prior knowledge about the sizes and shapes of the regions that are expected to be present in the image, in addition to knowledge about their gray levels, colors, or textures. In segmentation by pixel classification we can make use of information about gray level (etc.) but not about shape. Using region-based methods of segmentation makes it easier to take such geometric knowledge into account; we can use region-growing, merging, or splitting criteria that are biased in favor of the desired geometries, or, more generally, we can include geometry-based cost factors in searching for an optimal partition. Cost criteria involving the spatial relations among regions of different types can also be used.

XII. GEOMETRY

In this section we discuss ways of measuring geometric properties of image subsets (regions or features) and of decomposing them into parts based on geometric criteria. We also discuss ways of representing image subsets exactly or approximately.

A. Geometric Properties

Segmentation techniques based on pixel classification can yield arbitrary sets of pixels as "segments." One often wants to consider the connected pieces of a subset individually, in order to count them, for example.

Let P and Q be pixels belonging to a given subset S. If there exists a sequence of pixels $P = P_0, P_1, \ldots, P_n = Q$, all belonging to S, such that, for each i, P_i is a neighbor of P_{i-1}, we say that P and Q are connected in S. The maximal connected subsets of S are called its (connected) components. We call S connected if it has only one component.

Let $\bar{S}$ be the complement of S. We assume that the image is surrounded by a border of pixels all belonging to $\bar{S}$. The component of $\bar{S}$ that contains this border is called the background of S; all other components of $\bar{S}$, if any, are called holes in S.

Two image subsets S and T are called adjacent if some pixel of S is a neighbor of some pixel of T. Let $S_1, \ldots, S_m$ be a partition of the image into subsets. We define the adjacency graph of the partition as the graph whose nodes are $S_1, \ldots, S_m$ and in which nodes S_i and S_j are joined by an arc if and only if S_i and S_j are adjacent. It can be shown that if we take $S_1, \ldots, S_m$ to be the components of S and of $\bar{S}$, where S is any subset of the image, then the adjacency graph of this partition is a tree.

The border of S consists of those pixels of S that are adjacent to (i.e., have neighbors in) $\bar{S}$. More precisely, if C is any component of S and D any component of $\bar{S}$, the D-border of C consists of those pixels of C (if any) that are adjacent to D. In general, if $S = C$ is connected, it has an outer border composed of pixels adjacent to the background, and it may also have hole borders composed of pixels adjacent to holes.

The area of S is the number of pixels in S. The perimeter of S is the total length of all its borders; it can be defined more precisely, for any given border, as the number of moves from pixel to pixel required to travel completely around the border and return to the starting point. The compactness of S is sometimes measured by a/p^2, where a is area and p is perimeter; this quantity is low when S has a jagged or elongated shape.

For any pixel of P of S, let $d(P)$ be the distance (in pixel units) from P to $\bar{S}$. The thickness of S is twice the maximum value of $d(P)$. If the area of S is large compared with its thickness, it is elongated.

S is called convex if for all pixels P, Q in S, the line segment $\overline{PQ}$ lies entirely within distance 1 of S. It is easy to show that, if S is convex, it must be connected and have no holes. The smallest convex set containing S is called the convex hull of S; we denote it $H(S)$. The difference set $H(S) - S$ is called the convex deficiency of S, and its components are called the concavities of S.

If surface orientation information is available, we can compute properties of the visible surfaces in the scene (not just of their projections in the image); for example, we can compute surface area rather than just region area. Many of the properties discussed above are very sensitive to spatial orientation and are of only limited value in describing scenes in which the orientations of the surfaces are not known.

B. Geometry-Based Decomposition

The concepts defined in Section XII.A provide us with many ways of defining new subsets of an image in terms of a given subset S. Examples of such subsets are the individual connected components of S, the holes in S, subsets obtained by "filling" the holes (e.g., taking the union of a component and its holes), the borders of S, the convex hull of S or its concavities, and so on. One can also "refine" a subset by discarding parts of it based on geometric criteria; for example, one can discard small

components or concavities (i.e., those whose areas are below some threshold).

An important class of geometric decomposition techniques is based on the concept of expanding and shrinking a set. [These operations are the same as min–max filtering (Section V.D) for the special case of a binary image.] Let $S^{(1)}$ be obtained from S by adding to it the border of $\bar{S}$ (i.e., adding all points of $\bar{S}$ that are adjacent to S), and let $S^{(2)}, S^{(3)}, \ldots$ be defined by repeating this process. Let $S^{(-1)}$ be obtained from S by deleting its border (i.e., deleting all points of S that are adjacent to $\bar{S}$), and let $S^{(-2)}, S^{(-3)}, \ldots$ be defined by repeating this process. If we expand S and then reshrink it by the same amount, that is, we construct $(S^{(k)})^{(-k)}$ for some k, it can be shown that the result always contains the original S; but it may contain other things as well. For example, if S is a cluster of dots that are less than $2k$ apart, expanding S will fuse the cluster into a solid mass, and reshrinking it will leave a smaller, but still solid mass that just contains the dots of the original cluster. Conversely, we can detect elongated parts of a set S by a process of shrinking and reexpanding. Specifically, we first construct $(S^{(-k)})^{(k)}$; it can be shown that this is always contained in the original S. Let S_k be the difference set $S - (S^{(-k)})^{(k)}$. Any component of S_k has a thickness of at most $2k$; thus if its area as large relative to k (e.g., $\geq 10k^2$), it must be elongated.

Another method of geometric decomposition makes use of shrinking operations that preserve the connectedness properties of S, by never deleting a pixel P if this would disconnect the remaining pixels of S in the neighborhood of P. Such operations can be used to shrink the components or holes of S down to single pixels or to shrink S down to a "skeleton" consisting of connected arcs and curves; the latter process is called thinning.

Still another approach to geometric decomposition involves detecting features of a set's border(s). As we move around a border, each step defines a local slope vector, and we can estimate the slope of the border by taking running averages of these vectors. By differentiating (i.e., differencing) the border slope, we can estimate the curvature of the border. This curvature will be positive on convex parts of the border and negative on concave parts (or vice versa); zero-crossings of the curvature correspond to points of inflection that separate the border into convex and concave parts. Similarly, sharp positive or negative maxima of the curvature correspond to sharp convex or concave "corners." It is often useful to decompose a border into arcs by "cutting" it at such corners or inflections. Similar remarks apply to the decomposition of curves. Many of the methods of image segmentation described in Section XI can be applied to border segmentation, with local slope vectors playing the role of pixel gray levels. Analogues of various image-processing techniques can also be applied to borders; for example, we can take the 1-D Fourier transform of a border (i.e., of a slope sequence) and use it to detect global periodicities in the border or filter it to smooth the border, or we can match a border with a template using correlation techniques.

C. Subset Representation

Any image subset S can be represented by a binary image in which a pixel has value 1 or 0 according to whether or not it belongs to S. This representation requires the same amount of storage space no matter how simple or complicated S is; for an $n \times n$ image, it always requires n^2 bits. In this section we describe several methods of representing image subsets that require less storage space if the sets are simple.

1. *Run length coding*. Each row of a binary image can be broken up into "runs" (maximal consecutive sequences) of 1's alternating with runs of 0's. The row is completely determined if we specify the value of the first run (1 or 0) and the lengths of all the runs, in the sequence. If the row has length n and there are only k runs, this run length code requires only $1 + k \log n$ bits, which can be much less than n if k is small and n is large.
2. *Maximal block coding*. The runs of 1's can be regarded as maximal rectangles of height 1 contained in the set S. We can obtain a more compact code if we allow rectangles of arbitrary height. Given a set of rectangles $R_1, \ldots, R_m$ whose union is S, we can determine S by specifying the positions and dimensions of these rectangles. As a specific example, for each pixel P in S, let S_P be the maximal upright square centered at P and contained in S. We can discard S_P if it is contained in S_Q for some $Q \neq P$. The union of the remaining S_P's is evidently S. Thus the set of centers and radii of these S_P's completely determines S; it is called the medial axis transformation of S, since the centers tend to lie on the "skeleton" of S.
3. *Quadtree coding*. Consider a binary image of size $2^n \times 2^n$. If it does not consist entirely of 1's or 0's, we divide it into quadrants; if any of these is not all 1's or all 0's, we divide it into subquadrants, and so on, until we obtain blocks that are all 1's or all 0's. The subdivision process can be represented by a tree of degree 4 (a quadtree); the root node represents the entire image, and each time we subdivide a block, we give its node four children representing its quadrants. If we specify the values (1 or 0) of the blocks represented by the leaves, the image is completely determined. The number of leaves will generally be

greater than the number of blocks in the medial axis transformation, but the quadtree has the advantage of concisely representing the spatial relations among the blocks by their positions in the tree, rather than simply storing them as an unsorted list.

4. *Contour coding*. Given any border of an image subset, we can define a border-following process that, starting from any pixel P of the border, successively visits all the pixels of that border, moving from neighbor to neighbor, and, finally, returns to P. The succession of moves made by this process, together with the coordinates of P, completely determines the border. (Each move can be defined by a code designating which neighbor to move to; the sequence of these codes is called the chain code of the border. Curves in an image can also be encoded in this way.) If all the borders of a set S are specified in this way, S itself is completely determined; that is, we can "draw" its borders and then "fill" its interior. This method of representing a set S is called contour coding. If S has only a few borders and their total length (i.e., the perimeter of S) is not too great, this representation can be more compact than the representation of S by a binary image.

For all these representations, efficient algorithms have been developed for computing geometric properties directly from the representation, for computing the representations of derived subsets (unions, intersections, etc.) directly from the representations of the original subsets, and for converting one representation to another.

The representations defined above are all exact; they completely determine the given subset. We can also define approximations to an image, or to an image subset, using similar methods. For example, we can use maximal blocks whose values are only approximately constant; compare the method of segmentation by splitting described in Section XI.C. Similarly, we can approximate a border or curve—for example, by a sequence of straight-line segments. We can approximate a medial axis by a set of arcs and a radius function defined along each arc; a shape defined by a simple arc and its associated function is called a generalized ribbon. It is often useful to compute a set of approximations of different degrees of accuracy; if these approximations are refinements of one another, they can be stored in a tree structure.

Representations analogous to those described here can be defined for 3-D objects. Such 3-D representations are needed in processing information about surfaces (e.g., obtained from stereo, range sensing, or recovery techniques) or in defining the objects in a scene that we wish to recognize in an image.

XIII. DESCRIPTION

The goal of image analysis is usually to derive a description of the scene that gave rise to the image, in particular, to recognize objects that are present in the scene. The description typically refers to properties of and relationships among objects, surfaces, or features that are present in the scene, and recognition generally involves comparing this descriptive information with stored "models" for known classes of objects or scenes. This section discusses properties and relations, their representation, and how they are used in recognition.

A. Properties and Relations

In Section XII.A we defined various geometric properties of and relationships among image parts. In this section we discuss some image properties that depend on gray level (or color), and we also discuss the concept of invariant properties.

The moments of an image provide information about the spatial arrangement of the image's gray levels. If $f(x, y)$ is the gray level at (x, y), the (i, j) moment m_{ij} is defined as $\sum\sum x^i y^i f(x, y)$ summed over the image. Thus m_{00} is simply the sum of all the gray levels. If we think of gray level as mass, $(m_{10}/m_{00}, m_{01}/m_{00})$ are the coordinates of the centroid of the image. If we choose the origin at the centroid and let $\bar{m}_{ij}$ be the (i, j) central moment relative to this origin, then $\bar{m}_{10} = \bar{m}_{01} = 0$, and the central moments of higher order provide information about how the gray levels are distributed around the centroid. For example, $\bar{m}_{20}$ and $\bar{m}_{02}$ are sensitive to how widely the high gray levels are spread along the x and y axes, whereas $\bar{m}_{30}$ and $\bar{m}_{03}$ are sensitive to the asymmetry of these spreads. The principal axis of the image is the line that gives the best fit to the image in the least-squares sense; it is the line through the centroid whose slope $\tan\theta$ satisfies

$$\tan^2\theta + \frac{\bar{m}_{20} - \bar{m}_{02}}{\bar{m}_{11}} \tan\theta - 1 = 0.$$

Moments provide useful information about the layout or shape of an image subset that contrasts with its complement; they can be computed without the need to segment the subset explicitly from the rest of the image. (Many of the geometric properties defined in Section XII can also be defined for "fuzzy" image subsets that have not been explicitly segmented.)

If the gray levels in an image (or region) are spatially stationary (intuitively, the local pattern of gray levels is essentially the same in all parts of the region), various statistical measures can be used to provide information about the texture of the region. The "coarseness" of the texture can can be measured by how slowly the image's

autocorrelation drops off from its peak at zero displacement (or, equivalently, by how rapidly the image's Fourier power spectrum drops off from its peak at zero frequency), and the "directionality" of the texture can be detected by variation in the rate of dropoff with direction. Strong periodicities in the texture give rise to peaks in the Fourier power spectrum. More information is provided by the second-order probability density of gray levels, which tells us how often each pair of gray levels occurs at each possible relative displacement. (The first-order gray level probability density tells us about the population of gray levels but not about their spatial arrangement.) Some other possible texture descriptors are statistics of gray-level run lengths, gray-level statistics after various amounts and types of filtering, or the coefficients in a least-squares prediction of the gray level of a pixel from those of its neighbors. Information about the occurrence of local patterns in a texture is provided by first-order probability densities of various local property values (e.g., degrees of match to templates of various types). Alternatively, one can detect features such as edges in a texture, or segment the texture into microregions, and compute statistics of properties of these features or microregions (e.g., length, curvature, area, elongatedness, average gray level) and of their spatial arrangement. Overall descriptions of region texture are useful primarily when the regions are the images of flat surfaces oriented perpendicularly to the line of sight; in images of curved or slanted surfaces, the texture will not usually be spatially stationary (see Section X.B).

The desired description of an image is often insensitive to certain global transformations of the image; for example, the description may remain the same under stretching or shrinking of the gray scale (over a wide range) or under rotation in the image plane. This makes it desirable to describe the image in terms of properties that are invariant under these transformations. Many of the geometric properties discussed in Section XII are invariant under various geometric transformations. For example, connectivity properties are invariant under arbitrary "rubber-sheet" distortion; convexity, elongatedness, and compactness are invariant under translation, rotation, and magnification; and area, perimeter, thickness, and curvature are invariant under translation and rotation. (This invariance is only approximate, because the image must be redigitized after the transformation.) The autocorrelation and Fourier power spectrum of an image are invariant under (cyclic) translation, and similar transforms can be defined that are invariant under rotation or magnification. The central moments of an image are invariant under translation, and various combinations of moments can be defined that are invariant under rotation or magnification. It is often possible to normalize an image, that is, to transform it into a "standard form," such that all images differing by a given type of transformation have the same standard form; properties measured on the normalized image are thus invariant to transformations of that type. For example, if we translate an image so its centroid is at the origin and rotate it so its principal axis is horizontal, its moments become invariant under translation and rotation. If we flatten an image's histogram, its gray-level statistics become independent of monotonic transformations of the grayscale; this is often done in texture analysis. It is much more difficult to define properties that are invariant under 3-D rotation, since as an object rotates in space, the shape of its image can change radically, and the shading of its image can change nonmonotonically.

In an image of a 2-D scene, many types of relationships among image parts can provide useful descriptive information. These include relationships defined by the relative values of properties ("larger than," "darker than," etc.) as well as various types of spatial relations ("adjacent to," "surrounded by," "between," "near," "above," etc.). It is much more difficult to infer relationships among 3-D objects or surfaces from an image, but plausible inferences can sometimes be made. For example, if in the region on one side of an edge there are many edges or curves that abruptly terminate where they meet the edge, it is reasonable to infer that the surface on that side lies behind the surface on the other side and that the terminations are due to occlusion. In any case, properties of and relationships among image parts imply constraints on the corresponding object parts and how they could be related and, thus, provide evidence about which objects could be present in the scene, as discussed in the next subsection.

B. Relational Structures and Recognition

The result of the processes of feature extraction, segmentation, and property measurement is a collection of image parts, values of properties associated with each part, and values of relationships among the parts. Ideally, the parts should correspond to surfaces in the scene and the values should provide information about the properties of and spatial relationships among these surfaces. This information can be stored in the form of a data structure in which, for example, nodes might represent parts; there might be pointers from each node to a list of the properties of that part (and, if desired, to a data structure such as a chain code or quadtree that specifies the part as an image subset) and pointers linking pairs of nodes to relationship values.

To recognize an object, we must verify that, if it were present in the scene, it could give rise to an image having the observed description. In other words, on the basis of our knowledge about the object (shape, surface properties, etc.) and about the imaging process (viewpoint, resolution,

etc.), we can predict what parts, with what properties and in what relationships, should be present in the image as a result of the presence of the object in the scene. We can then verify that the predicted description agrees with (part of) the observed description of the image. Note that if the object is partially hidden, only parts of the predicted description will be verifiable. The verification process can be thought of as being based on finding a partial match between the observed and predicted data structures. Note, however, that this is not just a simple process of (say) labeled graph matching; for example, the predicted image parts may not be the same as the observed parts, due to errors in segmentation.

If the viewpoint is not known, recognition becomes much more difficult, because the appearance of an object (shape, shading, etc.) can vary greatly with viewpoint. A brute-force approach is to predict the appearance of the object from many different viewpoints and match the observed description with all of these predictions. Alternatively, we can use a constraint analysis approach such as the following. For any given feature or region in the image, not every feature or surface of the object could give rise to it (e.g., a compact object can never give rise to an elongated region in the image), and those that can give rise to it can do so only from a limited set of viewpoints. If a set of image parts could all have arisen from parts of the same object seen from the same viewpoint, we have strong evidence that the object is in fact present.

C. Models

In many situations we need to recognize objects that belong to a given class, rather than specific objects. In principle, we can do this if we have a "model" for the class, that is, a generic description that is satisfied by an object if and only if it belongs to the class. For example, such a model might characterize the objects as consisting of sets of surfaces or features satisfying certain constraints on their property and relationship values. To recognize an object as belonging to that class, we must verify that the observed configuration of image parts could have arisen from an object satisfying the given constraints.

Unfortunately, many classes of objects that humans can readily recognize are very difficult to characterize in this way. Object classes such as trees, chairs, or even handprinted characters do not have simple generic descriptions. One can "characterize" such classes by simplified, partial descriptions, but since these descriptions are usually incomplete, using them for object recognition will result in many errors. Even the individual parts of objects are often difficult to model; many natural classes of shapes (e.g., clouds) or of surface textures (e.g., tree bark) are themselves difficult to characterize.

We can define relatively complex models hierarchically by characterizing an object as composed of parts that satisfy given constraints, where the parts are in turn composed of subparts that satisfy given constraints, and so on. This approach is used in syntactic pattern recognition, where patterns are recognized by the detection of configurations of "primitive" parts satisfying given constraints, then configurations of such configurations, and so on. Unfortunately, even though the models defined in this way can be quite complex, this does not guarantee that they correctly characterize the desired classes of objects.

Models are sometimes defined probabilistically, by the specification of probability densities on the possible sets of parts, their property values, and so on. This allows recognition to be probabilistic, in the sense that we can use Bayesian methods to estimate the probability that a given observed image configuration arose from an object belonging to a given class. This approach is used in statistical pattern recognition. Unfortunately, making a model probabilistic does not guarantee its correctness, and in any case, the required probability densities are usually very difficult to estimate.

Real-world scenes can contain many types of objects. A system capable of describing such scenes must have a large set of models available to it. Checking these models one by one against the image data would be very time-consuming. The models should be indexed so the appropriate ones can be rapidly retrieved, but not much is known about how to do this.

D. Knowledge-Based Recognition Systems

Nearly all existing image analysis systems are designed to carry out fixed sequences of operations (segmentation, property measurement, model matching, etc.) on their input images. The operations are chosen by the system designer on the basis of prior knowledge about the given class of scenes. For example, we choose segmentation operations that are likely to extract image parts corresponding to surfaces in the scene, we measure properties that are relevant to the desired description of the scene, and so on.

A more flexible type of system would have the capacity to choose its own operations on the basis of knowledge about the class of scenes (and the imaging process) and about how the operations are expected to perform on various types of input data. At each stage and in each part of the image, the system would estimate the expected results of various possible actions, based on its current state of knowledge, and choose its next action to have maximum utility, where utility is a function of the cost of the action and the informativeness of the expected results.

Many examples of knowledge-based reocgnition systems have been demonstrated in the image understanding

and computer vision literature. A more recent idea is to design a supervisor for the knowledge-based system, so that by evaluating the system's current state (quality of image, required speed, etc.), the best possible algorithm, with optimal choice of parameters, can be chosen from among the available options. Designing such a system, however, requires active participation by the user.

XIV. ARCHITECTURES

Image processing and analysis are computationally costly because they usually involve large amounts of data and complex sets of operations. Typical digital images consist of hundreds of thousands of pixels, and typical processing requirements may involve hundreds or even thousands of computer operations per pixel. If the processing must be performed rapidly, conventional computers may not be fast enough. For this reason, many special-purpose computer architectures have been proposed or built for processing or analyzing images. These achieve higher processing speeds by using multiple processors that operate on the data in parallel.

Parallelism could be used to speed up processing at various stages; for example, in image analysis, one could process geometric representations in parallel or match relational structures in parallel. However, most of the proposed approaches have been concerned only with parallel processing of the images themselves, so we consider here only operations performed directly on digital images.

The most common class of image operations comprises local operations, in which the output value at a pixel depends only on the input values of the pixel and a set of its neighbors. These include the subclass of point operations, in which the output value at a pixel depends only on the input value of that pixel itself. Other important classes of operations are transforms, such as the discrete Fourier transform, and statistical computations, such as histogramming or computing moments; in these cases each output value depends on the entire input image. Still another important class consists of geometric operations, in which the output value of a pixel depends on the input values of some other pixel and its neighbors. We consider in this section primarily local operations.

A. Pipelines

Image processing and analysis often require fixed sequences of local operations to be performed at each pixel of an image. Such sequences of operations can be performed in parallel using a pipeline of processors, each operating on the output of the preceding one. The first processor performs the first operation on the image, pixel by pixel. As soon as the first pixel and its neighbors have been processed, the second processor begins to perform the second operation, and so on. Since each processor has available to it the output value of its operation at every pixel, it can also compute statistics of these values, if desired.

Let t be the longest time required to perform an operation at one pixel, and let kt be the average delay required, after an operation begins, before the next operation can start. If there are m operations and the image size is $n \times n$, the total processing time required is then $n^2t + (m-1)kt$. Ordinarily n is much greater than m or k, so that the total processing time is not much greater than that needed to do a single operation (the slowest one).

Pipelines can be structured in various ways to take advantage of different methods of breaking down operations into suboperations. For example, a local operation can sometimes be broken into stages, each involving a different neighbor, or can be broken up into individual arithmetical or logical operations. Many pipeline image-processing systems have been designed and built.

B. Meshes

Another way to use parallel processing to speed up image operations is to divide the image into equal-sized blocks and let each processor operate on a different block. Usually the processors will also need some information from adjacent blocks; for example, to apply a local operation to the pixels on the border of a block, information about the pixels on the borders of the adjacent blocks is needed. Thus processors handling adjacent blocks must communicate. To minimize the amount of communication needed, the blocks should be square, since a square block has the least amount of border for a given area. If the image is $n \times n$, where $n = rs$, we can divide it into an $r \times r$ array of square blocks, each containing $s \times s$ pixels. The processing is then done by an $r \times r$ array of processors, each able to communicate with its neighbors. Such an array of processors is called a mesh-connected computer (or mesh, for short) or, sometimes, a cellular array. Meshes are very efficient at performing local operations on images. Let t be the time required to perform the operation at one pixel, and let c be the communication time required to pass a pixel value from one processor to another. Then the total processing time required is $4cs + ts^2$. Thus by using r^2 processors we have speeded up the time required for the operation from tn^2 to about ts^2, which is a speedup by nearly a factor of r^2. As r approaches n (one processor per pixel), the processing time approaches t and no longer depends on n at all.

For other types of operations, meshes are not quite so advantageous. To compute a histogram, for example, each processor requires time on the order of s^2 to count the

values in its block of the image, but the resulting counts must still be transmitted to one location (e.g., to the processor in the upper left corner) so they can be added together. The transmission requires on the order of r relay stages, since the data must be communicated from one processor to another. Thus as r approaches n, the time required to compute a histogram becomes on the order of n. This is much faster than the single-processor time, which is on the order of n^2, but it is still not independent of n. It can be shown that the time required on a mesh to compute a transform or to perform a geometric operation is also on the order of n as r approaches n.

The speed of a mesh is limited in performing nonlocal operations, because information must be transmitted over long distances, and the processors in the mesh are connected only to their neighbors. Much faster processing times, on the order of $\log n$, can be achieved by the use of a richer interprocessor connection structure, based, for example, on a tree or a hypercube. Meshes augmented in this way allow a wide variety of operations to be performed on images at very high speeds.

C. Recent Trends

Given the impressive advances in the computing power of personal computers and other workstations, the role of computer architectures in image processing and analysis has changed. One can now cluster a set of PCs to realize the computational power of a supercomputer at a much lower cost. This, together with the ability to connect cameras and drivers to a PC, has made image processing feasible at every desk. Important advances are also being made in implementing image processing and understanding algorithms on embedded processors, field-programmable gate arrays, etc. Also, more emphasis is being given to low-power implementations so that battery life can be extended.

XV. SUMMARY

Given the advances that have been made over the last 15 years in image processing, analysis, and understanding, this article has been very difficult for us to revise. After completing the revision, we paused to ponder what has been achieved in these fields over the last 50 years. Based on our combined perspectives, we have formulated the following observations that encapsulate the excitement that we still feel about the subject.

1. Sensors—Instead of the traditional single camera in the visible spectrum, we now have sensors that can cover a much wider spectrum and can perform simple operations on the data.
2. Algorithms—Starting from heuristic, ad hoc concepts, we have reached a stage where statistics, mathematics, and physics play key roles in algorithm development. Wherever possible, existence and uniqueness issues are being addressed. Instead of the traditional passive mode in which algorithms simply process the data, we can now have the algorithms control the sensors so that appropriate data can be collected, leading to an active vision paradigm.
3. Architectures—In the early seventies it took considerable effort and money to set up an image processing/computer vision laboratory. Due to the decreasing costs of computers and memory and their increasing power and capacity, we are now not far away from the day when palm processing of images will be possible.
4. Standards—Advances in JPEG, JPEG-2000, and various MPEG standards have resulted in at least some image processing topics (image compression, video compression) becoming household concepts.
5. Applications—In addition to traditional military applications, we are witnessing enormous growth in applications in medical imaging, remote sensing, document processing, internet imaging, surveillance, and virtual reality. The depth and breadth of the field keep growing!

SEE ALSO THE FOLLOWING ARTICLES

COLOR SCIENCE • COMPUTER ALGORITHMS • COMPUTER ARCHITECTURE • IMAGE-GUIDED SURGERY • IMAGE RESTORATION, MAXIMUM ENTROPY METHODS

BIBLIOGRAPHY

Aloimonos, Y. (ed.) (1993). "Active Perception," Lawrence Erlbaum, Hillsdale, NJ.

Ballard, D. H., and Brown, C. M. (1982). "Computer Vision," Prentice–Hall, Englewood Cliffs, NJ.

Blake, A., and Zisserman, A. (1987). "Visual Reconstruction," MIT Press, Cambridge, MA.

Brady, J. M. (ed.) (1981). "Computer Vision," North-Holland, Amsterdam.

Chellappa, R., and Jain, A. K. (1993). "Markov Random Fields: Theory and Application," Academic Press, San Diego, CA.

Chellappa, R., Girod, B., Munson, D., Jr., Tekalp, M., and Vetterli, M. (1998). "The Past, Present and Future of Image and Multidimensional Signal Processing." *IEEE Signal Process. Mag.* **15**(2), 21–58.

Cohen, P. R., and Feigenbaum, E. A. (eds.) (1982). "Handbook of Artificial Intelligence," Vol. III, Morgan Kaufmann, Los Altos, CA.

Ekstrom, M. P. (ed.) (1984). "Digital Image Processing Techniques," Academic Press, Orlando, FL.

Faugeras, O. D. (ed.) (1983). "Fundamentals in Computer Vision—An Advanced Course," Cambridge University Press, New York.

Faugeras, O. D. (1996). "Three-Dimensional Computer Vision—A Geometric Viewpoint," MIT Press, Cambridge, MA.
Gersho, A., and Gray, R. M. (1992). "Vector Quantization and Signal Compression," Kluwer, Boston.
Grimson, W. E. L. (1982). "From Images to Surfaces," MIT Press, Cambridge, MA.
Grimson, W. E. L. (1990). "Object Recognition by Computer—The Role of Geometric Constraints," MIT Press, Cambridge, MA.
Hanson, A. R., and Riseman, E. M. (eds.) (1978). "Computer Vision Systems," Academic Press, New York.
Horn, B. K. P. (1986). "Robot Vision," MIT Press, Cambridge, MA, and McGraw–Hill, New York.
Horn, B. K. P., and Brooks, M. (eds.) (1989). "Shape from Shading," MIT Press, Cambridge, MA.
Huang, T. S. (ed.) (1981a). "Image Sequence Analysis," Springer, Berlin.
Huang, T. S. (ed.) (1981b). "Two-Dimensional Digital Signal Processing," Springer, Berlin.
Jain, A. K. (1989). "Fundamentals of Digital Image Processing," Prentice–Hall, Englewood Cliffs, NJ.
Kanatani, K. (1990). "Group-Theoretic Methods in Image Understanding," Springer, Berlin.
Koenderink, J. J. (1990). "Solid Shape," MIT Press, Cambridge, MA.
Marr, D. (1982). "Vision—A Computational Investigation into the Human Representation and Processing of Visual Information," Freeman, San Francisco.
Mitchell, J. L., Pennebaker, W. B., Fogg., C. E., and LeGall, D. J. (1996). "MPEG Video Compression Standard," Chapman and Hall, New York.
Pavlidis, T. (1982). "Algorithms for Graphics and Image Processing," Computer Science Press, Rockville, MD.
Pennebaker, W. B., and Mitchell, J. L. (1993). "JPEG Still Image Compression Standard," Van Nostrand Reinhold, New York.
Pratt, W. K. (1991). "Digital Image Processing," Wiley, New York.
Rosenfeld, A., and Kak, A. C. (1982). "Digital Picture Processing," 2nd ed., Academic Press, Orlando, FL.
Ullman, S. (1979). "The Interpretation of Visual Motion," MIT Press, Cambridge, MA.
Weng, J., Huang, T. S., and Ahuja, N. (1992). "Motion and Structure from Image Sequences," Springer, Berlin.
Winston, P. H. (ed.) (1975). "The Psychology of Computer Vision," McGraw–Hill, New York.
Young, T. Y., and Fu, K. S. (eds.) (1986). "Handbook of Pattern Recognition and Image Processing," Academic Press, Orlando, FL.
Zhang, Z., and Faugeras, O. D. (1992). "3D Dynamic Scene Analysis—A Stereo-Based Approach," Springer, Berlin.

Linear Systems of Equations (Computer Science)

Victor Pan
City University of New York

GLOSSARY

Condition (condition number) Product of the norms of a matrix and of its inverse; condition of the coefficient matrix characterizes the sensitivity of the solution of the linear system to input errors.

Error matrix (error vector) Difference between the exact and approximate values of a matrix (of a vector).

Gaussian elimination Algorithm that solves a linear system of equations via successive elimination of its unknowns, or, equivalently, via decomposition of the input coefficient matrix into a product of two triangular matrices.

***m* × *n* matrix** Two-dimensional array of mn entries represented in m rows and n columns. A sparse matrix is a matrix filled mostly with zeros. A sparse matrix

is structured if the locations of all its nonzero entries follow some regular patterns.

Norms of vectors (matrices) Nonnegative values that characterize the magnitudes of those vectors (matrices).

Pivot Entry in the northwest corner of the coefficient matrix, which defines the current elimination step of the solution to that system; pivoting is a policy of interchanging the rows and/or the columns of the matrix such that its certain entry of sufficiently large magnitude is moved to the northwest corner.

A LINEAR SYSTEM OF EQUATIONS is the set of equations of the form,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

where $a_{11}, a_{12}, \ldots, a_{mn}, b_1, b_2, \ldots, b_m$ are given constants, and $x_1, x_2, \ldots, x_n$ are unknown values. The main problem is to compute a solution to such a system, that is, a set of values, $c_1, c_2, \ldots, c_n$, such that the substitution of $x_1 = c_1, x_2 = c_2, \ldots, x_n = c_n$ simultaneously satisfies all the equations of that system, or to determine that the system is inconsistent, that is, it has no solution.

I. INTRODUCTION AND PRELIMINARIES

A. Subject Definition

Our subject is the systems

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned} \tag{1}$$

and algorithms for their solution, with particular attention to linear systems important in computational practice and to their solution on modern computers. We include inconsistent systems and systems having nonunique solution (compare Examples 2 and 3). [Nonunique solution of Eq. (1) always means infinitely many solutions.]

EXAMPLE 1. $m = n = 3$:

$$\begin{aligned} 10x_1 + 14x_2 + 0 * x_3 &= 7 \\ -3x_1 - 4x_2 + 6x_3 &= 4 \\ 5x_1 + 2x_2 + 5x_3 &= 6 \end{aligned}$$

where $x_1 = 0, x_2 = 0.5, x_3 = 1$ is the unique solution.

EXAMPLE 2. $m = 3, n = 2$ (overdetermined systems, $m > n$):

$$\begin{aligned} 2x_1 - x_2 &= 3.2 \\ -2x_1 - 2x_2 &= -10.6 \\ 0 * x_1 - 6x_2 &= -18 \end{aligned}$$

The system is inconsistent.

EXAMPLE 3. $m = 2, n = 3$ (underdetermined systems, $m < n$):

$$x_1 + x_2 + x_3 = 5, \qquad -x_1 + x_2 + x_3 = 3$$

The system has infinitely many solutions, $x_1 = 1, x_2 = 4 - x_3$ for any x_3.

Usually, we assume that all the inputs a_{ij} and b_i are real numbers; the extensions to complex and other inputs (Boolean rings, path algebras) being possible and frequently straightforward.

B. Variety of Applications, Ties with the Computer Technology. Vast Bibliography. Packages of Subroutines

Here are some large subjects, important in computational practice: numerical solution of differential and partial differential equations, mathematical programming and operations research, combinatorial computations, fitting data by curves, interpolation by polynomials and polynomial computations. These and many other practically important subjects have a common feature: Computationally, the problems are reduced to solving linear systems [Eq. (1)], probably the most frequent operation in the practice of numerical computing. Special packages of computer subroutines, such as LINPACK and LAPACK (see also MATLAB), are available for solving linear systems. The bibliography on the subject is vast; luckily, the book by G. H. Golub and C. F. van Loan systematically and successfully describes a variety of most important basic and advanced topics. The book (particularly, pages XIII–XXVII) contains an extensive list of bibliography up to 1996, including a sublist of introductory texts (pages XVII–XVIII). Some other texts, also containing further references, are listed at the end of this article. In particular, several major topics are covered in the two volumes edited by E. Spedicato (especially see pages 1–92) and by G. Winter Althaus and E. Spedicato and in the books by J. W. Demmel and L. N. Trefethen and D. Bau, III. The books by N. Higham, and by A. Greenbaum (advanced) and the volume edited by T. Kailath and A. Sayed are specialized in the treatment of numerical stability/roundoff errors, iterative algorithms and linear systems with Toeplitz, and other structured matrices, respectively (on

the latter topic, see also author's advanced book with D. Bini). Chapter 9 of the book by K. O. Geddes, S. R. Czapor, and G. Labahn covers error-free solution of linear systems by symbolic algebraic methods (see also Section 3 of chapter 3 of author's book with D. Bini).

The development and study of modern algorithms for systems [Eq. (1)] has been greatly influenced by practical and theoretical applications of linear systems and by the development of modern computer technology. More recent vector and parallel computers have turned out to be very effective for solving linear systems, which has motivated both further development of those computers and the study and design of algorithms for linear systems.

C. Sparsity, Structure, and Computer Representation of Linear Systems

A linear system [Eq. (1)] can be defined by its extended matrix,

$$W = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{bmatrix} \tag{2}$$

which occupies $m \times (n+1)$ working array in a computer. The first n columns of W form the $m \times n$ coefficient matrix A of the system. The last column is the right-hand-side vector $\mathbf{b}$.

EXAMPLE 4. The extended matrices of systems of Examples 1 and 2 are

$$\begin{bmatrix} 10 & 14 & 0 & 7 \\ -3 & -4 & 6 & 4 \\ 5 & 2 & 5 & 6 \end{bmatrix}, \qquad \begin{bmatrix} 2 & -1 & 3.2 \\ -2 & -2 & -10.6 \\ 0 & -6 & -18 \end{bmatrix}$$

Since the primary storage space of a computer is limited, the array [Eq. (2)] should not be too large; 100×101 or 200×201 can be excessively large for some computers. Practically, systems with up to, say, 100,000 equations, are handled routinely, however; because large linear systems arising in computational practice are usually sparse (only a small part of their coefficients are nonzeros) and well structured (the nonzeros in the array follow some regular patterns). Then special data structures enable users to store only nonzero coefficients (sometimes only a part of them). The algorithms for solving such special systems are also much more efficient than in the case of general dense systems.

Consider, for instance, tridiagonal systems, where $a_{ij} = 0$, unless $-1 \le i - j \le 1$. Instead of storing all the $n^2 + n$ input entries of A and $\mathbf{b}$, which would be required in case of a dense system, special data structures can be used to store only the $4n - 2$ nonzero entries. The running time of the program solving such a system (which can be roughly measured by the number of arithmetic operations used) is also reduced from about $\frac{2}{3}n^3$ for dense systems to $8n - 13$ for tridiagonal systems, (substitute $n = 10{,}000$ to see the difference). The structure of some linear systems may not be immediately recognized. For instance, in Section II.A solving the Laplace equation $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = 0$ is reduced to a system that is not tridiagonal but is block tridiagonal, that is, its coefficient matrix can be represented as a tridiagonal matrix whose entries are in turn matrices, specifically in the model example of a small size:

$$A = \begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix} = \begin{bmatrix} B_2 & I_2 \\ I_2 & B_2 \end{bmatrix} \tag{3}$$

$$B_2 = \begin{bmatrix} -4 & 1 \\ 1 & -4 \end{bmatrix}, \qquad I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Block tridiagonal structures can be also effectively exploited, particularly in cases where the blocks are well structured.

D. Specifics of Overdetermined and Underdetermined Linear Systems

Overdetermined linear systems [Eq. (1)] with m greatly exceeding n (say, $m = 1000; n = 2$) arise when we try to fit given data by simple curves, in statistics, and in many other applications; such systems are usually inconsistent. A quasi-solution $x_1^*, \ldots, x_n^*$ is sought, which minimizes the magnitudes of the residuals, $r_i = b_i - (a_{i1}x_1^* + \cdots + a_{in}x_n^*)$, $i = 1, \ldots, n$. Methods of computing such a quasi-solution vary with the choice of the minimization criterion, but usually the solution is ultimately reduced to solving some regular linear systems [Eq. (1)] (where $m = n$) (see Sections II.E; IV.A; and IV.C).

A consistent underdetermined system [Eq. (1)] always has infinitely many solutions (compare Example 3) and is frequently encountered as part of the problem of mathematical programming, where such systems are complemented with linear inequalities and with some optimization criteria.

E. General and Special Linear Systems. Direct and Iterative Methods. Sensitivity to Errors

Generally, the efforts to identify fully the structure of a system [Eq. (1)] are generously awarded at the solution stage. Special cases and special algorithms are so numerous, however, that we shall first study the algorithms that work for general linear systems [Eq. (1)]. We shall follow the customary pattern of subdividing the methods for

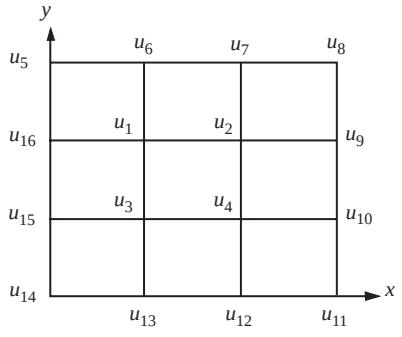
solving systems [Eq. (1)] into direct and iterative. The direct methods are more universal; they apply to general and special linear systems, but for many special and/or sparse linear systems, the special iterative methods are superior (see Section VIII). If the computations are performed with infinite precision, the direct methods solve Eq. (1) in finite time, whereas iterative methods only compute better and better approximations to a solution with each new iteration (but may never compute the solution exactly). That difference disappears in practical computations, where all arithmetic operations are performed with finite precision, that is, with round-off errors. In principle, the round-off errors may propagate and greatly, or even completely, contaminate the outputs. This depends on the properties of the coefficient matrix, on the choice of the algorithms, and on the precision of computation. A certain amount of study of the sensitivity of the outputs to round-off error is normally included in texts on linear systems and in current packages of computer subroutines for such systems; stable algorithms are always chosen, which keep output errors lower. In general, direct methods are no more or no less stable than the iterative methods.

II. SOME EXAMPLES OF APPLICATIONS

Next we present a few simple examples that demonstrate how frequently practical computations and important theoretical problems are reduced to solving linear systems of algebraic equations (more examples are presented in Sections VI.B and VII.D) (Those readers not interested in the present section may proceed to Section III, consulting Section II as they are referred to it.)

A. Numerical Solution of the Laplace Equation

We consider the Laplace equation $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = 0$ on the square region $0 \le x, y \le 1$, provided that the function $u(x, y)$ is given on the boundary of that region; $u(x, y)$ models the temperature distribution through a square plate with fixed temperature on its sides. To compute $u(x, y)$ numerically, we superimpose a mesh of horizontal and vertical lines over the region as shown by



so that u_1 denotes the point $(\frac{1}{3}, \frac{2}{3})$; u_2 denotes the point $(\frac{2}{3}, \frac{2}{3})$; u_7 denotes $(\frac{2}{3}, 1)$, and so on; $u_5, u_6, \ldots, u_{16}$ are given; and $u_1,\ u_2,\ u_3,\ u_4$ are unknowns. Then we replace the derivatives by divided differences,

$$\partial^2 u/\partial x^2 \quad \text{by} \quad u(x-h, y) - 2u(x, y) + u(x+h, y)$$
$$\partial^2 u/\partial y^2 \quad \text{by} \quad u(x, y-h) - 2u(x, y) + u(x, y+h)$$

where $h = \frac{1}{3}$. This turns the Laplace equation into a linear system that can be equivalently derived if we just assume that the temperature at an internal point of the grid equals the average of the temperatures at the four neighboring points of the grid;

$$\begin{aligned} -4u_1 + u_2 + u_3 &= -u_6 - u_{16} \\ u_1 - 4u_2 + u_4 &= -u_7 - u_9 \\ u_1 - 4u_3 + u_4 &= -u_{13} - u_{15} \\ u_2 + u_3 - 4u_4 &= -u_{10} - u_{12} \end{aligned}$$

The coefficient matrix A of the system is the block tridiagonal of Eq. (3). With smaller spacing we may obtain a finer grid and compute the temperatures at more points on the plate. Then the size of the linear system will increase, say to N^2 equations in N^2 unknowns for larger N; but its $N^2 \times N^2$ coefficient matrix will still be block tridiagonal of the following special form (where blank spaces mean zero entries),

$$\begin{bmatrix} B_N & I_N & & & \\ I_N & B_N & I_N & & \\ & I_N & B_N & \ddots & \\ & & \ddots & \ddots & I_N \\ & & & I_N & B_N \end{bmatrix}$$

$$B_N = \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & \ddots & & \\ & \ddots & \ddots & 1 & \\ & & 1 & -4 & \end{bmatrix}$$

Here, B_N is an $N \times N$ tridiagonal matrix, and I_N denotes the $N \times N$ identity matrix (see Section II.D). This example demonstrates how the finite difference method reduces the solution of partial differential equations to the solution of linear systems [Eq. (1)] by replacing derivatives by divided differences. The matrices of the resulting linear systems are sparse and well structured.

B. Solving a Differential Equation

Let us consider the following differential equation on the interval $\{t: 0 \le t \le 1\}$, $d^2x/dt^2 + x = g(t)$, $x(0) = 0$,

$x(1)=1$ [where $g(t)$ is a given function, say $g(t)=\log(t+1)$ or $g(t)=e^t$]. Let $t_0=0$; $t_1, t_2, t_3, t_4, t_5=1$ be the one-dimensional grid of six equally spaced points on the interval $\{t: 0\le t\le 1\}$, so $t_1=\frac{1}{5}$, $t_2=\frac{2}{5}$, $t_3=\frac{3}{5}$, $t_4=\frac{4}{5}$, $h=\frac{1}{5}$. Denote $x_0=x(t_0)=x(0)=0, x_1=x(t_1), \ldots, x_5=x(t_5)=x(1)=1$, replace the derivative d^2x/dt^2 by the divided differences so that $(d^2x/dt^2)|_{t=t_2}=(x_1-2x_2+x_3)/h^2$ and similarly at $t=t_1, t=t_3$, and $t=t_4$; and arrive at the tridiagonal system defined by the following extended matrix:

$$\begin{bmatrix} h^2-2 & 1 & 0 & 0 & h^2g(t_1)-x_0 \\ 1 & h^2-2 & 1 & 0 & h^2g(t_2) \\ 0 & 1 & h^2-2 & 1 & h^2g(t_3) \\ 0 & 0 & 1 & h^2-2 & h^2g(t_4)-x_5 \end{bmatrix}$$

Using a finer grid with more points t, we may compute the solution $x(t)$ at more points. The derived linear system would have greater size but the same tridiagonal structure.

C. Hitchcock Transportation Problem. Linear Programming Problem

We consider a communication system having three sources, 1, 2, and 3, with supplies s_1, s_2, and s_3 and two sinks, 1 and 2, with demands d_1 and d_2, respectively, such that $d_1+d_2=s_1+s_2+s_3$. Let every source be connected with every sink by a communication line. Suppose that the quantities x_{ij} must be delivered from source i to sink j for every pair i, j such that

$$\sum_{j=1}^{2} x_{ij} = s_i, \qquad i=1,2,3$$

$$\sum_{j=1}^{3} x_{ij} = d_j, \qquad j=1,2$$

One of these equations can be deleted, for the sums of the first three and of the last two equations coincide with one another. Thus, we arrive at an underdetermined system of four equations with six unknowns having two free variables; say x_{22} and x_{32}, which can be chosen arbitrarily, then $x_{11}, x_{12}, x_{21}, x_{31}$ will be uniquely defined. For instance, let $s_1=s_2=s_3=2, d_1=d_2=3$. Choose $x_{22}=x_{32}=1$, then $x_{11}=x_{12}=x_{21}=x_{31}=1$; choose $x_{22}=2, x_{32}=0$, then $x_{11}=x_{12}=1, x_{21}=0, x_{31}=2$, and so on. In such situations some additional requirements are usually imposed. Typically, it is required that all the variables x_{ij} be nonnegative and that a fixed linear function in those variables take its minimum value; for example,

$$\begin{aligned} \textit{minimize} \quad & x_{11}+x_{12}+x_{21}+2x_{22}+2x_{31}+x_{32} \\ \text{subject to} \quad & x_{11}+x_{12}=2 \\ & x_{21}+x_{22}=2 \\ & x_{31}+x_{32}=2 \\ & x_{11}+x_{21}+x_{31}=3 \\ & x_{12}+x_{22}+x_{32}=3 \\ & x_{ij}\ge 0 \qquad \text{for} \quad i=1,2,3, \quad j=1,2 \end{aligned}$$

In this case, $x_{11}=x_{12}=1, x_{21}=x_{32}=2, x_{22}=x_{31}=0$ is the unique solution. This example is a specific instance of the Hitchcock transportation problem, generally defined as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{p}\sum_{j=1}^{q} c_{ij}x_{ij} \\ \text{subject to} \quad & \sum_{j=1}^{q} x_{ij}=s_i, \qquad i=1,\ldots,p \\ & \sum_{i=1}^{p} x_{ij}=d_j, \qquad j=1,\ldots,q \\ & x_{ij}\ge 0 \qquad \text{for} \quad \text{all } i \quad \text{and } j \end{aligned}$$

Here s_i, d_j and c_{ij} are given for all i, j. (In our specific example above, $p=3, q=2, s_1=s_2=s_3=2, d_1=d_2=3, c_{11}=c_{12}=c_{21}=c_{32}=1, c_{22}=c_{31}=2$.) The linear equations form an underdetermined but sparse and well-structured system.

The Hitchcock transportation problem is in turn an important particular case of the linear programming problem (1.p.p.). The 1.p.p. is known in several equivalent forms, one of which follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{m} a_{ij}x_{ij}=b_i, \qquad i=1,\ldots,n \\ & x_j\ge 0, \qquad j=1,\ldots,m \end{aligned}$$

In this representation, the general 1.p.p. includes an underdetermined system of linear equations complemented with the minimization and nonnegativity requirements. Solving the 1.p.p. can be reduced to a finite number of iterations, each reduced to solving one or two auxiliary linear systems of equations; such systems either have n equations with n unknowns (in the simplex algorithms for 1.p.p.) or are overdetermined, in which case their least-squares solutions are sought (in ellipsoid algorithms, in Karmarkar's

algorithm). The least-squares solutions to overdetermined linear systems are studied in Sections II.E; IV.A; and IV.C.

D. Some Matrix Operations. Special Matrices

The study of linear systems [Eq. (1)] is closely related to the study of matrices and vectors. In this section we shall reduce $n \times n$ matrix inversion (which is theoretically important and has some practical applications, for instance, in statistics) to solving n systems [Eq. (1)], where $m = n$. We shall also list some special matrices and recall some basic concepts of matrix theory.

Generally, a $p \times q$ array is called a $p \times q$ matrix; in particular $p \times 1$ and $1 \times p$ matrices are called vectors of dimension p. Deletion of any r rows and s columns for $r < p$ and $s < q$ turns a $p \times q$ matrix into its $(p-r) \times (q-s)$ submatrix. The coefficient matrix A of Eq. (1) has size $m \times n$; the extended matrix W has size $m \times (n+1)$; the last column of W is vector $\mathbf{b}$ of dimension m. The unknowns $x_1, x_2, \ldots, x_n$ form a (column) vector $\mathbf{x}$ of dimension n. Of course, that notation may change; for instance, in the next section we shall replace $\mathbf{x}$ by $\mathbf{c}$ and $\mathbf{b}$ by $\mathbf{f}$.

The transpose of a matrix V is denoted by V^{T}, so $\mathbf{x}^{\mathrm{T}} = [x_1, \ldots, x_n]$, $\mathbf{x} = [x_1, \ldots, x_n]^{\mathrm{T}}$. For a complex matrix $V = [v_{gh}]$, its Hermitian transpose is defined $V^{\mathrm{H}} = [v_{hg}^*]$, v_{hg}^* being the complex conjugate of v_{hg}. $V^{\mathrm{T}} = V^{\mathrm{H}}$ for a real V. A matrix V is called symmetric if $V = V^{\mathrm{T}}$ and Hermitian if $V = V^{\mathrm{H}}$. For two column vectors $\mathbf{u}$ and $\mathbf{v}$ of the same dimension p, their inner product (also called their scalar product or their dot product), is defined as follows,

$$\mathbf{u}^{\mathrm{T}}\mathbf{v} = u_1 v_1 + u_2 v_2 + \cdots + u_p v_p$$

This is extended to define the $m \times p$ product of an $m \times n$ matrix A by an $n \times p$ matrix B, $AB = [a_{i1}b_{1k} + a_{i2}b_{2k} + \cdots + a_{in}b_{nk}, i = 1, \ldots, m; k = 1, \ldots, p]$; that is, every row $[a_{i1}, \ldots, a_{in}]$ of A is multiplied by every column $[b_{1k}, \ldots, b_{nk}]^{\mathrm{T}}$ of B to form the $m \times p$ matrix AB. For instance, if $A = [1, 2]$, $B = [1, 2]^{\mathrm{T}}$, then $AB = [5]$ is a 1×1 matrix, $BA = \left[\begin{smallmatrix} 1 & 2 \\ 2 & 4 \end{smallmatrix}\right]$ is a 2×2 matrix, $AB \neq BA$. The m equations of Eq. (1) can be equivalently represented by a single matrix–vector equation, $A\mathbf{x} = \mathbf{b}$. For instance, the system of Eq. (1) takes the following form,

$$\begin{bmatrix} 10 & 14 & 0 \\ -3 & -4 & 6 \\ 5 & 2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \\ 6 \end{bmatrix}.$$

(For control, substitute here the solution vector $[x_1, x_2, x_3]^{\mathrm{T}} = [0, 0.5, 1]^{\mathrm{T}}$ and verify the resulting equalities.)

Hereafter I_n denotes the unique $n \times n$ matrix (called the identity matrix) such that $AI_n = A$, $I_n B = B$ for all the matrices A and B of sizes $m \times n$ and $n \times p$, respectively. All the entries of I_n are zeros except for the diagonal entries, equal to 1. (Check that $I_2 A = A$ for $I_2 = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ and for arbitrary A.) In the sequel, I (with no subscript) denotes the identity matrix I_n for appropriate n; similarly, 0 denotes a null matrix (filled with zeros) of appropriate size.

An $n \times n$ matrix A may (but may not) have its inverse, that is, an $n \times n$ matrix $X = A^{-1}$ such that $XA = I$. For instance, $A^{-1} = \left[\begin{smallmatrix} 1 & -2 \\ 0 & 1 \end{smallmatrix}\right]$ if $A = \left[\begin{smallmatrix} 1 & 2 \\ 0 & 1 \end{smallmatrix}\right]$; but the matrix $\left[\begin{smallmatrix} 2 & 2 \\ 1 & 1 \end{smallmatrix}\right]$ has no inverse. Matrices having no inverse are called singular. All the nonsquare matrices are singular. Linear system $A\mathbf{x} = \mathbf{b}$ has unique solution $\mathbf{x} = A^{-1}\mathbf{b}$ if and only if its coefficient matrix A is nonsingular. If $XA = I$, then always $AX = I$; so computing the kth column of $X = A^{-1}$ amounts to solving the system of Eq. (1) where $\mathbf{b}$ is the kth coordinate vector whose kth entry is 1 and whose other entries are zeros, $k = 1, 2, \ldots, n$. Computing the inverse of an $n \times n$ matrix A is equivalent to solving n such linear systems with the common coefficient matrix A.

The maximum r, such that A has an $r \times r$ nonsingular submatrix, is called the rank of A and is denoted as $\mathrm{rank}(A)$. An $m \times n$ matrix A has full rank if $\mathrm{rank}(A) = \min\{m, n\}$.

Theorem 1. If the system of Eq. (1) is consistent, then its general solution is uniquely defined by the values of $n - r$ free parameters, where $r = \mathrm{rank}(A)$.

The sum $A + B = [a_{ij} + b_{ij}]$ and the difference $A - B = [a_{ij} - b_{ij}]$ are defined for two matrices A and B of the same size; the product $cA = Ac = [ca_{ij}]$ is defined for a matrix A and a scalar c. The customary laws of arithmetic (except for $AB = BA$) are extended to the case where the operations are performed with matrices, $A + B = B + A$, $(A + B) + C = A + (B + C)$, $-A = (-1)A$, $(AB)C = A(BC)$, $(A + B)C = AC + BC$, $C(A + B) = CA + CB$, $c(A + B) = cA + cB$.

Further, $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$, $(AB)^{-1} = B^{-1}A^{-1}$, $(A^{-1})^{\mathrm{T}} = (A^{\mathrm{T}})^{-1}$. Linear forms and linear equations can be defined over matrices, say $AX + BY = C$, where A, B, C, X, Y are matrices of appropriate sizes, A, B, C are given and X, Y are unknown. Such a matrix equation can be also rewritten as a system of linear equations, the entries of X and Y playing the role of unknowns. If $B = 0$, $C = I$, we arrive at the matrix equation $AX = I$, which defines $X = A^{-1}$.

Finally, definitions of and the customary notation for some special matrices used in special linear systems [Eq. (1)] are given in Table I.

E. Approximating Data by Curves. Overdetermined Linear Systems. Normal Equations, Reduction to Linear Programming Problems

In many applications we need to define a simple curve (function) that approximates to (that is, passes near) a

TABLE I Customary Notation for Special Matrices

1. Diagonal, $A = \text{diag}[a_{11}, \ldots, a_{nn}]$:	$a_{ij} = 0$, unless $i = j$
2. Lower triangular, $A = L$:	$a_{ij} = 0$, unless $i \geq j$
2a. Unit lower triangular:	$a_{ii} = 1$ for all i
3. (Unit) upper triangular, $A = U$, $A = R$:	The transpose of 2 (of item 2a)
4. Band with bandwidth (g, h):	$a_{ij} = 0$, unless $g \leq i - j \leq h$
4a. Tridiagonal:	$a_{ij} = 0$, unless $-1 \leq i - j \leq 1$
5. (Strictly) row-diagonally dominant:	$2\lvert a_{ii}\rvert > \sum_{j=1}^{n} \lvert a_{ij}\rvert$ for all i
6. (Strictly) column-diagonally dominant:	$2\lvert a_{jj}\rvert > \sum_{i=1}^{n} \lvert a_{ij}\rvert$ for all j
7. Hermitian (real symmetric):	$A = A^{\mathrm{H}}$(in real case, $A = A^{\mathrm{T}}$)
8. Positive definite:	$\bar{x}^{\mathrm{H}} A\bar{x} > 0$ for all vectors $\bar{x} \neq \bar{0}$
9. Unitary (real orthogonal), $A = Q$:	$A^{\mathrm{H}}A = I$ (real case, $A^{\mathrm{T}}A = I$)
10. Toeplitz:	$a_{ij} = a_{i+1j+1}$, for all i, $j < n$
11. Hankel:	$a_{ij} = a_{i+1j-1}$, for all $i < n$, $j > 1$
12. Vandermonde:	$a_{ij} = a_j^{i-1}$ for all i, j; a_j distinct

given set of points on a plane $\{(x_i, f_i), i = 1, \ldots, N\}$. [The objectives can be to compress $2 \times N$ array representing those points where N is large or to retrieve the information and to suppress the noise or to replace a function $f(x)$ by an approximating polynomial, which can be computed at any point using only few arithmetic operations.]

i:	1	2	3	4	5	6	7	8
x_i:	0.1	0.2	0.3	0.5	0.7	1.0	1.4	2.0
f_i:	0.197	0.381	0.540	0.785	0.951	1.11	1.23	1.33

A model example with eight points, where in fact $f_i = \tan^{-1} x_i$ (in practice, hundreds or thousands of input points are not an unusual case) is shown in the accompanying tabulation. To find a straight line $c_0 + c_1 x$ passing through all the eight points (x_i, f_i), we would have to satisfy the following system of eight equations with two unknowns c_0 and c_1, $c_0 + c_1 x_i = f_i$, $i = 1, \ldots, 8$. This is a typical example of an overdetermined linear system $A\mathbf{c} = \mathbf{f}$, that has no solution. Every (overdetermined) linear system $A\mathbf{c} = \mathbf{f}$, however, has a least-squares solution vector $\mathbf{c} = \mathbf{c}^*$ minimizing the Euclidean norm of the residual vector $\mathbf{r} = \mathbf{f} - A\mathbf{c}$, $\mathbf{r} = [r_1, r_2, \ldots, r_N]^{\mathrm{T}}$, $\|\mathbf{r}\|_2 = (\sum_{i=j}^{N} r_i^2)^{1/2}$. In our case, $N = 8$, and $\mathbf{r} = [r_1, r_2, \ldots, r_8]^{\mathrm{T}}$, $\mathbf{c} = [c_1, c_2]^{\mathrm{T}}$, $\mathbf{f} = [0.197, 0.381, 0.540, 0.785, 0.951, 1.11, 1.23, 1.33]^{\mathrm{T}}$, and

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0.1 & 0.2 & 0.3 & 0.5 & 0.7 & 1.0 & 1.4 & 2.0 \end{bmatrix}^{\mathrm{T}}$$

In some cases a vector $\mathbf{c}$ is sought that minimizes another norm of $\mathbf{r}$ (rather than Euclidean); but the minimization of the Euclidean norm of $\mathbf{r}$ [or of a weighted Euclidean norm $(\sum_{i=1}^{N} w_i r_i^2)^{1/2}$ for a fixed set of positive weights $w_1, \ldots, w_N$] is most customary because simple solution methods are available. Computing a least squares solution $\mathbf{c}$ to an overdetermined linear system $A\mathbf{c} = \mathbf{f}$ is called regression in statistics. When the computation is with finite precision, computing such a solution $\mathbf{c}$ is equivalent to solving the system of normal equations. $A^{\mathrm{T}}A\mathbf{c} = A^{\mathrm{T}}\mathbf{f}$. In the previous example the normal equations take the following form:

$$\begin{bmatrix} 8 & 6.2 \\ 6.2 & 7.84 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 6.524 \\ 6.8081 \end{bmatrix}$$

The latter system defines the desired least-squares solution vector $\mathbf{c}^{\mathrm{T}} = [c_0, c_1] = [0.368119, 0.577265]$ (here the entries of $\mathbf{c}$ are given with six decimals).

When the computation is done with finite precision, using normal equations is not always recommended because for some matrices A, higher precision of computation is required in order to solve the system of normal equations correctly. For example, let us consider the problem of least-squares fitting by a straight line, $c_0 + c_1 x$, to the set of data, shown in the accompanying tabulation.

i:	1	2	3	4
x_i:	970	990	1000	1040
f_i:	4	2	0	−3

Then, $\mathbf{f} = [4, 2, 0, -3]^{\mathrm{T}}$,

$$A^{\mathrm{T}} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 970 & 990 & 1000 & 1040 \end{bmatrix}$$

which gives the following system $A^{\mathrm{T}}A\mathbf{c} = A^{\mathrm{T}}\mathbf{f}$ of normal equations

$$\begin{bmatrix} 4 & 4000 \\ 4000 & 4002600 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 3 \\ 2740 \end{bmatrix}$$

The solution $c_1 = -0.1, c_0 = 100.75$ defines the straight line $-0.1x + 100.75$. Next, we assume that the computation is with chopping to four decimals, that is, every number is a floating-point decimal number, and its fraction (mantissa) is chopped to its four most significant digits. We then arrive at $c_1 = -0.13, c_0 = 130.7$. The substantial error arises because of the large entries of $A^{\mathrm{T}}A$. To counter this difficulty, we linearly transform the basis 1, x of the representation of the straight line into the more suitable basis 1, $x - 1000$ and seek the straight line $c_0^* + c_1^*(x - 1000)$. (That basis is in fact orthogonal: The two vectors [1, 1, 1, 1] and $[x_1 - 1000, x_2 - 1000, x_3 - 1000, x_4 - 1000] = [-30, -10, 0, 40]$, representing the values of the basis functions 1 and $x - 1000$ at x_1, x_2, x_3, x_4, are orthogonal to each other; that is, their inner product equals $-30 - 10 + 0 + 40 = 0$). Using that basis we may rewrite the previous tabulation as shown in the accompanying one here. We then arrive at

i:	1	2	3	4
$x_i - 1000$:	−30	−10	0	40
f_i:	4	2	0	−3

the normal equations, $A^{\mathrm{T}}A\mathbf{c}^* = A^{\mathrm{T}}\mathbf{f}$, where

$$A^{\mathrm{T}} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -30 & -10 & 0 & 40 \end{bmatrix}$$

so that $4c_0^* = 3$, $2600c_1^* = -260$, $c_0^* = 0.75$, and $c_1^* = c_1 = -0.1$. This defines the desired straight line, $c_0^* + c_1^*(x - 1000) = 0.75 - 0.1(x - 1000) = -0.1x + 100.75$. The latter approach, with the orthogonalization of the basis via its linear transformation, leads to the methods of QR-factorization of a matrix A (see Section IV.A).

Our model examples of the approximation to the given data by a straight line can be immediately extended to the approximation by algebraic polynomials of higher degree, by trigonometric polynomials, and so on. For instance, fitting by a quadratic curve $c_0 + c_1x + c_2x^2$ leads to an overdetermined system $c_0 + c_1x_i + c_2x_i^2 = f_i, i = 1, \ldots, N$, of the form $A\mathbf{c} = \mathbf{f}$, whose least-square solution can be found by the usual methods. Here, $\mathbf{c} = [c_0, c_1, c_2]^{\mathrm{T}}$, and A is a $3 \times N$ matrix $A = [x_i^j, i = 1, \ldots, N, j = 0, 1, 2]$.

Minimizing the least deviation norm of the residual vector $\|\mathbf{r}\|_1 = \sum_{i=1}^{N} |r_i|$ is a little harder than regression. The problem is equivalent to the linear programming problem (1.p.p.) of the following form:

$$\text{minimize} \quad \|\mathbf{q}\|_1 = \sum_{j=1}^{n} q_i$$

$$\text{subject to} \quad \mathbf{q} \geq \mathbf{f} - A\mathbf{c}, \qquad \mathbf{q} \geq -\mathbf{f} + A\mathbf{c}$$

The latter inequalities imply that every coordinate q_i of the vector $\mathbf{q}$ is not less than $|r_i| = |f_i - \mathbf{A}_i^{\mathrm{T}}\mathbf{c}|$ where $\mathbf{A}_i^{\mathrm{T}}$ denotes the ith row of A.

Similarly, the minimization of the maximum norm of $\mathbf{r}$, $\|\mathbf{r}\|_\infty = \max_{1 \leq i \leq N} |r_i|$, is equivalent to the 1.p.p. of the following form:

$$\begin{aligned} &\text{minimize} && q \\ &\text{subject to} && q \geq f_i - \mathbf{A}_i^{\mathrm{T}}\mathbf{c} \\ & && q \geq -f_i + \mathbf{A}_i^{\mathrm{T}}\mathbf{c}, \qquad i = 1, 2, \ldots, N \end{aligned}$$

The latter inequalities imply that $q \geq |f_i - A_i^{\mathrm{T}}\mathbf{c}|$ for all i.

III. GAUSSIAN ELIMINATION AND TRIANGULAR FACTORIZATION

Gaussian elimination and its modifications are the most customary direct algorithms for the general linear system of Eq. (1), $A\mathbf{x} = \mathbf{b}$. The subroutine versions usually include pivoting and rely on triangular factorizations of the coefficient matrix A; computing is performed with finite precision. We initially assume infinite precision and no pivoting in order to facilitate the presentation. Systems with nonsquare matrices A are studied further in Section IV; Section VI contains some special algorithms for special systems [Eq. (1)] that are most important in applications; and Section V contains further estimates for time and space required to solve a general linear system.

A. Solving Triangular Systems by Back Substitution

Let $A = [a_{ij}]$ be an $n \times n$ upper triangular matrix; that is, $a_{ij} = 0$ if $i > j$ (similarly, A is a lower triangular if $a_{ij} = 0$, where $i < j$). There can be two cases.

Case 1. All the diagonal entries a_{ii} are nonzero. In that case, the system is nonsingular and has a unique solution.

EXAMPLE 5.

$$\begin{aligned} x_1 + 2x_2 - x_3 &= 3 \\ -2x_2 - 2x_3 &= -10 \\ -6x_3 &= -18 \end{aligned}$$

Compute $x_3 = -18/(-6) = 3$. Substitute this value into the second equation and obtain $-2x_2 - 6 = -10$, so $-2x_2 = -4$, $x_2 = 2$. Substitute $x_2 = 2, x_3 = 3$ into the first equation and obtain $x_1 + 4 - 3 = 3$, $x_1 = 2$. In general, if A is an $n \times n$ triangular matrix, this back substitution algorithm can be written as follows.

For $i = n, n-1, \ldots, 1$

$$\text{If} \quad a_{ii} = 0, \qquad \text{end (the system is singular)}$$

$$\text{Else} \quad x_i := \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right) \Big/ a_{ii}$$

The computation consists mostly of operations of the form, $c := c + gh$. The amount of computational work required to perform such an operation on a computer using a floating-point finite-precision arithmetic is called a flop. The number of flops used is a customary measure for the amount of work required in algorithms for systems [Eq. (1)]. Frequently, the terms of lower order are ignored; for instance, we may say that the above back substitution algorithm requires $n^2/2$ flops. In that algorithm, operations are grouped into inner product computations; computing inner products can be simplified on some serial computers.

Case 2. Some diagonal entries are zeros. In that case the system is singular, that is, inconsistent or has infinitely many solutions (compare Theorem 1).

EXAMPLE 6.

$$\begin{aligned} 0 * x_1 + 2x_2 - x_3 &= 3 \\ -2x_2 - 2x_3 &= -10 \\ -6x_3 &= -18 \end{aligned}$$

Back substitution shows an inconsistency: $x_3 = 3$, $x_2 = 2$, $4 - 2 = 3$.

EXAMPLE 7.

$$\begin{aligned} x_1 + 2x_2 - x_3 &= 3 \\ 0 * x_2 - 2x_3 &= -6 \\ -6x_3 &= -18 \end{aligned}$$

Back substitution yields $x_3 = 3$; x_2 is a free variable; $x_1 = 6 - 2x_2$.

B. Forward Elimination Stage of Gaussian Elimination

Every system [Eq. (1)] can be reduced to triangular form using the following transformations, which never change its solutions:

1. Multiply equation i (row i of the extended matrix W) by a nonzero constant.
2. Interchange equations i and k (rows i and k of W).
3. Add a multiple of equation i to equation k (of row i to row k of W).

EXAMPLE 8.

$$\begin{aligned} 10x_1 + 14x_2 &= 7 \\ -3x_1 - 4x_2 + 6x_3 &= 4 \\ 5x_1 + 2x_2 + 5x_3 &= 6 \end{aligned}$$

Step 1. Multiply the first equation by the multipliers $m_{21} = -\frac{3}{10}$ and $m_{31} = \frac{5}{10}$; subtract the results from the second and the third equations, respectively; and arrive at the system

$$\begin{aligned} 10x_1 + 14x_2 &= 7 \\ 0.2x_2 + 6x_3 &= 6.1 \\ -5x_2 + 5x_3 &= 2.5 \end{aligned}$$

Step 2. Multiply the second equation by the multiplier $m_{32} = -5/0.2 = -25$ and subtract the results from the third equation. The system becomes triangular:

$$\begin{aligned} 10x_1 + 14x_2 &= 7 \\ 0.2x_2 + 6x_3 &= 6.1 \\ 155x_3 &= 155 \end{aligned}$$

For the general system [Eq. (1)], where $m = n$, the algorithm can be written as follows.

ALGORITHM 1. FORWARD ELIMINATION. Input the $n^2 + n$ entries of the extended matrix $W = [w_{ij}, i = 1, \ldots, n; j = 1, \ldots, n+1]$ of the system of Eq. (1), which occupies $n \times (n+1)$ working array:

For $k = 1, \ldots, n-1$,

For $i = k+1, \ldots, n$,

$w_{ik} := w_{ik}/w_{kk}$

For $j = k+1, \ldots, n+1$,

$w_{ij} := w_{ij} - w_{ik} w_{kj}$

In $n-1$ steps, the diagonal entries w_{ii} and the superdiagonal entries w_{ij}, $i < j$, of the working array W are overwritten by the entries of the extended matrix U of an upper triangular system equivalent to the original system [Eq. (1)], provided that in the kth step the denominator w_{kk} (called the pivot) is nonzero for $k = 1, \ldots, n-1$. For general system Eq. (1), the algorithm requires about $n^3/3$ flops, that is, more than $n^2/2$ flops used in back substitution. The multipliers m_{ik} usually overwrite the subdiagonal entries w_{ik} of the array (the previous contents are no longer needed and are deleted). Thus, the space of $n^2 + n$ units of an $n \times (n+1)$ array suffices to solve the general system Eq. (1), including the space for input and output. Additional n^2 (or $n^2 + n$) units of space are provided to save the inputs A (and $\bar{b}$) in some subroutines.

The entire computational process of forward elimination can be represented by the sequence of $n \times (n+1)$ matrices $W^{(0)}, \ldots, W^{(n-1)}$, where $W^{(k-1)}$ and $W^{(k)}$ denote the contents of the working arrays before and after elimination step k, respectively, $k = 1, \ldots, n-1$; $W^{(0)} = W$, whereas $W^{(n-1)}$ consists of the multipliers (placed under the diagonal) and of the entries of U.

EXAMPLE 9. The sequence $W^{(0)}, W^{(1)}, W^{(2)}$ represents forward elimination for a system of three equations of Example 1:

$$\begin{bmatrix} 10 & 14 & 0 & 7 \\ -3 & -4 & 6 & 4 \\ 5 & 2 & 5 & 6 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 14 & 0 & 7 \\ -0.3 & 0.2 & 6 & 6.1 \\ 0.5 & -5 & 5 & 2.5 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 10 & 14 & 0 & 7 \\ -0.3 & 0.2 & 6 & 6.1 \\ 0.5 & -25 & 155 & 155 \end{bmatrix}$$

The presented algorithm works if and only if the pivot entry (k, k) is zero in none step $k, k = 1, \ldots, n-1$, or, equivalently, if and only if for none k the $(n-k) \times (n-k)$ principal submatrix of the coefficient matrix A is singular. (A $p \times p$ submatrix of A is said to be the principal if it is formed by the first p rows and by the first p columns of A.) That assumption always holds (so the validity of the above algorithm is assured) in the two important cases where A is row or column diagonally dominant and where A is Hermitian (or real symmetric) positive definite (compare the list of special matrices in Section II.D). (For example, the system derived in Section II.A is simultaneously row and column diagonally dominant and real symmetric; multiplication of all the inputs by -1 makes that system also positive definite. The product $V^{\mathrm{H}}V$ for a nonsingular matrix V is a Hermitian positive definite matrix, which is real symmetric if V is real.)

Next, we assume that in some step k the pivot entry (k, k) is 0. Then we have two cases.

Case 1. The entries $(k, k), (k+1, k), \ldots, (s-1, k)$ are zeros; the entry (s, k) is nonzero, where $k < s \leq n$. In that case interchange rows s and k, bringing a nonzero entry into the pivot position; then continue the elimination.

EXAMPLE 10.

$$\begin{bmatrix} 10 & 14 & 0 & 7 \\ -3 & -4.2 & 6 & 4 \\ 5 & 2 & 5 & 6 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 14 & 0 & 7 \\ -0.3 & 0 & 6 & 6.1 \\ 0.5 & -5 & 5 & 2.5 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 10 & 14 & 0 & 7 \\ 0.5 & -5 & 5 & 2.5 \\ -0.3 & 0 & 6 & 6.1 \end{bmatrix}$$

Case 2. The pivot entry (k, k) and all the subdiagonal entries (s, k) for $s > k$ equal 0. In that case continue elimination, skipping the kth elimination step and leaving the (k, k) entry equal to 0. For underdetermined systems, apply complete pivoting (see Section III.C). Some subroutines end the computation in Case 2, indicating that the system is singular.

EXAMPLE 11.

$$\begin{bmatrix} 10 & 14 & 0 & 7 & 6 \\ -3 & -4.2 & 5 & 4 & 5 \\ 5 & 7 & 5 & 5 & 7 \\ 10 & 14 & 5 & 9 & 4 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 10 & 14 & 0 & 7 & 6 \\ -0.3 & 0 & 5 & 6.1 & 6.8 \\ 0.5 & 0 & 5 & 1.5 & 4 \\ 1 & 0 & 5 & 2 & -2 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 10 & 14 & 0 & 7 & 6 \\ -0.3 & 0 & 5 & 6.1 & 6.8 \\ 0.5 & 0 & 5 & 1.5 & 4 \\ 1 & 0 & 1 & 0.5 & -6 \end{bmatrix}$$

No row interchange is required in order to eliminate all the subdiagonal nonzero entries in the second column in Step 2.

The forward-elimination algorithm can be immediately extended to overdetermined and underdetermined systems.

EXAMPLE 12. Forward elimination for an underdetermined system:

$$\begin{bmatrix} 10 & -7 & 1 & 0 \\ -3 & 2 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & -7 & 1 & 0 \\ -0.3 & -0.1 & 0.3 & 1 \end{bmatrix}$$

Summarizing, forward elimination with pivoting reduces arbitrary linear system Eq. (1) to triangular form and, respectively, to either Case 1 or Case 2 of the previous section (see the end of Section III.C for complete classification).

Forward elimination and back substitution together are called Gaussian elimination algorithm. There exist several modifications of that algorithm. In one of them, Jordan's, the back substitution is interwoven throughout the elimination; that is, every pivot equation times appropriate multipliers is subtracted from all the subsequent and preceding equations; this turns the system into diagonal form in n elimination steps. [Each step, but the first, involves more flops, so the resulting solution of the general system of Eq. (1) becomes more costly.] In the case of systems of two equations, Jordan's version is identical to the canonical Gaussian elimination.

C. Gaussian Elimination Performed with Finite Precision. Pivoting Policies. Complete Pivoting for Singular Systems

Gaussian elimination, as presented in the previous sections, may easily fail in practical computations with finite precision, because such computations may greatly magnify the round-off errors if the pivot entries are close to 0.

EXAMPLE 13. Let the following sequence of working arrays represent Gaussian elimination performed on a computer with chopping to five decimals (see Section II.E about such chopping):

$$\begin{bmatrix} 10 & 14 & 0 & 7 \\ -3 & -4.198 & 6 & 3.901 \\ 5 & 2 & 6 & 7 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 10 & 14 & 0 & 7 \\ -0.3 & 0.002 & 6 & 6.001 \\ 0.5 & -5 & 6 & 3.5 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 10 & 14 & 0 & 7 \\ -0.3 & 0.002 & 6 & 6.001 \\ 0.5 & -2500 & 15006 & 15005 \end{bmatrix}$$

Solving the resulting upper triangular system, we obtain the following approximation to the solution:

$$x_3 = 0.99993, \qquad 0.002x_2 + 6 * (0.99993) = 6.001$$

$$x_2 = 0.75, \qquad 10x_1 + 14 * (0.75) = 7$$

$$x_1 = -0.35$$

This greatly differs from the correct solution, $x_1 = 0$, $x_2 = 0.5$, $x_3 = 1$, because the division (at the second elimination step) by the small diagonal entry 0.002 has magnified the round-off errors.

The algorithm can be made more *stable* (less sensitive to the errors) if the rows of working array are appropriately interchanged during the elimination. First, the following policy of row interchange is called (unscaled) partial pivoting. Before performing the kth elimination step, choose (the least) i such that $|w_{ik}|$ is maximum over all $i \geq k$ and interchage rows i and k. Row i is called pivotal, the entry w_{ik} is called the pivot entry of step k. Keep track of all the row interchanges. In some subroutines row interchanges are not explicit but are implicitly indicated by pointers. (Each step k may be preceded by scaling the equations by factors of 2^s on binary computers to make $\max_j |w_{ij}|$ in all rows $i \geq k$ lying between 1 and 2. Such scaling is expensive, however, so it is rarely repeated for $k > 1$.)

ALGORITHM 2. FORWARD ELIMINATION WITH UNSCALED PARTIAL PIVOTING.

For $h = 1, \ldots, n$

  $p_h = h$ (initialization)

For $k = 1, \ldots, n-1$

  Find the smallest $i \geq k$ such that $|w_{ik}| \geq |w_{lk}|$ for all $l \geq k$

  If $w_{ik} = 0$, end (A is singular)

  Else swap the contents of p_k and p_i

  Swap rows k and i of W

  For $i = k+1, \ldots, n$

    $w_{ik} := w_{ik}/w_{kk}$

    For $j = k+1, \ldots, n+1$

      $w_{ij} := w_{ij} - w_{ik}w_{kj}$

If $w_{nn} = 0$, end (A is singular).

In our previous example, partial pivoting after the first elimination step would change the working array W as follows:

$$\begin{bmatrix} 10 & 14 & 0 & 7 & 1 \\ -0.3 & 0.002 & 6 & 6.001 & 2 \\ 0.5 & -5 & 6 & 3.5 & 3 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 10 & 14 & 0 & 7 & 1 \\ 0.5 & -5 & 6 & 3.5 & 3 \\ -0.3 & 0.002 & 6 & 6.001 & 2 \end{bmatrix}$$

Here, the last column is the vector **p**, which monitors the row interchanges. Further computation with chopping to five decimals would give the correct solution, $x_3 = 1$, $x_2 = 0.5$, $x_1 = 0$.

Next, we discuss Complete (total) pivoting. Both rows and columns of W can be interchanged in elimination step k to bring the absolutely maximum coefficient in the left side of the last $n - k + 1$ equations of the current system to the pivot position (k, k); that is, after that interchange $|w_{kk}| \geq \max_{ij=k,\ldots,n} |w_{ij}|$. Two auxiliary vectors $[1, 2, \ldots, n]$ are formed to monitor the row and column interchanges.

EXAMPLE 14. Here are the pivot entries in the first elimination step for the system

$$200x_1 - 1000x_2 = 2200, \qquad x_1 - x_2 = 0$$

1. For unscaled partial pivoting, $a_{11} = 200$.
2. For complete pivoting, $a_{12} = -1000$.

Solution of some systems [Eq. (1)] (called ill-conditioned) is sensitive to input errors and to round-off errors,

no matter which algorithm is applied (see Sections III.I and III.J). For other systems (well-conditioned, such as in the first example of this section), the output errors in the presence of round-off depend on the algorithm, in particular, on pivoting. Theoretical estimates (Section III.J) show that complete pivoting prevents the solution from any substantial magnification of input and round-off errors in the case of well-conditioned systems. Practical computations show that even unscaled partial pivoting has similar properties for almost all well-conditioned systems [although it fails for some specially concocted instances of Eq. (1)]. Unscaled partial pivoting requires only $n(n-1)/2$ comparisons versus about $n^3/3$ in complete pivoting; therefore in practice, safe and inexpensive unscaled partial pivoting is strongly preferred, except for underdetermined systems, for which Gaussian elimination with complete pivoting or the methods of Section IV.B are recommended.

The triangular systems, to which the original systems [Eq. (1)] are reduced by forward elimination with complete pivoting, can be completely classified. We assume infinite precision of computation. Then there can be exactly two cases. (The reader may use examples from Section III.A.)

Case 1. All the diagonal (pivot) entries are nonzero; the triangular system has exactly r equations, $r = \text{rank}\,(A) \leq n$. In that case, $x_{r+1}, \ldots, x_n$ are free variables; for an arbitrary set of their values, back substitution defines unique set of values of $x_1, \ldots, x_r$ satisfying the original system of Eq. (1), (compare Theorem 1). If $r = n$, the system of Eq. (1) is nonsingular and has a unique solution.

Case 2. The triangular system includes one or more equations of the form, $0 = b_i^*$, where b_i^* is a constant. If at least one of those constants is nonzero, the original system is inconsistent. Otherwise, those equations become the identities, $0 = 0$ and can be deleted. The remaining triangular system belongs to Case 1.

D. Solving Several Systems with Common Coefficient Matrix. Matrix Inversion

Matrix inversion and many other problems are reduced to solving several linear systems [Eq. (1)] with the same coefficient matrix A and distinct vectors $\mathbf{b}$.

EXAMPLE 15. The two systems represented by their working arrays

$$\mathrm{W} = \begin{bmatrix} 10 & -7 & 1 \\ -3 & 2 & 0 \end{bmatrix}, \qquad \mathrm{W}^* = \begin{bmatrix} 10 & -7 & 0 \\ -3 & 2 & 1 \end{bmatrix}$$

define the inverse A^{-1} of their common coefficient matrix A. Represent these two systems by a 2×4 working array and apply forward elimination to that array (compare also Example 12 of Section III.B):

$$\begin{bmatrix} 10 & -7 & 1 & 0 \\ -3 & 2 & 0 & 1 \end{bmatrix} \to \begin{bmatrix} 10 & -7 & 1 & 0 \\ -0.3 & -0.1 & 0.3 & 1 \end{bmatrix}$$

Both systems have been simultaneously reduced to upper triangular form, so back substitution immediately gives the solutions $x_2 = -3, x_1 = -2$ to the first system and $x_2 = -10, x_1 = -7$ to the second. This defines the inverse matrix $A^{-1} = \begin{bmatrix} -2 & -7 \\ -3 & -10 \end{bmatrix}$. (Verify that $AA^{-1} = A^{-1}A = I$ for $A = \begin{bmatrix} 10 & -7 \\ -3 & 2 \end{bmatrix}$.)

In the elimination stage for k systems with a common $m \times n$ matrix A (as well as for an underdetermined system with $n = m + k - 1$), $m^3/3 + km^2/2$ flops and $(k+m)m$ units of storage space are used; for matrix inversion $k = m$, it is easier to solve the system $A\mathbf{x} = \mathbf{b}$ than to invert A. The back substitution stage involves $km^2/2$ flops for the k systems and $(k + m/2)m$ for the underdetermined system.

E. Block Matrix Algorithms

Arithmetic operations with matrices can be performed the same as those with numbers, except that singular matrices cannot be inverted, and the communitive law no longer holds for multiplications (see Section II.D). If the coefficient matrix is represented in block matrix form, as in Eq. (3), then we may perform block Gaussian elimination operating with matrix blocks the same as with numbers and taking special care when divisions and/or pivoting are needed. The block version can be highly effective. For instance, we represent the linear system of Section II.A as follows [compare Eq. (3)]:

$$\begin{bmatrix} B_2 & I_2 \\ I_2 & B_2 \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} \mathbf{c} \\ \mathbf{d} \end{bmatrix}, \qquad \mathbf{c} = \begin{bmatrix} -u_6 - u_{16} \\ -u_7 - u_9 \end{bmatrix}$$

$$\mathbf{d} = \begin{bmatrix} -u_{13} - u_{15} \\ -u_{10} - u_{12} \end{bmatrix}, \qquad B_2 = \begin{bmatrix} -4 & 1 \\ 1 & -4 \end{bmatrix}$$

where I_2 is the 2×2 identity matrix, and $\mathbf{y}$, $\mathbf{z}$ are two-dimensional vectors of unknowns. Then block forward elimination transforms the extended matrix as follows:

$$\begin{bmatrix} B_2 & I_2 & \mathbf{c} \\ I_2 & B_2 & \mathbf{d} \end{bmatrix} \to \begin{bmatrix} B_2 & I_2 & \mathbf{c} \\ B_2^{-1} & C_2 & \mathbf{f} \end{bmatrix}$$

Here, $C_2 = B_2 - B_2^{-1}$ and $\mathbf{f} = \mathbf{d} - B_2^{-1}\mathbf{c}$. Block back substitution defines the solution vectors

$$\mathbf{z} = C_2^{-1}\mathbf{f}, \qquad \mathbf{y} = B_2^{-1}(\mathbf{c} - \mathbf{z}) = B_2^{-1}\left(\mathbf{c} - C_2^{-1}\mathbf{f}\right)$$

The recent development of computer technology greatly increased the already high popularity and importance of block matrix algorithms (and consequently, of matrix multiplication and inversion) for solving linear systems, because block matrix computations turned out to be

particularly well suited and effective for implementation on modern computers and supercomputers.

F. *PLU* Factorization. Computing the Determinant of a Matrix

If Gaussian elimination requires no pivoting, then by the end of the elimination stage, the working array contains a lower triangular matrix L (whose subdiagonal entries are filled with the computed multipliers and whose diagonal entries are 1's) and the extended matrix $\hat{U} = A^{(n-1)}$ of the upper triangular system, equivalent to Eq. (1). The coefficient matrix U of that system is an upper triangular submatrix of $\hat{U}$. In Example 8,

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.3 & 1 & 0 \\ 0.5 & -25 & 1 \end{bmatrix},$$

$$\hat{U} = \begin{bmatrix} 10 & 14 & 0 & 7 \\ 0 & 0.2 & 6 & 6.1 \\ 0 & 0 & 155 & 155 \end{bmatrix},$$

$$U = \begin{bmatrix} 10 & 14 & 0 \\ 0 & 0.2 & 6 \\ 0 & 0 & 155 \end{bmatrix}$$

For that special instance of Eq. (1), $W = L\hat{U}$, $A = LU$; similarly for the general system of Eq. (1), unless pivoting is used. Moreover, Gaussian elimination with partial pivoting can be reduced to a certain interchange of the rows of W (and of A), defined by the output vector $\mathbf{p}$ (see Algorithm 2 in Section III.C), and to Gaussian elimination with no pivoting. Any row interchange of W is equivalent to premultiplication of W by an appropriate permutation matrix P^{-1} (say, if $P^{-1} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, then rows 1 and 2 are interchanged), so Gaussian elimination with pivoting computes matrices P, L, and $\hat{U}$ such that $P^{-1}W = L\hat{U}$, $P^{-1}A = LU$, $W = PL\hat{U}$, $A = PLU$; that is, the LU factors of $P^{-1}A$ and the $L\hat{U}$ factors of $P^{-1}W$ are computed. PLU factorization is not unique, it depends on pivoting policy; the elimination with no pivoting gives $P = 1$, $W = L\hat{U}$; $A = LU$.

When PLU factors of A are known, solving the system $A\mathbf{x} = \mathbf{b}$ is reduced to the interchange of the entries of $\mathbf{b}$ and to solving two triangular systems

$$L\mathbf{y} = P^{-1}\mathbf{b}, \qquad U\mathbf{x} = \mathbf{y} \tag{4}$$

for the total cost of n^2 flops. The back substitution of Section III.A saves $n^2/2$ of those flops, but computing the PLU factors of A saves $n^2/2$ of the "elimination flops," because the last column of the matrix $\hat{U}$ need not be computed. In fact, two ways of solving the system $A\mathbf{x} = \mathbf{b}$, that is, via PLU factorization and via Gaussian elimination of Sections III.A and III.B lead to exactly the same computations (within the order of performing the operations). In subroutine packages, the solution based on PLU factorization is usually preferred. Among its many applications, PLU factorization of A leads to very effective computation of the determinant of an $n \times n$ matrix A, $\det A = \det P \det U = (\det P)u_{11}u_{22}, \ldots, u_{nn}$, where $u_{11}, \ldots, u_{nn}$ denotes the diagonal entries of U and where $\det P = (-1)^s$, s being the total number of all the row interchanges made during the elimination.

Gaussian elimination applied to overdetermined or underdetermined systems also computes PLU factorizations of their matrices (see Sections IV.A–C about some other important factorizations in cases $m \neq n$).

G. Some Modifications of *LU* Factorization. Choleski's Factorization. Block Factorizations of a Matrix

If all the principal submatrices of an $n \times n$ matrix A are nonsingular, LU factors of A can be computed; furthermore, LDM^{T} factors of A can be computed where $DM^{\mathrm{T}} = U$, D is a diagonal matrix, $D = \mathrm{diag}\,(u_{11}, \ldots, u_{nn})$, so both L and M^{T} are unit triangular matrices (having only 1's on their diagonals). If the LDM^{T} factors of A are known, solving the system $A\mathbf{x} = \mathbf{b}$ can be reduced to solving the systems $L\mathbf{y} = \mathbf{b}$, $D\mathbf{z} = \mathbf{y}$, $M^{\mathrm{T}}\mathbf{x} = \mathbf{z}$, which costs only $n^2 + n$ flops, practically as many as in case where the LU factors are known. The following modification of Gaussian elimination computes the LDM^{T} factors of A.

ALGORITHM 3. LDM^{T} FACTORIZATION

For $k = 1, \ldots, n$

For $g = 1, \ldots, k-1$

$u_g := a_{gg}a_{gk}$

$v_g := a_{kg}a_{gg}$

$$a_{kk} := a_{kk} - \sum_{j=1}^{k-1} a_{kj}u_j$$

If $a_{kk} = 0$, end (A has no LDM^{T}-factors)
Else

For $i = k+1, \ldots, n$,

$$a_{ik} := \left(a_{ik} - \sum_{j=1}^{k-1} a_{ij}u_j\right) \Big/ a_{kk},$$

$$a_{ki} := \left(a_{ki} - \sum_{j=1}^{k-1} v_j a_{jk}\right) \Big/ a_{kk}$$

Algorithm 3 requires $n^3/3$ flops and $n^2 + 2n$ units of storage space. It keeps low the number of updatings of the

values a_{ij} store in the working array so that each a_{ij} is updated only in a single inner loop of the form $a_{kk} := a_{kk} - \sum_j a_{kj}u_j$ or $a_{ik} := (a_{ik} - \sum_j a_{ij}u_j)/a_{kk}$, or $a_{ki} := (a_{ki} - \sum_j v_j a_{ji})/a_{kk}$, where computing inner products is the main operation (easy on many serial computers).

Algorithm 3, for LDM^{T} factorization, is a simple extension of Crout's algorithm, which computes LD and M^{T}, and of Doolittle's algorithm, which computes L and $U = DM^{\mathrm{T}}$. If A is symmetric and has only nonsingular principal submatrices, then $L = M$, so Algorithm 3 computes the LDL^{T} factorization of A. If A is symmetric and positive definite, then all the diagonal entries of D are positive, so the matrix $\sqrt{D} = \mathrm{diag}\,[\sqrt{d_{11}}, \ldots, \sqrt{d_{nn}}]$ can be computed; then Choleski's factorization, $A = GG^{\mathrm{T}}$, $G = L\sqrt{D}$ can be computed. Algorithm 4 computes Choleski's factorization using $n^3/6$ flops and only $n(n+1)/2$ units of storage space.

ALGORITHM 4. CHOLESKI'S FACTORIZATION

For $k = 1, \ldots, n$

$$a_{kk} := \left(a_{kk} - \sum_{j=1}^{k-1} a_{kj}^2\right)^{1/2}$$

If $a_{kk} = 0$, end (A is not positive definite)
Else

For $i = k+1, \ldots, n$

$$a_{ik} := \left(a_{ik} - \sum_{j=1}^{k-1} a_{ij}a_{kj}\right) \Big/ a_{kk}$$

For complex Hermitian matrices, the factorizations LDL^{T} and GG^{T} are replaced by LDL^{H} and LL^{H}, respectively. The factorizations of A presented can be generally extended to the case where A is a block matrix, provided that the given and the computed block matrices can be inverted as required. For instance, let A be a 2×2 block-matrix,

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{5}$$

Then,

$$A = \begin{bmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & B \end{bmatrix} \begin{bmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \tag{6}$$

$$A^{-1} = \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & 0 \\ 0 & B^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix} \tag{7}$$

provided that $B = A_{22} - A_{21}A_{11}^{-1}A_{12}$ and A_{11} and B are nonsingular. This is the block version of the LDM^{T} factorization for A and of the respective triangular factorization of A^{-1}.

H. Error and Residual Vectors. Vector and Matrix Norms. Condition Number

The error of an approximation $\mathbf{x}^*$ to the solution $\mathbf{x}$ to Eq. (1) is measured by the error vector, $\mathbf{e} = \mathbf{x} - \mathbf{x}^*$. The error magnitude is measured by a norm $\|\mathbf{e}\|$ of $\mathbf{e}$ and by the relative error norm, $\|\mathbf{e}\|/\|\mathbf{x}\|$; $\mathbf{e}$ is not known until $\mathbf{x}$ is computed; as a substitution, the residual vector $\mathbf{r}$, its norm $\|\mathbf{r}\|$, and the relative residual norm $\|\mathbf{r}\|/\|\mathbf{b}\|$ are used. Here, $\mathbf{r} = \mathbf{r}(\mathbf{x}^*) = \mathbf{b} - A\mathbf{x}^*$, so $\mathbf{r} = A\mathbf{e}$, $\mathbf{e} = A^{-1}\mathbf{r}$ if A is nonsingular; a vector norm is nonuniquely defined by the following properties:

1. $\|\mathbf{v}\| \geq 0$ for all vectors $\mathbf{v}$, $\|\mathbf{v}\| = 0$ if and only if $\mathbf{v}$ is a null vector, filled with zeros.
2. $\|q\mathbf{v}\| = |q| * \|\mathbf{v}\|$ for all vectors $\mathbf{v}$ and all complex numbers q.
3. $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$ for all pairs of vectors $\mathbf{u}$ and $\mathbf{v}$ of the same dimension.

Maximum (Chebyshev's, uniform) norm, $\|\mathbf{v}\|_\infty = \max_i |v_i|$ and Hölder's norms, $\|\mathbf{v}\|_p = (\sum_i |v_i|^p)^{1/p}$, for $p = 1$ (least deviation norm) and $p = 2$ (Euclidean norm) are most customary vector norms. (Here, $\mathbf{v} = [v_i]$ is a vector.)

Every vector norm can be extended to its subordinate matrix norm,

$$\|A\| = \max_{\mathbf{v} \neq \mathbf{0}} \|A\mathbf{v}\|/\|\mathbf{v}\| \tag{8}$$

Chebyshev's and Hölder's norms define the matrix norms $\|A\|_\infty$, $\|A\|_p$. Also, the Frobenius norm (F norm) is a customary matrix norm, $\|A\|_F = (\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2)^{1/2}$, where A is an $m \times n$ matrix $[a_{ij}]$. All the matrix norms have properties 1–3 (see above) of the vector norms (where $\mathbf{v}$ and $\mathbf{u}$ are replaced by matrices); the subordinate matrix norms satisfy also the inequalities

$$\|AB\| \leq \|A\| * \|B\|, \quad \|A\mathbf{v}\| \leq \|A\| * \|\mathbf{v}\| \tag{9}$$

for all matrices A, B and vectors $\mathbf{v}$ such that AB and $A\mathbf{v}$ are defined. Here are some further properties of matrix norms.

Theorem 2. For any $m \times n$ matrix $A = [a_{ij}]$,

$$\begin{aligned}
\|A\|_\infty &= \max_i \sum_j |a_{ij}|, \\
\|A\|_1 &= \max_j \sum_i |a_{ij}|, \\
\|A\|_2^2 &\leq \|A\|_1 * \|A\|_\infty \\
\|A\|_\infty / \sqrt{n} &\leq \|A\|_2 \leq \sqrt{m}\,\|A\|_\infty \\
\|A\|_1 / \sqrt{m} &\leq \|A\|_2 \leq \sqrt{n}\,\|A\|_1 \\
\max_{ij} |a_{ij}| &\leq \|A\|_2 \leq \sqrt{mn}\, \max_{ij} |a_{ij}|, \\
\|A\|_2 &\leq \|A\|_F \leq \sqrt{n}\,\|A\|_2
\end{aligned}$$

The condition number of A [$\text{cond}(A) = \|A\| * \|A^{-1}\|$ if A is nonsingular, $\text{cond}(A) = \infty$ otherwise] is used in the error-sensitivity analysis for Eq. (1). Cond(A) depends on the choice of a matrix norm, but $\text{cond}(A) \geq \|I\| \geq 1$ in any of the cited norms.

EXAMPLE 16.

$$A = \begin{bmatrix} 10 & -7 \\ -3 & 2 \end{bmatrix}, \qquad A^{-1} = \begin{bmatrix} -2 & -7 \\ -3 & -10 \end{bmatrix}$$

$$\|A\|_1 = 17; \qquad \|A^{-1}\|_1 = 13,$$

$$\text{cond}_1(A) = \|A\|_1 \|A^{-1}\|_1 = 221$$

Hereafter, we shall use only the three subordinate matrix norms, $\|A\|_1$ (1-norm), $\|A\|_2$ (2-norm), $\|A\|_\infty$ (maximum norm). For the subordinate matrix norms, the equations $A\mathbf{x} = \mathbf{b}$, $\mathbf{x} = A^{-1}\mathbf{b}$, $\mathbf{r} = A\mathbf{e}$, $\mathbf{e} = A^{-1}\mathbf{r}$ imply that

$$(1/\text{cond}(A))(\|\mathbf{r}\|/\|\mathbf{b}\|) \leq \|\mathbf{e}\|/\|\mathbf{x}\| \leq \text{cond}(A)\|\mathbf{r}\|/\|\mathbf{b}\| \tag{10}$$

$$\text{cond}(A) \geq (\|\mathbf{e}\|/\|\mathbf{x}\|)/(\|\mathbf{r}\|/\|\mathbf{b}\|) \tag{11}$$

I. Sensitivity of Linear Systems to Input Errors

The solution to a linear system can be sensitive to input errors even if the computations are performed with infinite precision.

EXAMPLE 17.

$$\begin{bmatrix} 780 & 563 \\ 913 & 659 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 217 \\ 254 \end{bmatrix}$$

The correct solution is $x_1 = 1, x_2 = -1$. For the approximation $x_1^* = 0.341, x_2^* = -0.087$, we have the error and the residual vectors

$$\mathbf{e} = [0.659, -0.913]^{\mathrm{T}} \qquad \mathbf{r} = [-0.001, 0]^{\mathrm{T}}$$

so for that specific matrix, the addition of 0.001 to b_2, which is a very small perturbation of the vector $\mathbf{b} = [217; 254]^{\mathrm{T}}$, changes the solution from $\mathbf{x} = [1, -1]^{\mathrm{T}}$ to $\mathbf{x}^* = [0.341, -0.087]^{\mathrm{T}}$. Furthermore, $\|\mathbf{e}\|_\infty = 0.913$, $\|\mathbf{x}\|_\infty = 1$, $\|\mathbf{r}\|_\infty = 10^{-3}$, $\|\mathbf{b}\|_\infty = 254$, so Eq. (11) implies that

$$\text{cond}_\infty(A) \geq 0.913 * 254 * 10^3 > 0.2 * 10^6$$

Thus, in that example the linear system is ill-conditioned (its coefficient matrix A has a large condition number) and is sensitive to the input errors even if the computation is performed with infinite precision. Systems like Eq. (1) are not very sensitive to the input errors if the condition number of A is not large (then the system is called well conditioned). The latter fact follows from the next perturbation theorem, which bounds the output errors depending on the perturbation of inputs (on the input error) and on cond(A).

Perturbation Theorem. Let A, E be $n \times n$ matrices, $\mathbf{x}$, $\mathbf{e}$, $\mathbf{b}$, and $\boldsymbol{\Delta}$ be n-dimensional vectors, such that $A\mathbf{x} = \mathbf{b}$, $(A + E)(\mathbf{x} - \mathbf{e}) = \mathbf{b} + \boldsymbol{\Delta}$, $\text{cond}(A)\|E\| \leq \|A\|$. Then,

$$\frac{\|\mathbf{e}\|}{\|\mathbf{x}\|} \leq \frac{\text{cond}(A)}{1 - \text{cond}(A)\|E\|/\|A\|} \left(\frac{\|E\|}{\|A\|} + \frac{\|\boldsymbol{\Delta}\|}{\|\mathbf{b}\|} \right)$$

If $\boldsymbol{\Delta} = \mathbf{0}$, then $\|\mathbf{e}\|/\|\mathbf{x} - \mathbf{e}\| \leq \text{cond}(A)\|E\|/\|A\|$.

Remark 1. The $\text{cond}(A^{\mathrm{T}}A)$ may be as large as $(\text{cond}(A))^2$, so the transition from the system of Eq. (1) to $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$ is not generally recommended, even though the latter system is symmetric.

J. Sensitivity of Algorithms for Linear Systems to Round-Off Errors

The output errors may also be influenced by the round-off errors. The smaller such an influence, the higher *numerical stability* of the algorithm, which is a subject of major concern to users. J. H. Wilkinson applied his nontrivial techniques of backward error analysis to estimate that influence in cases where Gaussian elimination was applied. It turned out that the resulting output error bounds were the same as if the output errors stemmed entirely from some input errors. (This result is easily extended to computing LDM^{T} factorization and Choleski's factorization.) To state the formal estimates, we define (a) the matrix $|V|$ of the absolute values of the entries of a matrix $V = [v_{ij}]$, so $|V| = [|v_{ij}|]$; and (b) the unit round-off u for a given computer, u only depends on a floating-point finite precision (of d digits with a base β) u being the minimum positive value such that the computer represents $1 + u$ and 1 with finite precision as two distinct values; $u < \beta^{1-d}$. The desired estimates can be derived by analyzing the expressions $a_{ik} = \sum_{j=0}^{s} l_{ij}u_{jk}$, $s = \min\{i, k\}$, where $L = [l_{ij}]$, $U = [u_{jk}]$, $u_{ji} = l_{ij} = 0$ if $j > i$, $l_{ii} = 1$ for all i, $A = [a_{ij}]$, $A = LU$.

Theorem 3. Let L^*, U^*, and $\mathbf{x}^*$ denote approximations to the LU-factors of A and to the solution $\mathbf{x}$ to $A\mathbf{x} = \mathbf{b}$ computed on a computer with unit round-off u. Then

$$|L^*U^* - A| \leq 3\,\text{un}\{|A| + |L^*| * |U^*|\}$$

$(A - E)\mathbf{x}^* = \mathbf{b}$ where $|E| \leq \text{un}(3|A| + 5|L^*| * |U^*|\}$; here and hereafter the values of an order of $0(u^2)$ are ignored.

If PLU factorization of A has been computed, it suffices to replace A by PA^{-1} in Theorem 3. With pivoting, the entries of $|L|^*$ are bounded by 1, so $\|L^*\|_\infty \leq n$, and furthermore

$$\|E\|_\infty \leq \nu \left\{3\|A\|_\infty + 5n\|U^*\|_\infty\right\} \tag{12}$$

In the case where complete pivoting is applied, $\|U^*\|_\infty$ can be estimated by using Wilkinson's bound,

$$\max_{ij} |u_{ij}| \leq \left(2 \cdot 3^{1/2} \ldots n^{1/(n-1)} n\right)^{1/2} \max_{ij} |a_{ij}| \tag{13}$$

Many years of computational practice have convinced us that the latter bound is almost always pessimistic, even where unscaled partial pivoting is applied. The latter observation and the perturbation theorem of Section III.I imply that Gaussian elimination with complete pivoting never (and with partial pivoting rarely) greatly magnifies the input or round-off errors, unless the system of Eq. (1) is ill-conditioned, so that Gaussian elimination with pivoting is quite stable numerically. This analysis can be extended to some block matrix algorithms, in particular, to the block factorization (5)–(7) for Hermitian (real symmetric) and positive definite matrices A.

K. Iterative Improvement of Computed Solution and Condition Estimation

Let good approximations L^*U^* to the LU factors of $A = LU$ be computed. Then set $\mathbf{x}(0) = \mathbf{e}(0) = \mathbf{0}$, $\mathbf{r}(0) = \mathbf{b}$, and successively compute $\mathbf{r}(i+1) = \mathbf{r}(i) - A\mathbf{e}(i)$, in n^2 flops; $\mathbf{e}(i+1)$ from $L^*U^*\mathbf{e}(i+1) = \mathbf{r}(i+1)$, in n^2 flops [see Eq. (4)]; $\mathbf{x}(i+1) = \mathbf{x}(i) + \mathbf{e}(i+1)$, in n flops, for $i = 0, 1, 2, \ldots$. If the computations are done with infinite precision and if A is well conditioned, say, if $3\,\mathrm{cond}(A)\|A - L^*U^*\| < \|A\|$, then it can be shown that $\mathbf{x}(i+1)$ rapidly converges to the solution $\mathbf{x}$ to $A\mathbf{x} = \mathbf{b}$. Moreover, it can be shown that it is sufficient to use the d-bit precision with $d > \log_2 \mathrm{cond}(A)$ while computing $\mathbf{e}(i)$ and to use the (double) $2d$-bit precision while computing $\mathbf{r}(i+1)$ in the above algorithm in order to assure that every iteration decreases the error $\|\mathbf{x} - \mathbf{x}(i)\|$ by a constant factor, c, $0 < c < 1$. Thus, more and more correct digits of $\mathbf{x}$ are computed with each new iteration.

That algorithm of iterative improvement (by J. H. Wilkinson) can be extended to the case where an approximation C to A is available such that $\|C^{-1}A - I\|$ is sufficiently small and the system $C\mathbf{e} = \mathbf{r}$ can be quickly solved for $\mathbf{e}$. In particular $C = L^*U^*$ was implicitly defined above by its factors L^* and U^*. Since the progress in iterative improvement to $\mathbf{x}$ depends on $\mathrm{cond}(A)$, this suggests using the same algorithm of iterative improvement as an effective heuristic condition estimator. Indeed, to estimate the progress of iterative improvement, we only need an order of n^2 flops, whereas computing $\mathrm{cond}(A)$ generally requires us to invert A and costs more than n^3 flops.

IV. ORTHOGONAL FACTORIZATION AND SINGULAR LINEAR SYSTEMS

Solving linear systems via orthogonal factorization of the coefficient matrix is a stable method, particularly effective for singular and ill-conditioned linear systems.

A. Application to Overdetermined Systems

We again consider computing a least-squares solution to an overdetermined linear system, $A\mathbf{c} = \mathbf{f}$ (see Section II.E). We assume that A is an $m \times n$ matrix. The problem is equivalent to computing solution to the normal equations. $A^{\mathrm{T}}A\mathbf{c} = A^{\mathrm{T}}\mathbf{f}$, which amounts to the requirements that $\partial(\|\mathbf{r}(\mathbf{c})\|_2^2)/\partial c_j = 0$ for all j, where $\mathbf{r}(\mathbf{c}) = \mathbf{f} - A\mathbf{c}$ is the residual vector, $\mathbf{c} = [c_0, \ldots, c_n]^{\mathrm{T}}$, and $\|\mathbf{v}\|_2^2 = \sum_j v_j^2$. The symmetric matrix $A^{\mathrm{T}}A$ can be computed in $mn^2/2$ flops. The matrix $A^{\mathrm{T}}A$ is positive definite if it is nonsingular; then the normal equations can be solved via Choleski's factorization in $n^3/6$ flops, so computing a least-squares solution to $A\mathbf{c} = \mathbf{f}$ costs about $0.5n^2(m + n/3)$ flops. The best methods of orthogonal factorization require about $n^2(m - n/3)$ flops, but they are substantially more stable and can also be extended to the case of singular $A^{\mathrm{T}}A$. Iterative improvement of computed approximations to a least-squares solution to $A\mathbf{c} = \mathbf{f}$ is possible based on the following equivalent representation of the normal equations:

$$\begin{bmatrix} I & A \\ A^{\mathrm{T}} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{c} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{0} \end{bmatrix}$$

Remark 2. The problem of minimizing $\|\mathbf{f} - A\mathbf{c}\|_2$ can be immediately extended to the problem (called the weighted least-squares problem) of minimizing $\|D(\mathbf{f} - A\mathbf{c})\|_2$ for a diagonal matrix $D = \mathrm{diag}[d_1, \ldots, d_m]$ with positive entries on the diagonal. The methods of solution remain the same, they are just applied to the system $(DA)\mathbf{c} = D\mathbf{f}$. In Section II.E, we transformed the specific overdetermined system $A\mathbf{c} = \mathbf{f}$, with

$$A^{\mathrm{T}} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 970 & 990 & 1000 & 1040 \end{bmatrix}$$

$\mathbf{f}^{\mathrm{T}} = [4, 2, 0, -3]$, into the system $AU\mathbf{c}^* = \mathbf{f}$, such that $\mathbf{c} = U\mathbf{c}^*$,

$$c_0 = c_0^* - 1000_1^*, \qquad c_1 = c_1^*$$

$$U = \begin{bmatrix} 1 & -1000 \\ 0 & 1 \end{bmatrix};$$

$$(AU)^{\mathrm{T}} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -30 & -10 & 0 & 40 \end{bmatrix},$$

$$(AU)^{\mathrm{T}}AU = \begin{bmatrix} 4 & 0 \\ 0 & 2600 \end{bmatrix}$$

This changed the normal equations from the system $4c_0 + 4000c_1 = 3$, $4000c_0 + 4002600c_1 = 2740$ to the much more simple and more stable system, $4c_0^* = 3$, $2600c_1^* = -260$. Such a trick can be extended to the general overdetermined system $A\mathbf{c} = \mathbf{f}$ if a matrix U can be

computed such that $(AU)^{\mathrm{T}}AU$ is a nonsingular diagonal matrix. It is customary to apply that approach seeking $R = U^{-1}$ (rather than U) such that $A = QR$, $Q^{\mathrm{T}}Q$ is diagonal (or, furthermore, $Q^{\mathrm{T}}Q = I$) and R is an $n \times n$ upper triangular matrix. When such QR factors of A have been computed, the normal equations, $A^{\mathrm{T}}A\mathbf{c} = A^{\mathrm{T}}\mathbf{f}$, can be immediately simplified as follows. We substitute $A = QR$, $A^{\mathrm{T}} = R^{\mathrm{T}}Q^{\mathrm{T}}$, $Q^{\mathrm{T}}Q = I$ in the system $A^{\mathrm{T}}A\mathbf{c} = A^{\mathrm{T}}\mathbf{f}$ and arrive at the system $R^{\mathrm{T}}R\mathbf{c} = R^{\mathrm{T}}Q^{\mathrm{T}}\mathbf{f}$.

Case 1. $A^{\mathrm{T}}A$ is nonsingular. Then, all the diagonal entries of R^{T} are nonzeros, and the system can be premultiplied by $(R^{\mathrm{T}})^{-1}$, reduced to the upper triangular system $R\mathbf{c} = Q^{\mathrm{T}}\mathbf{f}$, and then solved by back substitution.

Case 2. $A^{\mathrm{T}}A$ is singular (see Section IV.C).

Algorithm 5 computes QR factorization of A using mn^2 flops and $n(m + n/2)$ space (A is overwritten by Q and $\mathbf{a}_{.h}$ denotes column h of A).

ALGORITHM 5 (MODIFIED GRAM–SCHMIDT).

For $h = 1, \ldots, n$

$r_{hh} := \|\mathbf{a}_{.h}\|_2$

For $i = 1, \ldots, m$,

$a_{ih} := a_{ih}/r_{hh}$

For $j = h + 1, \ldots, n$

$r_{hj} := \sum_{i=1}^{m} a_{ih}a_{ij}$

For $i = 1, \ldots, m$

$a_{ij} := a_{ij} - a_{ih}r_{hj}$

However, a faster algorithm, also completely stable (called the Householder transformation or Householder reflection), computes R and $Q^{\mathrm{T}}\mathbf{f}$ using $n^2(m - n/3)$ flops and mn space. The algorithm can be used for the more general purpose of computing an $m \times m$ orthogonal matrix $Q = Q_{m,m}$ and an $m \times n$ upper triangular matrix $R = R_{m,n}$ such that $A = QR$, $Q^{\mathrm{T}}Q = QQ^{\mathrm{T}} = I$. Previously we considered QR factorization, where Q had size $m \times n$ and R had size $n \times n$. Such QR factors of A can be obtained by deleting the last $m - n$ columns of $Q_{m,m}$ and the last $m - n$ rows of R (those last rows of R form a null matrix, for R is upper triangular). Householder transformation of A into R is performed by successive premultiplications of A by the Householder orthogonal matrices $H_k = I - 2\mathbf{v}_k\mathbf{v}_k^{\mathrm{T}}/\mathbf{v}_k^{\mathrm{T}}\mathbf{v}_k$, $k = 1, 2, \ldots, r$, where $r \le n$, and usually $r = n$. The vector $\mathbf{v}_k$ is chosen such that the premultiplication by H_k makes zeros of all the subdiagonal entries of column k of the matrix $A_k = H_{k-1}H_{k-1}\ldots H_1A$ and does not affect its columns $1, 2, \ldots, k - 1$. Such a choice of $\mathbf{v}(k)$ for $k = 1, 2$ is shown below for the matrix A of our previous example. Here is the general rule. Zero the first $k - 1$ entries of column k of A_k and let $\mathbf{a}(k)$ denote the resulting vector. Then $\mathbf{v}_k = \mathbf{a}(k) \pm \|\mathbf{a}(k)\|_2\mathbf{i}(k)$ where $\mathbf{i}(k)$ is the unit coordinate whose kth entry is 1 and whose other entries are zeros; the sign $+$ or $-$ is chosen the same as for the entry k of $\mathbf{a}(k)$ (if that entry is 0, choose, say, $+$)

Remark 3. A modification with column pivoting is sometimes performed prior to premultiplication by H_k for each k in order to avoid possible complications where the vector $\mathbf{a}(k)$ has small norm. In that case, column k of A is interchanged with column s such that $s \ge k$ and $\|\mathbf{a}(s)\|_2$ is maximum. Finally, QR factors are computed for the matrix AP, where P is the permutation matrix that monitors the column interchange. Column pivoting can be performed using $O(mn)$ comparisons.

When only the vector $Q^{\mathrm{T}}\mathbf{f}$ and the matrix $R_{m,n}$ must be computed, the vector $\mathbf{f}$ is overwritten by successively computed vectors $H_1\mathbf{f}$, $H_2H_1\mathbf{f}$, ..., and the matrices $H_1, H_2, \ldots$, are not stored. If the matrix Q is to be saved, it can be either explicitly computed by multiplying the matrices H_i together $[Q = (H_r H_{r-1} \ldots H_1)^{\mathrm{T}}]$ or implicitly defined by saving the vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_r$.

EXAMPLE 18. HOUSEHOLDER TRANSFORMATION WITH NO COLUMN PIVOTING. For the matrix A of Example 17, $\mathbf{v}_1^{\mathrm{T}} = [3, 1, 1, 1]$, so

$$A_1^{\mathrm{T}} = \begin{bmatrix} -2 & 0 & 0 & 0 \\ -2000 & 0 & 10 & 50 \end{bmatrix},$$

$$\mathbf{v}_2^{\mathrm{T}} = [0, \sqrt{2600}, 10, 50],$$

$$A_2^{\mathrm{T}} = \begin{bmatrix} -2 & 0 & 0 & 0 \\ -2000 & -\sqrt{2600} & 0 & 0 \end{bmatrix}$$

Givens transformation (Givens rotation) is sometimes applied to compute QR factors of A, although it requires $2n^2(m - n/3)$ flops and seems generally inferior to the Householder transformation. The Givens' method uses successive premultiplications of A by rotation matrices that differ from the identity matrix I only in their 2×2 submatrices of the following form, $\left[\begin{smallmatrix} c & d \\ -d & c \end{smallmatrix}\right]$, where $c^2 + d^2 = 1$. Each such premultiplication zeros a new entry of A.

B. Computing the Minimum 2-Norm Solution to an Underdetermined System

Gaussian elimination with complete pivoting solves an underdetermined system $A\mathbf{x} = \mathbf{b}$ with an $m \times n$ matrix A, $m \le n$, in $0.5m^2(n - m/3)$ flops, but does not define the unique solution having minimum 2-norm. The solution having minimum 2-norm can be computed by using $m^2(n - m/3)$ flops as follows. Apply the Householder transformation with column pivoting (see Remark 3) to

the *transposed* matrix A^{T} and compute its factorization $A^{\mathrm{T}}P = QR$, where

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

and R_{11} is an $r \times r$ nonsingular triangular matrix, $r = \mathrm{rank}(A)$, $Q = [Q_1, Q_2]$, and Q_1 is a square matrix. Then the minimum 2-norm solution $\mathbf{x}$ to the system $A\mathbf{x} = \mathbf{b}$ can be computed,

$$\mathbf{x} = Q_1\mathbf{y}, \qquad \begin{bmatrix} R_{11}^T \\ R_{12}^{\mathrm{T}} \end{bmatrix} \mathbf{y} = P^{-1}\mathbf{b}$$

unless the latter system (and then also the system $A\mathbf{x} = \mathbf{b}$) is inconsistent.

C. Applications to Overdetermined Systems of Deficient Rank

Householder transformation with column pivoting can be applied to a matrix A in order to compute the least-squares solution to $A\mathbf{c} = \mathbf{f}$ even where A does not have full rank, that is, where $r = \mathrm{rank}(A) < n \le m$. That algorithm first computes the factorization $AP = QR$ where

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

and R_{11} is an $r \times r$ nonsingular upper triangular matrix. Then the general solution to $A^{\mathrm{T}}A\mathbf{c} = A^{\mathrm{T}}\mathbf{f}$ (that is, the general least-squares solution to $A\mathbf{c} = \mathbf{f}$) is computed as follows:

$$\mathbf{c} = P^{-1}\begin{bmatrix} R_{11}^{-1}(\mathbf{g} - R_{12}\mathbf{b}) \\ \mathbf{b} \end{bmatrix}$$

where the vector $\mathbf{g}$ consists of the first r entries of $Q^{\mathrm{T}}\mathbf{f}$, and the vector $\mathbf{b}$ consists of the last $n - r$ entries of $P\mathbf{c}$. The latter $n - r$ entries can be used as free parameters in order to define a specific solution (the simplest choice is $\mathbf{b} = 0$). Formally, infinite precision of computation is required in that algorithm; actually, the algorithm works very well in practice, although it fails on some specially concocted instances, somewhat similarly to Gaussian elimination with unscaled partial pivoting.

A little more expensive [$n^2(m + 17n/3)$ flops and $2mn^2$ space versus $n^2(m - n/3)$ and mn^2 in the Householder transformation] but completely stable algorithm relies on computing the *singular value decomposition* (SVD) of A. Unlike Householder's transformation, that algorithm always computes the least-squares solution of the minimum 2-norm. The SVD of an $m \times n$ matrix A is the factorization $A = U\Sigma V^{\mathrm{T}}$, where U and V are two square orthogonal matrices (of sizes $m \times m$ and $n \times n$, respectively), $U^{\mathrm{T}}U = I_m$, $V^{\mathrm{T}}V = I_n$, and where the $m \times n$ matrix Σ may have nonzero entries only on the diagonal; the (i, i) diagonal; entry of Σ is denoted σ_i and is called the ith *singular value* of A; $\sigma_1 \ge \sigma_2 \ge \cdots \ge \sigma_p \ge 0$, $p = \min\{m, n\}$, so $p = n$ if $m \ge n$. Matrix Σ consists of its largest principal $p \times p$ diagonal submatrix, diag $[\sigma_1, \ldots, \sigma_p]$, banded with zeros.

EXAMPLE 19. The matrix $A = [0.4, 0.3]$ has the SVD.

$$[0.4, 0.3] = [1][0.5, 0]\begin{bmatrix} 0.8 & 0.6 \\ -0.6 & 0.8 \end{bmatrix}$$

The normal equations $A^{\mathrm{T}}A\mathbf{c} = A^{\mathrm{T}}\mathbf{f}$ are equivalent to the system $\Sigma^{\mathrm{T}}\Sigma^{\mathrm{T}}V\mathbf{c} = \Sigma^{\mathrm{T}}U^{\mathrm{T}}\mathbf{f}$; $\Sigma^{\mathrm{T}}\Sigma = \mathrm{diag}[\sigma_1^2, \ldots, \sigma_n^2]$ is an $n \times n$ matrix, for $m \ge n$. When the SVD of A is available, the minimum 2-norm solution is obtained by setting $\mathbf{c} = V(\Sigma^{\mathrm{T}}\Sigma)^*U^{\mathrm{T}}\mathbf{f}$; then $n \times n$ matrix $(\Sigma^{\mathrm{T}}\Sigma)^* = \mathrm{diag}[\sigma_1^{-2}, \ldots, \sigma_r^{-2}, 0, \ldots, 0]$ of rank $r = \mathrm{rank}(\Sigma)$ is called the *pseudo-inverse* of A.

As one of the by-products of computing the SVD of A, the number σ_1/σ_r can be computed. That number equals $\mathrm{cond}_2(A) = \|A\|_2 * \|A^{-1}\|_2$ for nonsingular matrices A and extends both the definition of the condition number of a matrix and its applications to error analysis to the case of singular matrices.

D. Orthogonalization Methods for a System with a Square Matrix

Methods of orthogonal factorization are also useful for systems $A\mathbf{x} = \mathbf{b}$ with a square $n \times n$ matrix A in the cases where A is ill-conditioned or singular. Householder orthogonalization solves such systems in $2n^3/3$ flops; SVD requires $6n^3$ flops.

V. ASYMPTOTIC AND PRACTICAL ACCELERATIONS OF SOLVING GENERAL LINEAR SYSTEMS

The upper estimates of $\ge n^3/3$ flops in algorithms for the linear systems of Eq. (1) for $m = n$ have led to a popular conjecture of the 1950s and 1960s that a nonsingular linear system of n equations cannot be solved using less than cn^3 arithmetic operations for some positive constant c. This was proven to be wrong in 1968. The result followed from Strassen's $n \times n$ matrix multiplication (hereafter referred to as MM) using $o(n^{2.81})$ arithmetic operations and recently successively implemented on the CRAY supercomputer. The block decomposition (5)–(7) of Section III.G reduces $2n \times 2n$ matrix inversion to two $n \times n$ matrix inversions and to six $n \times n$ MMs and recursively to several MMs of decreasing sizes; it follows that $o(n^{2.81})$ arithmetic operations suffice to invert an $n \times n$

matrix A and, consequently, to solve the system $A\mathbf{x}=\mathbf{b}$. Due to the new techniques of algorithmic design, the exponent 2.81 of *MM* of 1968 decreased to 2.522 in 1978–1979, below 2.52 in 1980, and below 2.4 in 1986. Consequently, in 1986 the asymptotic acceleration of *MM* already implied the complexity bound $o(n^{2.4})$ (rather than n^3) for solving linear systems.

Numerical stability of all such fast $n\times n$ matrix multipliers has been both theoretically proven in the most general case (by D. Bini and G. Lotti in 1980) and experimentally confirmed, but only the two algorithms (by Strassen and by Winograd), both supporting the upper bounds $cn^{2.81}$, became practical so far and compete with the classical algorithm for matrix multiplication [see Pan (1984); Golub and Van Loan (1996); Higham (1996)].

Here is a distinct approach, promising some practical improvement. Suppose that we seek the product $A\mathbf{b}$ of a 1000×1000 matrix A by a vector $\mathbf{b}$, both filled with zeros and ones and use a computer with 100 bit precision. Each entry of $A\mathbf{b}$ is at most 1000 and occupies at most 10 bits, so that 10 entries of $A\mathbf{b}$ can be easily computed and stored by using one arithmetic operation and a single word of memory, thus almost by 10 times improving the classical approach. The algorithm can be extended to matrices and vectors whose entries are bounded by integers [see Pan (1984); Bini and Pan (1994)].

The algorithms for matrix multiplication and inversion are applied to the solution of linear systems via block algorithms. The popularity and importance of such application are growing because of high efficiency of the implementation of matrix multiplication on modern supercomputers, particularly on loosely coupled multiprocessors.

VI. DIRECT SOLUTION OF SOME SPECIAL LINEAR SYSTEMS

Many special linear systems $A\mathbf{x}=\mathbf{b}$ can be solved by special efficient algorithms [recall, for example, Cholesky's method for symmetric positive-definite systems (Section III.G)]. Next, some typical special systems are considered (compare the end of Section II.D).

A. Banded, Block-Banded, Banded-Symmetric, and Symmetric Systems

A matrix $A=[a_{ij}]$ has bandwidth (p,q) if $a_{ij}=0$ whenever $i+p<j<i-q$. Such a matrix can be stored by using less than $(p+q+1)n$ memory words. The general $n\times n$ lower triangular matrix has bandwidth $(0,n-1)$; a tridiagonal matrix has bandwidth (1, 1). Banded systems frequently arise in applications. Their effective solution may rely on decompositions of A into the product of banded triangular matrices. In particular, if the bandwidth of A is (p,q) and if A has LU factors, then L and U have bandwidths $(0,q)$ and $(p,0)$, respectively; $pqn+r(n-(p^2+q^2)/2)+r^3/3$ flops, where $r=\min\{p,q\}$, suffice to compute the factors L and U. Then it remains to use $(p+q+1)n-(p^2+q^2)/2$ flops to solve the system $A\mathbf{x}=\mathbf{b}$. Partial pivoting partially destroys the band structure of A; however, the resulting PLU factorization of A defines matrices U still having bandwidth $(p+q,0)$ and L having at most $p+1$ nonzero entries per column. Consequently, a substantial flop saving is still possible.

Many banded systems are symmetric positive definite and/or diagonally dominant. In those cases, pivoting is unnecessary, and the band structure can be fully exploited; one-half of the flops used can be further saved if A has Cholesky's factorization.

$PLDL^{\mathrm{T}}P^{\mathrm{T}}$ factorization of an $n\times n$ symmetric matrix A (where P is a permutation matrix) can be computed, say, by Aasen's algorithm, using about $n^3/6$ (rather than $n^3/3$) flops and $O(n^2)$ comparisons for pivoting, even if A is not positive definite and has no Cholesky's factorization.

In many applications, linear systems have block-band structure. In particular, the numerical solution of partial differential equations is frequently reduced to solving block tridiagonal systems (see Section II.A). For such systems, block triangular factorizations of A and block Gaussian elimination are effective. Such systems can be also solved in the same way as usual banded systems with scalar coefficients. This would save flops against dense systems of the same size, but the algorithms exploiting the block structure are usually far more effective.

Apart from triangular factorization and Gaussian elimination, there exist other effective direct methods for block-banded systems. For instance, the odd–even reduction (the cyclic reduction) is effective for solving symmetric block tridiagonal systems $A\mathbf{x}=\mathbf{b}$ with $(2^s-1)\times(2^s-1)$, matrices of the form

$$A=\begin{bmatrix} D & F & & & \\ E & D & F & & \\ & \ddots & \ddots & \ddots & \\ & & & & F \\ & & & E & D \end{bmatrix}=PBP^{\mathrm{T}}$$

$$=P\begin{bmatrix} D & & & F & \ddots & \\ & \ddots & & E & \ddots & F \\ & & D & & & E \\ E & F & & D & & \\ & \ddots & \ddots & & \ddots & \\ & & F & & & \\ & & E & & & D \end{bmatrix}P^{\mathrm{T}}$$

Here, the blanks show the block entries filled with zeros; P and P^{T} are permutation matrices such that the matrix B obtained from the original matrix A by moving all the 2^{s-1} odd numbered rows and columns of A into the first 2^{s-1} positions. The first 2^{s-1} steps of Gaussian elimination eliminate all the subdiagonal nonzero blocks in the first 2^{s-1} columns of the resulting matrix. The $(2^{s-1}-1)\times(2^{s-1}-1)$ matrix in the right lower corner is again a block tridiagonal block Toeplitz matrix, so the reduction is recursively repeated until the system is reduced to a single block equation. (For an exercise, apply this algorithm to the system of Section II.A for the 4×6 grid, $n=15$, $s=4$.)

B. Toeplitz, Hankel, and Vandermonde Systems and Their Correlation to Polynomial Operations

Many scientific computations (for signal processing, for partial differential equations, in statistics, for approximation of functions by polynomials, and so on) are reduced to solving Toeplitz or Hankel systems $A\mathbf{x}=\mathbf{b}$, having Toeplitz or Hankel matrix A. A Hankel matrix becomes a Toeplitz matrix by appropriate row interchange (reflecting the rows about the median row), so we shall consider only Toeplitz systems. An $m\times n$ Toeplitz matrix is defined by its first row and its first column, which requires only $m+n-1$ units of storage space.

EXAMPLE 20. Toeplitz matrices:

$$\begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0\\ 0 & 1 & 0 & -1 & 0 & 0\\ 0 & 0 & 1 & 0 & -1 & 0 \end{bmatrix}, \qquad \begin{bmatrix} 1 & 4 & 5 & 6\\ 2 & 1 & 4 & 5\\ 3 & 2 & 1 & 4 \end{bmatrix},$$

$$\begin{bmatrix} 1 & 2 & 3\\ 2 & 1 & 2\\ 3 & 2 & 1 \end{bmatrix}$$

EXAMPLE 21. Polynomial division and solving a triangular Toeplitz system. Given two polynomials,

$$w(t)=u_4t^4+u_3t^3+u_2t^2+u_1t+u_0$$
$$v(t)=v_2t^2+v_1t+v_0$$

we compute the quotient $q(t)=q_2t^2+q_1t+q_0$ and the remainder $r(t)=r_1t+r_0$ of the division of $u(t)$ by $v(t)$ such that $u(t)=v(t)q(t)+r(t)$ or, equivalently, such that

$$\begin{bmatrix} v_2 & 0 & 0\\ v_1 & v_2 & 0\\ v_0 & v_1 & v_2\\ 0 & v_0 & v_1\\ 0 & 0 & v_0 \end{bmatrix}\begin{bmatrix} q_2\\ q_1\\ q_0 \end{bmatrix}+\begin{bmatrix} 0\\ 0\\ 0\\ r_1\\ r_0 \end{bmatrix}=\begin{bmatrix} u_4\\ u_3\\ u_2\\ u_1\\ u_0 \end{bmatrix}$$

Computing the coefficients, q_2, q_1, q_0 of $q(t)$ is equivalent to solving the triangular Toeplitz system,

$$\begin{bmatrix} v_2 & 0 & 0\\ v_1 & v_2 & 0\\ v_0 & v_1 & v_2 \end{bmatrix}\begin{bmatrix} q_2\\ q_1\\ q_0 \end{bmatrix}=\begin{bmatrix} u_4\\ u_3\\ u_2 \end{bmatrix}$$

When $q(t)$ is known, $r(t)$ is immediately computed, $r(t)=u(t)-v(t)q(t)$. For example, if $u(t)=t^4+t^3+t^2+t+1$, $v(t)=t^2-1$, then, $q(t)=t^2+t+2$, $r(t)=2t+3$, so the vector $\mathbf{q}=[1,1,2]^{\mathrm{T}}$ is the solution of the triangular Toeplitz system

$$\begin{bmatrix} 1 & 0 & 0\\ 0 & 1 & 0\\ -1 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1\\ 1\\ 2 \end{bmatrix}=\begin{bmatrix} 1\\ 1\\ 1 \end{bmatrix}$$

In general, computing the quotient of division of a polynomial of degree $n+k$ by a polynomial of degree $k+1$ is equivalent to solving a triangular Toeplitz system of size n. Solving both problems via fast Fourier transform involves $O(n\log n)$ arithmetic operations with a small overhead.

Solving a general Toeplitz system is also closely related to polynomial computations, namely, to computing Pade' approximants to a power series and to computing the greatest common divisor of two polynomials. Trench's algorithm of 1964 and other similar algorithms solve a Toeplitz system with an $n\times n$ matrix in $O(n^2)$ arithmetic operations (with a small overhead); algorithms of 1980 use only $O(n\log^2 n)$ with substantial overhead, and less than $8n\log^2 n$ are used in the more recent algorithms.

Vandermonde systems have the coefficient matrix A of the form $A=[a_{j+1}^i,\ i,\ j=0,1,\ldots,n]$. Solving system $A^{\mathrm{T}}\bar{p}=\bar{\mathbf{f}}$ with such a matrix A^{T} is equivalent to computing a polynomial $p(t)=p_0+p_1t+\cdots+p_nt^n$ that interpolates to a function $f(t)$ at some points $t_0,t_1,\ldots,t_n$, where $\mathbf{p}=[p_i]$, $\mathbf{f}=[f(t_i)]$, $i=0,1,\ldots,n$. That correlation can be exploited in order to solve the systems $A^{\mathrm{T}}\mathbf{p}=\mathbf{f}$ using n^2 flops, where A is an $n\times n$ Vandermonde matrix.

Numerical stability is frequently a problem in computations with polynomials and thus in the Toeplitz, Hankel, and Vandermonde computations.

Toeplitz, Hankel, and Vandermonde matrices are special cases of more general classes of structured matrices associated with operators of shift (displacement) and scaling; such an association enables us to extend the algorithms for Toeplitz linear systems to the systems of the cited general classes (see author's book with D. Bini).

C. Fast Fourier Transform Methods for Poisson Equations

Special linear systems arise from the Poisson equation,

$$\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = f(x, y)$$

on a rectangle, $0 \le x \le a$, $0 \le y \le b$. [The Laplace equation of Section II.A is a special case where $f(x, y) = 0$.] If finite differences with N points per variable replace the partial derivatives, the resulting linear system has N^2 equations. Such systems can be solved in $O(N^2 \log N)$ flops with small overhead by special methods using fast Fourier transform (FFT) versus an order of N^4 flops, which would be required by Gaussian elimination for that special system. Storage space also decreases from $2N^3$ to N^2 units. Similar saving of time and space from $O(N^7)$ flops, $2N^5$ space units to $O(N^3 \log N)$ flops and N^3 space units is due to the application of FFT to the solution of Poisson equations on a three-dimensional box.

VII. DIRECT ALGORITHMS FOR SPARSE AND WELL-STRUCTURED LINEAR SYSTEMS

A. Sparse Linear Systems and the Associated Graphs

The coefficient matrices of banded and block-banded linear systems of Section VI.A are sparse, that is, filled mostly with zeros, and have some regular patterns for all the nonzero entries. Linear systems with those two features arise most frequently in applications (compare the regular patterns of banded and block-banded systems, where nonzeros are grouped about the diagonal). Such sparse and structured systems can be solved efficiently using special fast direct or iterative algorithms and special data structures allowing the storage of only nonzero coefficients. To see the structure of the coefficient matrix A and to choose appropriate data structures for its representation, replace the nonzero entries of A by ones. The resulting $n \times n$ matrix, $B = [b_{ij}]$, filled with zeros and ones, can be interpreted as the adjacency matrix of a (directed) graph $G = (V, E)$ consisting of the vertex set $V = \{1, 2, \ldots, n\}$ and of the edge set E such that there exists are (i, j) from vertex i to vertex j if and only if $b_{ij} = 1$ or, equivalently, if and only if $a_{ij} \neq 0$. Note that the graph G is undirected if the matrix A is symmetric. The special data structures used in graph algorithms (linked lists, stacks, queues) are extended to the computations for linear systems.

B. Fill-In; Pivoting (Elimination Order) as a Means of Decreasing Fill-In

In the process of Gaussian elimination (with, say partial pivoting), applied to a sparse system, some zero entries of the working array may be filled with nonzeros. The set of such entries is called fill-in. Large fill-in leads to increasing both time and space used for solving linear systems. Some special pivoting policies (called orderings) of Gaussian elimination have been developed in order to keep fill-in low. Pivoting used for stabilization (see Section III) is respectively modified to avoid conflicts. Typically, a certain degree of freedom is introduced by applying threshold (rather than partial or complete) stabilization pivoting such that a pivot in the first elimination step can be any entry (i, l) such that $|a_{i1}| \ge t \max_h |a_{h1}|$, where t is a chosen tolerance value $0 < t < 1$ (and similarly in the further steps). The resulting freedom in pivoting is used to keep the fill-in low. In the important cases of symmetric positive-definite and/or symmetric diagonally dominant systems $A\mathbf{x} = \mathbf{b}$, no stabilization pivoting is needed, so the problem is simplified. To decrease the fill-in, rows and columns of A are interchanged such that the symmetry of A is preserved; the resulting triangular factorization takes the form $A = PLL^{\mathrm{T}}P^{\mathrm{T}}$, where L is a lower triangular matrix and P is a permutation matrix, which defines the elimination order.

C. Some Policies of Pivoting for Sparse Systems

The most universal elimination ordering is given by the Markowitz's rule, reduced to the minimum degree rule in the symmetric case. Let p_i and q_j denote the numbers of nonzeros in row i and column j of the coefficient matrix A, respectively. Then the Markowitz rule requires us to choose the nonzero entry (i, j) that minimizes the value $(p_i - 1)(q_j - 1)$ and to move that entry into the pivot position (1, 1) in the first elimination step. (The ties can be broken arbitrarily.) The same rule is applied to the subsystem of $n - k + 1$ remaining (last) equations in elimination step k for $k = 2, 3, \ldots, n - 1$. In the symmetric case, $p_i = q_i$ for all i, so the Markowitz rule is reduced to minimization of p_i [rather than of $(p_i - 1)(q_i - 1)$]. For instance, let A be symmetric.

$$A = \begin{bmatrix} x & x & x & x & \cdots & x \\ x & x & & & & \\ x & & x & & & \\ x & & & x & & \\ \vdots & & & & \ddots & \\ x & & & & & x \end{bmatrix}$$

where all the nonzero entries of A are located on the diagonal, in the first row, and in the first column and are denoted by x. Then the fill-in of Gaussian elimination with no pivoting would make the matrix dense, which would require an increase in the storage space from $3n - 2$ to n^2 units.

With the Markowitz rule for this matrix, no fill-in will take place.

There also exist general ordering policies that

1. Decrease the bandwidth (Cuthill–McKee) or the profile (reversed Cuthill–McKee, King) of a symmetric matrix A [the profile of a symmetric matrix A equals $\sum_{i=1}^{n}(i - \min_{a_{ij}\neq 0} j)$]
2. Reduce the matrix A to the block diagonal form or to the block triangular form with the maximum number of blocks (policies 1 and 2 amount to computing all the connected components or all the strongly connected components, respectively, of the associated graph G
3. Represent symmetric A as a block matrix such that elimination of the block level causes no fill-in (tree partitioning algorithms for the associated graph G). Effective dissection algorithms customarily solve linear systems whose associated graphs G have small separators, that is, can be partitioned into two or more disconnected subgraphs of about equal size by removing relatively few vertices. For instance, in many applications G takes the form of an $\sqrt{n}\times\sqrt{n}$ grid on the plane. Removing $2\sqrt{n}-1$ vertices of the horizontal and vertical medians separates G into four disconnected grids, each with $(n+1-2\sqrt{n})/4$ vertices. This process can be recursively repeated until the set of all the separators includes all the vertices. The nested dissections of this kind define the elimination orders (where the separator vertices are eliminated in the order-reversing the process of dissection), which leads to a great saving of time and space. For instance, for the $\sqrt{n}\times\sqrt{n}$ plane grid, the nested dissection method requires $O(n^{1.5})$ flops with small overhead, rather than $n^3/3$. Furthermore the triangular factors L and U of PA (or Cholesky's factor L and L^{T} of PAP^{T} in the symmetric case) are filled with only $O(n)$ nonzeros, so $O(n)$ flops suffice in the substitution stage, which makes the method even more attractive where the right side $\mathbf{b}$ varies and the coefficient matrix A is fixed.

D. Solving Path Algebra Problems via Their Reduction to Linear Systems. Exploiting Sparsity

Although effective algorithms for solving sparse linear systems exploit some properties of the associated graphs, many combinatorial problems can be effectively solved by reducing them to linear systems of equations whose coefficient matrix is defined by the graph, say, is filled with the weights (the lengths) of the edges of the graph. In particular, that reduction is applied to path algebra problems, such as computing shortest or longest paths between some fixed pairs or between all pairs of vertices, computing paths having bounded length, counting the numbers of circuits or of distinct paths between two given vertices, and testing graphs for connectivity. Numerous applications include the problems of vehicle routing, investment and stock control, network optimization, artificial intelligence and pattern recognition, encoding and decoding of information, and so on. The resulting linear systems are usually sparse; special techniques (such as the Markowitz rule and nested dissection) can be extended to this case.

VIII. ITERATIVE ALGORITHMS FOR SPARSE AND SPECIAL DENSE LINEAR SYSTEMS

Iterative algorithms are recommended for some linear systems $A\mathbf{x}=\mathbf{b}$ as an alternative to direct algorithms. An iteration usually amounts to one or two multiplications of the matrix A by a vector and to a few linear operations with vectors. If A is sparse, small storage space suffices. This is a major advantage of iterative methods where the direct methods have large fill-in. Furthermore, with appropriate data structures, arithmetic operations are actually performed only where both operands are nonzeros; then, $D(A)$ or $2D(A)$ flops per iteration and $D(A)+2n$ units of storage space suffice, where $D(A)$ denotes the number of nonzeros in A. Finally, iterative methods allow implicit symmetrization, when the iteration applies to the symmetrized system $A^{\mathrm{T}}A\mathbf{x}=A^{\mathrm{T}}\mathbf{b}$ without explicit evaluation of $A^{\mathrm{T}}A$, which would have replaced A by less sparse matrix $A^{\mathrm{T}}A$.

Consider the two classical iterative methods: Jacobi and Gauss–Seidel. Hereafter, $\mathbf{x}^{(s)}=[x_1^{(s)},\ldots,x_n^{(s)}]^{\mathrm{T}}$ denotes the approximation to the solution vector x computed in iteration s, $s=1,2,\ldots$.

ALGORITHM 6 (JACOBI)

$$x_i^{(s+1)}=\frac{b_i-\sum_{j=1}^{i-1}a_{ij}x_j^{(s)}-\sum_{j=i+1}^{n}a_{ij}x_j^{(s)}}{a_{ii}}$$
$$i=1,\ldots,n$$

ALGORITHM 7 (GAUSS–SEIDEL)

$$x_i^{(s+1)}=\frac{b_i-\sum_{j=1}^{i-1}a_{ij}x_j^{(s+1)}-\sum_{j=i+1}^{n}a_{ij}x_j^{(s)}}{a_{ii}}$$
$$i=1,\ldots,n$$

Example 22. $4x_1-x_2=7$, $-x_1+4x_2=2$, $x_1=2$, $x_2=1$ is the solution. Let $x_1^{(0)}=x_2^{(0)}=0$. Then, the Jacobi iterations give $x_1^{(1)}=1.75$, $x_2^{(1)}=0.5$, $x_1^{(2)}=1.875$, $x_2^{(2)}=$

0.9375, and so on. The Gauss–Seidel iterations give $x_1^{(1)} = 1.75$, $x_2^{(1)} = 0.9375$, $x_1^{(2)} = 1.984375$, $x_2^{(2)} = 0.99609375$, and so on.

The Jacobi and Gauss–Seidel iterations can be expressed in matrix form. Let us represent A as follows: $A = L + D + U$, where $D = \text{diag}[a_{11}, \ldots, a_{nn}]$, and L and U are lower and upper triangular matrices whose diagonals are filled with zeros.

Example 23

$$A = \begin{bmatrix} 4 & -1 \\ -1 & 4 \end{bmatrix}, \qquad D = \text{diag}[4, 4]$$

$$L = \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix}, \qquad U = \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix}$$

Then, we may rewrite Algorithm 6 (Jacobi) as $D\mathbf{x}^{(s+1)} = \mathbf{b} - (L + U)\mathbf{x}^{(s)}$, and Algorithm 7 (Gauss–Seidel) as $(D + L)\mathbf{x}^{(s+1)} = \mathbf{b} - U\mathbf{x}^{(s)}$. Here is a more general iteration scheme.

ALGORITHM 8. Let $A = P - N$, where A and P are nonsingular (and P is readily invertible). Let $P\mathbf{x}^{(s+1)} = \mathbf{b} + N\mathbf{x}^{(s)}$, $s = 0, 1, \ldots$.

Examples. In Jacobi iteration, $P = D$, $N = -(L + U)$; in Gauss–Seidel iteration, $P = D + L$, $N = -U$.

THEOREM 3. Algorithm 8 converges to the solution $\mathbf{x} = A^{-1}\mathbf{b}$ to the system $A\mathbf{x} = \mathbf{b}$ if there exists a matrix norm such that $\rho = \|P^{-1}N\| < 1$. Furthermore in that norm $\|\mathbf{x} - \mathbf{x}^{(s)}\| \le \rho^s \|\mathbf{x} - \mathbf{x}^{(0)}\|$.

COROLLARY 1. Jacobi iteration converges if A is a diagonally dominant matrix; Gauss-Seidel iteration converges if A is symmetric positive definite.

It is known that the greatest lower bound $\rho(W)$ on all the norms $\|W\|$ of a matrix W equals the absolute value of the absolutely largest eigenvalue λ of W; the eigenvalues of W are the values λ such that $\det(W - \lambda I) = 0$; $\rho(W)$ is called the spectral radius of W. Theorem 3 implies that $\rho(P^{-1}N)$ defines the convergence rate of Algorithm 8. Estimating $\rho(P^{-1}N)$ is generally harder than solving linear systems $A\mathbf{x} = \mathbf{b}$, but for many specific choices of P and N, such estimates are readily available.

Appropriate variation of the splitting $A = P - N$ may imply smaller spectral radius $\rho(P^{-1}N)$ and thus faster convergence. In particular, we may try to accelerate the Jacobi or Gauss–Seidel iteration, choosing a positive β and modifying the splitting $A = P^* - N^*$ of those iteractions as follows: $A = (1 + \beta)P^* - (N + \beta P^*)$. In fact the customary variation of Gauss-Seidel, called the successive overrelaxation (SOR) method, is a bit different: $A = P - N$, $P = (D + \omega L)$, $N = ((1 - \omega)D - \omega U)$, $\omega > 1$; ω is called the relaxation parameter [$\omega = 1$ means Gauss-Seidel splitting, $A = (D + L) - (-U)$]. For some linear systems, we know how to choose appropriate ω in order to obtain dramatic acceleration of the convergence of Gauss–Seidel; in other cases, ω is defined by additional analysis or by heuristics. There exists a modification of SOR called the symmetric SOR (SSOR) method, which amounts to combining SOR with implicit symmetrization of the system $A\mathbf{x} = \mathbf{b}$.

Another modification (frequently combined with SSOR) is the Chebyshev semi-iterative acceleration, which replaces the approximation $x^{-(k+1)}$ by

$$\mathbf{y}^{(k+1)} = \omega_{k+1}\left[\mathbf{y}^{(k)} - \mathbf{y}^{(k-1)} + \gamma P^{-1}\left(\mathbf{b} - A\mathbf{y}^{(k)}\right)\right] + \mathbf{y}^{(k-1)}$$

where $\mathbf{y}^{(0)} = \mathbf{x}^{(0)}$, $\mathbf{y}^{(1)} = \mathbf{x}^{(1)}$, and γ and ω_{k+1} are some scalars, responsible for the acceleration and somewhat similar to the relaxation parameter ω of SOR. Some of these and other iterative methods are sometimes applied in block form.

The algorithms solve the linear systems that arise from a PDE discretized over a sequence of d-dimensional grids $G_0, \ldots, G_k$, rather than over a single grid, as in Section II.A. The grid G_{i+1} refines G_i and has by about 2^d times more points. The solution on a coarser grid G_i is simpler, but it supplies a good initial approximation to the solution on G_{i+1}, and then $O(1)$ iteration steps refine this approximation. It was recently observed that adding $O(1)$ bits of storage space per a solution point in each transition to a finer grid also suffices, which means the overall storage space of $O(n)$ binary bits ($n = N_k$ is the number of points in the finest grid G_k), and for constant coefficient linear PDEs, this also means that only a constant number, $O(1)$, of binary bits are needed in each of $O(n)$ arithmetic operations of this process. The search for the most effective preconditioning is the area of active research. Here is one of the versions, where $(\mathbf{u}, \mathbf{v})$ denotes $\mathbf{u}^{\mathrm{T}}\mathbf{v}$, and the matrix B is precomputed. If $B = I$, this becomes the original conjugate gradient algorithm.

PRECONDITIONED CONJUGATE GRADIENT ALGORITHM

$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$, $\mathbf{q}^{(0)} := B^{-1}\mathbf{r}^{(0)}$, $\mathbf{r}^{(0)} := \mathbf{q}^{(0)}$

For $k = 0, 1, \ldots$

$c_k := \left(\mathbf{r}^{(k)}, \mathbf{r}^{(k)}\right) / \left(\mathbf{p}^{(k)}, A\mathbf{p}^{(k)}\right)$

$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + c_k\mathbf{p}^{(k)}$

If not converged, do

$\mathbf{r}^{(k+1)} := \mathbf{r}^{(k)} - c_k A\mathbf{p}^{(k)}$, $\mathbf{q}^{(k+1)} := B^{-1}\mathbf{r}^{(k+1)}$

$d_k = \left(\mathbf{r}^{(k+1)}, \mathbf{q}^{(k+1)}\right) / \left(\mathbf{r}^{(k)}, \mathbf{q}^{(k)}\right)$,

$\mathbf{p}^{(k+1)} = \mathbf{q}^{(k+1)} + d_k\mathbf{p}^{(k)}$

Conjugate gradient methods are closely related to the Lanczos method, also used as an iterative algorithm for symmetric linear systems. A large and growing class of highly effective methods for linear systems arising from partial differential equations (PDEs) is known as multigrid and multilevel methods. For some linear systems (defined on grids) the alternating direction implicit (ADI) method is also effective. The multigrid methods solve a large class of problems on a grid of n points with sufficiently high precision within $O(n)$ flops.

Several most popular iterative algorithms for linear systems $A\mathbf{x}=\mathbf{b}$ rely on the residual minimization in the Krylov subspace, so that the successive approximations $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k$ to the solution minimize the residual norms $\|\mathbf{r}_i\|$ where $\mathbf{r}_i = C(\mathbf{b} - A\mathbf{x}_i)$, $i=0, 1, \ldots, k$, and $C=I$ or $C=A^H$ (conjugate gradient and orthogonalization techniques). Sometimes the minimization is replaced by a related but weaker requirement, such as orthogonality of $\mathbf{r}_i$ to $B^j\mathbf{r}_0$ for all $j<i$, where $B=A^*$ (BIOMIN or CGS). The power of this approach is due to the three term relations expressing $\mathbf{x}_i$ through $\mathbf{x}_j$ for $j<i$, so that every iteration step is essentially reduced to one or two matrix-by-vector multiplications. Many algorithms of this class compute the exact solution $\mathbf{x}=A^{-1}\mathbf{b}$ in n steps (in the absence of round-off errors), but much fewer steps are needed in order to compute a good approximation to many linear systems (even under round-off). The convergence rate is defined by the eigenvalues of A or of A^HA (singular values of A), and there are various techniques of preconditioning, which replace A by matrices CAD for appropriate readily invertible matrices C and D and which accelerate the convergence of such methods for many linear systems.

IX. INFLUENCE OF THE DEVELOPMENT OF VECTOR AND PARALLEL COMPUTERS ON SOLVING LINEAR SYSTEMS

The development of vector and parallel computers has greatly influenced methods for solving linear systems, for such computers greatly speed up many matrix and vector computations. For instance, the addition of two n-dimensional vectors or of two $n\times n$ matrices or multiplication of such a vector or of such a matrix by a constant requires n or n^2 arithmetic operations, but all of them can be performed in one parallel step if n or n^2 processors are available. Such additional power dramatically increased the previous ability to solve large linear systems in a reasonable amount of time. This development also required revision of the previous classification of known algorithms in order to choose algorithms most suitable for new computers. For instance, Jordan's version of Gaussian elimination (Section III.B) is slow on serial computers, but seems more convenient on many parallel machines because $n-1$ nonzeros in a column are eliminated in each elimination step. This is better suited to parallel computation with, say $n-1$ processors than the usual Gaussian elimination, where $n-k$ nonzeros are eliminated in step k and k varies from 1 to $n-1$. Variation of k complicates synchronization of the computation on all processors and communication among them. Similar problems characterize the additive steps of computing inner products, so the designer of parallel algorithms does not always adopt usual tendency of the designer of sequential algorithms to exploit maximally the inner product computation. In another example, computing the inverse A^{-1} of A (and, consequently, computing the solution $\mathbf{x}=A^{-1}\mathbf{b}$ to $A\mathbf{x}=\mathbf{b}$) can be performed via Newton's iterations, $B_{h+1}=2B_h - B_hAB_h$, $h=0, 1, \ldots$. If A is a well-conditioned $n\times n$ matrix, then the choice $B_0=tA^{\mathrm{T}}$, $t=1/(\|A\|_1\|A\|_\infty)$ ensures that $\|A^{-1}-B_h\|$ is already sufficiently small, when $h=O(\log n)$. Matrix multiplications can be performed rapidly on some parallel computers, so Newton's algorithm can be useful for solving linear systems on some parallel machines, although as a sequential algorithm it is certainly inferior to Gaussian elimination.

It is frequently effective to use block representation of parallel algorithms. For instance, a parallel version of the nested dissection algorithm of Section VIII.C for a symmetric positive-definite matrix A may rely on the following recursive factorization of the matrix $A_0=PAP^{\mathrm{T}}$, where P is the permutation matrix that defines the elimination order (compare Sections III.G–I):

$$A_h=\begin{bmatrix} X_h & Y_h^{\mathrm{T}} \\ Y_h & Z_h \end{bmatrix}$$

where $Z_h = A_{h+1} + Y_hX_h^{-1}Y_h^{\mathrm{T}}$ for $h=0, 1, \ldots, d-1$, and X_h is a block diagonal matrix consisting of square blocks of rapidly decreasing sizes, say, $(2^{-h}n)^{0.5} \times (2^{-h}n)^{0.5}$, so

$$A_h^{-1}=\begin{bmatrix} I & -X_h^{-1}Y_h^{\mathrm{T}} \\ 0 & I \end{bmatrix}\begin{bmatrix} X_h^{-1} & 0 \\ 0 & A_{h+1}^{-1} \end{bmatrix}\begin{bmatrix} I & 0 \\ -Y_hX_h^{-1} & I \end{bmatrix}$$

Such a reclassification of the available algorithms for linear systems due to the recent and current development of computer technology is an area of active research. Making the final choice for practical implementation of parallel algorithms requires some caution. The following quotient q is a good measurement for the speedup due to using a certain parallel algorithm on p processors:

$$q=\frac{\text{Execution time using the fastest known sequential algorithm on one processor}}{\text{Execution time using a parallel algorithm on } p \text{ processors.}}$$

The execution time includes various synchronization and communication overheads, which means that in fact only part of the whole computational work can be performed in parallel. The overheads are usually machine dependent and are harder to estimate, so theoretical analysis frequently ignores them and tends to give overly optimistic estimates for the power of various parallel algorithms.

SEE ALSO THE FOLLOWING ARTICLES

COMPUTER ALGORITHMS • DIFFERENTIAL EQUATIONS • LINEAR OPTIMIZATION • NONLINEAR PROGRAMMING • NUMERICAL ANALYSIS

BIBLIOGRAPHY

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1995). "LAPACK, User's Guide," release 2.0, second edition, SIAM, Philadelphia.

Bini, D., and Pan, V. Y. (1994). "Polynomial and Matrix Computations," Birkhauser, Boston.

Chvatal, V. (1983). "Linear Programming," Freeman, San Francisco, CA.

Demmel, J. W. (1996). "Numerical Linear Algebra," SIAM, Philadelphia.

Dongarra, J., Bunch, J. R., Moler, C. B., and Stewart, G. W. (1978). "LINPACK Users Guide," SIAM, Philadelphia.

Golub, G. H., and van Loan, C. F. (1996). "Matrix Computations," third edition, Johns Hopkins Press, Baltimore, MD.

Gondran, M., and Minoux, M. (1984). "Graphs and Algorithms," Wiley (Interscience), New York.

Greenbaum, A. (1997). "Iterative Methods for Solving Linear Systems," SIAM, Philadelphia.

Higham, N. J. (1996). "Accuracy and Stability of Numerical Analysis," SIAM, Philadelphia.

Kailath, T., and Sayed, A., eds. (1999). "Fast Reliable Algorithms for Matrices with Structure," SIAM, Philadelphia.

Pan, V. (1984). "How to Multiply Matrices Faster," Lecture Notes in Computer Science 179, Springer-Verlag, Berlin.

Pissanetsky, S. (1984). "Sparse Matrix Technology," Academic Press, New York.

Spedicato, E., ed. (1991). "Computer Algorithms for Solving Linear Algebraic Equations (the State of Art)," NATO ASI Series, Series F: Computer and Systems Sciences, Vol. 77, Springer-Verlag, Berlin, 1991.

Trefethen, L. N., and Bau III, D. (1997). "Numerical Linear Algebra," SIAM, Philadelphia.

Winter Althaus, G., and Spedicato, E., eds. (1998). "Algorithms for Large Scale Linear Algebriac Systems: Applications in Science and Engineering," NATO Advanced Science Institute, Kluwer Academic, Dordrecht, The Netherlands.

Project Management Software

Robert T. Hughes
University of Brighton

GLOSSARY

Activity network (AN) A diagram showing the activities required to complete a project and the dependences among activities that will govern the sequence in which they can be undertaken.
Function points A method of assessing the theoretical size of a computer-based information system by counting a number of different types of externally apparent features, applying weights to the counts, and aggregating them.
Product breakdown structure (PBS) A representation of the products that are to be created by a project.
Product flow diagram (PFD) A diagram indicating the order in which the products of a project must be created because of the technical dependences among them.
Project life cycle The sequence of phases needed to accomplish a particular project.
Risk exposure An indicator of the seriousness of a risk, based on consideration of the likelihood of the risk occurring and the extent of the damage if it did occur.
Risk reduction leverage An indicator of the usefulness of a proposed risk reduction activity in reducing risk exposure.
Software breakage Changes that have to be made to the existing functionality of a software component in order to accommodate the addition of new features.
Software development life cycle (SDLC) The sequence of generic activities needed for software to be created according to a particular development method.
Work breakdown structure (WBS) A representation of the activities that will need to be undertaken to complete a project.

SOME PRINCIPLES OF MANAGEMENT are applicable to projects of all kinds. A convergence of opinion on these principles is reflected in, among other things,

the documentation of a "body of knowledge" (BOK) relating to project management produced by the Project Management Institute (PMI) in the United States. Comparable BOKs have also been published in many other countries, including Australia and the United Kingdom. The applicability of these principles to software development projects is illustrated by the integration by the software engineering community in the United States of the PMI BOK into the proposed Software Engineering Body of Knowledge. The PMI, for its part, recognizes that the core practices and techniques of project management may need to be extended to deal with specialist projects in particular environments. As software development projects are notoriously prone to delay and cost overruns, the question therefore arises of whether, because of their inherent risks, they require specialist software project management techniques. In an influential paper, Brooks (1987) argued that the products of software engineering differ from the products of most other engineering disciplines in their inherent complexity, conformity, changeability, and invisibility. Software project management can therefore be seen as extending the range of techniques offered by generic project management to deal with the particular difficulties in software development.

Software products are more complex than other engineered artifacts in relation to the effort expended on their creation and hence their cost. In part, this is because software tends to be composed of unique components. Clearly software products can be reproduced, as when millions of copies of an operating system are produced and sold, but the focus of software development (as with the authorship of books) is on the development of the initial unique product. The inherent complexity of software crucially affects such aspects of development as the definition of requirements and the testing of completed products.

The problem of conformity centers on the need for software to reflect the requirements of human beings and their institutions. Brooks pointed out that while the physicist has to deal with complexity, the physical world does seem to conform to consistent rules, while human institutions can often promulgate inconsistent, ambiguous, and arbitrary rules for which the software engineer has to make provision.

In most fields of engineering, it is commonly accepted that once a product is built, changes to it are likely to be at least expensive to implement and might in extreme cases be technically impossible. Having built a road, for example, one would not expect that a change in the routing of the road would be undertaken lightly. Yet with software products, changes in functionality are expected to be incorporated without undue difficulty. Paradoxically, it is often the most successful software products that are most vulnerable to the demands for modification, as users try to extend the use of the software to new areas of application.

Compared to the products of, say, civil engineering, the products of software engineering tend not to be directly visible. The progress that is being made in the construction of a road is plain for all to see, while the appearance of an office where software is being developed does not change regardless of the stage of the project that has been reached.

All these factors, it is argued, call special attention to particular aspects of project management. We suggest that software project management has to respond to the following specific demands.

- The need, because of the "invisible" nature of software and hence the difficulty of tracking of the activities which create software, to make the products of the software process visible and then to control the software process by tracking these now-visible intermediate and deliverable products. The visibility of the products of thought processes is commonly achieved through the use of modeling techniques, such as those developed for structured analysis and design and the object-oriented approach.
- The need to map the phases of particular software development life cycles onto the stages of a managed project.
- The organization of teams of specialists to design and implement the software.
- The requirement for a configuration management regime that will ensure that the products of the software development process are kept in step, so that the components of a software application are compatible and consistent and generate desired outcomes. The configuration management system needs to be complemented by an effective change control mechanism.
- The need to consider the risks inherent in the development of software that can threaten the successful completion of a project.
- The need for techniques that can assess the "size" of the often intangible products of software development as a basis for projecting the likely cost of creating those products.
- The selection of the most appropriate development methods so that the quality attributes of software, especially in safety critical applications, can be assured.

These topics will be addressed in more detail in the following sections.

I. PRODUCT-BASED PROJECT MANAGEMENT

A. Product Breakdown Structures

The design of software is primarily a mental activity. It is also largely iterative in nature. One consequence of this is that it is difficult to judge, as a development activity takes place, the time that will be required to complete the task. One symptom of this is the "90% completion syndrome," where an activity is reported as being on schedule at the end of each week, say, until the planned completion date is reached, at which point it is reported as being "90% complete" for however remaining weeks are actually required for the accomplishment of the task. The first step in avoiding this is to commence project planning with the creation of a product breakdown structure (PBS) identifying all the products that the project is to create.

The products identified may be technical ones, such as software components and their supporting documentation, management products, such as plans and reports of various types, or items that are the by-products of processes to ensure software quality, such as testing scripts. The products may be ones that will be delivered to the customer (deliverables) or intermediate products created at various stages of the project for internal purposes, for example, to clarify and communicate design decisions between members of the development team. The concept of a product is a broad one, so that a person could be a product (e.g., "a trained user") or the product could be a revised version of a previously existing product (e.g., "a tested program").

A simplistic example of how a fragment of a PBS might appear is shown in Fig. 1. In the hierarchy a higher level box assigns a name to the grouping of lower level products that belong to it, but does not itself add any new products.

The PBS may be compared with the work breakdown structure (WBS), which is the more traditional way of decomposing a project. Some advantages of the PBS over the WBS have been touched upon above, but it should be noted that the PBS is easier to check for completeness than a WBS, as the customer might, for example, be able to identify a deliverable that they will need (such as user manuals) even if they are not conversant with a particular development methodology. As each product will need a task or tasks to create it, there would clearly be a close mapping between the PBS and WBS for a project. Indeed, a composite structure is sometimes used which at the higher levels is a PBS, but where at the lower levels the tasks needed to create each atomic product are identified.

B. Product Flow Diagrams

Having created a PBS, it is possible to draw a product flow diagram (PFD) which portrays the sequence in which products have to be created. An example of a PFD is shown in Fig. 2, which corresponds to and develops the PBS in Fig. 1. The "flow" of dependences is primarily from top to bottom and from left to right. The PFD should not have any backward flows to indicate iterations. This is because the assumption is that a jump back to a preceding product can be made at any point—for example, the examination of a test completion report might trigger an amendment to a design document, which would lead to the re-creation of this document and generate new versions of the products that are dependent on the design document.

The PFD can be used to identify the details of the methodology or methodologies that are employed on the project. Each methodology, including those implied by various structured and object-oriented approaches to system development, will have associated with it a particular generic PFD.

C. Activity Networks

As each identified product will need activities to create it, the PFD that has been produced can be used to create a first-cut activity network. This shows the activities that

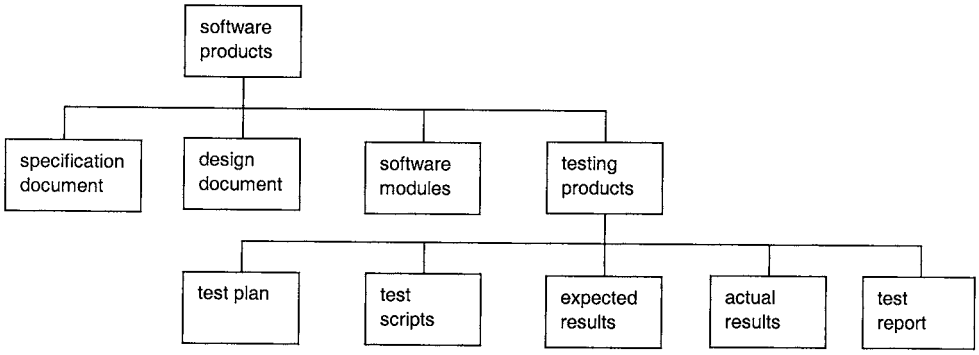


FIGURE 1 Example of a product breakdown structure.

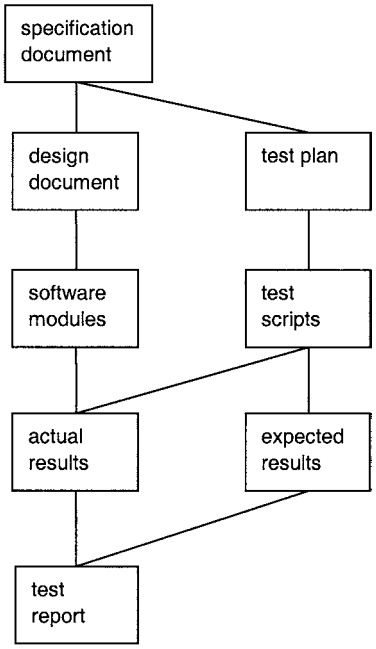


FIGURE 2 Example of a product flow diagram.

are required to accomplish the project and the sequence in which they have to be carried out. The mathematical properties of such networks are well established, and when durations for each of the activities are allocated, techniques exist which can calculate the overall time duration of the project and the latest time each of the activities can start without the overall duration of the project being affected. There are a number of software tools available, such as Microsoft Project, which permit the entry of activity network details and their subsequent analysis. The detail of such analyses is outside the scope of this article, but attention is drawn to the fact that the mathematics of activity networks depends upon the assumption that each activity is only executed once. Where activities of relatively short duration can be iterated, as in the case of software testing and correction cycles, then this can be dealt with by hiding the repeating activity within some higher level activity which has a fixed end point. If it is a major activity that is iterated, as when successive versions of a prototype are produced and exercised, then each iteration can be treated as an individual activity or even as an individual project.

To make the control of projects easier, it is convenient to have component activities that are of about the same duration. Where a large number of very small, but important activities have been identified, these can be bundled into a larger, umbrella activity. Where there is a very large activity ("write code" is often an example), then an attempt should be made to decompose the activity, either by identifying components of the activity's main product or by identifying intermediate stages in the activity which can be promoted to activities in their own right.

The product-based approach to the planning of software development projects is advocated as it means that the subsequent control of the project can be accomplished by identifying which products have been successfully completed. The generation of products also allows the quality control of intermediate products to be carried out, which contributes to the early elimination of system defects. Where activities are large and monolithic, the product-based approach encourages the decomposition of an activity into smaller ones, each of which should produce its own intermediate or component products which can then be subjected to control and quality checking.

II. SOFTWARE DEVELOPMENT LIFE CYCLES

A. The Need for Defined Development Life Cycles

Since the early 1960s, when management started to become aware of the potential applications of electronic computers, there has been a rapidly increasing demand for software. In the early days, the techniques needed to develop good software were not well defined and the newness of the field also meant that projects were inevitably staffed by the largely inexperienced. Overambitious schemes on the one hand and relative incompetence on the other led to many failed projects. One of the solutions to these problems was seen to be development of structured methods, step-by-step procedures for the design and construction of software which, if followed, would ensure good software products delivered on time. As a consequence, a number of different software design and implementation methods have been devised. The features, merits, and disadvantages of various approaches are discussed elsewhere, but it has already been noted that it is possible to draw up at least one PBS and PFD for each of these methods. Thus, for each method, it should be possible to designate a development life cycle showing the sequence in which activities are carried out and the scope that exists for iteration. Within these frameworks, project managers need to know when specific activities will start and end and be reassured that an activity, once completed, will not be repeated. This may conflict with the need for developers to repeat processes in order to discard imperfect designs and to perfect promising ones.

For the purposes of management control, managers will also want to group activities into phases of some kind, each of which will terminate with a milestone. A milestone is

an event which is selected as requiring special attention from the project and business management functions.

B. The Waterfall Model

The classic engineering approach which encapsulates the manager's viewpoint is the waterfall model. An example of a project life cycle that conforms to this model is shown in Fig. 3. Each activity can, at least in theory, only be legitimately started when its predecessor has been completed. The analogy is to a stream of water descending a mountain-side which drops from one pool to another with no return. There is some scope for splashing back, for feedback from a later stage to an earlier one. For example, consideration of systems design might unearth further user requirements; however, such feedback is generally regarded as something of an aberration.

It is important to stress that the waterfall model does not imply that it is either possible or desirable to attempt to plan every stage of a project in detail at the outset. An obvious example of the difficulties that this would entail is with the actual coding of software, where work can only be allocated to individual members of the development team when the structure of the components of the software has emerged from design.

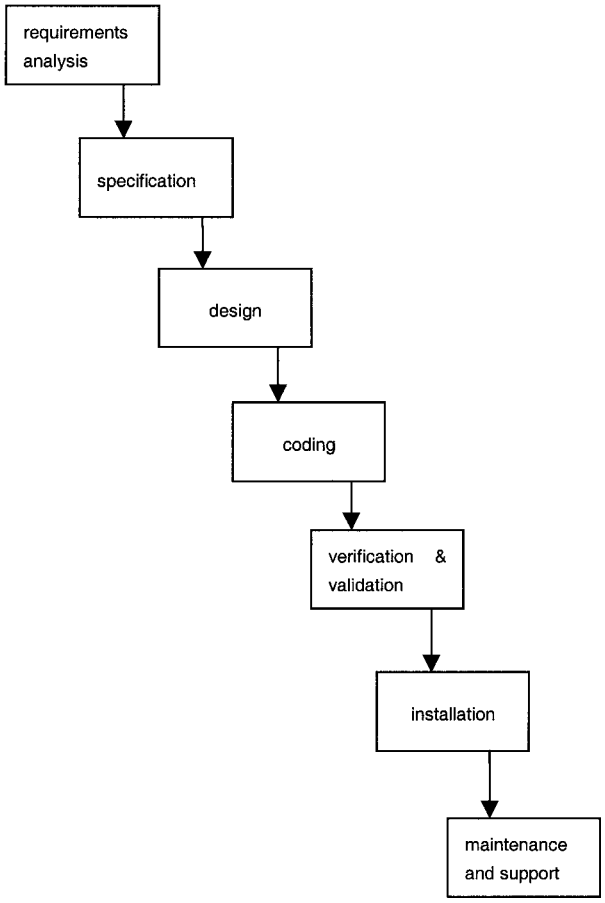


FIGURE 3 The "waterfall" project life cycle.

The advantage of the waterfall approach is that, if all goes well, management control is facilitated: once a stage has been completed, it should remain completed and all involved in the project should know where they stand. However, it is argued that the emphasis of the waterfall approach on designing everything just once encourages the initiation of projects that are large and monolithic in nature. With large projects there is a danger that by the time the project is completed, the world has moved on and the original requirements have changed. The need for a faster response to user requirements has encouraged a more incremental approach to the delivery of software.

C. The Incremental Model

The incremental approach is based on the principle that those involved in a project should at the outset focus on the key business objectives that the project is to achieve and be willing to suspend detailed consideration of the minutiae of a selected solution. This willingness to suspend consideration of the detail of a solution does not mean that the need for a proof of concept is ignored; the general feasibility of the overall project should have been established, even if the details of all the requirements have not. This initial focus on the overall desired outcomes, plus the adoption of software and hardware platforms that allow applications to be easily modified and extended, enables the partition of the project into a number of increments, each of which will deliver some benefit to the customer for the software. Although it may appear that many projects at first sight may not be amenable to this kind of decomposition, it is suggested that with practice it is usually possible to identify in the region of 20–100 such increments. Increasingly, software projects involve the extension or replacement of an existing software application and this means that very often the functions of the old system can be replaced segment by segment.

A major advantage of this approach is the ability to deliver a subset of the customers' requirements, preferably the most valuable ones, at an early stage. The fact that benefits of the development are felt earlier in the project may mean that income (or savings) may be generated earlier and thus improve an organization's cash flow. A further advantage is that technical risks may be reduced as the knowledge gleaned from earlier increments can be used to plan and deliver later stages with more certainty.

The incremental approach also facilitates "time boxing," the use of a fixed time period for each increment to

dictate the amount of functionality to be delivered: if time runs out, rather than extending the deadline, the delivery of some of the less urgent functionality is deferred until a later increment.

The incremental approach is not without possible disadvantages. Each increment can be perceived as a small project in its own right. Projects have startup costs and there is a risk that the incremental approach can lead to a reduction in productivity because of the loss of economies of scale. There is also the risk of "software breakage": because of the lack of an overall detailed design, later increments may require the software written for earlier increments to be rewritten so that it is compatible with this later functionality. The occurrence of software breakage may be a symptom of a deeper problem: the piecemeal development of the overall application may lead to insufficient effort being given to the creation of a robust unifying architecture. The focus on a succession of individual increments, especially where the minds of developers are being concentrated by the tight deadlines implied by time boxing, might mean that opportunities for different subsystems to share code may be overlooked. There is also a risk that the short-term nature of the concerns of the developers of increments leads to the neglect of longer term concerns relating to maintainability.

D. Evolutionary Models

Although it has been suggested that all the details of every stage of a project planned using the waterfall model do not have to be determined before the start of the project, managers usually need to provide the client with a clear idea of when the project will be completed. The history of computer disasters indicates this is often problematic. A large part of the difficulty stems from the uncertainties that are inherent in the environments of many projects. These uncertainties might reside in the nature of the customer requirements or in the nature of the technical platform that is to be used to deliver functionality. The rational approach to the reduction of such uncertainty is to commit resources to buying knowledge through trial and experiment, usually by building prototypes. Prototypes, working models of the software, can be used to explore the nature of user requirements, or to trial alternative ways of using technical platforms. Prototypes can also be used to assess the impact of the adoption of a particular information technology (IT) solution on the operations of the host organization.

A prototype can be a "throwaway," in that, once the lessons learnt have been documented, it is discarded and development starts afresh. Alternatively, a prototype can be "evolutionary," and gradually modified until it is transformed into the final operational product.

In the case of prototypes that are designed to elicit and elucidate requirements, two major types can be distinguished. In some cases there are uncertainties about the way that the core software functions should operate. For example, there might be a requirement for some software to simulate some phenomenon in the real world. The algorithms that execute the simulation would normally be based on some preexisting theory, but might then be adjusted after comparing the results of the model with the corresponding outcomes in the real world. In other applications, the internal functionality that the software is to have is well defined (by, for example, established business and accounting practices), but the interface between the users and the software may need to be the subject of prototyping.

E. Rapid Application Development

The development of structured approaches has, in conjunction with the discipline imposed by the waterfall model, brought some order to software development. However, these approaches also have costs: order is imposed by introducing bureaucracy. This can stifle development by putting an excessive focus on the next step in a procedure rather than on the larger goals to be achieved and by slowing down communication by insisting on documentation and formal meetings.

Approaches like rapid application development (RAD) and the dynamic systems development method (DSDM) try to overcome these obstacles by exploiting the opportunities offered by incremental development and prototyping. Development is carried out as far as possible in small, discrete, incremental steps which last for only a few weeks. Software products are produced by small, empowered teams of developers and user representatives so that agreement over requirements and design decisions can be quickly reached through intensive communication. Delivery on time is facilitated by the use of time boxes. These approaches are not without risk, as noted in the section on the incremental approach. In particular, there is the risk that the emphasis on very focused incremental developments can result in an overall architecture that is not maintainable in the longer term. It could be that the adoption of a strong object-oriented technique with some carefully considered global design may be able to alleviate some of the risks.

RAD, DSDM, and associated development regimes would seem to require more project management rather than less. For example, configuration management becomes even more important as detailed requirements and design will be volatile and may even be reversed on occasion. The time-boxing approach may mean that requirements may suddenly be deferred to later increments, making plans subject to frequent modifications.

F. The Spiral Model

This development model is based on the idea that when risks are prominent, the prudent way to proceed is by a series of iterations where each cycle involves carrying out activities that explore the problem domain and or develop potential solutions in more detail. Each iteration is completed by a control process where the desirability of the next iteration is considered. Each iteration is more detailed and involves a bigger commitment of resources, but should have an increased probability of a successful overall outcome. This model is compatible with a waterfall model if provision is made at the end of each step in the waterfall for the consideration of the advisability of either continuing the project or abandoning it.

III. SOFTWARE PROJECT ORGANIZATION

A. Relationship between Client and Developer

Most software development projects are commissioned by a single client who requires a software application. There may be cases where there is no client as such, as when software is being developed speculatively to meet what has been perceived as a need in the market. Even in this latter case, the developers will need some surrogate users, perhaps from the marketing function, to assist in deciding what functionality the software should have.

In the situation where there is a clearly defined client who has sponsored the project, development could be carried out in-house by staff employed by the same organization as the client or could be carried out by a supplier from outside. With both types of structure there is likely to be a range of different stakeholders, both individual and group, who have a legitimate interest in the project. This may be because they will be affected by the operation of the new system once implemented, or because their assistance will be needed for the development or implementation of the application. With in-house projects these stakeholders can be categorized as business managers who will be concerned that the application should further the business's objectives, users who will have to interact with the operational system, and various people, not just technical staff, who will be needed to develop the application.

With all these different voices wanting to be heard, a project, to be successful, needs a single, unified project authority which establishes and adapts as necessary the project's objectives and authorizes expenditure on it. The project authority could be one person, but in more complex situations may be a steering committee or project board with representatives of the major stakeholders. This project board does not deal with the day-to-day running of the project: this is delegated to a project manager who reports to the project board.

Where software is being supplied by an external organization, project management will be split into customer and supplier elements. In these cases, there may be effectively two managers. One would belong to the customer and would have responsibility for supervising the fulfillment of the contractors' obligations and also for coordinating those elements of the project for which the customer has responsibility, of which the populating of databases with standing information would be a typical example. On the supplier side, there would be a project manager coordinating the technical activities needed to deliver to the customer those products specified in the contract. The executives negotiating such contract need to make sure that the relationship between the two sides during the project is agreed and understood by all concerned at the outset. Projects to develop software almost invariably involve nonsoftware elements and so there should be a clear allocation of responsibility between customer and supplier for each necessary activity.

B. Creating the Project Team

Projects, by definition, are unique, time-bound enterprises. The project typically brings together practitioners who are experts in different fields for the duration of the project and then disperses them once the project is done. To do this, staff might be brought together physically or may be left in dispersed locations with channels of communication established between them. Some enterprises adopt a functional organization where practitioners are grouped physically by their specialties. For example, there might be a central pool of code developers. This can lead to the more effective use of staff as specialists can be more easily switched between projects to cope with peaks and troughs in demand. In the longer term, it allows specialists to further their careers without having to move out of a technical specialty at which they might be good. Innovative technical ideas might also be able to circulate more speedily between the specialists in this type of environment.

Disadvantages of the functional organization are that there can be communication barriers between the different specialists involved in project, and software maintenance may be hampered by there not being code developers dedicated to the enhancement and perfection of specific components in the organization's inventory of software. One way of resolving the latter problem is by having specialist teams of maintenance programmers who are separate from the initial development teams.

Some organizations have attempted to have both a task-based and functional organization at the same time by means of a matrix structure. Here a developer has two

managers: a project leader who gives day-to-day direction about the work in hand, and a specialist manager concerned with such matters as technical training needs.

A major problem with software development continues to be the dearth of good software developers. Many observers have noted the wide difference in productivity between the most and the least capable developers. It has been suggested that the best way of achieving success with a software project is to ensure that the best staff are hired and that then they are used to best effect. This train of thought leads to the chief programmer team. The chief programmer is a highly talented and rewarded developer, who designs and codes software. They are supported by their own team designed to maximize the chief programmer's personal effectiveness. There is a "co-pilot" with whom the chief programmer can discuss problems and who writes some of the code. There is an editor to write up formally the documentation sketched out by the chief programmer and a program clerk to maintain the actual code and a separate tester. The general idea is that the team is under the control of a single unifying intellect. The major problem with this strategy is the difficulty of obtaining and retaining the really outstanding software engineers to fulfil the role of chief programmer.

A more practical and widespread approach is to have small groups of programmers under the leadership of senior programmers. Within these groups there is free communication and a practice of reviewing each other's work.

The structures above assume that developers work in isolation from users and other, more business-orientated analysis specialists. The adoption of a rapid application development strategy would require this approach to be radically rethought.

IV. CONFIGURATION MANAGEMENT

The innate qualities of invisibility, changeability, and complexity that software possesses can create problems of ensuring that all the components of a software system are up to date and compatible. The technical discipline that is designed to ensure that the right versions of each component are assembled to create the correct working product is known as configuration management. The growing prevalence of development regimes that favor the creation of increments or successive generations of prototypes makes the need for effective configuration management even more pressing.

Particularly problematic from the point of view of configuration management is the situation where there are not only different consecutive versions of a software product over time as enhancements are made, but also different concurrent versions ("variants") of the same base software. These variants are typically to cater for different national or market sectors.

Effective configuration management depends on a person or persons being allocated the responsibility for this activity. The products that need to be controlled should be identified as configuration items (CIs) and their details should be recorded in a configuration management database (CMDB). The CMDB record will include, among other things, the status of the CI (for example, whether under development, under test, or released for operation) and, where appropriate, the developer currently carrying out any changes to it. This avoids situations where more than one developer might inadvertently be making potentially incompatible changes to the same software. The set of components that make up an application that is released to the users needs to be "baselined." This is the freezing of development on this set of components. Further development of any of these components effectively constitutes the creation of a new, different product.

The essentially technical discipline of configuration management is associated with the managerial and business concerns of change control. Although it is important for developers to satisfy user needs and to recognize that the precise nature of those needs is likely to emerge incrementally as the project progresses, an uncritical blanket acceptance of all requests for changes would lead to an uncontrollable project. There are risks that, without the careful control of changes, the scope of the project will grow, existing work will be wasted, costs will increase, and time scales will be extended. Projects are normally based on a business case where the benefits of the project have been judged to exceed the costs. The constraints on changes should not be so tight that there is an insistence on producing software in a previously agreed form when it is now clear that the benefits originally envisaged will not be reaped. On the other hand, unnecessary changes which could increase costs so that they exceed the value of the projected benefits need to be resisted.

The execution of a properly managed change control procedure is normally initiated via a mechanism for requesting a change through the raising of a request for change (RFC) document. The RFC represents a user consensus that the change is desirable from their point of view. The technical and resource implications of the change envisaged in the RFC then need to be assessed by developers. Where the software development is being carried out for a customer by a separate organization and is subject to a contract, any additional work above that specified in the contract would attract additional payments. In major projects, arrangements may be put in place for an independent evaluation of the impact of a change so that an unscrupulous supplier is not able to take advantage of a customer who may not have any real choice but to go

ahead with a change. Once the cost of the change has been assessed, then a decision is needed on going ahead with the change. A positive decision would lead to an authorization for work to proceed and for copies of the baselined products affected to be released to developers assigned to making the changes.

V. RISK MANAGEMENT AND SOFTWARE DEVELOPMENT PROJECTS

A. The Nature of Risk

The particular characteristics of software that were identified by Brooks and which were reviewed in the introduction suggest that software development projects are likely to be particularly risky. Repeatedly surveys of software projects have confirmed this: in one case a survey of 200 IT organizations found 90% of IT projects were overbudget; 98% of the organizations had had to modify the original requirements; 60% of the projects were late and 20% were eventually found to be inappropriate. It should therefore follow that the management of risk ought to be an integral part of software and IT project management. However, the same survey found that only 30% of organizations in the survey carried out any kind of analysis of risk.

Risk has always been a feature of projects in more traditional engineering fields and generic approaches to project risk have been developed. Software engineering thinkers, such as Barry Boehm, have considered how these generic principles can be applied to software development.

One dictionary definition of risk is the "possibility of loss or injury." Another way of perceiving risk is as the possibility that an assumption underlying a planned project is in fact incorrect. When this happens then the project manager has a problem. Obviously problems can occur at any time and at any level of seriousness. For example, a project might be completed on time, within budget, and to specification but might still be deemed a failure because the business environment within which it was to operate has changed so that the new software is no longer of use. Apart from these business risks, there are other risks that can apply to the project as a whole. Many of these will arise because the situation in which the project is to take place has some inherent dangers. These types of risk are susceptible to identification at the planning stage, allowing a strategy to be adopted which is likely to reduce risks arising from situational factors. An incremental delivery strategy might, for example, reduce the risks involved in the development of large, complex systems, while an evolutionary approach could reduce uncertainties in requirements or the technical platform. Other risks will be attached to individual tasks within the project. Here the planning process will have to identify first these activities and then the risks associated with each activity.

B. Prioritizing Risk

If a risk is simply a possible problem, then the number of risks that might affect a project is infinite. The size of each risk (or "risk exposure") therefore needs to be assessed so that attention can be focused on the most important ones. This can be done by estimating the probability of the unsatisfactory outcome, $P(\mathrm{UO})$, and the loss, $L(\mathrm{UO})$, if this unsatisfactory outcome were to occur. $P(\mathrm{UO})$ will be a number with the value of 0.00 if the outcome were impossible (i.e., there is in fact no risk) and of 1.00 for an absolute certainty. A value of 0.20 would imply that this risk is likely to occur in one of five projects of this type. $L(\mathrm{UO})$ would typically be measured in money, although lost time might be an alternative measure. The risk exposure (RE) is then calculated as $P(\mathrm{UO}) \times L(\mathrm{UO})$. Thus if the probability of failure were 0.20 and the potential loss were \$10,000, the RE would be \$2000. One way of looking at RE is as the amount that should be put aside to deal with this risk. Simplistically, if a similar project were repeated 10 times and \$2000 for each of the projects were put aside to cover this risk, one might expect, on average, the risk to occur on two of the occasions and use up the money that had been held in reserve. (Obviously in practice the same project is never repeated five times! However, a project will have several risks and money put aside for a risk that has not materialized can be used to alleviate one that has). In practice, allocating a $P(\mathrm{UO})$ and an $L(\mathrm{UO})$ for a risk might not be easy. $P(\mathrm{UO})$ can only be realistically assessed by examining a history of past cases, while the actual loss that an unsatisfactory outcome might cause will depend on particular circumstances. Because of these difficulties, a more qualitative approach where $P(\mathrm{UO})$ and $L(\mathrm{UO})$ for a risk are rated as high, medium, or low is often adopted.

An instance of a software project can normally be categorized as belonging to a certain type that tends to employ common techniques and methods. Many risks for projects of a particular type will therefore be similar, allowing generic risks of the most likely hazards to be assembled. For example, Conrow and Shishido devised the list shown in Table I, which is an aggregation and summary of 150 candidate risk issues that they have identified.

These generic risks will not threaten all software projects to the same degree, but such lists can be used as a memory prompt suggesting the possible hazards that may face a specific project. The importance of each of these possible risks for a project can be prioritized by assessing the risk exposure (RE) for each one in the way described above.

TABLE I A Summary of Key Risk Issues[a]

Risk grouping	Software risk issues
Project level	Excessive, immature, unrealistic, or unstable requirements
	Lack of user involvement
	Underestimation of project complexity or dynamic nature
Project attributes	Performance shortfalls (includes errors and quality)
	Unrealistic cost or schedule
Management	Ineffective project management (multiple levels possible)
Engineering	Ineffective integration, assembly and test, quality control, speciality engineering, or systems engineering
	Unanticipated difficulties associated with the user interface
Work environment	Immature or untried design, process, or technologies selected
	Inadequate work plans or configuration control
	Inappropriate methods or tool selection or inaccurate metrics
	Poor training
Other	Inadequate or excessive documentation or review process
	Legal or contractual issues (such as litigation, malpractice, ownership)
	Obsolescence (includes excessive schedule length)
	Unanticipated difficulties with subcontracted items
	Unanticipated maintenance and/or support costs

[a] From Conrow, E. H., and Shishido, P. S. (1997). "Implementing risk management on software intensive projects." *IEEE Software* **14**(3) (May/June), 83–89.

C. Planning Risk Avoidance or Reduction

Activities that might avoid the most likely risks occurring can now be planned. Thus the risk that there might be problems caused by the unfamiliarity of the development staff with a certain software tool might be avoided, or at least reduced, by hiring experts in the use of the tool. These risk avoidance/reduction activities would require changes to project plans and would themselves need to be subjected to risk assessment. In the example where certain software tool specialists have been hired, the risk of overreliance on the experts who might then leave might need to be considered.

Risk reduction actions need to be cost effective. To assess whether this is the case, the risk reduction leverage can be calculated as $(RE_{before} - RE_{after})/(\text{cost of risk reduction})$, where RE_{before} is the risk exposure before the risk reduction action and RE_{after} is the risk exposure that will remain after the risk reduction. Both risk exposures are expressed in terms of money. A risk reduction leverage greater than 1.00 indicates that the avoidance/reduction activity is financially worthwhile. Risk reduction activities with values above but still close to 1.00 would need to be considered very carefully.

D. Monitoring Risk

Like most aspects of software project planning, the balance of risk will not remain static during the lifetime of a project. Some fears at the planning stage may turn out to be groundless, while new risks can emerge unexpectedly. Some risks related to specific events, the delivery of equipment, for example, will simply disappear because the activity has been successfully accomplished. Hence risks need to be carefully monitored throughout the execution of the project: one method of doing this is to maintain a project risk register, or inventory, which is reviewed and updated as part of the general project control process.

VI. SOFTWARE EFFORT ESTIMATION

A. A Taxonomy of Software Effort Estimation Methods

Both the invisibility and complexity of software identified by Brooks contribute to the difficulty of judging the effort needed to complete software development tasks. This, and the rapidly changing technical environment in which software development takes place, has led to the almost proverbial habit of software projects of being late and overbudget. This in turn has motivated considerable research effort into the problem.

Barry Boehm (1981), in the classic work in this area, "Software Engineering Economics," set out a basic taxonomy of estimating methods:

- *Algorithmic models.* These use "effort drivers" reflecting quantitative aspects of the software application to be built (such as the lines of code to be

written) and the development environment (such as the experience of the development team) to predict effort.

- *Expert judgment.* The advice of knowledgeable staff is solicited.
- *Analogy.* A previous completed project that seems to have similarities with the one under consideration is selected and the actual effort for that project is used as the basis for an estimate for the new project.
- "*Parkinson.*" The staff effort that is available to carry out the project is identified and this is used as the basis for an "estimate."
- *Price-to-win.* The "estimate" is a figure set at a level low enough to have a good chance of winning the contract.
- *Top-down.* An overall estimate of effort for the project as a whole is ascertained, often by means of an algorithmic model, and this figure is then divided up into the effort needed for lower level, component activities.
- *Bottom-up.* The project is iteratively decomposed into its component activities and the effort for each of these activities is then assessed and accumulated to obtain the overall estimated effort.

Two of these methods, Parkinson and price-to-win, are clearly not true effort estimation techniques, and are rejected by Boehm as such. They can, however, be seen as ways of setting management targets. Having produced a target effort figure based on the Parkinson approach, a manager might legitimately seek to reduce the scope of the requirements of the project so that they can be implemented with the resources to hand. This would be an example of the sound engineering practice of designing to cost.

A true estimate, rather than being a single value, should rather take the form of a probability graph where a probability is assigned to each value in a range of estimates. Such a probability curve could plausibly conform to a gamma distribution (see Fig. 4). A project manager using the effort estimation presented in the form of such a graph might decide to select a target which is aggressively low, but where there is a substantial risk that the target might be missed, or a larger effort figure which has a greater chance of actually being met. The manager's decision might well be influenced by the need to win work from a potential customer by a competitive price and by awareness that aggressive targets can motivate higher productivity from staff.

With the remaining, "true" methods there are numerous overlaps. Top-down approaches, for example, will often be based on algorithmic models. An "expert," when asked to produce an estimate of effort, might in fact use any method, including asking another expert. However, there is some evidence that experts tend to adopt an analogy approach. If a previous application that is a reasonable match

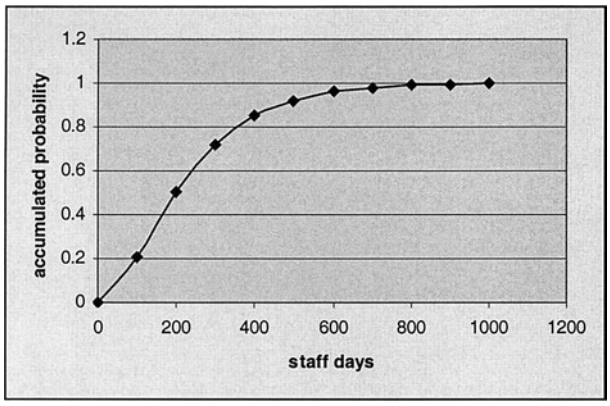


FIGURE 4 An accumulative probability curve showing the probability that a software component will be completed within different numbers of staff-days.

cannot be found, then analogies might be sought by informally breaking the application down into component parts and then seeking analogies for these components. Boehm drew the conclusion that the various methods were complementary, so that while algorithmic models could produce objective predictions of effort that were not subject to bias, expert judgment might be able to identify exceptional circumstances. The best practice was therefore to use the techniques in combination, compare their results, and analyze the differences among them.

B. COCOMO: An Example of an Algorithmic Model

Nearly all the techniques listed above, especially the algorithmic models, depend on some measure of software size. If the size of implemented software applications can be measured in some manner, for example, in lines of code, and the actual effort needed for these applications is also recorded, then a productivity rate can be derived as size/effort (e.g., as lines of code per day). If, for a new project, the probable size is known, then by applying the historical productivity rate an estimate of effort for the new project can be arrived at.

Early approaches to software effort estimation were based on this principle, a sophistication tending to be an adjustment by means of the application of exponentiation to take account of diseconomies of scale. These diseconomies can be caused by larger projects needing disproportionately more effort to deal with communication and management overheads.

Boehm's COCOMO (constructive cost model) illustrates the principle. In its basic, organic mode it is expressed as

$$pm = 3.2(kdsi)^{1.05}, \tag{1}$$

where *pm* is person-months and *kdsi* is thousands of delivered source code instructions.

TABLE II COCOMO 81 Cost Drivers

Product attributes	RELY	Required software reliability
	DATA	Database size
	CPLX	Product complexity
Computer attributes	TIME	Execution time constraints
	STOR	Main storage constraints
	VIRT	Virtual machine volatility—degree to which the operating system, etc., changes
	TURN	Computer turnaround time
Personnel attributes	ACAP	Analyst capability
	AEXP	Application experience
	PCAP	Programmer capability
	VEXP	Virtual machine (operating system, etc.) experience
	LEXP	Programming language experience
Project attributes	MODP	Use of modern programming practices
	TOOL	Use of software tools
	SCED	Required development schedule

A problem with this basic approach is that productivity rates vary considerably between development environments and indeed between projects within the same environment. One solution to this is to use statistical techniques to build local models. An alternative, favored by the COCOMO community, is to locate a development environment and application type within the totality of projects executed by the software development community along a number of dimensions. The dimensions as identified by the initial COCOMO are shown in Table II. Each point in this n-dimensional matrix would have associated with it an expected productivity rate relating to the software development industry as a whole. For example, an application might have a high reliability requirement compared to the nominal or industry-average type of project: this would justify 15% additional effort. However, the programmers involved might have higher than average capabilities, which would justify a reduction in the effort projected.

In addition to the need for calibration to deal with local circumstances, a problem with the COCOMO-type approach is that the number of lines of code that an application will require will be difficult to assess at the beginning of the software development life cycle and will only be known with certainty when the software has actually been coded. Lines of code are also difficult for the user community to grasp and thus validate.

C. Function Points

An alternative size measure, function points, was first suggested by Alan Albrecht. This measurement is based on counts of the features of a computer-based information system that are externally apparent. These include counts of the files that are maintained and accessed by the application ("logical internal files"), and files that are maintained by other applications but are accessed by the current application ("external interface files"). Three different types of function are also counted. These are transactions that take inputs and use them to update files ("external inputs"), transactions that report the contents of files ("external outputs"), and transactions that execute inquiries on the data that are held on files. The counts of the different types of feature are each weighted in accordance with the perceptions of the designers of function points of their relative importance. The particular weighting that applies is also governed by the perception of whether the instance of the feature is simple, average, or complex. These basic ideas and rules have been subsequently taken up and expanded by an International Function Point User Group (IFPUG).

An advantage of function points is that they can be counted at an earlier stage of a project than lines of code, that is, once the system requirements are determined. During the course of software development there is a tendency, known as "scope creep," for the requirements of the proposed system to increase in size. This is partly because the users are likely to identify new needs during the gathering of the details of the required features of the new system. These will clearly require additional effort for their implementation and are a frequent cause of cost and time overruns. Counting and recounting function points during the course of a project can help to keep this phenomenon under control. When the software application is being constructed for a client under a contract and extra features are required, then the original contract will need to be modified to take account of the extra work and cost to be incurred by the contractor. One option for dealing with this is to agree at the outset on a price per function

point for work in addition to that specified in the original specification.

Accurate effort estimates can only proceed on the basis of good historical information and so the accurate recording of details of project cost and size is one of the indicators of a well-managed project environment.

VII. SOFTWARE QUALITY ISSUES AND PROJECT MANAGEMENT

A. Software Product Quality

The quality of software products is seen as being increasingly important, especially as software is embedded into more and more safety-critical applications. As a consequence, attention has been devoted to ways of defining the quality requirements of software with the same rigor as has been previously given to the definition of the functions of the required software. The continuing development of the ISO 9126 standard is one example of this trend. This standard has defined five generic software quality characteristics in addition to functionality, namely reliability, usability, efficiency, maintainability, and portability. Despite the slogan that "quality is free," in practice the incorporation of, for example, high levels of reliability into software can add considerably to costs.

For this reason, the level of effort and expenditure that is invested in the quality of a software product must be justified by the use to which the software will be put. Where human life could be at risk, for instance, then high levels of expenditure on the quality of the software would be justified. These considerations will influence many of the decisions that have to be addressed at the stage when the project is planned. The qualities defined by such standards as ISO 9126 are those that are expected in the final delivered product and this places the emphasis on the evaluation of the completed product. From the point of view of the managers of the development process, this is too late: they need ways during the development process of assessing the likely quality of the final product. One way of doing this is by evaluating the quality of the *intermediate* products that are created during the course of a software development project.

B. Software Defect Accumulation

Software development can be usefully perceived as a chain of processes where the output from one process is the input to the next. Errors and defects can enter the development chain during any of the constituent activities. Once the defect has been introduced into the development chain it is likely to remain there if care is not taken. An error in the specification of a software product, for example, could feed through to its design and then into the way that it is coded. Furthermore, the defects in the developing application will, if left uncontrolled, accumulate as more processes add their own defects. One consequence of this will be that most of these defects will only become apparent at one of the final testing stages. This can be a serious problem for project management, as testing is one of the most difficult activities to control, driven as it is by the number of defects remaining in the software, which is an unknown quantity. It is also generally more expensive to correct errors at the later stages of development as more rework is needed and later products tend to be more detailed than earlier ones.

We have noted that an effective way to reduce these difficulties is by the careful examination of intermediate products before they are passed onto subsequent processes. To enable this to be done effectively, the following process requirements should be specified for each activity.

- *Entry requirements* which have to be satisfied before work on a process in relation to a product can be authorized. For example, it might be laid down that a comprehensive set of test data and expected results must have been prepared, checked, and approved before testing is permitted to commence.
- *Implementation requirements* which define how the process in question is to be carried out. In the case of testing, it could be stated that whenever errors are found and removed, all test runs must be repeated, even those that have previously been found to run correctly. The justification for this would be to ensure that the error corrections have not themselves introduced new errors.
- *Exit requirements* which have to be met before an activity can be signed off as being completed. Thus, for testing to be regarded as completed, a requirement might be specified that all test runs must have been run in sequence with no errors.

C. Reviews, Inspections, and Walkthroughs

The most common methods of checking the appropriateness of intermediate products are reviews, inspections, and walkthroughs. Although these are technically different, many practitioners use the terms interchangeably.

The IEEE (1997) has defined a review as "an evaluation of the software element(s) or project status to ascertain discrepancies from planned results and to recommend improvements. This evaluation follows a formal process." Broadly, technical reviews have the objective of examining the quality of products being created by a project, while

a management review focuses on the effectiveness of the processes.

A technical review is directed by a review leader who is familiar with the methods and technologies used on the product to be examined. The review leader must select reviewers and arrange a review meeting when the product is in a fit state for examination. In addition to the product, the objectives of the review, the specification which the product is fulfilling, and any relevant standards should be available. Reviewers should carefully examine these documents before the meeting. At the meeting itself defects found by the reviewers are recorded on a technical review issues list. The temptation to suggest ways of resolving the issues at this stage should be resisted. If a large number of defects is found, a second meeting may be needed to review the reworked product, but otherwise there should be a management arrangement to ensure that the technical issues noted by the review are dealt with.

Inspections are similar to reviews in principle, but the focus is on the scrutiny of a specific document. There must be one or more other documents against which it can be checked. A design document would, for example, need a specification with which it should be compatible. Inspections are associated with M. E. Fagan, who developed the approach at IBM. He drew particular attention to the conditions needed to make inspections effective. For instance, the defect detection rate was found to fall off if more than about 120 lines of a document was reviewed in 1 hr or the review took longer than 2 hr. Inspections can thus be very time-consuming, but Fagan was able to produce evidence that, despite this, the use of inspections could be massively cost-effective. Both reviews and inspections can be made more effective by the use of checklists of the most probable errors that are likely to occur in each type of software product.

Reviews, inspections, and other quality-enhancing techniques are well established but are potentially expensive. Project managers need to weigh the cost of conformance, that is, the cost of the measures taken to remove defects during development, against the cost of nonconformance, that is, the cost of remedying defects found during the final testing phase. There would also be costs associated with the potential damage an application could occasion if it were defective when in operation.

D. Quality Plans

Standard entry, exit, and implementation requirements could be documented in an organization's quality manual. Project planners would then select those standards that are appropriate for the current project and document their decisions in a software quality assurance plan for their project.

The ANSI/IEEE standard for software quality plans states that such plans should have the following sections.

1. Purpose. This section documents the purpose and scope of the plan. It identifies the software product(s) to which the plan relates.

2. Reference Documents. This section lists other documents that are referred to in the plan.

3. Management. This section describes the organization of the project in terms of who will be responsible for what. The tasks that the project will need to accomplish should be documented. This would seem to be an example of where a reference to another, existing planning document would be appropriate.

4. Documentation. The documentation that is to be produced or used by the project should be identified. This includes specifications, designs, testing plans and reports, and user manuals.

5. Standards, Practices and Conventions. These specify the standards that will be used during the project. These are often in the form of references to preexisting standards, either formulated externally or from the organization's own quality manual.

6. Reviews and Audits. This section lists the reviews and audits to be carried out along with their scheduled dates. Audits differ from reviews in that they tend to be performed retrospectively. For example, audits might be conducted on all the deliverables of a project immediately prior to hand-over to the customer.

7. Configuration Management. This has been touched upon in Section IV above.

8. Problem Reporting and Corrective Action. A familiar problem with management systems is that although problems and issues might be effectively and accurately identified and recorded, their resolution can sometimes escape proper control. Procedures need to be in place to ensure that the quality loop is complete so that defect reports are properly tracked to their final resolution.

9. Tools, Techniques, and Methodologies. The purpose and use of any special tools, techniques and methodologies employed should be described.

10. Code Control. This section describes the procedures and organization in place to maintain a correct and secure library of software and documentation.

11. Media Control. This section describes the measures that ensure the physical security of the products of the project, especially those that are in electronic format.

12. Supplier Control. Some elements of the software product might not be created by the internal team, but might be bought from subcontractors. The measures to ensure the quality of these bought-in components need to be planned and documented.

13. Records Collection, Maintenance, and Retention. The person or persons are named who are responsible for ensuring that the appropriate documentation is produced and stored and that the situations which should generate documentation are identified.

The core purpose of this document can thus be seen as ensuring that the actions needed to ensure quality are incorporated into the fabric of the project and executed as part of the day-to-day running of the project and not as some desirable add-on.

SEE ALSO THE FOLLOWING ARTICLES

COMPUTER ALGORITHMS • REQUIREMENTS ENGINEERING • SOFTWARE ENGINEERING • SOFTWARE MAINTENANCE AND EVOLUTION • SOFTWARE TESTING

BIBLIOGRAPHY

Brooks, F. (1987). "No silver bullet, essence and accidents of software engineering," *IEEE Computer* **20**(4), 10–19.

Brooks, F. P. (1995). "The Mythical Man-Month: Essays on Software Engineering (Anniversary Edition)," Addison-Wesley, Reading MA.

Boehm, B. (1981). "Software Engineering Economics," Prentice-Hall, Englewood Cliffs, NJ.

Hughes, B., and Cotterell, M. (1999). "Software Project Management," 2nd ed., McGraw-Hill, Maidenhead, U.K.

Humphrey, W. S. (1990). "Managing the Software Process," Addison-Wesley, Reading, MA.

IEEE. (1997). "Managing risk," *IEEE Software* **14**(3), 17–89.

Jones, C. (1998). "Estimating Software Costs," McGraw-Hill, New York.

Kemerer, C. F. (ed.). (1997). "Software Project Management: Readings and Cases," Irwin, Chicago.

Project Management Institute (1996). "A Guide to the Project Management Body of Knowledge," Project Management Institute, Upper Darby, PA.

Prolog Programming Language

Heimo H. Adelsberger
University of Essen

GLOSSARY

Backtracking If the basic control structure of Prolog (i.e., calling procedures and using clauses in a top-down, left-to-right fashion) leads to a goal that cannot be satisfied, Prolog goes back to a previous choice point with an unexplored alternative and moves forward again.

Clause Alternative formulation of statements expressed in first-order predicate calculus, having the form $q_1, q_2, \ldots q_m \leftarrow p_1, p_2, \ldots p_n$.

Constraint Logical relation among several variables, each taking a value in a given domain.

Definite clause grammar Formalism for describing languages, both natural and artificial. Definite clause grammars are translated into Prolog yielding a recursive-descent, top-down parser.

Horn clause Clause having one left-hand side literal at most. The right-hand side literals have to be positive. Prolog facts, rules, and goals are in the form of Horn clauses.

Logical variable Variable that stands for some definite but unidentified object like a pronoun in natural language.

Logic programming Representation of a program in the form of Horn clauses, and interpretation of these clauses in a declarative and in a procedural form.

Predicate calculus Formal language to express and reason with statements over a domain of discourse.

Resolution Computationally advantageous inference rule for proving theorems.

Unification Makes two terms identical by finding appropriate substitutions for variables. It is the basic parameter-passing method of Prolog, which may also be viewed as pattern matching using logical variables.

PROLOG is a computer programming language based on the ideas of logic programming. Logic programming, like functional programming, is radically different from conventional, imperative (or procedural) programming languages. Rather than mapping the von Neumann machine model into a programming language and prescribing how the computer has to solve the problem, logic programming is derived from an abstract model for describing the logical structure of a problem with no relationship to a machine model. Prolog describes objects and the relationships between them in a form close to mathematical logic. In this context, computation means the deduction of consequences from a program. Prolog manipulates pure symbols with no intrinsic meaning. Constraint logic programming extends the purely abstract logical framework with objects that have meaning in an application domain: numbers, along with their associated algebraic operations and relations.

I. BRIEF HISTORY OF PROLOG

As early as the 1950s, computationally inclined logicians were investigating techniques for automating proofs of mathematical theorems. Then, in the mid-1960s, J. A. Robinson formulated the resolution principle, a powerful method of drawing inferences that makes it easy to reason about facts and deduce new knowledge from old.

In the early 1970s the first Prolog interpreter was designed and implemented by A. Colmerauer at the University Aix-Marseilles, France. However, the use of Prolog was restricted primarily to a rather small academic community. R. A. Kowalski popularized the idea of logic programming—which is really what Prolog is about—with the publication of his book, "Logic for Problem Solving," in 1979. During the second half of the 1970s David H. D. Warren developed a very efficient Prolog compiler written almost entirely in Prolog. The Prolog syntax used for this project became a de facto standard called the Edinburgh syntax. Since the announcement of the Japanese Fifth Generation Computer Project in 1981, when Prolog was chosen as the kernel language, Prolog has been accepted as the second general artificial intelligence programming language (with LISP being the first).

Prolog was also influenced by the development of other artificial intelligence languages. Dissatisfaction with the theorem-proving techniques of the 1960s provided impetus for the development of the language Planner, which could explicitly represent assertions and theorems. Then Popler was developed, a Planner-type language for the POP-2 language environment. Subsequently, Conniver provided additional language features for expressing the order in which a proof of a theorem was to be attempted. Prolog then followed with its representation of theorems and axioms in the form of Horn clauses.

The founding work on constraint logic programming (CLP) was done at Monash University in Melbourne around 1987 by J. Jaffar and J. L. Lassez. In Europe, research was originally concentrated at the European Computer-Industry Research Center in Munich. CHIP (Constraint Handling in Prolog) is a result of this effort. A. Colmerauer contributed to this field by creating Prolog III. Eclipse, a language coming from IC Parc (Imperial College, London), shares many features with CHIP. It is written in itself, employs a powerful generalized propagation technique, and supports parallelism.

The crucial features of Prolog are unification and backtracking. Through unification, two arbitrary structures can be made equal, and Prolog processors employ a search strategy which tries to find a solution to a problem by backtracking to other paths if any one particular search comes to a dead end.

Since 1995, Prolog has been an international standard (ISO/IEC 13211).

II. APPLICATION AREAS

Prolog has been applied successfully for miscellaneous tasks in artificial intelligence, such as theorem proving, problem solving, mathematical logic, computer algebra, pattern matching, knowledge representation, and expert systems. Other major application areas are database management, software prototyping, compiler writing, computer-aided instruction, and design automation, including architecture, chemistry, and VLSI design.

Constraint programming has been successfully applied to problem areas as diverse as DNA structure analysis, time-tabling for hospitals, and industry scheduling. It is well adapted to solving real-life problems because many application domains evoke constraint description naturally. Some examples follow:

Assignment problems: Typical examples are (1) stand allocation for airports, where an aircraft must be parked on an available stand during its stay at the airport, (2) counter allocation for departure halls in airports, (3) berth allocation to ships in a harbor.

Personnel assignment where work rules and regulations impose difficult constraints: Examples include (1) production of rosters for nurses in hospitals, (2) crew assignments to flights, (3) staff assignments in railways.

Scheduling problems: Examples are (1) the petroleum industry, (2) forest treatment scheduling, (3) production scheduling in the plastic industry, (4) planning the production of military and business jets. The use of constraints

in advanced planning and scheduling systems is increased due to current trends of on-demand manufacturing.

Network management and configuration: Examples are (1) the planning of cabling of telecommunication networks in buildings, (2) electric power network reconfiguration for maintenance scheduling without disrupting customer services.

III. PROLOG'S ELEMENTARY CONSTRUCTS

Prolog is very different from traditional programming languages such as Ada, BASIC, COBOL, C, C++, FORTRAN, and Pascal. These languages are procedural, that is, a program specifies explicitly the steps that must be performed to reach a solution to the problem under consideration. Prolog is also different from functional programming languages such as pure LISP. A functional language is based on values and expressions, and computation means the evaluation of expressions until the final value is determined. Prolog is a declarative or descriptive language for describing the logical structure of the problem in a form close to mathematical logic. In Prolog, computation means the deduction of consequences from the program. Computer programming in Prolog consists of (1) declaring some facts about objects and their relationships, (2) defining rules about objects and their relationships, and (3) asking questions about objects and their relationships.

A. Facts

In a statement like "Mozart composed Don Giovanni" a relation ("composed") links two objects ("Mozart" and "Don Giovanni"). This is expressed in Prolog in the form of an assertion:

```
composed(mozart, don_giovanni).
```

A relationship such as "composed" is also called a predicate. Several facts together form a database.

Example. A database for operas

```
composed(beethoven,  fidelio).
composed(mozart,     don_giovanni).
composed(verdi,      rigolesso).
composed(verdi,      macbeth).
composed(verdi,      falstaff).
composed(rossini,    guillaume_tell).
composed(rossini,    il_barbiere_di_siviglia).
composed(paisiello,  il_barbiere_di_siviglia).
```

B. Questions

It is possible to ask questions in Prolog. The symbol used to indicate a question is ?-. There are two different types of questions: is-questions and which-questions. A typical is-question is "Did Mozart compose *Falstaff*?" In Prolog one would write

```
?- composed(mozart, falstaff).
```

Prolog's answer would be

```
no.
```

A typical which-question is "Who composed *Falstaff*?" In Prolog one would write

```
?- composed(X, falstaff).
```

Prolog's answer would be

```
X = verdi.
```

If there are more solutions to a question, as in "Which operas have been composed by Verdi?"

```
?- composed(verdi, X).
```

all solutions are presented one by one:

```
X = rigoletto;
X = macbeth;
X = falstaff;
no
```

1. Prolog Environment

Prolog waits after each solution for a user input.

A semicolon means: "Present one more solution." Hitting the return-key terminates the query. The "no" at the end indicates that there are no more solutions.

Actually, the top-level behavior of a Prolog system is not defined in the ISO standard. For the rest of this article we will use the following convention for better readability: if there are one or more solutions to a query, all solutions are listed, separated by semicolons; the last solution is terminated by a full stop. The values of the variables for one solution are separated by a comma. If there is no solution, this is indicated by the word "no."

The text of a Prolog program (e.g., the opera database) is normally created in a file or a number of files using one of the standard text editors. The Prolog interpreter can then be instructed to read in programs from these files; this is called consulting the file. Alternatively, the Prolog compiler can be used for compiling the file.

2. Closed-World Assumption

The answer to a query with respect to a program is a logical consequence of the program. Such consequences

are obtained by applying deduction rules. The assumption that all relevant knowledge is in the database is called the closed-world assumption. Under this assumption it is valid that Prolog comes back with "no" if a question cannot be answered.

C. Terms, Constants, Variables, and Structures

1. Terms

The data objects of Prolog are called terms. A term is a constant, a variable, or a compound term. A constant denotes an individual entity such as an integer or an atom, while a variable stands for a definite but unidentified object. A compound term describes a structured data object.

2. Constants

Atoms. An atom is a named symbolic entity. Any symbolic name can be used to represent an atom. If there is a possibility of confusion with other symbols, the symbol has to be enclosed in single quotes. The following are examples:

Quoted: `'Socrates' 'end of file' '('`
Unquoted: `composed mozart --> ?- :- **`

Strings. A string is formed by a sequence of characters enclosed between quotation characters, as in `` `this is a string" ``.

Numbers. An integer is represented by a sequence of digit characters, possibly preceded by a plus or minus; examples include the following:

```
123 -123 +123
```

A floating point number is represented by a sequence of digit characters, possibly preceded by a plus or minus and followed by a decimal part, or an exponent part, or a decimal part followed by an exponent part. Examples are

```
2.0   -2.718   5.5E8   -0.34e+8   45.0e-8
```

3. Variables

A variable name may contain letters, numbers, and the underscore character. It must start with an uppercase letter or the underscore character.

```
X Var Grand_dad X12 _abc _123 _
```

If a variable is only referred to once in a clause, it does not need to be named and may be written as an anonymous variable, indicated by the underscore character "_". A clause may contain several anonymous variables; they are all read and treated as distinct variables. For example,

```
composed(_, _).
```

A variable should be thought of as standing for some definite but unidentified object, which is analogous to the use of a pronoun in natural language.

4. Compound Terms

Compound terms are structured data objects. A structure consists of a name and a number of arguments, separated by commas, enclosed by parentheses. The name is an atom, called the principal functor. The number of arguments is called the arity of the structure. Each argument is a term. An atom can be considered to be a structure of arity zero.

Examples.

```
human(socrates)
father(Dad, Child)
line(point(10,10,20), point(X,Y,Z))
s(np(john), vp(v(likes), np(mary)))
s(np(dt(the), n(boy)), vp(v(kicked),
  np(dt(the),  n(ball))))
```

D. Rules

Rules enable us to define new relationships in terms of existing ones. For example, to explain that a person is someone's grandfather one could say

```
Grand_dad is a grandfather of  Grand_child
  if
    Child is parent of Grand_child
  and
    Grand_dad is father of Child.
```

In Prolog syntax one would write

```
grandfather_of(Grand_dad, Grand_child):-
  parent_of(Child, Grand_child),
  father_of(Grand_dad, Child).
```

The symbol ":-" is pronounced "if." The comma between "parent(...)" and "father(...)" is pronounced "and" and must not be confused with the comma between arguments of a structure. A rule has the form of an implication. The left-hand side of a rule, the conclusion, is called the rule-head; the right-hand side, the conjunction of conditions, is called the rule-body.

E. Conjunctions

Given the following database (comments in Prolog come in two forms: % up to end of line or `/*···*/`)

```
/* A small example about
   people, food, wine, and love */

likes(mary, food).         % mary
likes(mary, wine).
likes(peter, wine).        % peter
likes(peter, mary).
likes(paul, Something) :-  % paul
  likes(mary, Something).
```

in Prolog one could ask "Is there anything that Paul and Mary both like?" in the following form:

```
?- likes(mary, X), likes(paul, X).
```

It can easily be verified that only the object 'wine' satisfies the conditions. Therefore, Prolog's answer is

```
X = wine.
```

IV. BASIC CONCEPTS OF LOGIC PROGRAMMING

A. Nondeterminism

A very important aspect of logic programming is that, when several facts or rules match a given goal, the strategy by means of which these alternatives are tried is not determined. Sometimes this nondeterminism can be interpreted as a "don't care" nondeterminism.

This is the case if every answer is acceptable. More often it is a "don't know" nondeterminism. In the above example, "?-likes(mary, X), likes(paul, X)," the first subgoal, "?-likes(mary, X)," can be matched with two facts in the database: "likes (mary, food)" and "likes(mary, wine)." At this point we do not know which fact will lead to a solution.

Another form of nondeterminism is when several subgoals must be satisfied in a single goal statement. The order in which these subgoals are satisfied is also not determined. Again, the example "?-likes(mary, X), likes (paul, X)" can be used: it is logically equivalent to satisfying the subgoal "?-likes(mary, X)" first and then "likes (paul, X)" or vice versa or even both in parallel.

B. Backward Reasoning

Prolog deals with these forms of nondeterminism with a simple depth-first search strategy. The first subgoal, "likes(mary, X)," can be matched with the first fact in the database. This leads to the instantiation of X to "food." The second subgoal, "likes(paul, X)" cannot be satisfied for the following reason: X is instantiated to "food," to prove "likes(paul, food)" the rule "likes(paul, Something):-likes(peter, Something)" must be used, instantiating "Something" to "food." The statement "likes(paul, food)" is only true if "likes(peter, food)" is true. This is not the case, however, since "likes(peter, food)" is not in the database. Therefore, the subgoal fails. Prolog now backtracks, that is, it goes back to a previous choicepoint. This was the goal "likes(mary, X)." Prolog uninstantiates X and matches "likes(mary, X)" with the next appropriate fact in the database, that is, "likes(mary, wine)." This leads to the instantiation of X to wine. The second subgoal, "likes(paul, X)," can now be proven since X is instantiated to "wine," and "likes(paul, wine)" can be deduced from the fact "likes(peter, wine)," found in the database.

This form of reasoning is called *backward reasoning*: Prolog scans the database for matching facts and rule-heads. A matching fact indicates success. A matching rule-head reduces the problem to one of solving the conditions at the right-hand side of the rule. If several facts or rule-heads match a given goal, Prolog takes the first one from the database, then the second, and continues until all matching facts or rule-heads have been processed. A conjunction of goals is processed from left to right. The first goal of the conjunction is proved, then the second, and so on.

C. Unification and Logical Variables

The process of matching terms is called unification. The goal is to make two (or more) terms identical, replacing variables with terms as necessary. In the example above, the two terms `likes(mary, X)` and `likes(mary, food)` can be unified; the substitution is {`X=food`}. To make, for example, the terms

```
parents(peter, Y, Z)
parents(X, paul, Z)
```

equal, more than one set of possible substitutions exists. Possible substitutions are

```
{X=peter, Y=paul, Z=mary},
{X=peter, Y=paul, Z=elizabeth},
{X=peter, Y=paul, Z=xzvky} or
{X=peter, Y=paul}
```

to name only some. The last unifier, {`X=peter`, `Y=paul`}, is called a *most general unifier* (mgu) since it leads to the most general instance `parents(peter, paul, Z)`. Such an mgu is unique (up to renaming of variables). Variables in logic programs are different from variables in procedural programming languages in which they stand for a store location in memory. Variables in procedural programming languages can be changed using an assignment statement.

A logical variable, in contrast, has the following properties:

1. There is no explicit assignment (the system does it through unification).
2. Values of instantiated variables cannot be changed (except by the system through backtracking).
3. When two or more uninstantiated variables are matched together, they become linked as one.

D. First-Order Predicate Calculus

Predicate calculus is a formal language for expressing statements that are built from atomic formulas. Atomic formulas can be constants, variables, functions, and predicates. Atomic formulas can be connected to form statements by using the following connectives: $\vee$ (logical or), $\wedge$ (logical and), and $\Rightarrow$ (implication). The symbol $\sim$ (not) is used to negate a formula. Atomic formulas and their negations are called literals. Variables in a statement can be quantified by $\forall$ (the universal quantifier, "for all") and $\exists$ (the existential quantifier, "for at least one"). The syntactically correct expressions of the predicate calculus are called well-formed formulas (wff's). First-order predicate calculus is an important subset of predicate calculus. Statements are restricted in that quantification is not allowed over predicates or functions. For example, the sentence "Siblings are not married" can be represented by the first-order predicate calculus statement

$$(\forall X)(\forall Y)(\text{siblings}(X, Y) \Rightarrow \sim \text{married}(X, Y)).$$

E. Clauses

Clauses are alternative notations of statements expressed in first-order predicate calculus. They have the form of a logical implication. A clause is expressed as a pair of sets of terms in the form

$$q_1, q_2, \ldots q_m \leftarrow p_1, p_2, \ldots p_n.$$

Variables are considered to be universally quantified. The q_i are called the consequents and the p_j are called the antecedents. This can be read as "q_1 or q_2 or $\ldots q_m$ is implied by p_1 and p_2 and $\cdots p_n$." Clauses can also be written in form of a disjunction with the antecedents as negated literals

$$q_1 \vee q_2 \vee \ldots \vee q_m \vee \sim p_1 \vee \sim p_2 \vee \ldots \vee \sim p_n.$$

Three special clauses deserve attention. When the consequent is empty,

$$\leftarrow p_1, p_2, \ldots p_n$$

the clause is interpreted as a negation. Such a clause is also called a goal. When the antecedent is empty, as in

$$q_1, q_2, \ldots q_m \leftarrow,$$

the clause is called a fact, and it states unconditionally that q_1 or q_2 or $\ldots q_m$ is true. The empty clause

$$\leftarrow$$

is a contradiction and can never be satisfied.

F. Horn Clauses

Prolog's rules, facts, and queries are Horn clauses. Horn clauses, also called definite clauses, are a subset of the clausal form of logic. The consequent of the clause is restricted to one term at maximum:

$$\begin{array}{lll} q \leftarrow p_1, p_2, \ldots p_n & \text{a rule} & (n > 0) \\ q \leftarrow & \text{a fact} & \\ \leftarrow p_1, p_2, \ldots p_n & \text{a goal} & (n > 0). \end{array}$$

Horn clauses do not have the expressive power of the full clausal form of logic and, therefore do not have the power of first-order predicate calculus. The main disadvantage is that it is impossible to represent negative information like "Siblings are not married" with Horn clauses.

Horn-clause logic is known to be Turing-complete, that is, it provides a universal computing formalism. Horn clauses are closely related to conventional programming languages since they can be given a simple procedural interpretation.

There are two ways in which a Horn clause like

$$a \leftarrow b, c, d$$

can be interpreted:

1. *Declarative or descriptive interpretation*. This is a statement in logic saying "a is true if b and c and d are true."
2. *Procedural or prescriptive interpretation*. This is the definition of the procedure "In order to execute a, all procedures b, c, and d have to be executed."

The declarative aspect emphasizes the static knowledge that is represented. The procedural aspect emphasizes how to use the knowledge to solve the problem.

Example.

```
grandfather_of(Grand_dad, Grand_child) :-
  parent_of(Child, Grand_child),
  father_of(Grand_dad, Child).
```

Declarative interpretation. `Grand_dad` is the grandfather of `Grand_child` if (there exists `a Child` so that) `Child` is the parent of `Grand_child` and `Grand_dad` is the father of `Child.`

Procedural interpretation. There are many possible interpretations, for example, given `Grand_child`, seeking `Grand_dad:` To find the `Grand_dad` of a given `Grand_child`, find (compute) a parent of `Grand_child` and then find (compute) his/her father.

G. Invertibility

Prolog is different from most other languages according to the input–output behavior of arguments. A parameter in a procedural programming language is of type "in," "out," or "in–out." This means that either a value is passed to a subroutine, a result is returned from the subroutine, or a value is passed first and a result is returned afterward. In Prolog, the same argument of a predicate can be used for both input and output, depending on the intention of the user.

In a question like "?-grandfather(paul, peter)" both arguments are used as input parameters. In a question like "?-grandfather(paul, X)" the second argument may function as an output parameter, producing all grandchildren of Paul, if any. This aspect of Prolog, that an argument can be used sometimes as an input parameter but at other times as an output parameter, is called invertibility.

H. Resolution

Resolution is a computationally advantageous inference rule for proving theorems using the clausal form of logic. If two clauses have the same positive and negative literal (after applying appropriate substitutions as needed), as in $(p \vee q_1 \vee q_2 \vee \ldots q_m)$ and $(\sim p \vee \sim r_1 \vee r_2 \vee \ldots r_n)$, then

$$(q_1 \vee q_2 \vee \ldots q_m \vee r_1 \vee r_2 \vee \ldots r_n)$$

logically follows. This new clause is called the resolvent of the two parent clauses.

Resolution refutation is a proof by contradiction. To prove a theorem from a given set of consistent axioms, the negation of the theorem and the axioms are put in clausal form. Then resolution is used to find a contradiction. If a contradiction can be deduced (i.e., the negated theorem contradicts the initial set of axioms), the theorem logically follows from the axioms. An advantage of resolution refutation is that only one inference rule is used. A problem is combinatorial explosion: Many different candidates can be selected for resolution at each stage of the proof, and worse, each match may involve different substitutions. Using only the most general unifier eliminates one disadvantage mentioned above. Restricting clauses to Horn clauses that have only one positive literal drastically reduces the number of possible reduction candidates. The specific form of resolution used in Prolog systems is called SLD-resolution (Linear resolution with Selector function for Definite clauses). The most recent resolvent and a clause from the database, determined by a selection function, must always be used for each resolution step.

V. PROGRAMMING IN PROLOG

A. Lists

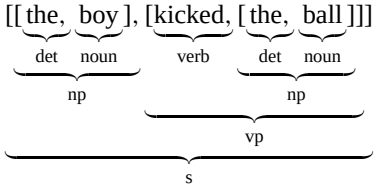
A list is an ordered sequence of elements that can have any length. Lists are written in Prolog using square brackets to delimit elements separated by commas, as in

```
colors([red, blue, green]).
```

Some other lists are the following:

```
[2,3,5,7]
[]
[[the, boy], [kicked, [the, ball]]]
```

The first list consists of the first four prime numbers, the next list is the empty list. The last one represents the grammatical structure of the sentence "The boy kicked the ball":



where s denotes sentence; np, noun phrase; vp, verb phrase; and det, determiner.

The first element of a list is called the head of the list; the remaining elements are called the tail; a vertical bar separates the list head from its tail:

```
?- colors([X|Y]).
   X = red,
   Y = [blue, green].
```

or it can be used to build a new list:

```
?- colors(X), sort([black|X], Z).
```

Procedural interpretation. Get the list of colors (called `X`), add black in front of it (`[black|X]`), and sort this list. The sorted list is returned as `Z`.

Table I shows the heads and tails for miscellaneous lists.

Table II shows the instantiations of the variables if `List_1` is matched with `List_2`.

Prolog's notation of a list using brackets and the vertical bar is only a syntactical convenience. Fundamentally, lists are structures having two arguments, the head and the tail of the list.

TABLE I Head and Tail for Selected Examples of Lists

List	Head	Tail
[a,b,c,d]	a	[bc,d]
[a]	a	[]
[a,[b,c]]	a	[[b,c]]
[[],[a],[b],[a,b]]	[]	[[a],[b],[a,b]]

TABLE II Matching of Two Lists

List_1	List_2	Instantiated variables
[a, b, c, d]	[X, Y, Z, U]	X = a Y = b Z = c U = d
[a, b, c, d]	[X|Y]	X = a Y = [b, c, d]
[a, b, c, d]	[X, Y|Z]	X = a Y = b Z = [c, d]
[a, b]	[X, Y|Z]	X = a Y = b Z = []
[a, b, c, d]	[X, Y|Z, W]	Incorrect syntax
[prolog, lisp]	[lisp, X]	Fails

B. Recursion

Data structures and procedures that are defined in terms of themselves are called recursive. In many cases the use of recursion permits the specification of a solution to a problem in a natural form. A simple example is the membership relation:

```
member(Element, [Element|_Tail]).
member(Element, [Head|Tail]):-
  member(Element, Tail).
```

This can be read as "The element given as the first argument is a member of the list given as the second argument if either the list starts with the element (the fact in the first line) or the element is a member of the tail (the rule in the second line)."

The following are possible questions (with answers):

```
?- member(d, [a,b,c,d]).
   yes
?- member(e, [a,b,c,d]).
   no
```

It is possible to produce all members of a list by asking

```
?- member(X, [a,b,c,d]).
   X = a;
   X = b;
   X = c;
   X = d.
```

C. Example: Permutations

The following Prolog program can be used to compute all permutations of a given list:

```
permutation([], []).
permutation(X, [H|T]):-
  append(Y, [H|Z], X),
  append(Y, Z, P),
  permutation(P, T).

append([], X, X).
append([A|B], C, [A|D]) :-
  append(B, C, D).
```

Actually, considering the declarative aspect of Prolog, it is better to state that this program defines a relation between two lists, one being a permutation of the other. Possible questions are

```
?- permutation([a,b,c], [b,c,a]).
   yes
```

and

```
?- permutation([a,b,c], [a,b,c,d]).
   no
```

The following question produces all possible permutations of the list `[a,b,c]`:

```
?- permutation([a,b,c], X).
   X = [a,b,c];
   X - [a,c,b];
   X = [b,a,c];
   X = [b,c,a];
   X = [c,a,b];
   X = [c,b,a].
```

Explanation. The predicate "append" defines a relation between three lists: the list given as the third argument is the concatenation of the lists given as the first and second arguments. For example, the following holds:

```
?- append([a,b,c], [i,j], [a,b,c,i,j]).
   yes
```

With the first two arguments instantiated and a variable as the third argument, "append" can be used to join two lists:

```
?- append([1,2,3], [4,5,6], X).
   X = [1,2,3,4,5,6].
```

This relation is traditionally called "append" in Prolog in accordance with the append-function in LISP. Append as the relation name, however, is disadvantageous, since it stresses the procedural aspect of concatenation over the declarative one. Different from LISP, Prolog's append can be used to split a list into all possible sublists as well:

```
?- append(X, Y,   [1,2,3]).
   X = [],        Y = [1,2,3];
   X = [1],       Y = [2,3];
   X = [1,2],     Y = [3];
   X = [1,2,3],   Y = [].
```

With the help of "append" the predicate "permutation" is easy to understand. The first clause says that the permutation of an empty list is an empty list. This seems to be trivial but it is essential since the second rule is recursive and therefore depends on this rule to terminate. The second rule says that to compute a permutation one has first to split the given list `(X)` into two sublists `(Y` and `[H|Z])`. The head of the second list `(H)` becomes the first element of the result `([H|T])`, but note that `T` is not yet instantiated. Then the first list `(Y)` and the tail-end of the second list `(Z)` are concatenated, forming list `P`. List `P` is permuted and this permuted list `(T)` forms the tail-end of the result `([H|T])`.

D. Terminology

The above Prolog program to compute permutations consists of two procedures ("permutation" and "append"). Each procedure comprises one or more *clauses* (two for "permutation" and two for "append"). A clause is terminated by a period. The procedure name is called a *predicate*. The number of arguments is called the *arity* (two for "permutation" and three for "append"). Each clause has a *head*, which is also called a procedure entry point, and may have a *body*. The head defines the form of the predicate's arguments. The body of a clause consists of *goals* or procedure calls, which impose conditions for the head to be true.

E. Operators

Operators are a syntactical convenience to make programs more readable. Instead of saying

```
composed(mozart, don_giovanni).
```

it is also possible to say

```
mozart composed don_giovanni.
```

if "composed" is defined as an infix operator. Operators have the following properties: (1) precedence, (2) position, (3) associativity, and (4) name.

Operators in Prolog are defined by executing the directive "op":

```
:- op(Prec, Spec, Name).
```

For example ("Name" can be a single name or a list of names),

```
:- op(1200,   fx, [:-, ?-]).
:- op(500,   yfx, +).
:- op(400,   yfx, *).
:- op(200,   xfy, ^).
```

Generally, directives allow one to specify how Prolog should behave in specific situations, and are indicated by the term ":-."

1. Precedence

A lower number (like 400 compared to 1200 in the example above) indicates a higher precedence:

$$3 * 4 + 5 \ \cdots \ (3 * 4) + 5 \ \cdots \ + (*(3, 4), 5).$$

Parentheses can be used to get a different result:

$$3 * (4 + 5) \ \cdots \ * (3, \ + (4, 5)).$$

2. Position and Associativity

Prefix operators are placed in front of the term, as in "-5"; postfix operators are placed after the term, as in "5!"; and infix operators are placed between two terms, as in "$2 + 3$." If two operators with the same precedence are used without parentheses, the meaning of such an expression has to be clarified. If the left operator binds more strongly, we speak of left-associativity; otherwise, of right-associativity. To define position and associativity for a Prolog operator, the following atoms are used:

Infix:	`xfx xfy yfx`
Prefix:	`fx fy`
Postfix:	`xf yf`

The meanings of "x" and "y" are as follows:

"x": Only operators of strictly lower precedence are allowed at this side of the term.
"y": Operators of lower or the same precedence are allowed at this side of the term.

yfx: left-associative
`a + b + c ... (a + b) + c ... +(+(a,b),c)`
xfy: right-associative
`a^b^c ... a^(b^c) ... ^(a,^(b,c))`
xfx: nonassociative
`a :- b :- c` ... this is an invalid term.

TABLE III Standard Operator Definitions

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).
:- op(  900,  fy, [ \+ ]).
:- op(  700, xfx, [ =, \=]).
:- op(  700, xfx, [==, \==, @<, @>, @=<, @>= ]).
:- op(  700, xfx, [ =.. ]).
:- op(  700, xfx, [ is, =:=, =\=, <, >, =<, >= ]).
:- op(  500, yfx, [ +, -, /\, \/ ]).
:- op(  400, yfx, [ *, /, //, rem, mod, <<, >> ]).
:- op(  200, xfx, [ ** ]).
:- op(  200, xfy, [ ^ ]).
:- op(  200,  fy, [ - \]).
```

A list of standard operator definitions is shown in Table III. Some of the symbols have already been introduced, like `:-, ?-,` etc.; others are ordinary mathematical symbols, like `+, *, <, **,` etc., and for bit-oriented manipulation, `/\, \/, <<, >>, \`. Some will be introduced later, like `-->` (used with grammar rules), `->` ("if"), `;` ("or"), `\+` ("not provable"), `\=` ("not unifiable"), `==, \==, @<, @>, @=<,` and `@>=` (comparison operators for terms), `=..` (manipulation of structured terms), `is, =:=, =\=` (evaluation of arithmetic expressions), and `^` (used with findall, setof, and bagof).

F. Control Constructs

1. Cut

The cut allows one to control the procedural behavior of Prolog programs. The cut succeeds only once. In case of backtracking, it not only fails, but causes the parent goal to fail as well, indicating that choices between the parent goal and the cut need not be considered again. By pruning computation paths in this form, programs operate faster or require less memory space.

Interpretation of the Cut

First interpretation of the cut. "If you get this far, you have picked the correct rule for this goal, there is no point in ever looking for alternatives (there are no more or these are not the correct ones)."

Example. Transliterate an English sentence into German:

```
english_german(you, du)    :- !.
english_german(are, bist)  :- !.
english_german(a, ein)     :- !.
english_german(X, X).      /* catch-all */

list_transliterated([], []).
list_transliterated([E_Word|E_WordList],
            [G_Word|G_WordList]) :-
  english_german(E_Word, G_Word),
  list_transliterated(E_WordList, G_WordList).

?-list_transliterated([you,are,a,computer],X).
  X = [du,bist,ein,computer].
```

We would get incorrect solutions (like `[du, bist, a, computer]`) without the cuts in the first three clauses of "english_german."

Problems with the cut. There are many problems caused by the cut: A program such as

```
append([], X, X) :- !.
append([X|Xl], Y, [X|Zl]) :-
append(Xl, Y, Zl).
```

would still work on questions like

```
?- append([l,2,3], [4,5], X).
```

but would produce only one solution in

```
?- append(X, Y, [l,2,3,4,5]).
   X = [], Y = [1,2,3,4,5].
```

Second interpretation of the cut. The cut–fail combination: "If you get this far, you should stop trying to satisfy this goal, you are wrong!" This helps to solve the problem if a sentence has no Horn-clause representation such as "Siblings are not married." The expression "siblings (X, Y) $\Rightarrow$ not married (X, Y)" would lead to a rule with a negated left-hand side, which is not allowed in Horn-clause logic. The solution is to write a Prolog program such as

```
married(X, Y) :- siblings(X, Y), !, fail.
married(X, Y).
```

If a pair of siblings can be found, the cut will be executed and the first rule fails due to the predicate "fail" (see next subsection). Now the cut prevents two things. First, Prolog will not look for further solutions for siblings, and, second, Prolog will not look for other solutions for the "married" predicate.

2. True and Fail

The predicate "fail" was used in the above example. It never succeeds. The opposite behavior holds for "true." It

succeeds, but fails on backtracking. Using "true," a fact can be represented as a rule:

```
composed(mozart, don_giovanni):-true.
```

3. Conjunction, Disjunction, If-Then, and If-Then-Else

Prolog's basic control structure is the conjunction (",", an infix operator), which is true if both arguments are true:

```
a :- b, c.
```

Disjunctions are normally expressed by two clauses

```
a :- b.
a :- c.
```

but can be written also using ";" as follows:

```
a :- b;c.
```

If-then is expressed by '`->`'

```
a :- b -> c.
```

"a" is true, if "b" is true and "c" is true, but only for the first solution of "b"!

If-then-else is expressed by

```
a :- b -> c ; d.
```

"a" is true if (i) "b" is true and "c" is true, but only for the first solution of "b," or (ii) "b" is false and "d" is true!

VI. BUILT-IN PREDICATES

Built-in predicates provide facilities that cannot be obtained by definitions in pure Prolog. In addition, they provide convenient facilities to save each programmer from having to define them.

A. Logic and Control

1. Not Provable: \+

The predicate `\+` is defined in analogy to the second interpretation of the cut; therefore, the above example about siblings could also be written as

```
married(X,Y) :- \+ siblings(X,Y).
```

The predicate `\+` is defined as a prefix operator. The goal `\+ X` must be instantiated to an executable goal when `\+ X` is executed. The goal `\+ X` succeeds if the goal `X` fails, and fails if the goal `X` succeeds. This is based on the idea to interpret "negation as failure," and this is different from negation in logic; it means only: X is not provable. Note: Executing `\+ X` can never result in X becoming more instantiated. Any unbound variable which is part of X is still unbound after executing `\+ X`.

2. Repeat

The goal "repeat" always succeeds and can always be resatisfied. This goal can be used to build looplike control structures if used in conjunction with a goal that fails. Such a loop is called a failure-driven loop:

```
repeat,
  goal1,
  goal2,
  ...,
end_test.
```

The loop is terminated if the `end_test` succeeds. If `end_test` fails, Prolog backtracks. Since repeat always succeeds, Prolog executes `goal1` and `goal2` again. There is a difference, however, between such a construct in Prolog and a normal repeat loop if goals in the loop can be resatisfied or fail. In that case the execution can bounce several times between these resatisfiable goals and the `end_test` before going up to the repeat. Furthermore, the execution can bounce between the repeat at the beginning and a goal in the loop body that fails. Finally, failure-driven loops are only useful when used in conjunction with built-in predicates that cause side effects, such as read and write predicates. For an example, the user is referred to Program I.

Program I

```
copy(In_file, Out_file) :-
  open(In_file, read, IN),
  open(Out_file, write, OUT),
  /* begin loop */
  repeat,
    get_code(IN, Ch),
    char_mapped(Ch, Ch_new),
    put_code(Ch_new),
    Ch = -1, % end of file test
  !,
  /* end loop */
  close(IN),
  close(OUT).
char_mapped(Lower, Upper) :-
  Lower >= 97,  /* a */
  Lower =< 122, /* z */
  !,
  Upper is Lower - 32.
char_mapped(X, X).
```

3. Once

To find only one (the first) solution, once(X) can be used. If no solutions can be found, it fails. However, on backtracking, it explores no further solutions.

B. All Solutions

Normally, a goal like

```
?- composed(verdi, X).
```

produces a solution, and on backtracking the next one, and so on. The predicates "findall," "bagof," and "setof" allow one to collect all solutions to a query to be collected in a list. They all have the same arguments "(Template, Goal, Result)," but have different semantics. The predicate "bagof" assembles the result list in the order in which the results are found, "setof," in addition, sorts the list and removes duplicates. Predicates "bagof" and "setof" treat variables differently from "findall." An example shall clarify this. Based on the database for operas from the introductory example, the goal

```
?- findall(X,composed(verdi, X),Operas).
```

produces

```
Operas = [rigoletto, macbeth, falstaff].
```

The question

```
?- bagof(X, composed(verdi, X), Operas).
```

produces the same result, whereas

```
?- setof(X, composed(verdi, X), Operas).
```

produces the sorted list

```
Operas = [falstaff, macbeth, rigoletto].
```

A question like

```
?-findall(X, composed(einstein, X), Operas).
```

succeeds and produces the solution

```
Operas = [].
```

The corresponding queries with "bagof" and "setof" fail.

A more delicate query is

```
?-setof(Work,composed(Composer,Work),Operas).
```

The variables "Composer" and "Work" are not of the same kind. One ("Work") appears in "Template," the other ("Composer") does not! The predicates "bagof" and "setof" bind such a free variable, and then produce a list of all solutions for this binding. On backtracking, they produce the next list, and so on:

```
Operas = [fidelio],
Composer = beethoven;

Operas = [don_giovanni],
Composer = mozart;

Operas = [il_barbiere_di_siviglia],
Composer = paisiello;

Operas = [guillaume_tell,
          il_barbiere_di_siviglia],
Composer = rossini;

Operas = [falstaff, macbeth, rigoletto],
Composer = verdi.
```

The corresponding query with "findall,"

```
?- findall(Work, composed(Composer, Work),
           Operas).
```

treats a variable like "Work" not as free, but as existentially quantified. This results in only one solution:

```
Operas =[fidelio, don_giovanni, rigoletto,
         macbeth, falstaff,guillaume_tell,
         il_barbiere_di_siviglia,
         il_barbiere_di_siviglia].
```

The exactly same result can be achieved with "bagof" by stating that the free variable shall be existentially quantified:

```
?- bagof(Work,
         Composer^composed(Composer, Work)
         Operas).
```

"Composer ^ composed(Composer, Work)" can be read as: "There exists a Composer such that composed(Composer, Work) is true."

C. Input/Output

Prolog predicates that lie outside the logic programming model are called extra-logical predicates. The main reason for using them is the side effect they achieve. The Prolog predicates for I/O belong to this group. For a practical reason, these predicates cannot be resatisfied; they fail on backtracking. Only the major aspects are covered here.

1. Input/Output for Terms

Read. The predicate for input is "read." The argument is a term. The goal "read(X)" succeeds if X matches the next term appearing on the current input stream:

```
?- read(X).
```

The user enters `female(mary).`

```
X = female(mary).
```

Write. The predicate for output is "write." The argument is a Prolog term. The term is written to the current output stream.

2. Layout

New line. The predicate "nl" produces a new line on the current output stream.

Example. We can write a list of terms

```
write_list([H|T]) :-
  write(H),nl,write_list(T).
write_list([]) :- nl.
```

```
?- write_list([a,b,c(d)]).
      a
      b
      c(d)
   yes
```

3. Input/Output for Characters

Prolog provides I/O based on characters as well. Predicates with suffix "_code" use character codes (ASCII), and those with suffix "_char" use character atoms. Each predicate comes with two variants: with an explicit first argument, indicating the stream, or without one; in this case, the standard input or output stream is used.

Reading characters. The goal "get_code(X)" succeeds if "X" matches the ASCII code of the next printing character, "get_char(X)" returns an atom. The end of the file is indicated by the integer "-1" for "get_code" and by "end_of_file" for "get_char."

Writing characters. The goal "put_code(X)" prints out the character whose ASCII code corresponds to "X":

```
?- putcode(65).      prints `A'
?- putchar(a).       prints `a'
```

D. File Handling

Prolog supports the transfer of data to and from one or more external files. The predicates to open and close streams are, respectively,

```
open(Filename, Mode, Stream)
close(Stream)
```

Mode is either read, write, or append. An example of copying a file and mapping all lowercase letters into uppercase letters is given in Program I.

E. Arithmetic

Expressions are evaluated using "is," which is defined as an infix operator (xfx). The right-hand side of the "is" goal is evaluated. If variables are used, they have to be instantiated to numbers, otherwise the Prolog system produces an error. The result of the evaluation is then unified with the term on the left-hand side. Prolog supports the usual mathematical functions like abs(), sign(), round(), truncate(), sin(), cos(), atan(), exp(), log(), sqrt(), and power(**).

Example.

```
?- X is (2 + 3) * 5.
   X = 25.
```

Arithmetic comparison operators cause evaluation of expressions as well. Depending on the results, the goal succeeds or fails. Arithmetic operators and comparison operators are listed in Tables IV and V, respectively.

Examples.

`X is 1 + 2 + 3.`	$X = 6$
`X is 4 - 5.`	$X = -1$
`X is sqrt(2.3).`	$X = 1.51657508881031$
`X is 2**3.`	$X = 8.0$
`6 is 1 + 2 + 3.`	succeeds
`7 is 1 + 2 + 3.`	fails
`X is 4 + Y.`	gives an error
`Y is 5, X is 4 + Y.`	succeeds
`X is a + 2 + 3.`	gives an error
`7 is 4 + Y.`	gives an error
`1 + 3 =:= 2+2.`	succeeds
`1 + 3 =\= 2 + 5.`	succeeds
`1 + 3 < 2 + 4.`	succeeds

TABLE IV Arihmetic Operators

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division (float)
`//`	Division (integer)
`rem`	Remainder
`mod`	Modulo
`/\`	Bitwise and
`\/`	Bitwise or
`\`	Bitwise complement
`<<`	Shift left
`>>`	Shift right

TABLE V Arithmetic Comparison Operators

Operator	Description
<	Less
>	Greater
=<	Less-equal (the syntax avoids arrows!)
>=	Greater-equal
=:=	Equal
=\=	Not equal

F. Manipulating and Testing the Database

Prolog allows one to check the existence of clauses in the database and to add and remove clauses if the predicate is declared "dynamic."

1. Checking Clauses in the Database

The predicate "clause" is used to check if a clause is in the database. It requires two terms for its arguments. The first term is to be matched against the heads of clauses in the database, the second against the body of the clause. The goal may be resatisfied. A clause with no body is treated as if it had a body consisting of the single goal "true."

Examples. In the example in Section III.E ("A small example about people, food, wine, and love"), the last clause is

```
likes(paul, Something) :-
  likes(peter, Something).
```

If "likes/2" was declared "dynamic," by executing the directive

```
:- dynamic(likes/2).
```

then the following questions could be asked:

```
?- clause(
     likes(paul, Something), likes(peter,
       Something)).
   yes

?- clause(likes(X, Y), Z).
   X = mary, Y = food, Z = true;
   X = mary, Y = wine, Z = true;
   X = peter, Y = wine, Z = true;
   X = peter, Y = mary, Z = true;
   X = paul, Z = likes(peter, Y).
```

2. Asserting New Clauses

Clauses may be added to the database using assert. The predicate "asserta" adds a clause before all other clauses for the predicate, and "assertz" adds the clause after it.

Example.

```
?- asserta( composed(bizet, carmen) ).
?- assertz( (sign(X, 1)  :- X > 0) ).
?- assertz( (sign(X, 0)  :- X =:= 0) ).
?- assertz( (sign(X, -1) :- X < 0) ).
```

3. Retracting Clauses

The predicate "retract" removes a clause from the database. The goal can be resatisfied. The predicate "abolish" removes all clauses for a given predicate. The argument for abolish is a predicate indicator, i.e., a term of the form Name/Arity.

Example.

```
?- retract( (likes(X, Y) :- Z) ).
```

The first clause for predicate "likes" with arity 2 and arbitrary body will be removed.

Example.

```
?- abolish(likes/2).
```

All facts and rules for predicate "likes" with arity 2 are removed.

G. Manipulating, Creating, and Testing Terms

1. Testing Terms

The predicates for testing terms are shown in Table VI. These predicates are meta-logical since they treat variables, rather than the terms they denote, as objects of the language.

2. Manipulating Structured Terms

a. Functor. The predicate "functor" is called with three arguments, "functor(S, F, A)," where S is a structure or atom, F is an atom indicating the name of principal functor of S, and A is the arity of S.

The predicate "functor" is used in most cases in either one of the following ways:

1. Get the principal functor and arity for a given structure, that is, S input, F and A output:

TABLE VI Predicates for Testing Terms

var(X)	Succeeds if X is uninstantiated when the goal is executed
atom(X)	Succeeds for atoms ([] is an atom)
integer(X)	Succeeds for integers
float(X)	Succeeds for reals
atomic(X)	Succeeds if X is an atom or number
compound(X)	Succeeds if X is instantiated to a compound term
nonvar(X)	Succeeds if X is instantiated when the goal is executed
number(X)	Succeeds for numbers

```
?- functor(append([], X, X), Name, Arity).
   Name = append,
   Arity = 3.
```

2. Construct a structure for a given name and arity, that is, S output, F and A input:

```
?- functor(X, append, 3).
   X = append(_A, _B, _C).
```

b. `=...` The predicate ``=..'' (pronounced "univ") is defined as an infix operator whose arguments are a structure on the left-hand side and a list on the right-hand side.

Example.

```
?- composed(mozart, don_giovanni) =.. X.
   X = [composed, mozart, don_giovanni].
```

This predicate is necessary in Prolog since goals and structures cannot be manipulated directly. This is possible, however, for the corresponding list. A typical action in a grammar rule translator (see Sections VII and VIII) is to expand a term with two additional arguments. In the following example, `expr(Value)` is expanded into `expr(Value,S0,S)`:

```
?- expr(Value) =.. L,
   append(L, [S0,S], L1),
   G =.. L1.

   G  = expr(Value, S0, S),
   L  = [expr, Value],
   L1 = [expr, Value, S0, S].
```

c. arg. The predicate "arg" has three arguments. The goal ``arg(N,S,C)'' succeeds if C matches the Nth argument of the structure S.

Example.

```
?- arg(2, composed(verdi, falstaff), X).
   X = falstaff.
```

3. Term Unification

The predicates for testing and forcing equality are defined as infix operators.

`=` *and* `\=`. Two terms are defined as being equal or not equal depending on the success of the unification process. The following goals will succeed:

```
?- a = a.
?- a = X.
?- X = Y.
?- a \= b.
```

4. Term Comparison

a. `==` *and* `\==`. A different concept is identity. The main difference concerns variables. Two terms are identical if they are:

1. Variables which are linked together by a previous unification process, or
2. The same integer, or
3. The same float, or
4. The same atom, or
5. Structures with the same functor and identical components.

The following goals will succeed:

```
?- X == X.
?- X \== Y.
?- a == a.
?- a(X,Y) == a(X,Y).
?- a(3) \== a(X).
```

b. `@=<, @=<, @<, @>,` *and* `@>=.` These five operators are based on a precedence relation between arbitrary terms. The following goals will succeed:

```
?- abc @< xyz.
?- 2 @>= 1.
?- foo(a) @< foo(b).
?- foo(a) @< foo(a,a).
```

c. `=:=` *and* `=\=.` A third form of equality was covered in the section on arithmetics. Two expressions are considered numerically equal if they evaluate to the same value, otherwise they are numerically not equal. The following goals will succeed:

```
?- 1+3 =:= 2+2.
?- 1+3 =\= 1+4.
```

VII. DEFINITE CLAUSE GRAMMARS

A. Parsing

The formal description of the syntax of a language (natural language or programming language) is called a grammar. Certain rules define which sequences of words are valid in the language and which are not. The process of recognizing a correct sentence of a language is called parsing. The structure of the sentence thereby produced is called the parse tree. An important class of grammars consists of context-free grammars (CFGs), developed by the linguist Noam Chomsky. At the same time John Backus used a similar grammar form to define ALGOL. Grammars of this type are called BNF (Backus-Naur-Form) grammars. Both types are equivalent in power, and the difference is only notational.

A CFG is described by three sets: a set of terminals, which are basic words of the language; a set of nonterminals, which describe categories (verb-phrase, determiner, expression, statement, etc.); and a set of rules (also called productions). There is a natural correspondence between this approach and first-order predicate logic expressing context-free production rules in Horn clause form. Since Horn clauses are also called definite clauses, this form for describing a grammar is called a definite clause grammar (DCG). DCGs are extensions of CFGs. They have the advantages of CFGs, but they also overcome some of the disadvantages of CFGs, as follows:

1. DCGs can implement context dependence. This means that production rules can be restricted to a specific context where a phrase may appear.
2. DCGs can produce data structures aside from the recursive structure of the grammar.
3. DCGs permit a flexible mixture of production rules and auxiliary computation.

Since Horn clauses have a procedural interpretation, such a DCG implemented in Prolog yields a parser for the language described by the grammar. Since Prolog procedures are called in a depth-first fashion, such a parser is a recursive-descent, top-down parser.

B. Definite Clause Grammars in Prolog

DCGs in a Prolog environment use a Prolog-like syntax. Terminals are written in brackets; nonterminals are written in the form of atoms. A small example defining the syntax of a simple expression can illustrate this:

```
expr --> term, [+], expr.
expr --> term.

term --> numb, [*], term.
term --> numb.

numb --> [1].
numb --> [2].
numb --> [3].
numb --> [4]. /* etc. */
```

This grammar defines simple expressions consisting of numbers and the two arithmetic operators "plus" and "times," obeying the usual hierarchy of operators used in mathematics. The second group of rules (or productions), for example, define a term. A term is a number, followed by a times symbol, followed by a term; or a term is simply a number. Generally, a rule like

```
head --> body.
```

can be read as "a possible form for 'head' is 'body'."

The above grammar is not yet a Prolog program. To produce the Prolog facts and rules, most Prolog systems have a built-in grammar rule processor that automatically translates a production rule into a correct Prolog rule (DCGs are not part of the ISO standard). To determine whether a given sentence is valid for a given grammar, the predicate "phrase" is used. This predicate has two arguments; the first is the name of the production to be used, and the second is the list of words of the sentence. To check if $2 * 3 + 4$ is a valid sentence, the goal

```
?- phrase(expr,[2,*,3,+,4]).
```

has to be executed. The answer is yes.

C. Arguments of Nonterminals and Mixing Grammar Rules and Prolog Code

Grammar rules may also have arguments and can execute Prolog goals. To execute a sequence of Prolog goals, these goals must be put between braces. We shall extend the above example so that not only can the validity of an expression be checked, but the value of the expression can be computed:

```
expr(Z) --> term(X), [+], expr(Y),
            {Z is X+Y|.

expr(Z) --> term(Z).
term(Z) --> numb(X), [*], term(Y),
            {Z is X*Y|.
term(Z) --> numb(Z).

numb(C) --> [C], {integer (C)|.
```

A goal like

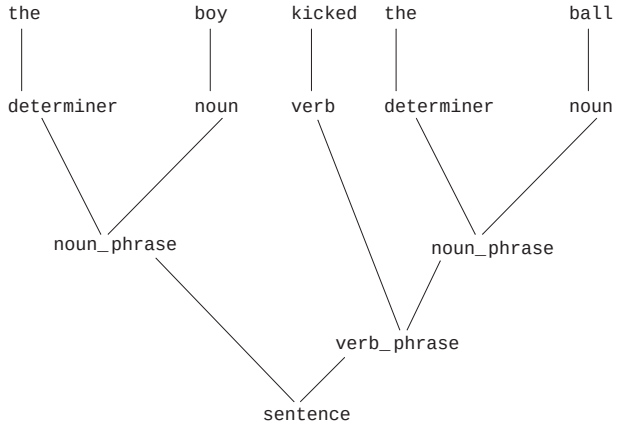
```
?- phrase(expr(X),[3,*,4,+,5]).
```

will now produce the answer

```
X = 17.
```

D. Translation of Grammar Rules into Prolog Rules

As an example of the way grammar rules work in Prolog, we shall use the following sentence: "The boy kicked the ball." The grammatical structure is:



The complete grammar for parsing a sentence like this is as follows:

```
sentence     -->  noun_phrase, verb_phrase.
noun_phrase  -->  determiner, noun.
verb_phrase  -->  verb, noun_phrase.
determiner   -->  [the].
verb         -->  [kicked].
noun         -->  [boy].
noun         -->  [ball].
```

A grammar rule like

```
noun_phrase --> determiner, noun.
```

is translated into the Prolog rule

```
noun_phrase(S0,S) :-
  determiner(S0,S1),
  noun(S1,S).
```

The goal

```
?- noun_phrase([the, boy, kicked, the, ball],S).
```

would succeed and instantiate S to `[kicked,the, ball]`. This also gives the correct interpretation of such a rule: The difference between S0 and the tail-end sublist S is a `noun_phrase` if there is some S1 so that the difference between S0 and the tail-end sublist S1 is a `determiner`, and the difference between S1 and S is a `noun`. More succinctly, a `noun_phrase` extends from S0 to S if there is a `determiner` from S0 to S1 and a noun from S1 to S.

The first parameter can be considered to be the input parameter, the second to be the output parameter. The first parameter is a list of words, which is split into two sublists. The first sublist, the words at the beginning of the sentence, is parsed by the rule. The second sublist forms the second parameter. It consists of the remaining words of the sentence that are not parsed by the rule.

The values of the variables S0, S1, and S for the example are

```
S0 = [the,boy,kicked,the,ball],
S1 = [boy,kicked,the,ball],
S  = [kicked,the,ball].
```

This demonstrates the correctness of the following interpretation: The difference between S0 and S is the list `[the, boy]` which is the result of parsing the input sentence S0 as a noun-phrase.

To parse a sentence completely by a rule, the second parameter has to be set to the empty list:

```
?- sentence([the,boy,kicked,the,ball],[]).
   yes
```

E. A Grammar Example

The arguments of the nonterminals can be used to produce data structures during the parsing process. This means that such a grammar can be used not only to check the validity of a sentence, but also to produce the corresponding parse tree. Another use of arguments is to deal with context dependence. To handle, e.g., number agreement, certain nonterminals will have an extra argument, which can take the values "singular" or "plural."

The following grammar demonstrates some aspects of writing a DCG in Prolog. It produces the complete parse tree of the sentence. It handles some form of number agreement. (A sentence like "The boy kick the ball" would be rejected.). Finally, it separates grammar rules from the dictionary. In this form it is easier to maintain the grammar:

```
/* A simple grammar */
sentence(s(NP, VP))
  --> noun_phrase(NP, Number),
      verb_phrase(VP, Number).
noun_phrase(np(Det,Noun), Number)
  --> determiner(Det, Number),
      noun(Noun, Number).
verb_phrase(vp(V,NP), Number)
  --> verb(V, Number, transitive),
      noun_phrase(NP, _).
determiner(det(Word), Number)
  --> [word],
      {is_determiner(Word, Number)|.
```

```
  noun(n(Root), Number)
    --> [Word],
        {is_noun(Word, Number, Root)|.
  verb(v(Root, Tense), Number, Transitivity)
    --> [Word],
        {is_verb(Word, Root, Number,
                 Tense, Transitivity)|.

            /* the dictionary */
/* determiner */
is_determiner(a, singular).
is_determiner(every, singular).
is_determiner(the, singular).
is_determiner(all, plural).

/* nouns */
is_noun(man, singular, man).
is_noun(men, plural, man).
is_noun(boy, singular, boy).
is_noun(boys, plural, boy).

is_noun(woman, singular, woman).
is_noun(women, plural, woman).
is_noun(ball, singular, ball).
is_noun(balls, plural, ball).

/* verbs */
is_verb(Word, Root, Number,
      Tense, Transitivity) :-
  verb_form(Word, Root, Number, Tense),
  infinitive(Root, Transitivity).

infinitive(kick, transitive).
infinitive(live, intransitive).
infinitive(like, transitive).

verb_form(kicks, kick, singular, present).
verb_form(kick, kick, plural, present).
verb_form(kicked, kick, _, past).
verb_form(lives, live, singular, present).
verb_form(live, live, plural, present).
verb_form(lived, live, _, past).
verb_form(likes, like, singular, present).
verb_form(like, like, plural, present).
verb_form(liked, like, _, past).
```

This grammar allows one to parse the original sentence "The boy kicked the ball," producing the following parse tree:

```
?- phrase(sentence(X),
         [the, boy, kicked, the, ball]).

   X = s(np(det(the), n(boy)),
        vp(v(kick, past),np(det(the),n(ball)))).
```

In addition, one can parse many other sentences, most of which may not make sense, such as

```
a man kicks a man
a man kicks a boy.
a boy kicked a man.
every man likes every woman.
```

VIII. META-LEVEL PROGRAMMING

Program and data in Prolog have the same syntactical form. Predicates such as `clause, assert, ==..,` and `arg` permit the manipulation of programs, even the manipulation of the program being executed. This language aspect, which was already an outstanding feature of LISP, makes Prolog a very powerful tool. Programs that analyze, transform, or even simulate other programs are called meta-programs. The main characteristic is that such programs treat other programs as data. A good illustration is a grammar rule translator, mentioned in the section above. Other examples are expert systems shells or Prolog interpreters, which use different unification algorithms, search strategies, or simulate parallel execution.

The purpose of a grammar rule translator is to transform a syntax rule such as

```
noun_phrase -> determiner, noun.
```

into the corresponding Prolog clause

```
noun_phrase(S0,S) :-
  determiner(S0,S1), noun(S1,S).
```

An example of a simple grammar rule translator is given in Program II, covering the essential aspects only. Advanced features, such as the mixing in of Prolog code or cuts, are not considered.

In the following examples, variables produced by the system are represented in the form of uppercase characters preceded by an underscore character. The same variables are represented by the same character:

```
?- expand_term(
      (sentence --> noun_phrase,
      verb_phrase), X).

   X = sentence (_A,_B) :-
    noun_phrase(_A, _C),verb_phrase(_C,_B).

?- expand_term( (noun --> [ball]), X).
   X = noun([ball}_A],_A) :- true.

?- expand_term((expr --> term,[+],expr), X).
   X = expr(_A,_B) :-
       term(_A, [+}_C]), expr(_C,_B).
```

Program II

```
expand_term((Head --> Body), (Head1:-Body1)) :-
  expand_goal(Head, S0, S, Head1),
  expand_body(Body, S0, S, Body1).

expand_goal(Goal, Before, After, NewGoal) :-
  Goal =.. GoalList,
append(GoalList, [Before,After], NewGoalList),
  !,
  NewGoal =.. NewGoalList.

expand_body((P1,P2), S0, S, G) :-
  !,
  expand_body(P1, S0, S1, G1),
  expand_body(P2, S1, S, G2),
  expand_and(G1, G2, G).

expand_body(Terminals, S0, S, true) :-
  append(Terminals, S, S0),
  !.

expand_body(NonTerm, S0, S, Goal) :-
  expand_goal(NonTerm, S0, S, Goal).

expand_and(true, A, A) :- !.
expand_and(A, true, A) :- !.
expand_and(A, B, (A,B)).
```

```
  % The first argument of expand_term,
% a grammar rule is translated into
% the corresponding Prolog clause.

  % expand_goal adds the two
% additional parameters.

  % expand_body consists of three clauses:
% The first one deals with conjunctions.
% P1 is either a goal or a terminal.
% expand_body(P1,..) will use the second
% or third clause accordingly. If P2 is a
% conjunction, the first clause is called
% recursively. If P2 is a goal or a terminal,
% it is treated in the same manner
% as P1. expand_and puts the two partial
% results G1 and G2 together, yielding G
% as result.

    % The second clause deals with terminals
% that are written in list form. To check
% if Terminals is really a list, a small
% trick is used. The append goal succeeds
% only with lists. If `Terminals' is a
% goal, append fails, but if it is a list
% the desired effect is achieved, and the
% list of terminals is expanded for the
% two additional parameters.

    % The third clause deals with simple
% nonterminals.

    % expand_and forms a conjunction out of
% two goals. The first two clauses deal
% with the special case that one argument
% is the goal true in which case the
% other argument is returned.
```

IX. PROGRAMMING WITH CONSTRAINTS OVER FINITE DOMAINS

A. Introduction

The following queries will demonstrate some deficiencies of Prolog (we assume that all numbers are integers):

```
?- Z is X + Y, X=3, Y=4.                    (1)
   {INSTANTIATION ERROR: _71 is _68+_69 - arg 2}

?- X=3, Y=4, Z is X + Y.                    (2a)
   X = 3, Y = 4, Z = 7.

?- Z=7, Y=4, Z is X + Y.                    (2b)
   {INSTANTIATION ERROR: _71 is _68+4 - arg 2}

?- X >= 3, X < 4.                           (3)
   {INSTANTIATION ERROR: _68>=3 - arg 1}

?- (X=1; X=3), (Y=1; Y=3), X \= Y.          (4)

   X = 1, Y = 3;
   X = 3, Y = 1.

?- X \= Y, (X=1; X=3), (Y=1; Y=3).          (5)
no
```

Query (1) produces an instantiation error since at the time the first goal is executed, X and Y are not bound. Rearranging the goals (2a) produces the correct solution: $Z = 7$. Rearranging the goals would not help for (2b), even though simple arithmetic would lead to the desired solution: X is $Z - Y$. Query (3) produces an instantiation error, but it is trivial that X should be 3 (assuming X to be an integer!). Prolog easily finds solutions for query (4); however, query (5) fails because of the first subgoal (X and Y are unbound, therefore they can be unified!) Prolog would be a more powerful language if all these cases would be handled in the intuitive way: a goal like "Z is $X + Y$" should be delayed until X and Y are bound; 7 is $X + 4$ should produce a solution for X; if the constraints narrow a variable to a single value, the variable should be bound to this value, and there should be a predicate "different_terms" delayed until the arguments are sufficiently bound to decide on final failure or success of the goal. All this is achieved by extending Prolog with the concept of constraints.

B. Constraint Logic Programming (CLP)

Prolog manipulates pure symbols with no intrinsic meaning. Numbers, on the other hand, have a rich mathematical structure: algebraic operations (e.g., addition and multiplication) and order (e.g., =, <, and >). Taking advantage of this for Prolog means extending the purely abstract logical framework by introducing domains for variables and constraints for those variables which have to be obeyed. This is called constraint logic programming (CLP). In such a CLP system the simple unification algorithm that lies at the heart of Prolog is augmented by a dedicated "solver," which can decide at any moment whether the remaining constraints are solvable. For efficiency's sake, solvers for CLP systems need to be monotonic, so that adding a new constraint to an already solved set does not force them all to be re-solved. From a simple user's perspective, CLP allows one to do mathematics with unbound variables.

C. Constraints

Constraints arise in most areas of human endeavor. They formalize the dependences in physical worlds and their mathematical abstractions naturally and transparently. A constraint simply is a logical relation among several variables, each taking a value in a given domain. The constraint thus restricts the possible values that variables can take; it represents partial information about the variables of interest. The important feature of constraints is their declarative manner, i.e., they specify what relationship must be fulfilled without specifying a computational procedure to enforce that relationship.

D. Constraint Satisfaction

Constraint satisfaction deals with problems defined over finite domains. A constraint satisfaction problem is defined as follows:

- A set of *variables* $X = x_1, \ldots, x_n$.
- For each variable x_i a finite set D_i of possible values (called the *domain*).
- A set of *constraints* restricting the values that the variables can simultaneously take.

A solution for a constraint satisfaction problem is a set of assignments of values from its domain to every variable so that all constraints are satisfied at once.

E. Constraint Logic Programming and Prolog

Constraint solvers are not part of the ISO Prolog standard, but it is easy to add such a solver as a module. Most Prolog systems provide those solvers, the most important being clp(FD), where the FD stand for "finite domain." Only the most basic aspects are covered here.

1. Imposing Domains on Variables

A domain consists of an upper and a lower bound for a variable. There are two different ways of defining domains for variables. One is to use the range operator ".." imposing an upper and lower bound as in `X in 1..5`. In this way it is possible to impose domains on unbound variables by constraining them with already bound variables (#= denotes an equity constraint).

```
?- X in 1..5, Y in 2..8, X+Y #= T.
   X in 1..5,
   Y in 2..8,
   T in 3..13.
```

The second way to impose domains upon variables is designed for larger problems. It produces domains for a list of variables:

```
?- domain(VarList,Lower,Upper).
```

VarList is a list of unbound variables, while Lower is an integer or the atom "inf" and Upper is an integer or the atom "sup" ("inf" means no lower bound and "sup" means no upper bound).

2. Imposing Constraints on Variables

Arithmetic constraints. The syntax for arithmetic constraints is "Expression ComparisonOperator Expression." An "Expression" is an arithmetic term, consisting of integers and/or variables, connected by arithmetic operators, e.g., `+, -, *, /`. The comparison operators in clp(FD)

TABLE VII Comparison Operators in clp(FD)

Operator	Description
#<	Less
#>	Greater
#=<	Less-equal
#>=	Greater-equal
#=	Equal
#\=	Not equal

are distinguished from the ordinary ones by a leading "#" (Table VII).

Propositional constraints. Propositional constraints are used to combine constraints into phrases, thus making it possible to define conditional constraints. Examples are negating a constraint or defining an "either constraint 1 or constraint 2" constraint (Table VIII).

Symbolic constraints. The constraint `all_different(List)` ensures that each variable in the list is constrained to take a value that is unique among the variables. "List" is a list of domain variables.

The constraint "all_different" is the most important of the group of so-called symbolic constraints. The rest of this group is not explained here, only the names are given: "count," "element," "relation," "assignment," "circuit" (Hamiltonian circuit), and constraints specifically designed for scheduling problems: "serialized," "cumulative," "disjoint1," and "disjoint 2."

F. clp(FD)

With this instrumentarium, it is possible to formulate the examples from above in clp(FD) form:

TABLE VIII Propositional Operators in clp(FD)

Operator	Description
`#\ C`	True if the constraint `C` is false
`C1 #/\ C2`	True if the constraints `C1` and `C2` are both true
`C1 #\ C2`	True if exactly one of the constraints `C1` and `C2` is true (xor)
`C1 #\/ C2`	True if at least one of the constraints `C1` or `C2` is true
`C1 #=> C2`	True if constraint `C2` is true or the constraint `C1` is false
`C1 #<= C2`	True if constraint `C1` is true or the constraint `C2` is false
`C1 #<=> C2`	True if the constraints `C1` and `C2` are both true or both false

```
?- Z #= X + Y, X=3, Y=4.                                    (1)
   X = 3, Y = 4, Z = 7.

?- X=3, Y=4, Z #= X + Y.                                    (2a)
   X = 3, Y = 4, Z = 7.

?- Z=7, Y=4, Z #= X + Y.                                    (2b)
   X = 3, Y = 4, Z = 7.

?- X #>= 3, X #< 4.                                         (3)
   X = 3.

?- (X#=1 #\/ X#=3),(Y#=1 #\/ Y#=3),X #\= Y.
                                                            (4)
   X in inf..sup, Y in inf..sup.

?- X #\= Y, (X#=1 #\/ X#=3), (Y#=1 #\/ Y#=3).
                                                            (5)
   X in inf..sup, Y in inf..sup.
```

Queries (1)–(3) exhibit the expected behavior. Instead of `X=3`, also `X#=3` could be used. Queries (4) and (5) demonstrate that not everything that could be deduced is also deduced by the solver. One possibility to overcome this is to formulate the query in the following form [and similarly for (4)]:

```
?- X #\= Y, (X=1; X=3), (Y=1; Y=3).                         (5a)
   X = 1, Y = 3;
   X = 3, Y = 1.
```

What we actually do is post a constraint and then try to find a set of assignments for the variables satisfying the constraint ("generate-and-test"); this is called labeling.

G. Labeling Domain Variables

Let us define a small problem: x is an integer in the range 1–5, y is an integer in the range 1–10, x is even, $y = 3x$. It is trivial that there exists only one solution: $x = 2$, $y = 6$.

The clp(FD)-program produces this solution:

```
?-A in 1..5, B in 1..10, A mod 2 #= 0, B#=3*A.
  A = 2, B = 6.
```

However, if we now enlarge the domain, e.g.,

```
?-A in 1..10, B in 1..20, A mod 2 #= 0, B#=3*A.
  A in 2..6, B in 6..18.
```

the result is now an answer constraint, not a single value for each variable. In addition, not all values of the interval are valid, since x has to be even. To get single values, one has to bind the domain variables to valid values. This process is called labeling, and in clp(FD) the predicate "labeling(OptList, List)" is used. "List" is the list of domain variables, "OptsList" is a list of atoms or

predicates which control the order in which the domain variables are labeled. If this list is empty, the default labeling strategies are used.

1. Little Example

```
/* A little constraint problem */
little_problem(L) :-
  L=[A,B],
  A in 1..10,
  B in 1..20,
  A mod 2 #= 0,
  B#=3*A.

solution(L) :-
  little_problem(L),
  labeling([],L).

?- little_problem(L).
   L = [_A,_B],
   _A in 2..6,
   _B in 6..18.

?- solution(L).
   L = [2,6];
   L = [4,12];
   L = [6,18].
```

2. Controlling the Labeling Process

The first parameter of "labeling(OptList, List)" is a list of atoms or predicates which control the order in which the domain variables are labeled.

`leftmost`: The leftmost variable is selected first. This is the default value.
`min`: The leftmost variable with the smallest bound is selected.
`max`: The leftmost variable with the greatest upper bound is selected.
`ff`: The first-fail principle is used. The leftmost variable with the smallest domain is selected.
`ffc`: The variable with the smallest domain is selected, breaking ties by selecting the variable with the most constraints suspended on it.
`variable(Sel)`: Provides the programmer with direct control on how the next domain variable is selected.

Furthermore, there are atoms which control the way the integer for each domain variable is selected.

`step`: Chooses the upper or the lower bound of a domain variable first. This is the default.
`enum`: Chooses multiple integers for the domain variable.
`bisect`: Uses domain splitting to make the choices for each domain variable.
`Value(Enum)`: Enum is a predicate provided by the programmer to narrow the choices for each domain variables.

The next atoms define which integer is selected for each variable.

`up`: Explores the domain in ascending order. This is the default.
`down`: Explores the domain in descending order.

The next atoms control whether all solutions should be enumerated or a single solution that minimizes or maximizes a domain variable is returned.

`all`: All solutions are generated. This is the default.
`minimize(X)`: Search for a solution that minimizes the domain variable X.
`maximize(X)`: Search for a solution that maximizes the domain variable X.

The last option is an atom that binds a variable to the number of assumptions made to find that solution.

`assumption(K)`: When a solution is found, K is bound to the number of choices made to find it.

For example, to produce the results in reverse order, one has to select the option "down":

```
?- little_problem(L), labeling([down],L).
   L = [6,18];
   L = [4,12];
   L = [2,6].
```

3. Rectangle with Maximum Area

A small problem from geometry: Find the dimensions of the rectangle of greatest area for a given circumference. The circumference is given as 40 m, only integer solutions are valid. The clp(FD) program is simple:

```
/* Rectangle with maximum area
   for given circumference */

rectangle(Length,Width,Area) :-
  Length in 0 ..  20,
  Width in  0 ..  20,
  Length+Width #= 20,
  Area #= Length*Width.

maximum(Length,Width,Area) :-
  rectangle(Length,Width,Area),
  labeling([maximize(Area)],
           [Length,Width,Area]).

?- maximum(Length,Width,Area).

Area   = 100,
Width  =  10,
Length =  10.
```

H. Send + More = Money

The `send + more = money` problem is a well-known puzzle often used in the literature. A young, dynamic student is short on money. He also knows that his father will not easily give him the extra cash he needs, but under certain circumstances his father may yield the money. He knows that his father loves to solve riddles, especially numerical ones. So he comes up with the following message to his father:

```
      SEND
  +   MORE
  =  MONEY
```

Each letter represents a different digit ranging from zero to nine, and the digits build numbers which in turn build the above-mentioned mathematical exercise.

Gathering additional information. Each letter has a domain with a lower bound of zero and an upper bound of nine. Each domain variable is different from all others. We know that a number does not start with the digit 0. This narrows the domains for the variables `S` and `M`.

An arithmetic constraint for all letters which represents the expression `SEND + MORE = MONEY` must be formulated.

The predicate "sum(List)" builds the arithmetical constraint over the domain variables. It reflects the expression "SEND + MORE = MONEY." The List variable is a list of the domain variables.

```
sum([S,E,N,D,M,O,R,Y]) :-
            1000*S+100*E+10*N+D   %      SEND
  +         1000*M+100*O+10*R+E   % +    MORE
  #= 10000*M+1000*O+100*N+10*E+Y.  % =   MONEY
```

The predicate `riddle(Solution,Variables)` defines the puzzle. The purpose of the first parameter is only to improve the readability of the result (without this it would be difficult to recognize which variable assumes which value):

```
riddle(Solution,Variables) :-
    % A little trick (readability!)
  Solution=[s:S,e:E,n:N,d:D,m:M,o:O,r:R,y:Y],
    % The list of the domain variables:
  Variables=[S,E,N,D,M,O,R,Y],
    % Defines the domains for the variables:
  domain(Variables,0,9),
    % We know, M and S can not be 0!
  M #> 0,  S #> 0,
    % Different letters, different numbers!
all_different(Variables),
    % And finally, the arithmetic constraint:
  sum(Variables).
```

If we now pose the question

```
?- riddle(Solution,Variables).
```

we get the following answer:

```
Solution=[s:9,e:_A,n:_B,d:_C,m:1,o:0,r:_D,y:_E],
Variables=[9,_A,_B,_C,1,0,_D,_E],
_A in 4..7,
_B in 5..8,
_C in 2..8,
_D in 2..8,
_E in 2..8.
```

Three of the eight variables are found ("s" = 9, "m" = 1, "o" = 0), "e" is narrowed down to the interval [4..7], "n" to [5..8]. For the remaining variables, the clp(FD)-solver finds no additional information besides that the values 0, 1, and 9 are excluded. This is also the information an experienced puzzle solver would immediately recognize. To find the complete solution, we have to use the labeling predicate:

```
?- riddle(Solution, Variables),
   labeling([],Variables).

Solution = [s=9,e=5,n=6,d=7,m=1,o=0,r=8,y=2],
Variables = [9,5,6,7,1,0,8,2].
```

Efficient implementations of constraint solvers are available, e.g, the clp(FD) solver implemented in Sictus Prolog.

SEE ALSO THE FOLLOWING ARTICLES

ARTIFICIAL INTELLIGENCE • BASIC PROGRAMMING LANGUAGE • C AND C++ PROGRAMMING LANGUAGE • LINEAR SYSTEMS OF EQUATIONS • SOFTWARE ENGINEERING • SOFTWARE TESTING

BIBLIOGRAPHY

Barták, R. (1999). "Constraint Programming: In Pursuit of the Holy Grail." *In* "Proceedings of WDS99, Prague."

Campbell, J. A. (ed.). (1984). "Implementations of Prolog," Wiley, New York.

Carlsson, M., Ottoson, G., and Carlson, B. (1997). "An open-ended finite domain constraint solver." *In* "Proceedings Programming Languages: Implementations, Logics, and Programs," Springer-Verlag, New York.

Clark, K. L., and Tarnlund, S.-A. (eds.). (1982). "Logic Programming," Academic Press, New York.

Clocksin, W. F., and Mellish, C. S. (1984). "Programming in Prolog," 2nd ed. Springer-Verlag, New York.

Hogger, C. J. (1984). "Introduction to Logic Programming," Academic Press, Orlando, FL.

ISO/IEC 13211 (1995). "Prolog International Standard," ISO, Geneva.
Kowalski, R. A. (1979). "Logic for Problem Solving," Elsevier North-Holland, Amsterdam.
Jaffar, J., and Lassez, J. L. (1987). "Constraint Logic Programming." *In* "Proceedings Conference on Principles of Programming Languages," ACM, Munich.
Lloyd, J. W. (1984). "Foundations of Logic Programming," Springer-Verlag, New York.
Pereira, F. C. N., and Warren, D. H. D. (1980). "Definite clause grammars for language analysis," *Artif. Intell*. **13,** 231–278.
Sterling, L., and Shapiro, E. (1986). "The Art of Prolog," MIT Press, Cambridge, MA.
Swedish Institute of Computer Science (2000). "SICStus Prolog User's Manual," Kista, Sweden.

Real-Time Systems

A. Burns
University of York

GLOSSARY

Aperiodic process A process that is released at nonregular time intervals.
Deadline The time by which a process must complete its execution.
Event-triggered The triggering of an activity by the arrival of an event.
Jitter Variability in the completion (or release) time of repeating processes.
Periodic process A process that is released at regular time intervals.
Real-time systems A computing system with explicit timing requirements that must be satisfied for correct execution.
Response time analysis A form of schdulability analysis that computes the worst-case completion time for a process.
Scheduling The scheme by which resource are allocated to processes.
Sporadic process An aperiodic process where consecutive requests are separated by a minimum interarrival time.
Temporal firewall A partitioning of a distributed system such that an error in the time domain in one part of the system does not induce timing error elsewhere.
Time-triggered The triggering of an activity by the passage of time.

THE EVENT-TRIGGERED (ET) model of computation is presented as a generalization of the time-triggered (TT) approach. It supports hard real-time and flexible soft real-time services. The ET model is built upon a number of key notions: temporal firewalls, controlled objects, temporarily valid state data, and unidirectional communications between isolated subsystems. It uses the producer/consumer rather than client/server model of interaction. In addition to describing a systems model and computation model, this article considers issues of schedulability and fault tolerance. The ET model is not radically different from the TT approach (as in many systems most events will originate from clocks) but it does provide a more appropriate architecture for open adaptive applications.

I. INTRODUCTION

At the 1998 Real-Time Systems Symposium, Kopetz (1998) presented a cogent argument for the use of a time-triggered model of computation for hard real-time systems. In this model, all computation and communication activities are controlled by the passage of time, i.e., they are triggered by a clock. His argument in support of a time-triggered approach is based upon the following:

- A rejection of the client-server model of computation
- The need to emphasize the real-time properties of data
- The need to support the partitioning of systems with temporal firewalls and effective interfaces

In this article we also support these observations but provide a more general computation model. Again, all activities are released by events, but these events can either be generated by a clock, the system's environment, or internally. However, all event sources that are not directly linked to a clock can have temporal constraints imposed or derived. By this means, hard real-time systems can be supported.

The debate between proponents of time-triggered and event-triggered models of computation has been a lively and at times contentious one (Kopetz and Verissimo, 1993; Kopetz, 1995). In this article we argue that the event-triggered model is more general; it subsumes the time-triggered approach. Hence the event-triggered model is not an alternative to the time-triggered one, but, rather, is a controlled means of increasing the flexibility of a purely clock-driven architecture. With an event-triggered approach it is possible, for example, to support a system in which all but a small number of events are sourced from clocks. However, it also allows a system to deal with uncertainty in its environment.

Although it is as predictable, the drawback of using events to control computation is that it leads to variability in temporal behavior. This variability can be bounded but can cause significant output jitter, which may be an issue for certain control applications. Means of minimizing output jitter will be addressed later in this article. Note, however, that time-triggered systems can also exhibit jitter (due to variations in execution time) especially if they employ a cyclic executive for implementation. It should also be remembered that jitter is not always detrimental; it allows systems to be more versatile. Event-triggered systems can go as quickly as possible, finish as quickly as possible, and hence deliver services as quickly as possible. This is a significant advantage for many applications.

The need to provide effective models of computation for hard real-time applications has been intensified by the advent of FieldBus technology in the process control domain, distributed microcomputer-based systems for body-electronics in automobiles, and the integrated modular avionics (IMA) effort in aerospace. A wide range of high-integrity applications, from railway signaling to nuclear power-plant control, are looking to make use of generic distributed computing architectures that facilitate composability, analysis, fault tolerance, and effective performance. The key to engineering these generic architectures is to make use of an appropriate computational model. For simple systems, the time-triggered architecture is one such model. For more dynamic and open applications the event-triggered model described in this article is necessary.

The motivation for the event-triggered model is presented in the next section, where a systems model is proposed. Section III then gives the computational model and Section IV an implementation model. Approaches to fault tolerance are presented in Section V and conclusions are developed in Section VI.

II. SYSTEM MODEL

All real-time systems monitor and most at least partially control the environment in which they execute. The computer system is said to interact with *controlled objects* (COs) in its environment. These objects may represent a vessel's temperature, the position of a rotator, the setting of a switch, etc. All have a temporal dimension in that the passage of sufficient time will cause their states to change. However, controlled objects can clearly be categorized into two distinct behavioral types: those that exhibit continuous changes and those that exhibit discrete changes. Temperature, pressure, airflow, chemical concentration, and rotor position are all continuous variables; switch positions, operator input, electrical flow in earth wires, and proximity sensors are examples of discrete controlled objects.

Linked to each external controlled object is an entity within the real-time computer system that shadows its behavior. The real-time state variable (RTSV) must always be sufficiently up to date for the control functions of the system to be discharged. This applies to output as well as input activities. Having computed the required valve setting, for example, there is a temporal constraint on the time that can be taken before the actual physical valve has taken up that position. The use of COs and RTSVs in the event-triggered model is identical to that in the time-triggered approach (Kopetz, 1998).

For controlled objects that are subject to continuous change, the relationship between the CO and its RTSV is a straightforward one. The required measurement variable (e.g., temperature) is communicated as an electrical signal

to an analog-to-digital converter (ADC) device at the interface to the computer systems. The device converts the voltage level to a digital signature representing its value. By polling the register holding this value, the RTSV can obtain the effective temperature reading. It is reliable if the sensor is reliable and is up to date apart from the time it takes for the reading to become stable in the ADC. Of course the value in the RTSV will immediately start to age and will eventually become stale if a new value is not obtained.

The rate of change of the controlled object will dictate the rate of polling at the interface. Polling is usually a time-triggered activity and hence time-triggered architectures are really only focused on systems with a fixed set of continuously changing external objects. A typical controlled object of this type will give rise to a single sampling rate. Others will execute in different phases and will ideally be supported by time-varying polling rates. Interestingly, even for this essentially time-triggered arrangement the interface to the ADC is often event-triggered. The software sends a signal to the device to initiate an input action from a particular input channel. It then waits a predefined time interval before taking the reading.

For external objects that exhibit discrete changes of state there are two means of informing the computer system of the state change. Either the state change event is made available at the interface, or the interface just has the state value. In the former case, the computer system can directly deal with the event; in the latter case polling must again be used to "check" the current state of the external object. A time-triggered system must use the polling approach.

Note that with applications that have discrete-change external objects, events are a reality of the system. The computer system may choose to be only time-triggered by restricting its interface, but at the system level it is necessary to deal with the temporal consequences of events. The real world cannot be forced to be time-triggered. So the real issue is when to transform an event into state information. A time-triggered approach always does it at the interface, an event-triggered model allows an event to penetrate the system and to be transformed into a state only when it is most appropriate. Hence, the event can have an impact on the internal scheduling decisions affecting computation. Indeed the event could have a priority that will directly influence the time at which the state change occurs and the consequences of this change.

For the computer system designer, the decision about the interface (whether to accept events or to be just state based) is often one that is strongly influenced by the temporal characteristics of the external object. It depends on the rarity of the event and the deadline on dealing with the consequences of the event. To poll for a state change that only occurs infrequently is very inefficient. The air-bag controller in an automobile provides an extreme example of this. The deadline for deploying an air-bag is 10 msec. It would therefore be necessary to poll for the state change every 5 msec and complete the necessary computations within 5 msec. If 1 msec of computation is needed to check deployment, then 20% of the processing resource must be dedicated to this activity. Alternatively, an event-trigger response will have a deadline of 10 msec, the same computational cost of 1 msec, but negligible processor utilization as the application only requires one such event ever to be dealt with. Clearly the event-triggered approach is the more efficient.

A. Interface Definition

It is imperative that the interface between the computer-based control system and the environment it is controlling is well defined and appropriately constrained. This is one of the attractive features of the time-triggered architecture and is equally important for the event-triggered (ET) approach. An ET interface consists of state data and event ports. State data are written by one side of the interface and read by the other. Concurrent reads and writes are noninterfering. The temporal validity of the data is known (or can be ascertained) by the reading activity. This may require an associated time-stamp but may alternatively be a static property that can be validated prior to execution. An event port allows an event to pass through the interface. It is unidirectional. The occurrence of an event at the system interface will result in a computational process being released for execution. The effect of an output event into the computer system's environment cannot be defined; it is application-specific.

Although event ports are allowed in the interface, it remains state-based as events must carry an implicit state with them. So, for example, a switch device will not give rise to a single event "Change" but will interface with its state variable via two events: "SwitchUp" and "SwitchDown."

In order to manage the impact of incoming events the computer system must be able to control its responses, either by bounding the arrival patters or by only meeting temporal constraints that are load sensitive. A bound on the input traffic may again be a static property requiring no run-time monitoring. Alternatively, an interface may have the capability to disable the event port for periods of time and thereby impose rate control over the event source (see Section V.A). Flexible temporal constraints can take the form of load-sensitive deadlines (e.g., the greater the concentration of events, the longer the deadline requirement on dealing with each event) or event rationing. In the latter case an overloaded system will drop certain events; it is therefore crucial that such events be

state-based. A time-triggered system has no such robustness when it comes to unforeseen dynamics.

The above discussion has concentrated on the computer system and its environment. Equally important are the interfaces between distinct subsystems of the computer system itself. On a distributed platform it is useful to construct "temporal firewalls" (Kopetz and Nossal, 1997) around each node or location with local autonomy. An ET interface with its state and event port definition is therefore required at the boundary of any two subsystems.

The existence of event ports as well as state data would appear to make the ET interface more complicated than its TT equivalent. However, Kopetz (1998) recognised that to meet some timing requirements it is necessary to coordinate the execution of linked subsystems. He terms these interfaces *phase-sensitive*. To acquire this sensitivity, the time-triggered schedules in each subsystem must be coordinated, which therefore necessitates a common time base. That is, some notion of global time must exist in the system. With an ET interface, this coordination can be achieved though the events themselves and hence no global time base is needed for the architecture itself.

Having said that, many applications will require a global time service (although not necessarily of the granularity required by the TT mechanisms), for example, to achieve an ordering of external events in a distributed environment, or to furnish some forms of fault tolerance will require global time (see Section V). Nevertheless, the ET architecture itself is not fundamentally linked to the existence of global time. This is important in facilitating the development of open real-time systems (Stankovic *et al*., 1996).

B. Summary

Our system model recognizes the need to support interfaces to external controlled objects (COs) that either change continuously or in discrete steps. Within the computer system real-time entities (RTSV) will track the behavior of these external objects. A system designer has the choice of making the computer system entirely time-triggered, in which case all COs are polled, or to be event-triggered, in which case continuously COs are polled but discrete COs give rise to events that are directly represented and supported. The decision of which methods to use must take into account the temporal characteristics of the external COs and the structure and reliability of the electronics needed to link each external object to the computer system's interface.

We have defined an ET interface that allows state data and events ports to be used. All communication through an interface is unidirectional. Such interfaces link the computer system to its environment and are used internally to link distinct subsystems. Events themselves give rise to state changes in which the final state of the RTSV is specified (not a delta change).

In the next section we develop the computational model that allows the internal behavior of the computer system (or subsystem) to be specified and analyzed. This will enable the temporal obligation defined in the system's interface to be realized.

III. COMPUTATIONAL MODEL

Within the external interfaces, the behavior of the computer system (or subsystem) is defined and constrained by the computational model. The model allows an application to be described using processes and shared state data. Each process is event-triggered. Once released it may read input state data from the system's interface, write data to different elements of the interface, read and/or write to shared internal data, or generate events for external consumption via the interface or internally to trigger the release of a local process. The shared data, as the name implies, are used for communication between processes. The implementation model must ensure atomic updates to shared data.

The use of the term "process" does not imply a heavyweight construct with memory protection, etc. It merely implies an abstract concurrent activity. Each process experiences a potentially infinite number of invocations (each in response to its event-trigger). The event may originate from the clock, the interface, or another process. If the clock event is regular, then the process is termed *periodic*.

The software architecture that links the processes together is best described as a set of *producer–consumer* relations. The more general *client–server* model of interaction is not well suited to real-time systems (Kopetz, 1998). In particular the server is not easily made aware of the temporal requirements of the client. With the producer–consumer model, data flow and control flow (via event chains releasing a sequence of precedence related processes) allows an end-to-end temporal requirement to be specified and subsequently verified.

The temporal requirements themselves are expressed as attributes of the components of the application model. But there is an interesting duality between a data-oriented focus or a process-oriented focus to these specifications:

- *Data-oriented focus*. Here the state variables are decorated with the temporal attributes. These take the form of either absolute or relative temporal constraints (Audsley *et al*., 1995) over the allowed age of the data. For example, data X must never be older than 100 msec; or following event E, data Y must be

updated within 50 msec. By fully specifying the required properties of all events and state variables at the interface, a complete specification of the required temporal behavior of the system is achieved.
- *Process-oriented focus*. Here the processes have deadlines relating their completion to the event that caused their invocation, for example, the process must be completed within 30 msec. Deadlines can also be attached to events during execution, for example, between invocation and the writing to shared data (in general between the invocation event and any event being initiated by the process). If a process is periodic, then the period of invocation must also be defined.

The duality of these two approaches comes from the relationships between the attributes. If there is a data focus, then one can derive the necessary period and deadlines of the processes. Similarly, if all processes have deadlines (and periods where necessary), then the absolute or relative temporal constraints of the state data can be obtained. For example, a periodic process with period 20 msec and deadline on a write event of 10 msec will sustain an absolute temporal constraint of 30 msec. However, a temporal constraint of 30 msec can be met by a number of process attributes, for example, period 15 deadline 15, period 20 deadline 10, period 25 deadline 5; even period 10 deadline 20. The choice of these attributes can be used to increase schedulability and flexibility (Burns *et al*., 1994; Burns and Davis, 1996).

In a more complex example there may be a relative temporal constraint between an event at the interface and an output state variable also at the interface. The computations necessary to construct this state value may involve a number of processes. Nevertheless, the combined deadlines of the processes involved must be sufficient to satisfy the required constraint.

With either a data focus or a process focus (or a combination of the two) all the necessary constraints of hard real-time systems can be specified. In addition, soft and firm constraints with run-time tradeoffs can also be articulated. Also "value" or "utility" attributes can be added to allow resource (e.g., CPU) overflow and underflow management. Tradeoffs can be made between a choice of algorithm and deadline. Indeed all the flexible and adaptive schemes associated with the term "imprecise computation" can be specified (Liu *et al*., 1991).

Even if all events originate from clocks it is still possible to construct more flexible and dynamic systems than can be achieved with a time-triggered architecture where all processes are periodic and periods are statically fixed (Burns and McDermid, 1994). For example, a "periodic" process can choose its next time trigger based on the current state of the system. Although many time-triggered systems can accommodate some flexibility by the use of modes (with different schedules within each mode) this is very limited when compared with what is possible in event-triggered scheduling.

A. Communication Model

A key aspect of the computational model is the communication facilities that links the interfaces of the various subsystems of the distributed real-time system. The classic *remote procedure call* that is at the heart of the client–server paradigm is not an ideal real-time protocol. It requires the caller to wait, which induces a variable suspension time, and the called object will need a surrogate process to manage the call. This invokes a scheduling action on the server. Although in the functional domain this interaction is straightforward, for a real-time system it is problematic. The time taken by the surrogate process will depend on the number of external calls and local scheduling events. The delay experienced by the client will depend on the time taken by the surrogate process, and hence there is a circular set of temporal dependences that makes it difficult/impossible to undertake effective timing analysis.

With the producer–consumer model no surrogate processes are needed as only data and events are been communicated. A single unidirectional message send facility is all that is required.[1] The role of the communication system is to transfer state data between subsystem interfaces and to transport events from one subsystem to another. At the receiver, data need to be placed in the correct interface object (conceptually via dual port memory between the communication controller and the host processor) and events need to invoke application processes (typically via an interrupt mechanism).

As the only communication protocol needed is a simple unidirectional broadcast, a number of simple schemes are possible (Tindell *et al*., 1995). For example, the CAN protocol is particularly suited to real-time communication (Tindell *et al*., 1994b). Many of these schemes can be easily analyzed for their worst-case temporal characteristics. Together with the analysis of the processing units this enables complete systems to be verified (at least in the absence of faults; see Section V).

B. Formal Modeling and Verification

The computational model described in this section is similar to that in a number of structural design methods such as MASCOT (Simpson, 1986) and HRT-HOOD (Burns

[1]To facilitate multiple clients for the data, a broadcast or multicast protocol is normally used.

and Wellings, 1994). It is also possible to define its semantics formally. Events are assumed to be instantaneous (i.e., have no duration), whereas processing activities must always have duration. Time could be modeled as a continuous entity or as a discrete notion (this depends on the form of verification that will be applied). The notation RTL (Jahanian and Mok, 1986) exactly fits this requirement, as do the modeling techniques represented by timed automata (Alur and Dill, 1994).

Although proof-theoretic forms of verification are powerful, they require theorem provers and a high level of skill on behalf of the user. They have also yet to be used on large temporal systems. Model-theoretic schemes are, however, proving to be effective. For example, the combination of model checking and timed automata in tools such as Uppaal (Larsen *et al*., 1995, 1997) does allow verification to be applied to nontrivial applications. Although state explosion remains a problem with large systems, the ability to break a system down into effectively isolated zones (using temporal firewalls) does allow a composable approach.

The verification activity links the attributes of the application code, for example, periods and deadlines with the requirements of the interface. It does not prove that an implementation is correct; it does, however, allow the correctness of the model to be explored. If the subsequent implementation is able to support the process periods and deadlines, then the system will work correctly in the temporal domain.

This form of verification is required for hard systems. For soft or firm components the model can be exercised to determine its average performance or other behavior metrics. The ability to combine worst-case and average-case behavior is a key aspect of the computational model.

C. Summary

The event-triggered model of computation introduced in this section consists of the following:

- Processes, which are released for execution by invocation events; they can read/write state variables and can produce events
- Invocation events, which originate from a local clock, the system/subsystem interface, or another local process
- State variables, which hold temporally valid data corresponding to external entities, interface entities, or shared data between local processes
- Interfaces, which provide a temporal firewall and contain state variables and event ports
- A communication system, which provides a unidirectional broadcast facility for linking subsystem interfaces
- Attributes, which capture the temporal requirement of the state variables, processes, and communications.

Together these provide the mechanisms necessary to produce distributed real-time systems that are predictable but also have the flexibility needed to provide adaptive value-added services.

IV. SCHEDULING MODELS

A. Introduction

A scheduling model describes how resources are allocated to the components of the computation model. It also facilitates the analysis of processes that adhere to the resource allocation scheme. Within a temporal firewall a fixed set of hard processes will have well-defined temporal characteristics. At the simplest level these will be as follows:

- *Maximum invocation frequency.* Usually given as the minimum time between any two releases and denoted by T. For a periodic process, T is the period. For other event-triggered processes it is the shortest time between any two invocation events.
- *Deadline.* The time by which the process must complete its execution, or have completed some key aspect of its computation. Usually denoted by D and measured relative to the process's release time.

The implemented code will also give rise to a measured or estimated worst-case computation time, C.

More complex applications may have values of T, D, and C that are variable and interdependent.

There are a number of applicable scheduling schemes that are predictable and use resources effectively. Fixed-priority scheduling has been the focus of considerable research activity over the last decade. Initial work centered upon rate monotonic scheduling (Liu and Layland, 1973), where each process is required to be completed before its next release. In effect this implies that each process τ_i has a deadline D_i equal to its period T_i. More recent work has considered systems with $D_i < T_i$ (Leung and Whitehead, 1982) or $D_i > T_i$ (Lehoczky, 1990; Tindell *et al*., 1994b). In the former case a simple algorithm provides an optimal priority assignment: process priorities are ordered according to deadline; the shorter the deadline, the higher the priority. When processes have $D_i > T_i$ a more complex priority assignment scheme is needed (Audsley *et al*., 1993c).

The work of Joseph and Pandya (1986) and Audsley *et al.* (1993a, b) gives the following equation for the

worst-case response time of a given task τ_i (for $R_i \leq T_i$):

$$R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j, \quad (1)$$

where hp(i) is the set of all tasks of higher priority than task τ_i.

Equation (1) is solved by forming a recurrence relation. Having calculated the worst-case response time, it is trivial to check that $R_i \leq D_i$.

The value of this *response time analysis* (RTA) is that is can be easily modified to accommodate a variety of application characteristics and platform constraints (Burns, 1994), including irregular release patterns, phased executions (i.e., offsets), processes with multiple deadlines, processes with forward or backward error recovery, soft and firm deadlines, aperiodic processes, etc. The necessary mutual exclusive protection of shared state data is provided by priority ceiling protocols.

Of course, in addition to fixed priority scheduling there are a number of dynamic approaches, earliest deadline first (EDF) being the most widely discussed.

B. Jitter Control

As indicated in the introduction, one of the drawbacks of using events is that it induces variations in the time of execution. This can result in poor schedulability and excessive input and output jitter with the environment. Many control applications require regular output, and signal processing routines work best if data are obtained with precise periods.

Within fixed-priority scheduling, the processes with the highest priorities suffer the least jitter, and processes with the shortest deadlines are those that are assigned the top priorities. It follows therefore that the actual I/O activity should be placed in a process that does not contain other computations (and hence has low C) and is given a deadline equal to the maximum jitter that is allowed. The rest of computation is placed in a process with a longer deadline. For example, consider a 50 msec periodic activity that has an output jitter requirement of 5 msec. The activity is realized by two processes, each with $T = 50$. The main process has $D = 45$; the I/O process has an offset of 45 and $D = 5$. The ability to control jitter in this way has been demonstrated on safety-critical control systems (Bate *et al.*, 1996).

V. APPROACHES TO FAULT TOLERANCE

Most real-time systems also have high-integrity requirements and hence need to support various forms of error detection and fault tolerance. In this section we consider some of the schemes that can be built upon the event-triggered model of computation. Three issues are considered: how a system (or subsystem) can protect itself against an overload of event occurrences, how to build a fault-tolerant system when there exists a global time base, and how to provide fault-tolerant services when there is no such time base.

A. Protection against Event Overloads

The notion of a temporal firewall is built on the assumption that the external environment of a system or subsystem cannot induce a failure in the temporal domain. With a state-only interface this is straightforward, although coordinated executions between subsystems requires a global time base. Event ports, however, introduce the possibility that assumptions about the frequency of events may be violated at run-time. If static analysis is not adequate, then the source and/or the destination of events must be monitored and controlled.

To control exports, a subsystem must monitor its event production and be able to recognize overproduction, a phenomena known as *babbling*. The simplest action to take on recognizing this error is to close the subsystem down. Action at a system level via, for example, replication will then be needed if availability is to be preserved.

To control imports requires the ability to close an event port (e.g., disable interrupts) or to internally drop events so that an upper bound on the event traffic is guaranteed.

Combining export and import controls facilitates the production of fault containment regions, at least for temporal faults.

B. Fault Tolerance with Global Time

If a global time base is available (either directly or via a protocol that bounds relative drift between the set of system clocks), then it is relatively straightforward to build fault-tolerant services on either the time-triggered or event-triggered models.

The first requirement is that the broadcast facility provided by the communication system is actually atomic. Either all receivers get a message (embodying state data or an event) or none do; moreover, they get the message only once. The ability to support atomic broadcast with global time (to the required level of integrity) is well illustrated by the time-triggered approach. However, it is also possible to provide an atomic broadcast service on top of an event-triggered communication media that does not have a global time base. For example, it has been shown that the CAN protocol can support atomic broadcast (Ruffino *et al.*, 1998; Proenza and Miro-Julia, 1999).

The second requirement, if active replication is required, is for replicated subsystems to exhibit replica determinism. If active replication is not required, i.e., faults can be assumed to give rise to only fail-stop behavior and there is time to execute recovery, then replica determinism is not necessary and cold or warm standbys can be used to increase availability.

Replica determinism requires all nonfaulty replicas to exhibit identical external behavior (so that voting can be used to identify faulty replicas). With the producer/consumer model, divergent behavior can only occur if replicated consumers do not read the same data from single or replicated producers.[2]

The use of lock-stepped processors will furnish replica determinism but it is inflexible. If replicated processes are allowed to be more free-running, then different levels of code replication can be achieved and indeed replicated and nonreplicated software can coexist. If replicated consumers can execute at different times (though within bounds), it is possible for an early execution to "miss" data that a later execution will obtain. To prevent this, a simple protocol can be used based on the following (Poledna *et al*., 2000):

- Data are state-based, i.e., a read operation does not block, as there is always something to read.
- Data are time-stamped with the time when the data become valid.
- Clocks are bounded in terms of their relative drift by an amount d.

So, for example, if process P produces the data and process C consumes it and they are both replicated, then the data become valid at time $L + R$, where R is the longest response time of the replicated Ps and L is the common release time of the Ps. The data can be read by the replicas of C if the release time of all the Cs is earlier than $L + R + d$. Otherwise an older copy of the data must be used. A full set of algorithms for different schemas is presented by Poledna *et al*. (2000). All algorithms are based on *simulated common knowledge* that is known or can be calculated statistically.

C. Fault Tolerance without Global Time

In the event-triggered model of compilation, messages are used to communicate events and states between subsystems that are protected by their temporal firewalls. Within such a firewall a central time base is assumed (i.e., is required). However, it may be that the subsystems themselves do not share a single time source or implement a global time-base via some clock synchronization protocol. As communications can never be perfect, this presents a difficulty as it is not obvious how to distinguish between a message that has failed and one that is late.

Between the two extremes of synchronous systems (i.e., global time-base and easy recognition of a failure) and asynchronous systems (i.e., no assumptions about time and hence no means of easily recognizing failure) is the *timed asynchronous system* model developed by Fetzer and Cristian (1996a, b; Cristian and Fetzer, 1998).[3] In practice, most practical distributed systems can be described as timed asynchronous. The model can be used to build an atomic broadcast facility and indeed fail-aware clock synchronization. The timed asynchronous model is built on the following assumptions:

- Each processing node has a hardware clock with bounded, known, drift; high-integrity applications can use duplicated local clocks to achieve the required reliability for this device.
- Regular traffic is passed between nodes; a couple of messages per second is sufficient for the protocol to work.

From the communication media, two time values are derived:

- $\delta_{\min}$, the minimum transmission time for any message; a value of zero is valid but nonzero values make the protocol more efficient.
- Δ, the assured maximum time it will take a message to be transmitted when no failure occurs.

Note that Δ cannot be derived exactly. It is chosen to make the likelihood of a message being delivered within Δ to be suitably high.

The timed asynchronous protocol uses the transmission of local clock values and the bounded drift values to classify all messages (when received) to be either "fast" or "slow." Fast messages are correct messages, i.e., they are never too fast. The protocol guarantees the following:

- All fast messages were delivered in less than Δ.
- All messages that took longer than Δ are "slow."
- No messages are duplicated

Note the protocol recognized that some messages delivered in less than Δ may be designated "slow." This is a performance issue; it may effect the quality of the communication service but not its fundamental behavior.

[2]The direct manipulation of clock values must also be prohibited.

[3]Another approach is to define quasi-synchronous systems (Verissimo and Almeida, 1995; Almeida and Verissimo, 1998).

As all messages are now "fail aware," appropriate fault tolerance mechanisms can be built. Indeed the time-stamping schemes described in the previous section can be accommodated. Necessary hand-shaking protocols also can be built if necessary. In an event-triggered model, the failure of an event message can be catastrophic as the receiver is obviously unaware that the event has not been delivered. The use of an acknowledgment/resend protocol solves this problem as long as the original event message (which is assumed to have failed) is not delivered late and is acted upon. The fail-aware scheme reported here prevents this. Alternatively, if the acknowledgment/resend protocol is inappropriate for scheduling reasons it is possible to use *diffusion*, in which the message is sent a sufficient number of times to "ensure" delivery within the system's fault model.

The advantage with synchronous systems is that the assumption of global time and bounded message communication makes it relative easy to build application services. The disadvantage occurs if the assumption of synchronous behavior breaks down—the upper-level services (and the application) cannot recover. With the timed asynchronous approach, bounded communication is guaranteed if there are no faults, and knowledge of failure is available if there are. The flexibility this provides (in terms of dealing with faults) is best exploited by the flexible computational model furnished by the event-triggered approach.

VI. CONCLUSION

This article has presented the event-triggered model of computation. It is a generalization of the time-triggered approach in that it allows actions to be invoked by the passage of time and by events originating in the system's environment or in other actions. By bounding the occurrences of events, predictable hard real-time systems can be produced. By not requiring an event to be statically mapped on to a predetermined time line, flexible, adaptive, value-added, responsive systems can be defined. For example, a polling process that can dynamically change its rate of execution when the controlled object is close to a critical value is easily supported by the event-triggered model of computation. A statically scheduled time-triggered architecture cannot furnish this flexibility.

As the computational model is supporting real-time applications it is obvious that a local clock (time source) is required in each subsystem, but the event-triggered model does not require global time or a strong synchronous assumption about communication. Figure 1 depicts the architecture implied by the event-triggered approach. Within temporal firewalls processes and state variables are protected from external misuse. Events and state data are communicated between subsystems; the messages embodying these entities are time stamped (with local clock values).

The computational model can be supported by a number of flexible scheduling schemes. We have shown how the

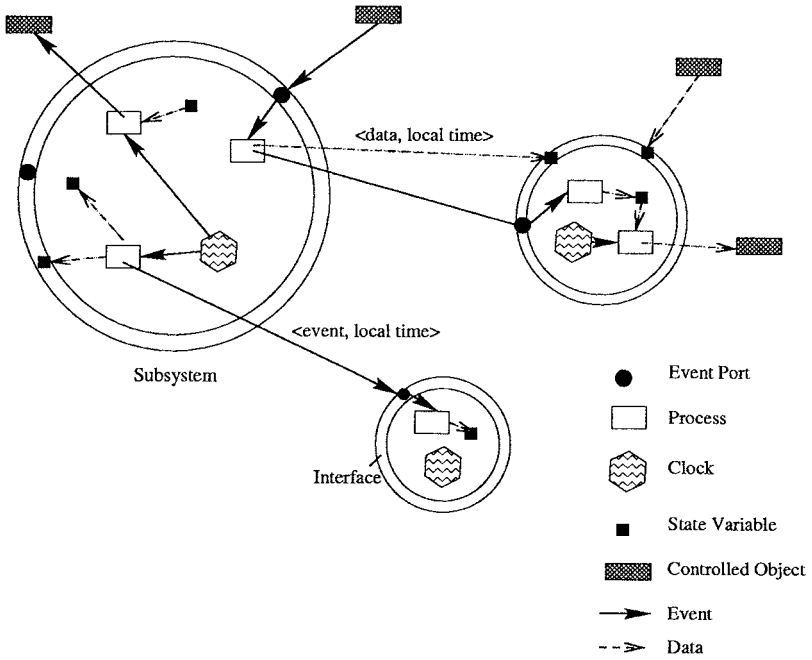


FIGURE 1 Part of an event-trigger system.

fixed-priority approach is well suited to event-triggered computation.

ACKNOWLEDGMENTS

The author would like to express his thanks for useful comments made on an earlier version of this article by Neil Audlsey, Paulo Verissimo, and Andy Wellings.

SEE ALSO THE FOLLOWING ARTICLES

PROCESS CONTROL SYSTEMS • SOFTWARE ENGINEERING • SOFTWARE RELIABILITY

BIBLIOGRAPHY

Almeida, C., and Verissimo, P. (1998). "Using light-weight groups to handle timing failures in quasi-synchronous systems." *In* "Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain," pp. 430–439, IEEE Computer Society Press.

Alur, R., and Dill, D. L. (1994). "A theory of timed automata," *Theor. Computer Sci.* **126**(2), 183–236.

Audsley, N. C., Burns, A., Richardson, M., Tindell, K., and Wellings, A. J. (1993a). "Applying new scheduling theory to static priority preemptive scheduling," *Software Eng. J.* **8**(5), 284–292.

Audsley, N. C., Burns, A., and Wellings, A. J. (1993b). "Deadline monotonic scheduling theory and application," *Control Eng. Pract.* **1**(1), 71–78.

Audsley, N. C., Tindell, K., and Burns, A. (1993c). "The end of the line for static cyclic scheduling?" *In* "Proceedings of the Fifth Euromicro Workshop on Real-Time Systems," pp. 36–41, IEEE Computer Society Press.

Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. (1995). "Data consistency in hard real-time systems," *Informatica* **9**(2), 223–234.

Bate, I. J., Burns, A., and Audsley, N. C. (1996). "Putting fixed priority scheduling theory into engineering practice for safety critical applications." *In* "Proceedings of the 2nd Real-Time Applications Symposium."

Burns, A. (1994). "Preemptive priority based scheduling: An appropriate engineering approach." *In* "Advances in Real-Time Systems" (Son, S. H., ed.), pp. 225–248, Prentice-Hall, Englewood Cliffs, NJ.

Burns, A., and Davis, R. I. (1996). "Choosing task periods to minimise system utilisation in time triggered systems," *Information Processing Lett.* **58,** 223–229.

Burns, A., and McDermid, J. A. (1994). "Real-time safety critical systems: Analysis and synthesis," *Software Eng. J.* **9**(6), 267–281.

Burns, A., and Wellings, A. J. (1994). "HRT-HOOD: A design method for hard real-time ada," *Real-Time Syst.* **6**(1), 73–114.

Burns, A., Tindell, K., and Wellings, A. J. (1994). "Fixed priority scheduling with deadlines prior to completion," *In* "6th Euromicro Workshop on Real-Time Systems, Vaesteraas, Sweden," pp. 138–142.

Cristian, F., and Fetzer, C. (1998). "The timed asynchronous system model." *In* "Digest of Papers, The 28th Annual International Symposium on Fault-Tolerant Computing," pp. 140–149, IEEE Computer Society Press.

Fetzer, C., and Cristian, F. (1996a). "Fail-aware detectors," Technical Report CS96-475, University of California, San Diego, CA.

Fetzer, C., and Cristian, R. (1996b). "Fail-awareness in timed asynchronous systems." *In* "Proceedings of the 15th ACM Symposium on Principles of Distributed Computing," pp. 314–321a.

Jahanian, F., and Mok, A. K. (1986). "Safety analysis of timing properties in real-time systems," *IEEE Trans. Software Eng.* **12**(9), 890–904.

Joseph, M., and Pandya, P. (1986). "Finding response times in a real-time system," *BCS Computer J.* **29**(5), 390–395.

Kopetz, H. (1995). "Why time-triggered architectures will succeed in large hard real-time systems," *Proc. IEEE Future Trends* **1995,** 2–9.

Kopetz, H. (1998). "The time-triggered model of computation." *In* "Proceedings 19th Real-Time Systems Symposium," pp. 168–177.

Kopetz, H., and Nossal, R. (1997). "Temporal firewalls in large distributed real-time systems." *In* "Proceedings IEEE Workshop on Future Trends in Distributed Systems, Tunis."

Kopetz, H., and Verissimo, P. (1993). "Real-time and dependability concepts." *In* "Distributed Systems," 2nd ed. (Mullender, S. J., ed.), Chapter 16, pp. 411–446. Addison-Wesley, Reading, MA.

Larsen, K. G., Pettersson, P., and Yi, W. (1995). "Compositional and symbolic model-checking of real-time systems." *In* "Proceedings of the 16th IEEE Real-Time Systems Symposium," pp. 76–87, IEEE Computer Society Press.

Larsen, K. G., Pettersson, P., and Yi, W. (1997). "Uppaal in a nutshell," *Int. J. Software Tools Technol. Transfer* **1**(1/2), 134–152.

Lehoczky, J. P. (1990). "Fixed priority scheduling of periodic task sets with arbitrary deadlines." *In* "Proceedings 11th Real-Time Systems Symposium," pp. 201–209.

Leung, J. Y.-T., and Whitehead, J. (1982) "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation (Netherlands)* **2**(4), 237–250.

Liu, C. L., and Layland, J. W. (1973). "Scheduling algorithms for multiprogramming in a hard real-time environment," *JACM* **20**(1), 46–61.

Liu, J. W. S., Lin, K. J., Shih, W. K., Yu, A. C. S., Chung, J. Y., and Zhao, W. (1991). "Algorithms for scheduling imprecise computations," *IEEE Computer* **1991,** 58–68.

Poledna, S., Burns, A., Wellings, A. J., and Barrett, P. A. (2000). "Replica determinism and flexible scheduling in hard real-time dependable systems," *IEEE Trans. Computing* **49**(2), 100–111.

Proenza, J., and Miro-Julia, J. (1999). "MajorCAN: A modification to the controller area network protocol to achieve atomic broadcast." *In* "Offered to RTSS99," IEEE Computer Society Press.

Ruffino, J., Verissimo, P., Arroz, G., Almeida, C., and Rodrigues, L. (1998). "Fault-tolerant braodcast in can." *In* "Proceedings of the 28th FTCS, Munich," IEEE Computer Society Press.

Simpson, H. R. (1986). "The mascot method," *Software Eng. J.* **1**(3), 103–120.

Stankovic, J. A., *et al.* (1996). "Real-time and embedded systems," *ACM Surv. Spec. Iss. Strategic Directions Computer Sci. Res.* **28**(4), 751–763.

Tindell, K., Burns, A., and Wellings, A. J. (1994a). "An extendible approach for analysing fixed priority hard real-time tasks," *Real-Time Syst.* **6**(2), 133–151.

Tindell, K. W., Hansson, H., and Wellings, A. J. (1994b). Analysing real-time communications: Controller area network (CAN)." *In* "Proceedings 15th IEEE Real-Time Systems Symposium," pp. 259–265, San Juan, Puerto Rico.

Tindell, K., Burns, A., and Wellings, A. J. (1995). "Analysis of hard real-time communications," *Real-Time Syst.* **7**(9), 147–171.

Verissimo, P., and Almeida, C. (1995). "Quasi-synchronisation: A step away from the traditional fault-tolerant real-time system models," *Bull. Tech. Committee Operating Syst. Appl. Environ. (TCOS)* **7**(4), 35–39.

Requirements Engineering

Bashar Nuseibeh
Open University

Steve Easterbrook
University of Toronto

GLOSSARY

Analysis The process of examining models or specifications for the purposes of identifying their state of correctness, completeness, or consistency.
Elicitation Sometimes also referred to as acquisition or capture, it is the process of gathering requirements about an envisioned system.
Modeling In the context of requirements engineering, it is process of constructing abstract descriptions of the desired structure or behavior of a system and/or its problem domain.
Problem domain The environment in which a desired system will be installed.
Prototyping The process of developing partial models or implementations of an envisioned system, for the purposes of eliciting and validating requirements.
Requirements Describe the world of the problem domain as the stakeholders would like it to be.
Specification A description of the requirements for a system.
Stakeholders Individuals, groups, or organizations who stand to gain or lose from the development of an envisioned system.
Validation The process of establishing whether or not the elicited requirements are indeed those desired by the stakeholders.

I. INTRODUCTION

Pamela Zave provides a concise definition of *Requirements Engineering (RE)*:

"Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families." [Zave]

Simply put, software requirements engineering is the process of discovering the purpose of a software system, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis,

communication, and subsequent implementation. There are a number of inherent difficulties in this process. Stakeholders (who include paying customers, users, and developers) may be numerous and geographically distributed. Their goals may vary and conflict, depending on their perspectives of the environment in which they work and the tasks they wish to accomplish. Their goals may not be explicit or may be difficult to articulate, and, inevitably, satisfaction of these goals may be constrained by a variety of factors outside their control.

The primary measure of success of a software system, and by implication its quality, is the degree to which it meets its requirements. RE therefore plays a pivotal role in determining the quality of a software system and its fitness for purpose. It lies at the heart of software development projects, since the outcomes of an RE process provide crucial inputs into project planning, technical design, and evaluation. Of course, RE is not an activity that only takes place at the start of a software development project only. The discovery of requirements often continues well into the development process, and can be influenced by the presence of existing technical solutions and constraints. Broadly speaking, however, RE is a "front-end" development activity—determining subsequent development decisions and used as a measure of success for the final software system that is delivered.

It is worth noting in passing that software requirements engineering is often regarded as part of *systems engineering*, in that the software to be developed has to fit into a wider technical, social, organizational, and business context. Thus, RE is often also called *software systems requirements engineering*, or simply *systems requirements engineering*, to emphasise that it is a system-wide scope, and not merely a software engineering one. The consequences of this are that RE is very much a multidisciplinary activity, whose techniques draw upon many areas of research and practice. These include traditional areas such as computer science and mathematical logic, systems theory, and information systems analysis, and the cognitive and social sciences such as psychology, anthropology, sociology, and linguistics.

II. OVERVIEW OF REQUIREMENTS ENGINEERING

The process by which software development organizations conduct RE varies from one organization to the next. However, there are a number of core activities that are common. These include requirements elicitation (gathering), specification (modeling), and analysis, which will be elaborated in the next two sections. The term *requirements management* is often used to denote the handling of these activities, although many industrial requirements management tools actually only support the handling of the different kinds of requirements documents generated by the RE process. In other words, these tools have limited elicitation support and analysis capabilities.

Underpinning the above activities are some broader goals for RE. They include communicating, agreeing, and evolving requirements.

A. Communicating Requirements

RE is not only a process of discovering and specifying requirements, it is also a process of facilitating effective communication of these requirements among different stakeholders. Requirements documentation remains one of the primary ways of communicating requirements. The way in which requirements are documented plays an important role in ensuring that they can be read, analyzed, (re)written, and validated.

The focus of requirements documentation research is often on specification languages and notations, with a variety of formal, semiformal, and informal languages suggested for this purpose. From logic to natural language, different languages have been shown to have different expressive and reasoning capabilities. Requirements documentation is also often based on commercial or industrial standards. These standards provide templates that can be used to structure the documentation and trigger information gathering activity. Of course, the templates should not constrain the requirements development activity, except in the context of particular projects with rigid contractual constraints.

Requirements traceability (RT) is another important factor that determines how easy it is to read, navigate, query, and change requirements documentation. RT refers to the ability to describe and follow the life of a requirement in both forward and backward direction; that is, from its origins in the problem world, through its development and specification, to its subsequent deployment and use, and through all periods of ongoing refinement and iteration in any of these phases. RT lies at the heart of requirements management practice in that it can provide a rationale for requirements and is the basis for tools that analyze the consequences and impact of change. Supporting RT in requirements documentation is a means of achieving integrity and completeness of that documentation, and has an important role to play in the management of changing requirements.

B. Agreeing Requirements

As requirements are gathered, maintaining agreement with all stakeholders can be a problem, especially where

stakeholders have divergent goals. *Validation* is the process of establishing that the requirements elicited provide an accurate account of stakeholder actual needs. Explicitly describing the requirements is a necessary precondition not only for validating requirements, but also for resolving conflicts between stakeholders.

Techniques such as inspection and formal analysis tend to concentrate on the coherence of the requirements descriptions, asking questions such as: are they consistent, and are they structurally complete? In contrast, techniques such as prototyping, specification animation, and the use of scenarios are geared toward testing a correspondence with the real world problem. For example, they may ask: have all the aspects of the problem that the stakeholders regard as important been covered? Requirements validation is difficult for two reasons. The first reason is philosophical in nature, and concerns the question of truth and what is knowable. The second reason is social, and concerns the difficulty of reaching agreement among different stakeholders with conflicting goals.

C. Evolving Requirements

Successful software systems always evolve as the environment in which these systems operate changes and stakeholder requirements change. Therefore, *managing change* is a fundamental activity in RE.

Changes to requirements documentation need to be managed. Minimally, this involves providing techniques and tools for configuration management and version control, and exploiting traceability links to monitor and control the impact of changes in different parts of the documentation. Typical changes to requirements specifications include adding or deleting requirements and fixing errors. Requirements are added in response to changing stakeholder needs, or because they were missed in the initial analysis. Requirements are deleted usually only during development, to forestall cost and schedule overruns, a practice sometimes called *requirements scrubbing*.

Managing changing requirements is not only a process of managing documentation, it is also a process of recognizing change through continued requirements elicitation, re-evaluation of risk, and evaluation of systems in their operational environment. In software engineering, it has been demonstrated that focusing change on program code leads to a loss of structure and maintainability. Thus, each proposed change needs to be evaluated in terms of existing requirements and architecture so that the trade-off between the cost and benefit of making a change can be assessed.

Finally, the development of software system *product families* has become an increasingly important form of development activity. For this purpose, there is a need to develop a range of software products that share similar requirements and architectural characteristics, yet differ in certain key requirements. The process of identifying *core requirements* in order to develop architectures that are (a) stable in the presence of change, and (b) flexible enough to be customized and adapted to changing requirements is one of the key research issues in software engineering.

III. ELICITING REQUIREMENTS

The elicitation of requirements is perhaps the activity most often regarded as the first step in the RE process. The term "elicitation" is preferred to "capture," to avoid the suggestion that requirements are out there to be collected simply by asking the right questions. Information gathered during requirements elicitation often has to be interpreted, analyzed, modeled, and validated before the requirements engineer can feel confident that a complete enough set of requirements of a system have been collected. Therefore, requirements elicitation is closely related to other RE activities—to a great extent, the elicitation technique used is driven by the choice of modeling scheme, and vice versa: many modeling schemes imply the use of particular kinds of elicitation techniques.

A. Requirements to Elicit

One of the most important goals of elicitation is to find out what problem needs to be solved, and hence identify system *boundaries*. These boundaries define, at a high level, where the final delivered system will fit into the current operational environment. Identifying and agreeing a system's boundaries affects all subsequent elicitation efforts. The identification of stakeholders and user classes, of goals and tasks, and of scenarios and use cases all depend on how the boundaries are chosen.

Identifying *stakeholders*—individuals or organizations who stand to gain or lose from the success or failure of a system—is also critical. Stakeholders include customers or clients (who pay for the system), developers (who design, construct, and maintain the system), and users (who interact with the system to get their work done). For interactive systems, users play a central role in the elicitation process, as usability can only be defined in terms of the target user population. Users themselves are not homogeneous, and part of the elicitation process is to identify the needs of different user classes, such as novice users, expert users, occasional users, disabled users, and so on.

Goals denote the objectives a system must meet. Eliciting high-level goals early in the development process is crucial. However, goal-oriented requirements elicitation

is an activity that continues as development proceeds, as high-level goals (such as business goals) are refined into lower-level goals (such as technical goals that are eventually operationalized in a system). Eliciting goals focuses the requirements engineer on the problem domain and the needs of the stakeholders, rather than on possible solutions to those problems.

It is often the case that users find it difficult to articulate their requirements. To this end, a requirements engineer can resort to eliciting information about the *tasks* users currently perform and those that they might want to perform. These tasks can often be represented in *use cases* that can be used to describe the outwardly visible interactions of users and systems. More specifically, the requirements engineer may choose a particular path through a use case, a *scenario*, in order to better understand some aspect of using a system.

B. Elicitation Techniques

The choice of elicitation technique depends on the time and resources available to the requirements engineer, and of course, the kind of information that needs to be elicited. We distinguish a number of classes of elicitation technique:

- *Traditional techniques* include a broad class of generic data gathering techniques. These include the use of questionnaires and surveys, interviews, and analysis of existing documentation such as organizational charts, process models or standards, and user or other manuals of existing systems.
- *Group elicitation techniques* aim to foster stakeholder agreement and buy-in, while exploiting team dynamics to elicit a richer understanding of needs. They include brainstorming and focus groups, as well as RAD/JAD workshops (using consensus-building workshops with an unbiased facilitator).
- *Prototyping* has been used for elicitation where there is a great deal of uncertainty about the requirements, or where early feedback from stakeholders is needed. Prototyping can also be readily combined with other techniques, for instance, by using a prototype to provoke discussion in a group elicitation meeting, or as the basis for a questionnaire or think-aloud protocol.
- *Model-driven techniques* provide a specific model of the type of information to be gathered, and use this model to drive the elicitation process. These include goal-based and scenario-based methods.
- *Cognitive techniques* include a series of techniques originally developed for knowledge acquisition for knowledge-based systems. Such techniques include protocol analysis (in which an expert thinks aloud while performing a task, to provide the observer with insights into the cognitive processes used to perform the task), laddering (using probes to elicit structure and content of stakeholder knowledge), card sorting (asking stakeholders to sort cards in groups, each of which has the name of some domain entity), repertory grids (constructing an attribute matrix for entities, by asking stakeholders for attributes applicable to entities and values for cells in each entity).
- *Contextual techniques* emerged in the 1990s as an alternative to both traditional and cognitive techniques. These include the use of ethnographic techniques such as participant observation. They also include ethnomethodology and conversation analysis, both of which apply fine-grained analysis to identify patterns in conversation and interaction.

C. The Elicitation Process

With a plethora of elicitation techniques available to the requirements engineer, some guidance on their use is needed. *Methods* provide one way of delivering such guidance. Each method itself has its strengths and weaknesses, and is normally best suited for use in particular application domains.

Of course, in some circumstances a full-blown method may be neither required nor necessary. Instead, the requirements engineer needs simply to select the appropriate technique or techniques most suitable for the elicitation process in hand. In such situations, technique-selection guidance is more appropriate than a rigid method.

IV. MODELING AND ANALYZING REQUIREMENTS

Modeling—the construction of abstract descriptions that are amenable to interpretation—is a fundamental activity in RE. Models can be used to represent a whole range of products of the RE process. Moreover, many modeling approaches are used as elicitation tools, where the modeling notation and partial models produced are used as drivers to prompt further information gathering. The key question to ask for any modeling approach is "what is it good for?," and the answer should always be in terms of the kind of analysis and reasoning it offers. We discuss below some general categories of RE modeling approaches and give some example techniques under each category. We then list some analysis techniques that can be used to generate useful information from the models produced.

A. Enterprise Modeling

The context of most RE activities and software systems is an *organization* in which development takes place or

in which a system will operate. Enterprise modeling and analysis deals with understanding an organization's structure; the business rules that affect its operation; the goals, tasks, and responsibilities of its constituent members; and the data that it needs, generates, and manipulates.

Enterprise modeling is often used to capture the purpose of a system, by describing the behavior of the organization in which that system will operate. This behaviour can be expressed in terms of organizational objectives or goals and associated tasks and resources. Others prefer to model an enterprise in terms of its business rules, workflows, and the services that it will provide.

Modeling goals is particularly useful in RE. High-level business goals can be refined repeatedly as part of the elicitation process, leading to requirements that can then be operationalized.

B. Data Modeling

Large computer-based systems, especially information systems, use and generate large volumes of information. This information needs to be understood, manipulated, and managed. Careful decisions need to be made about what information the system will need to represent, and how the information held by the system corresponds to the real world phenomena being represented. Data modeling provides the opportunity to address these issues in RE. Traditionally, Entity-Relationship-Attribute (ERA) modeling is used for this type of modeling and analysis. However, object-oriented modeling, using class and object models, are increasingly supplanting ERA techniques.

C. Behavioral Modeling

Modeling requirements often involves modeling the dynamic or functional behavior of stakeholders and systems, both existing and required. The distinction between modeling an existing system and modeling a future system is an important one, and is often blurred by the use of the same modeling techniques for both. Early structured analysis methods suggested that one should start by modeling how the work is currently carried out (the current physical system), analyze this to determine the essential functionality (the current logical system), and finally build a model of how the new system ought to operate (the new logical system). Explicitly constructing all three models may be overkill, but it is nevertheless useful to distinguish which of these is being modeled.

A wide range of modeling methods are available, from structured to object-oriented methods, and from soft to formal methods. These methods provide different levels of precision and are amenable to different kinds of analysis. Models based on formal methods can be difficult to construct, but are also amenable to automated analysis. On the other hand, soft methods provide *rich* representations that nontechnical stakeholders find appealing, but are often difficult to check automatically.

D. Domain Modeling

A significant proportion of the RE process is about developing *domain descriptions*. A model of the domain provides an abstract description of the world in which an envisioned system will operate. Building explicit domain models provides two key advantages: they permit detailed reasoning about (and therefore validation of) what is assumed about the domain, and they provide opportunities for requirements reuse within a domain. Domain-specific models have also been shown to be essential for building automated tools, because they permit tractable reasoning over a closed world model of the system interacting with its environment.

E. Modeling Non-functional Requirements (NFRs)

Non-functional requirements (also known as *quality requirements*) are generally more difficult to express in a measurable way, making them more difficult to analyze. In particular, NFRs tend to be properties of a system as a whole, and hence cannot be verified for individual components. Recent work by both researchers and practitioners has investigated how to model NFRs and to express them in a form that is measurable or testable. There also is a growing body of research concerned with particular kinds of NFRs, such as safety, security, reliability, and usability.

F. Analyzing Requirements Models

A primary benefit of modeling requirements is the opportunity this provides for analyzing them. Analysis techniques that have been investigated in RE include requirements animation, automated reasoning (e.g., analogical and case-based reasoning and knowledge-based critiquing), and consistency checking (e.g., model checking).

V. CONTEXT AND GROUNDWORK

RE is often regarded as a front-end activity in the software systems development process. This is generally true, although it is usually also the case that requirements change during development and evolve after a system has been in operation for some time. Therefore, RE plays an important role in the management of change in software development. Nevertheless, the bulk of the effort of RE does

occur early in the lifetime of a project, motivated by the evidence that requirements errors, such as misunderstood or omitted requirements, are more expensive to fix later in project life cycles.

Before a project can be started, some preparation is needed. In the past, it was often the case that RE methods assumed that RE was performed for a specific customer, who could sign off a requirements specification. However, RE is actually performed in a variety of contexts, including market-driven product development and development for a specific customer with the eventual intention of developing a broader market. The type of product will also affect the choice of method: RE for information systems is very different from RE for embedded control systems, which is different again from RE for generic services such as networking and operating systems.

Furthermore, some assessment of a project's feasibility and associated risks needs to be undertaken, and RE plays a crucial role in making such an assessment. It is often possible to estimate project costs, schedules, and technical feasibility from precise specifications of requirements. It is also important that conflicts between high-level goals of an envisioned system surface early, in order to establish a system's concept of operation and boundaries. Of course, risk should be re-evaluated regularly throughout the development lifetime of a system, since changes in the environment can change the associated development risks.

Groundwork also includes the identification of a suitable process for RE, and the selection of methods and techniques for the various RE activities. We use the term *process* here to denote an instance of a process model, which is an abstract description of how to conduct a collection of activities, describing the behavior of one or more agents and their management of resources. A *technique* prescribes how to perform one particular activity—and, if necessary, how to describe the product of that activity in a particular notation. A *method* provides a prescription for how to perform a collection of activities, focusing on how a related set of techniques can be integrated, and providing guidance on their use.

VI. INTEGRATED REQUIREMENTS ENGINEERING

RE is a multidisciplinary activity, deploying a variety of techniques and tools at different stages of development and for different kinds of application domains. Methods provide a systematic approach to combining different techniques and notations, and *method engineering* plays an important role in designing the RE process to be deployed for a particular problem or domain. Methods provide heuristics and guidelines for the requirements engineer to deploy the appropriate notation or modeling technique at different stages of the process.

To enable effective management of an integrated RE process, automated tool support is essential. Requirements management tools, such as DOORS, Requisite Pro, Cradle, and others, provide capabilities for documenting requirements, managing their change, and integrating them in different ways depending on project needs.

VII. SUMMARY AND CONCLUSIONS

The 1990s saw several important and radical shifts in the understanding of RE. By the early 1990s, RE had emerged as a field of study in its own right, as witnessed by the emergence of two series of international meetings—the IEEE sponsored conference and symposium, held in alternating years, and the establishment of an international journal published by Springer. By the late 1990s, the field had grown enough to support a large number of additional smaller meetings and workshops in various countries.

During this period, we can discern the emergence of three radical new ideas that challenged and overturned the orthodox views of RE. These three ideas are closely interconnected:

- The idea that modeling and analysis cannot be performed adequately in isolation from the organizational and social context in which any new system will have to operate. This view emphasized the use of contextualized enquiry techniques, including ethnomethodology and participant observation.
- The notion that RE should not focus on specifying the functionality of a new system, but instead should concentrate on modeling indicative and optative properties of the environment.[1] Only by describing the environment, and expressing what the new system must achieve in that environment, can we capture the system's purpose, and reason about whether a given design will meet that purpose. This notion has been accompanied by a shift in emphasis away from modeling information flow and system state, and toward modeling stakeholders' goals and scenarios that illustrate how goals are (or can be) achieved.
- The idea that the attempt to build consistent and complete requirements models is futile, and that RE has to take seriously the need to analyze and resolve conflicting requirements, to support stakeholder

[1] *Indicative* descriptions express things that are currently true (and will be true irrespective of the introduction of a new system), while *optative* descriptions express the things that we wish the new system to make true.

negotiation, and to reason with models that contain inconsistencies.

A number of major challenges for RE still remain to be addressed in the years ahead:

1. Development of new techniques for formally modeling and analyzing properties of the environment, as opposed to the behavior of the software. Such techniques must take into account the need to deal with inconsistent, incomplete, and evolving models. We expect such approaches will better support areas where RE has been weak in the past, including the specification of the expectations that a software component has of its environment. This facilitates migration of software components to different software and hardware environments, and the adaptation of products into product families.
2. Bridging the gap between requirements elicitation approaches based on contextual enquiry and more formal specification and analysis techniques. Contextual approaches, such as those based on ethnographic techniques, provide a rich understanding of the organizational context for a new software system, but do not map well onto existing techniques for formally modeling the current and desired properties of problem domains. This includes the incorporation of a wider variety of media, such as video and audio, into behavioral modeling techniques.
3. Richer models for capturing and analyzing NFRs. These are also known as the "ilities" and have defied a clear characterization for decades.
4. Better understanding of the impact of software architectural choices on the prioritization and evolution of requirements. While work in software architectures has concentrated on how to express software architectures and reason about their behavioral properties, there is still an open question about how to analyze what impact a particular architectural choice has on the ability to satisfy current and future requirements, and variations in requirements across a product family.
5. Reuse of requirements models. We expect that in many domains of application, we will see the development of reference models for specifying requirements, so that the effort of developing requirements models from scratch is reduced. This will help move many software projects from being creative design to being normal design, and will facilitate the selection of commercial off-the-shelf (COTS) software.
6. Multidisciplinary training for requirements practitioners. In this paper, we have used the term "requirements engineer" to refer to any development participant who applies the techniques described in the paper to elicit, specify, and analyze requirements. While many organizations do not even employ such a person, the skills that such a person or group should possess is a matter of critical importance. The requirements engineer must possess both the social skills to interact with a variety of stakeholders, including potentially nontechnical customers, and the technical skills to interact with systems designers and developers.

Many delivered systems do not meet their customers' requirements due, at least partly, to ineffective RE. RE is often treated as a time-consuming, bureaucratic, and contractual process. This attitude is changing as RE is increasingly recognized as a critically important activity in any systems engineering process. The novelty of many software applications, the speed with which they need to be developed, and the degree to which they are expected to change all play a role in determining how the systems development process should be conducted. The demand for better, faster, and more usable software systems will continue, and RE will therefore continue to evolve in order to deal with different development scenarios. Therefore, effective RE will continue to play a key role in determining the success or failure of projects, and in determining the quality of systems that are delivered.

ACKNOWLEDGMENTS

The content of this article is drawn from a wide variety of sources, and is structured along the lines of the roadmap by Nuseibeh and Easterbrook (2000). Nuseibeh would like to acknowledge the financial support the UK EPSRC (projects GR/L 55964 and GR/M 38582).

SEE ALSO THE FOLLOWING ARTICLES

HUMAN–COMPUTER INTERACTIONS • PROJECT MANAGEMENT SOFTWARE • SOFTWARE ENGINEERING • SOFTWARE RELIABILITY • SOFTWARE TESTING

BIBLIOGRAPHY

Chung, L., Nixon, B., Yu, E., and Mylopoulos, J. (2000). *Non-Functional Requirements in Software Engineering.* Kluwer Academic, Boston.

Davis, A. (1993). *Software Requirements*: *Objects*, *Functions and States.* Prentice Hall, New york.

Finkelstein, A. (1993). Requirements Engineering: An Overview, *2nd Asia-Pacific Software Engineering Conference (APSEC'93)*, Tokyo, Japan.

Gause, D. C., and Weinberg, G. M. (1989). *Exploring Requirements*: *Quality before Design*, Dorset House.

Goguen, J., and Jirotka, M., eds. (1994). *Requirements Engineering*: *Social and Technical Issues*, Academic Press London.

Graham, I. S. (1998). *Requirements Engineering and Rapid Development*: *A Rigorous*, *Object-Oriented Approach*, Addison-Wesley, Reading, MA.

Jackson, M. (1995). *Software Requirements and Specifications*: *A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, Reading, MA.

Jackson, M. (2001). *Problem Frames*: *Analyzing and Structuring Software Development Problems*, Addison-Wesley, Reading, MA.

Kotonya, G., and Sommerville, I. (1998). *Requirements Engineering*: *Processes and Techniques*, Wiley, New York.

Kovitz, B. L. (1999). *Practical Software Requirements*: *A Manual of Contents & Style.* Manning.

Loucopoulos, P., and Mylopoulos, J., eds. (1996). *Requirements Engineering Journal*, Springer Verlag, Berlin/New York.

Loucopoulos, P., and Karakostas, V. (1995). *System Requirements Engineering*, McGraw-Hill, New York.

Macaulay, L. M. (1996). *Requirements Engineering*, Springer Verlag, Berlin/New York.

Nuseibeh, B., and Easterbrook, S. (2000). Requirements Engineering: A Roadmap, In *ICSE-2000 The Future of Software Engineering* (A. Finkelstein, ed.), ACM Press, New York.

Pohl, K. (1996). *Process-Centered Requirements Engineering*, Research Studies Press.

Robertson, S., and Robertson, J. (1999). *Mastering the Requirements Process.* Addison-Wesley, Reading, MA.

Sommerville, I., and Sawyer, P. (1997). *Requirements Engineering*: *A Good Practice Guide*, Wiley, New York.

Stevens, R., Brook, P., Jackson, K., and Arnold, S. (1998). *Systems Engineering*: *Coping with Complexity*, Prentice Hall Europe.

Thayer, R., and Dorfman, M., eds. (1997). *Software Requirements Engineering* (2nd Ed.), IEEE Computer Society Press, Los Alamitos, CA.

van Lamsweerde, A. (2000). Requirements Engineering in the Year 00: A Research Perspective, Keynote Address, In *Proceedings of 22nd International Conference on Software Engineering (ICSE-2000), Limerick, Ireland, June 2000*, ACM Press, New York.

Wieringa, R. J. (1996). *Requirements Engineering*: *Frameworks for Understanding*, Wiley, New York.

Zave, P. (1997). Classification of Research Efforts in Requirements Engineering, *ACM Computing Surveys* **29**(4): 315–321.

Software Engineering

Mehdi Jazayeri
Technische Universität Wien

GLOSSARY

Component-based software engineering An approach to software engineering based on the acquisition and assembly of components.
Component interface The description of what services are provided by the component and how to request those services.
Inspections and reviews Organized meetings to review software components and products with the aim of uncovering errors early in the software life cycle.
Process maturity The capability of an organization to follow a software process in a disciplined and controlled way.
Programming language A primary tool of the software engineer. It is a notation used to write software.
Software architecture The overall structure of a software system in terms of its components and their relationships and interactions.
Software component A well-defined module that may be combined with other components to form a software product.
Software process An ideal description of the steps involved in software production.
Software requirements A document that describes the expected capabilities of a software product.
Validation and verification Procedures that help gain confidence in the correct functioning of software.

SOFTWARE ENGINEERING is the application of engineering principles to the construction of software. More

precisely, the IEEE Std 610.12-1990 Standard Glossary of Software Engineering Terminology defines software engineering as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

Software engineering deals with the building of software systems that are so large or so complex that they are built by a team or teams of engineers. Usually, these software systems exist in multiple versions and are in service for many years. During their lifetime, they undergo many changes—to fix defects, to enhance existing features, to add new features, to remove old features, or to be adapted to run in a new environment.

We may appreciate the issues involved in software engineering by contrasting software engineering with computer programming. A programmer writes a complete program, while a software engineer writes a software component that will be combined with components written by other software engineers to build a system. The component written by one software engineer may be modified by other software engineers; it may be used by others to build different versions of the system long after the original engineer has left the project. Programming is primarily a personal activity, while software engineering is essentially a team activity.

The term "software engineering" was invented in the late 1960s after many large software projects failed to meet their goals. Since then, the field has grown to include many techniques and methods for systematic construction of software systems. These techniques span the entire range of activities starting from the initial attempts to understand the customer requirements for a software system to the design and implementation of that system, validation of the system against the requirements, and the delivery of the system and its deployment at the customer site.

The importance of software has grown over the years. Software is now used to control virtually every sophisticated or even not so sophisticated device. Software is used to control transportation systems including subways, trains, and airplanes; control power plants; everyday devices such as ovens, refrigerators, and television sets; medical devices such as pace makers and diagnostic machines. The Internet is certainly powered by sophisticated software. The whole society is dependent upon correct functioning of software. Software is also of growing economic impact: in 1985, around $140 billion was spent annually on software worldwide. In 2000, the amount is estimated at $800 billion worldwide. As a result, the software engineering discipline has gained importance. The emphasis of software engineering is on building dependable software economically.

This article reviews the two major elements of software engineering: products and processes. Software engineers follow a process to build a product. We review techniques for building a high-quality product. Some of these are product related and others are process related. We start with a brief history of software engineering followed by a classification of the kinds of software in Section II. Sections III and IV deal with the software process and the remaining sections deal with activities in the software process that produce the software product, including requirements analysis, architectural design, and testing and verification of software.

I. A BRIEF HISTORY OF SOFTWARE ENGINEERING

The birth and evolution of software engineering as a discipline within computer science can be traced to the evolving and maturing view of the programming activity. In the early days of computing, the problem of programming was viewed essentially as how to place a sequence of instructions together to get the computer to do something useful. The problems being programmed were well understood—for example, how to solve a differential equation. The program was written by, say, a physicist to solve an equation of interest to him or her. The problem was just between the user and the computer—no other person was involved.

As computers became cheaper and more common, more and more people started using them. Programming languages were invented in the late 1950s to make it easier to communicate with the machine. But still, the activity of getting the computer to do something useful was essentially done by one person who was writing a program for a well-defined task.

It was at this time that "programming" attained the status of a profession: you could ask a *programmer* to write a program for you instead of doing it yourself. This introduced a separation between the user and the computer. Now the user had to specify the task in a form other than the precise programming notation used before. The programmer then read this specification and translated it into a precise set of machine instructions. This, of course, sometimes resulted in the programmer misinterpreting the user's intentions, even in these usually small tasks.

Very few large software projects were being done at this time—the early 1960s—and these were done by computer pioneers who were experts. For example, the CTSS operating system developed at MIT was indeed a large project, but it was done by highly knowledgeable and motivated individuals.

In the middle to late 1960s, truly large software systems were attempted commercially. The best documented of these projects was the OS 360 operating system for

the IBM 360 computer family. The people on these large projects quickly realized that building large software systems was significantly different from building smaller systems. There were fundamental difficulties in scaling up the techniques of small program development to large software development. The term "software engineering" was invented around this time, and conferences were held to discuss the difficulties these projects were facing in delivering the promised products. Large software projects were universally over budget and behind schedule. Another term invented at this time was "software crisis." Although the term software crisis is still used, it is now agreed that "crisis" is caused by misplaced and unrealistic expectations.

It became apparent that the problems in building large software systems were not a matter of putting computer instructions together. Rather, the problems being solved were not well understood, at least not by everyone involved in the project or by any single individual. People on the project had to spend a lot of time communicating with each other rather than writing code. People sometimes even left the project, and this affected not only the work they had been doing but also the work of the others who were depending on them. Replacing an individual required an extensive amount of training about the "folklore" of the project requirements and the system design. Any change in the original system requirements seemed to affect many parts of the project, further delaying system delivery. These kinds of problems just did not exist in the early "programming" days and seemed to call for a new approach.

Many solutions were proposed and tried. Some suggested that the solution lay in better management techniques. Others proposed different team organizations. Yet others argued for better programming languages and tools. Many called for organization-wide standards such as uniform coding conventions. A few called for the use of mathematical and formal approaches. There was no shortage of ideas. The final consensus was that the problem of building software should be approached in the same way that engineers had built other large complex systems such as bridges, refineries, factories, ships, and airplanes. The point was to view the final software system as a complex product and the building of it as an engineering job. The engineering approach required management, organization, tools, theories, methodologies, and techniques. And thus was software engineering born.

II. KINDS OF SOFTWARE

Software comes in an amazing variety. Certainly the software inside a television set differs from that which controls an airplane which is still different from the software that makes up a word processor. Each of these different types requires a different software engineering approach in terms of the process model and in terms of requirements that it must achieve.

A software system is often a component of a much larger system. For example, a telephone switching system consists of computers, telephone lines and cables, other hardware such as satellites, and finally, software to control the various other components. It is the combination of all these components that is expected to meet the requirements of the whole system.

We may classify the types of software along different dimensions. Along one dimension we may distinguish software as system or application software. Another classification distinguishes among embedded, networking, and application software. We will now discuss these different kinds of software and their special characteristics.

A. Application Software

Application software refers to software that is written to perform a particular function of general utility. While the software is of course run on a computer, the function it performs is not particularly computer related. For example, a word processor is a typical example of application software. It helps users to create text documents. It deals with concepts from a domain of interest to users in the document processing domain such as documents, pages, paragraphs, words, justification, hyphenation, and so on. The requirements for such software may be defined entirely in terms of these concepts independently of the execution platform. Of course, the execution platform does exert some influence on the software in terms of what the software can reasonably achieve. For example, the software has access to different amounts of resources if it runs on a powerful desktop computer or a small hand-held device. But the functionality requirements may be stated in terms of the application domain.

Another example of application software is a banking system. Such software also deals with a particular application domain and its requirements may be stated in terms of the domain concepts such as customer, account, interest rate, deposit, and so on.

The different kinds of application software may be further subdivided as shrink-wrapped and custom software. Shrink-wrapped software is developed for a mass market and sold on the market. Custom software is developed for a particular customer. A word processor is typically shrink-wrapped and banking software is typically custom-made.

A main difference between process models for shrink-wrapped and custom software is in the requirements phase. The requirements phase for shrink-wrapped software is

done by a marketing group perhaps with the help of focus groups. There is no specific customer that may be consulted about the requirements. In contrast, the requirements of custom software are driven by a customer. Shrink-wrap software must be developed in different versions for different hardware platforms and perhaps in different levels of functionality (e.g., in light and professional versions), whereas custom software is developed and optimized for a particular platform (with a view toward future evolutions of the platform). These considerations affect the initial statements of the requirements and the techniques used to achieve those requirements.

B. System Software

System software is intended to control the hardware resources of a computer system to provide useful functionality for the user. For example, an operating system tries to optimize the use of resources such as processor, memory, and input–output devices to enable users to run various programs efficiently. A database system tries to maximize the use of memory and disk resources to allow different users to access data concurrently.

As opposed to application software, the requirements of system software are tied directly to the capabilities of the hardware platform. The requirements must deal with computer concepts such as hardware resources and interfaces. The software engineering of system software typically requires specialized knowledge such as transaction processing for databases or process scheduling for operating systems.

Despite their dependence on the hardware platform, system software such as operating systems and databases these days are developed independently of the hardware platform. In earlier days, they had to be developed hand in hand with the hardware. But advances in techniques for developing abstract interfaces for hardware devices have enabled system software to be developed in a portable way. By setting appropriate configuration parameters, the software is specialized for a particular platform.

System software often provides interfaces to be used by other programs (typically application programs) and also interfaces to be used by users (interactively). For example, an operating system may be invoked interactively by a user or by database systems.

An increasingly important class of system software is communication software. Communication software provides mechanisms for processes on different computers to communicate. Initially, communication software was included in operating systems. Today, it is being increasingly packages as "middleware." Middleware provides facilities for processes on different computers to communicate as well as other facilities for finding and sharing resources among computers. For writing distributed applications, software engineers rely on standard middleware software.

C. Embedded Software

Embedded software is software that is not directly visible or invokable by a human user but is part of a system. For example, the software is embedded in television sets, airplanes, and videogames. Embedded software is used to control the functions of hardware devices. For example, a train control system reads various signals produced by sensors along tracks to control the speed of the train. The characteristic of embedded software is that it is developed hand in hand with the hardware. The designers of the system face tradeoffs in placing a given functionality in hardware or software. Generally, software offers more flexibility. For example, a coin-operated machine could be designed with different-sized slots for different coins or a single slot with control software that determines the value of the coin based on its weight. The software solution is more flexible in that it can be adapted to new coins or new currencies.

A particular kind of embedded software is real-time software. This kind of software has requirements in terms of meeting time constraints. For example, the telephone software must play the dial tone within a certain time after the customer has taken the phone off hook. Often real-time systems are responsible for critical functions such as patient monitoring. In such cases special design techniques are needed to ensure correct operation within required time constraints. Real-time software is among the most challenging software to construct.

III. THE SOFTWARE LIFE CYCLE

From the inception of an idea for a software system, until it is implemented and delivered to a customer, and even after that, a software system undergoes gradual development and evolution. The software is said to have a *life cycle* composed of several phases. Each of these phases results in the development of either a part of the system or something associated with the system, such as a test plan or a user manual. The phases may be organized in different orders, for example, performed sequentially or in parallel. The choice of organization and order of the phases define a particular software process model. Depending on the kind of software being built, and other requirements on the project, such as the number of people or the length of the schedule, a different process model may be appropriate. The traditional life cycle model is called the "waterfall model," because each phase has well-defined starting and

ending points, with clearly identifiable deliverables that "flow" to the next phase. In practice, it is rarely so simple and the phases and their deliverables have to be managed in a more complicated way.

The following phases typically comprise a software process model:

- **Feasibility study.** Starting with an idea for a software product, a feasibility study tries to determine whether a software solution is practical and economical. If the project appears feasible, alternative solutions are explored such as whether it is possible to buy the software, develop it in-house, or contract for it to be developed externally. The impact of the software on the existing environment must also be considered in this phase. Sometimes, new software will be replacing existing manual procedures in an organization. As a result, the introduction of the software will change the way people work.
- **Requirements analysis and specification.** The purpose of this phase is to identify and document the exact requirements for the system. Such study may be performed by the customer, the developer, a marketing organization, or any combination of the three. In cases where the requirements are not clear—e.g., for a system that has never been done before—much interaction is required between the user and the developer. Depending on the kind of software being produced, this phase must also produce user manuals and system test plans.
- **Architecture design and system specification.** Once the requirements for a system have been documented, software engineers design a software system to meet them. This phase is sometimes split into two subphases: *architectural or high-level design* and *detailed design*. High-level design decomposes the software to be built into subsystems or components called modules; detailed design then works on the design of the components.
- **Coding and module testing.** In this phase the engineers produce the actual software code and programs that will be delivered to the customer as the running system. Individual modules developed in the coding phase are also tested before being delivered to the next phase.
- **Integration and system testing.** All the modules that have been developed before and tested individually are put together—integrated—in this phase and tested as a whole system.
- **Delivery and maintenance.** Once the system passes all the tests, it is delivered to the customer and is deployed. Following delivery, from the developer's viewpoint the product enters a maintenance phase. Any modifications made to the system after initial delivery are usually attributed to this phase.

These phases may be ordered and organized differently according to the needs of the project. Each particular organization reflects a particular modeling of the development process. In the next section we present the most prevalent software process models. In parallel to these phases, two other activities are essential to a software engineering organization. *Project management* monitors the timely and orderly progress of the project and *quality assurance* is concerned with ensuring that the final product meets the quality standards of the organization. These activities are not specific to any particular phase of the software life cycle and must be in place throughout the whole life cycle. Indeed, they are not even specific to software engineering but to any engineering activity.

IV. PROCESS MODELS

A process model is an ideal definition of the software life cycle or of how actual software projects work. As such, they are prescriptions for how to organize the software engineering activity. In reality, actual software projects deviate from these models but the models nevertheless give both managers and engineers a framework in which to plan and schedule their work.

A. The Waterfall Model

The classic and traditional process model is the waterfall model. This model assumes a sequential order of the phases and the completion of one phase before the next phase starts.

Figure 1 gives a graphical view of the waterfall software development life cycle that provides a visual explanation of the term "waterfall." Each phase yields results that "flow" into the next, and the process ideally proceeds in an orderly and linear fashion.

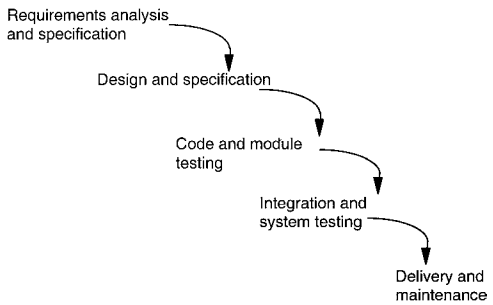


FIGURE 1 The waterfall model of the software life cycle.

The figure gives the simplest model of the waterfall model. For example, it is clear that if any tests uncover defects in the system, we have to go back at least to the coding phase and perhaps to the design phase to correct some mistakes. In general, any phase may uncover problems in previous phases; this will necessitate going back to the previous phases and redoing some earlier work. For example, if the system design phase uncovers inconsistencies or ambiguities in the system requirements, the requirements analysis phase must be revisited to determine what requirements were really intended. Such problems require us to add feedback loops to the model, represented as arrows that go back from one phase to an earlier phase and the need to repeat some earlier work.

A common shortcoming of the waterfall model is that it requires a phase to be completed before the next one starts. While feedback loops allow the results of one phase to affect an earlier phase, they still do not allow the overlapping of phases. In practice, a common technique to shorten development times is to carry out activities in parallel. The strict sequential nature of the waterfall model is one of its most severe drawbacks. Other process models try to alleviate this problem.

B. Incremental Models

The waterfall model defines the requirements at the beginning and delivers the product at the end. During the whole development time, the customer is not involved and does not gain any visibility into the state of the product. Some models try to remedy this problem by introducing different stages in which partial deliveries of the product are made to the customer.

One such model is the prototyping approach. The first delivery to the customer is a prototype of the envisaged system. The purpose of the prototype is to assess the feasibility of the product and to verify that the requirements of the customer have been understood by the developer and will be met by the system. The prototype is then thrown away (in fact, it is sometimes called a throwaway prototype), and development starts on the real product based on the now firmly established requirements. The prototyping approach addresses the difficulty of understanding the real requirements but it does not eliminate the time gap between the definition of requirements and delivery of the application.

Incremental process model addresses the delivery gap. It produces the product in increments that implement the needed functionality in increments. Increments may be delivered to the customer as they are developed; this is called *evolutionary*, or *incremental*, *delivery*.

Increments to be delivered to the customer consist not only of code and internal project documentation but also of user-oriented documentation. In other words, a *delivered increment* is a self-contained functional unit of software that performs some useful purpose for the customer, along with all supporting material (requirements and design specifications, test plans and test cases, a user manual, and training material). This approach requires a phase in which the requirements are carefully subdivided into functionalities that may be decomposed and delivered independently. The decision as to what order to deliver the increments is a key negotiation point between the developer and the customer. One possibility is to deliver the basic functionality of the system first and increasingly sophisticated functionality later. Another is to deliver what can be delivered quickly and delay to later stages those features that take longer to build. A third option is to deliver the functionality that is most critical to the customer early.

C. The Transformation Model

The transformation model is rooted in theoretical work on formal specifications. This model views software development as a sequence of steps that starts from a formal (mathematical) specification of the requirements and gradually transforms it into an implementation. First, informal requirements are analyzed and functions are specified formally, possibly in an incremental way. Then, the development process takes this formal description and transforms it into a more detailed, less abstract formal description.

The software engineer may perform the transformations. In this case, the formal nature of the derivation may provide a form of mathematical check that one step is a correct transformation of the previous. It is also possible, however, that the support system performs transformations automatically, possibly under the software engineer's direction.

The transformation model is rather a theoretical one and is best viewed as an ideal model. Its contribution to practice has been limited. The requirement to specify the requirements formally is rather stringent and unrealistic for typical products.

D. Spiral Model

The goal of the spiral model of the software process is to provide a framework for designing such processes, guided by the risk levels in the project at hand. As opposed to the previously presented models, the spiral model may be viewed as a *metamodel*, because it can accommodate any development process model. By using it as a reference,

one may choose the most appropriate development model (e.g., evolutionary versus waterfall). The guiding principle behind such choice is risk management.

The spiral model focuses on identifying and eliminating high-risk problems by careful process design, rather than treating both trivial and severe problems uniformly. The spiral model is cyclic rather than linear like the waterfall model. Each cycle of the spiral consists of four stages.

Stage 1 identifies the objectives of the portion of the product under consideration, in terms of qualities to achieve. Furthermore, it identifies alternatives—such as whether to buy, design, or reuse any of the software—and the constraints on the application of the alternatives. The alternatives are then evaluated in Stage 2 and potential risk areas are identified and dealt with. Risk assessment may require different kinds of activities to be planned, such as prototyping or simulation. Stage 3 consists of developing and verifying the next level product; again, the strategy followed by the process is dictated by risk analysis. Finally, Stage 4 consists of reviewing the results of the stages performed so far and planning for the next iteration of the spiral, if any.

If the requirements for the application are understood reasonably well, a conventional waterfall process model may be chosen, which leads to a simple one-turn spiral. In less understood applications, however, the next step may be evolutionary in nature; i.e., several spiral turns may be required in order to achieve the desired results. It can also accommodate any mixture of the previously discussed models, with the appropriate mix chosen so as to minimize the risks involved in development.

E. Open Source Model

The traditional software process models are based on the assumption that what is delivered to the customer is an executable module that may be installed and run on the customer's computer. The source code remains the property of the software developer. In the 1990s, the open source process model became popular. This model spurns the proprietary nature of the source code. It holds that the source code must be placed in the public domain for anyone to use. The goal is to encourage others to find the defects in the software and produce enhancements and variations of the product. System software, such as the Linux operating system, was produced in this way. In effect, this process encourages software developers anywhere to cooperate in *ad hoc* workgroups. Anyone who finds a defect or thinks of an enhancement can send a contribution to the unofficial keeper of the software who decides whether and when to insert the enhancement in the software. This model has gained considerable support in the software engineering community as an effective way to enhance the reliability of software products. The rationale is that the more independent people that examine the source of the software, the more defects will be found.

V. PROCESS QUALITY

The process followed in building software affects the qualities of the software product. For example, a disciplined process with reviews and inspections along the way is more likely to produce reliable software than a process that does not rely on any reviews of the intermediate steps. Several standards have been developed to measure the quality of software processes and thus gain confidence in the quality of the produced product. Such process quality standards help enterprises such as government agencies evaluate software contractors.

A typical such standard is the Capability Maturity Model. This model defines software process *capability* as the range of results that may be expected by following a software process. The process capability of an organization may be used to predict the most likely results of a project undertaken by the organization. The model also defines software process maturity as the extent to which the process is defined explicitly, managed, measured, controlled, and effective. The notion of maturity implies that an organization's software process capability can grow as it matures. The model defines five levels of software capability maturity. The least capable organizations are graded at Level 1, called initial. An organization at Level 1 has no defined process model. It produces software as best it can without any apparent planning. As a result, projects sometimes succeed and sometimes fail. Any given project's outcome is unpredictable. At Level 2, the organization has stable and established project management controls. As a result, even though the organization does not have a documented process, it consistently produces the intended product. This capability is referred to as "repeatable." Level 3 organizations have a defined process that consistently leads to successful projects. Projects in the organization tailor the defined process to the needs of the project. A specific group is responsible for the organization's software engineering activities. Management has good visibility into the progress and status of projects. Level 4 organizations not only have a defined process but also collect metrics on the performance of their processes and can therefore predict with good confidence how their projects will perform in the future. This level is called "managed." At the highest level of process maturity, organizations not only measure their process performance but also aim to improve their performance continuously.

This level is called "optimizing." The focus in optimizing organizations is no longer merely the production of the software product—which is considered a certainty—but on continuous improvement of the organization.

An initial study of software organizations around the world showed that very few organizations perform at Level 5. Most were at Level 3 or below. But once such criteria for process performance have been defined, they can be used by organizations to strive for improvement.

Each level of maturity is defined with the help of a detailed list of characteristics. There are organizations that perform "process assessment," that is, measure the maturity of a software development organization and provide it with guidelines on how to try to improve to the next level of maturity. An organization's process maturity level gives its customers a basis for realistic expectations of its performance. This is especially important if a company wants to outsource the development of key parts of its software.

Measuring the maturity level of an organization is one way of assessing the performance of a software development organization. Other standards also exist that measure the process differently. In all cases, such measurements are difficult to make and should be evaluated carefully.

VI. REQUIREMENTS ENGINEERING

The most critical phase of software engineering is the requirements analysis and specification phase. In this phase, the representatives of the developing organization and the customer organization attempt to define a complete set of requirements for the software product to be built. This phase is critical because once the requirements have been defined, the software product will be built according to those requirements. Any errors in the requirements will lead to a product that does not meet the customer's expectations.

There are many reasons why requirements are often not correct. First, the customer does not always know exactly what is needed or what is possible. In this case, the developer and the customer try to come up with an approximation of what could be needed. Second, the customer and the developer often come from different backgrounds and experience domains, which leads to miscommunication and misunderstandings. Third, the customer's requirements may be conflicting and inconsistent.

The activities involved with determining the requirements are collectively called "requirements engineering" and consist of the activities of eliciting, understanding, and documenting the requirements. The success of an application depends heavily on the quality of the results of requirements engineering. The ultimate purpose of these activities is to understand the goals of the system, the requirements it should meet, and to specify the qualities required of the software solution, in terms of functionality, performance, ease of use, portability, and so on, that would lead to the satisfaction of the overall system requirements.

A general guideline for what should be included in a requirements document, which is not always easy to follow, is that the requirements document should describe *what* qualities the application must exhibit, not *how* such qualities should be achieved by design and implementation. For example, the document should define what functions the software must provide, without stating that a certain distributed architecture, or module structure, or algorithm should be used in the solution. The goal of the guideline is to avoid over-specification of the system. Over-specification will constrain the engineers who will implement the requirements and possibly prevent them from finding better solutions. In practice, sometimes this guideline must be broken. For example, government regulations may require that a certain security algorithm is or is not used, or that a certain architecture is used to achieve reliability. Often, existing environments constrain the design choices to work with existing computers or architectures.

If the software to be developed is part of a more general system, such as a television system, which includes many devices, a crucial requirements activity will be responsible for decomposing the overall system requirements into software and other requirements. The responsibilities of the software component within the overall system must be defined. The choice of what functionality to place in software and what to do in hardware must sometimes remain flexible because it could be dependent on current technology.

The main goal of the requirements activities is to understand precisely the interface between the application to develop and its external environment. Such an environment can be—say—a physical plant that the application is supposed to monitor and automate, or it can be a library where librarians interact with the system to insert new acquisitions in the catalog and to lend books to customers, and where customers can browse through the catalog to find the books they are interested in. The requirements activity must clearly identify the main *stakeholders*, i.e., all those who have an interest in the system and will be eventually responsible for its acceptance. For example, in the case of the library, the software engineers must understand who are the expected users of the system (librarians, customers of the library), and what different access rights to the system they will have. They will need to understand the mechanisms of book acquisition, of borrowing and returning books, etc. The main stakeholders will include the librarians and sample customers. Since the library is usually part of a larger system, the individuals responsible

for the library in the larger system are also relevant stakeholders. As an example, in the case of the library of a university department, the department head is a stakeholder, whose goals and requirements should be properly taken into account. The department head's goal might be to encourage students to borrow both books and journals, to stimulate their interest in reading the current scientific literature. Or, alternatively, he or she might require that students may only borrow books, limiting the ability to borrow journals to only staff members. An important stakeholder is the person or organization that will have the budget to pay for the development. In this case, it may be the dean or the president at the university. As this example shows, the various stakeholders have different *viewpoints* on the system. Each viewpoint provides a partial view of what the system is expected to provide. Sometimes the different viewpoints may be even contradictory. The goal of requirements engineering is to identify all conflicts and integrate and reconcile all the different viewpoints in one coherent view of the system.

The result of the requirements activities is a *requirements specification document*, which describes what the analysis has produced. The purpose of this document is twofold: on the one hand, it must be analyzed and confirmed by the various stakeholders in order to verify whether it captures all of the customers' expectations; on the other hand, it is used by the software engineers to develop a solution that meets the requirements.

The way requirements are actually specified is usually subject to standardized procedures in software organizations. Standards may prescribe the form and structure of the requirements specification document, the use of specific analysis methods and notations, and the kind of reviews and approvals that the document should undergo.

A possible checklist of the contents of the requirements specification document that might guide in its production is the following:

1. *Functional requirements*. These describe what the product does by using informal, semiformal, formal notations, or a suitable mixture. Various kinds of notations and approaches exist with different advantages and applicability. The Unified Modeling Language (UML) is increasingly used as a practical standard because it contains different notations for expressing different views of the system. The engineer can select and combine the notations best suited to the application or the ones he or she finds the most convenient or familiar.
2. *Quality requirements*. These may be classified into the following categories: reliability (availability, integrity, security, safety, etc.), accuracy of results, performance, human–computer interface issues, operating constraints, physical constraints, portability issues, and others.
3. *Requirements on the development and maintenance process*. These include quality control procedures—in particular, system test procedure—priorities of the required functions, likely changes to the system maintenance procedures, and other requirements.

VII. SOFTWARE ARCHITECTURE

The software architecture describes a software system's overall structure in terms of its major constituent components and their relationships and interactions. The design of an architecture for complex systems is a highly challenging task, requiring experience and, often, creativity and ingenuity. The goal of the architect is to design and specify a system that satisfies the user requirements and can be produced and maintained efficiently and economically. The software architecture has many uses in the software life cycle. The software architecture is the first concrete description of the system that will be produced. It describes the gross structure of the system and its major parts. As a result, it can be analyzed to answer questions about the final system. For example, it may be used to estimate the performance of the system or the expected cost of producing the system. It may therefore be used to perform initial assessment of whether the system will meet the functional, performance, and cost requirements.

Another use of the software architecture is to enable the division of the subsequent work of designing the subsystems. As the architecture specifies the relationships and constraints among the major subsystems, the design of each such subsystem may be assigned to a different designer, with the requirements specified by the overall architecture.

One of the approaches to combating the challenges of software architecture design is to develop and use standard architectures. One of the most successful such standard architectures for distributed platforms is the client–server architecture. In this architecture, some components play the role of servers, providing specific services. Other components play the role of clients and use the services of the servers to provide other services to their clients, possibly to the users of the system. This architecture has been used successfully in the design of the Worldwide Web system. In that system, *Web servers* manage information organized as *pages* on their respective sites. Client modules, called *Web browsers* in this case, interact with the human user and request access to information from appropriate Web servers.

The client–server architecture strongly separates the clients from the servers, allowing them to be developed

independently. Indeed, different Web browsers, produced by different people and organizations work with different Web servers, developed by different people and different organizations. The requirement for this independence between client and server, and indeed between any two components is that each component has a well-defined *interface* with which it can be used. The interface specifies what services are available from a component and the ways in which clients may request those services. Thus, the designer of any client knows how to make use of another component simply based on that component's interface. It is the responsibility of the component's implementer to implement the interface. The interface thus specifies the responsibilities of the designers and implementers of components and users of components.

Standard architectures capture and codify the accepted knowledge about software design. They help designers communicate about different architectures and their tradeoffs. They may be generic, such as the client–server architecture, or application- or domain-specific. Application-specific architectures are designed for particular domains of application and take advantage of assumptions that may be made about the application. For example, a domain-specific architecture for banking systems may assume the existence of databases and even particular data types such as bank accounts and operations such as depositing and withdrawing from accounts.

At a lower level of detail, *design patterns* are a technique for codifying solutions to recurring design problems. For example, one may study different techniques for building high-performance server components that are capable of servicing large numbers of clients. Such solutions may be documented using patterns that describe the problem the pattern solves, the applicability of the patterns, the tradeoffs involved, and when to use or not to use the pattern. Both standard architectures and design patterns are aimed at helping designers and architects cope with the challenges of the design activity and avoid reinventing the wheel on every new project.

The notion of a component is central to both architectures and design patterns. Just as we can identify standard architectures and patterns, we can also identify components that are useful in more than one application. The use of standard components is standard practice in other engineering disciplines but it has been slow in software engineering. Their importance was recognized early in the development of software engineering but its realization was only made possible in the 1990s after advances in software architecture and design and programming languages made it possible to build such components and specify their interfaces.

Component-based software engineering considers software engineering to consist of assembling and integrating components to build an application. It relies mostly on components that are acquired from outside developers. It divides software engineering into two distinct activities of component development and application or system development. Application developers buy so-called off-the-shelf components (COTS) from component developers and assemble them into the desired application.

Component-based software engineering places the additional constraint on the software architect to produce a design that utilizes available COTS as much as possible. The use of COTS and component-based software engineering presumably reduces the design time thus improving productivity and reducing overall costs of software engineering.

VIII. SOFTWARE DESIGN AND SPECIFICATION

The software architecture describes the overall structure of a software system. The activity of software design further refines this structure and produces a complete set of components and their interfaces. The decomposition of the architecture into supporting components and the design of interfaces for components are challenging design tasks. There are several different approaches and methodologies to approach this task. Two main approaches are function-oriented design and object-oriented design.

In function-oriented design, the required functionality is analyzed and decomposed into smaller units. This decomposition is repeated until the units are small enough that can be implemented as a single component in the overall system design. The interfaces for the components are documented so that implementers know exactly what to implement. The job of implementing a component involves designing appropriate algorithms and data structures to accomplish the task of the component.

In object-oriented design, the system is viewed as consisting of a set of objects that cooperate to accomplish the required functionality. Each object has a set of attributes and a set of functions that it can perform. The attributes hold values that represent the state of the object. For example, attributes for an "employee" object may include name, address, skill, and salary. The objects interact and cooperate by sending each other messages that request functions to be performed. The task of object-oriented design consists of identifying what objects are needed, and the objects' attributes and functions. Object behavior refers to the functions that an object supports.

In more detail, an object-oriented designer starts by identifying the needed objects, grouping similar objects into classes, designing the attributes and behavior of

classes, and the relationships among the classes. A class describes the structure and behavior of a group of objects. Each object is an instance of a particular class. Techniques exist for designing new classes based on existing classes. One such technique relies on the concept of "inheritance" in which a class inherits all the properties of an existing class and adds some more specific properties. For example, we might define a new class hourly-employee as an extension of an existing class employee. The new class might define an hourly-pay attribute and a new function for calculating the employee's salary.

Several object-oriented design methodologies exist and notations have been developed for documenting object-oriented designs. The Unified Modeling Language (UML) is standard notation that combines many previously designed notations into a single language. It consists of notations for documenting classes and their relationships, cooperation patterns among objects, synchronization constraints on parallel activities, the deployment requirements for the system, as well as a notation for describing the user visible functions of the system.

Collections of related classes may be packaged together to form application-oriented *frameworks*. Such frameworks generally assume an implicit architecture for the final application and provide classes that fit into that architecture. More general frameworks address the needs of a particular domain such as banking or finance. For example, a financial-domain framework would consist of classes such as account, interest, mortgage, and so on that support the development of financial software. One of the most successful uses of frameworks has been in the graphical user interface domain.

IX. SOFTWARE QUALITY

With the critical role played by software in our society, the issue of software quality has gained well-deserved attention. Software is used in many diverse areas and applications with varying demands for quality. For example, the software responsible for landing an airplane has different quality requirements from the software that controls a vide-game console. Both must provide their specified functionality but the users of the software have different tolerance levels for errors in the software and are willing to pay different amounts for higher quality.

To deal with variations in quality requirements, we can define a set of quality factors and metrics for the factors. The choice of factors that are important and their level depends on the application. For example, *security* is an important factor for e-commerce applications and *fault-tolerance* is important for flight software. Typical quality factors are performance, reliability, security, usability, and so on. These factors may be further subdivided. For example, reliability is subdivided into correctness, fault-tolerance, and robustness. Correctness means that the product correctly implements the required functionality; fault- tolerance means that the product is tolerant of the occurrence of faults such as power failure; robustness means that the product behaves reasonably even in unexpected situation such as the user supplying incorrect input. For each of these factors, we can define metrics for measuring them.

A complete requirements document includes not only the functional requirements on the product but also the quality requirements in terms of factors and the levels required for each factor. Software engineers then try to apply techniques and procedures for achieving the stated quality requirements. There are two general approaches to achieving quality requirements: product-oriented methods and process-oriented methods. The process-oriented approach concentrates on improving the process followed to ensure the production of quality products. We have already discussed process quality, its assessment and improvement in Section V. Here we review product-oriented techniques for achieving software quality.

A. Validation and Verification

We refer to activities devoted to quality assessment as validation and verification or V&V. Validation is generally concerned with assessing whether the product meets external criteria such as the requirements. Verification is generally concerned with internal criteria such as consistency of components with each other or that each component meets its specification.

The approaches to validation and verification may be roughly classified into two categories: formal and informal. Formal approaches include mathematical techniques of proving the correctness of programs and the more common approach of testing. Informal techniques include reviews and inspections of intermediate work products developed during the software life cycle.

B. Testing

Testing is a common engineering approach in which the product is exercised in representative situations to assess its behavior. It is also the most common quality assessment technique in software engineering. There are, however, several limitations to testing in software engineering. The most important limitation is that software products do not exhibit the "continuity" property we are accustomed to in the physical world. For example, if we test drive a car at 70 miles an hour, we reasonably expect that the car

will also operate at speeds below 70 miles an hour. If a software product behaves correctly for a value of 70, however, there is no guarantee at all about how it will behave with any other value! Indeed, due to subtle conditions, it may not even behave properly when run with the value of 70 a second time! This discontinuity property of software creates challenges for the testing activity. An accepted principle in software engineering is that testing can only be used to show the presence of errors in the software, never their absence.

The primary challenge of testing is in deciding what values and environments to test the software against. Since it is impossible to test the software exhaustively—on all possible input values—the engineer must apply some criteria for selecting test cases that have the highest chances of uncovering errors in the product. Two approaches to selecting test cases are called white box testing and black box testing. White box testing considers the internals of the software, such as its code, to derive test cases. Black box testing ignores the internal structure of the software and selects test cases based on the specification and requirements of the software. Each approach has its own advantages and uses. *Test coverage* refers to the degree to which the tests have covered the software product. For example, the tests may have exercised 80% of the code of the software. The amount of the test coverage gives the engineers and managers a figure on which to base their confidence in the software's quality.

There are different focuses for testing. Functional testing concentrates on detecting errors in achieving the required functionality. Performance testing measures the performance of the product. Overload testing tries to find the limits of the product under heavy load.

Testing is carried out throughout the entire life cycle, applied to different work products. Two important tests applied before final delivery of the product are alpha test and beta test. In alpha test, the product is used internally by the developing organization as a trial run of the product. Any errors found are corrected and then the product is placed in a beta test. In a beta test, selected potential users of the product try a pre-release of the product. The primary aim of the beta test is to find any remaining errors in the product that are more likely to be found in the customer's environment. A benefit of the beta test for the software's developer is to get early reaction of potential customers to a product or its new features. A benefit of beta test for the customer is to get an early look at a future product or its new features.

Testing consumes a large part of the software life cycle and its budget. Engineers use software tools such as test generators and test execution environments to try to control these costs.

C. Reviews and Inspections

As opposed to testing, which attempts to uncover errors in a product by running the product, analytic techniques try to uncover errors by analyzing a representation of the product such as its software code. Program proof techniques fall in this category. A more common approach involves reviews of a work product by peer engineers. Two such techniques are code walk-throughs and inspections.

A code walk-through is an informal analysis of code as a cooperative, organized activity by several participants. The participants select some test cases (the selection could have been done previously by a single participant) and simulate execution of the code by hand, "walking through" the code or through any design notation.

Several guidelines have been developed over the years for organizing this naive but useful verification technique to make it more systematic and reliable. These guidelines are based on experience, common sense, and subjective factors.

Examples of recommended rules are:

- The number of people involved in the review should be small (three to five).
- The participants should receive written documentation from the designer a few days before the meeting.
- The meeting should last a predefined amount of time (a few hours).
- Discussion should be focused on the *discovery* of errors, not on fixing them, nor on proposing alternative design decisions.
- Key people in the meeting should be the designer, who presents and explains the rationale of the work, a moderator for the discussion, and a secretary, who is responsible for writing a report to be given to the designer at the end of the meeting.
- In order to foster cooperation and avoid the feeling that the designers are being evaluated, managers should not participate in the meeting.

The success of code walk-throughs hinges on running them in a cooperative manner as a team effort: they must avoid making the designer feel threatened. Their purpose is to examine the code, not the coder.

Another organized activity devoted to analyzing code is called *code inspection*. The organizational aspects of code inspection are similar to those of code walk-throughs (i.e., the number of participants, duration of the meeting, psychological attitudes of the participants, etc., should be about the same), but there is a difference in goals.

In code inspection, the analysis is aimed explicitly at the discovery of common errors. In other words, the code—or, in general, the design—is examined by checking it for

the presence of errors, rather than by simulating its execution.

Many errors in programs can be classified according to well-known categories. Such categorizations help support and focus inspections.

There is some evidence that walk-throughs and inspections are cost-effective and many software engineering organizations are adopting them as standard procedures in the life cycle.

X. MANAGEMENT OF SOFTWARE ENGINEERING

In addition to common project management issues such as planning, staffing, monitoring, and so on, there are several important challenges that face a software engineering manager due to the special nature of software. The first is that there are no reliable methods for estimating the cost or schedule for a proposed software product. This problem leads to frequent schedule and cost over-runs for software projects. The factors that contribute to this problem are the difficulty of measuring individual engineer's software productivity and the large variability in the capability and productivity of software engineers. It is not uncommon for the productivity of peer engineers to differ by an order of magnitude. Dealing with the cost estimation challenge requires the manager to monitor the project vigilantly and update estimates and schedules as necessary throughout the project.

Another factor that complicates the scheduling and estimation tasks is that requirements for a software system can rarely be specified precisely and completely at the start of a project. The product is often built for an incompletely known environment, possibly for a new market, or to control a novel device. Thus, it is typical for requirements to change throughout the software life cycle. One of the ways to deal with this challenge is to use a flexible process model that requires frequent reviews and incremental delivery cycles for the product. Feedback from the delivered increments can help focus and complete the requirements.

A third challenge of managing a software project is to coordinate the interactions and communications among the engineers and the large numbers of work products produced during the life cycle. Typically, many work products exist in different versions. They must be shared, consulted, and modified by different engineers. The solution to this challenge is to establish strict guidelines and procedures for configuration management. Configuration management tools are some of the most widely used and effective software engineering tools.

Configuration management tools and processes provide a repository to hold documents. Procedures exist for controlled access to documents in the repository, for creating new versions of documents, and for creating parallel and alternative development paths for a product consisting of some of the documents. Typically, a software engineer checks out a document such as a source code file from the repository. This action locks out any other accesses to the document by others. The engineer applies some changes to the document and checks the document back in. At this point this new version of the document becomes available for others to access. Such procedures ensure that engineers can share documents while avoiding unintended conflicting updates.

With the support of configuration management, several releases of a product may exist in the repository and indeed under development. A particular release of a product may be built by knowing which versions of individual components are necessary. Configuration management tools typically provide mechanisms for documenting the contents of a release and build tools for automatically building a given release. Such build tools support parameterization of components. For example, a given product may be built for a particular country by specifying a country-specific component as a parameter.

Configuration management tools are essential for orderly development of software projects that produce myriads of components and work products. They are even more important in distributed software engineering organizations. It is now common for teams of software engineers to be dispersed geographically, located at different sites of an organization, possibly in different countries and continents. Configuration management tools provide support for such distributed teams by providing the illusion of a central repository that may be implemented in a distributed fashion.

One of the product management and development trends of the 1990s was to look for organizational efficiency by outsourcing those parts of a business that are not in the core business of the company. Outsourcing of software is indeed compatible with component-based software engineering. Development of whole classes of components can be outsourced with the responsibility lines clearly specified. Successful outsourcing of software engineering tasks can be based on well-specified architectures and interface specifications. Clear and complete specifications are difficult to produce but they are a necessary prerequisite to a contract between the contractor and a contracting organization. The specification must state quality requirements in addition to functional requirements. In particular, the contract must specify who performs the validation of the software, and how much, including test coverage criteria and levels.

XI. SUMMARY

Software engineering is an evolving *engineering* discipline. It deals with systematic approaches to building large software systems by teams of programmers. We have given a brief review of the essential elements of software engineering including product-related issues such as requirements, design, and validation, and process-related issues including process models and their assessment.

With the pervasiveness of software in society, the importance of software engineering is sure to grow. As technologies in diverse areas are increasingly controlled by software, challenges, requirements, and responsibilities of software engineers also grow. For example, the growth of the Internet has spurred the need for new techniques to address the development and large-scale deployment of software products and systems. The development of e-commerce applications has necessitated the development of techniques for achieving security in software systems. As new applications and technologies are constantly emerging, the software engineering field promises to stay a vibrant and active field in a constant state of flux.

ACKNOWLEDGMENT

Sections of this material were adapted from the textbook Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002). "Fundamentals of Software Engineering," 2nd edition, Prentice Hall, Englewood-Cliffs, NJ.

SEE ALSO THE FOLLOWING ARTICLES

COMPILERS ● COMPUTER ALGORITHMS ● COMPUTER ARCHITECTURE ● COMPUTER NETWORKS ● OPERATING SYSTEMS ● REQUIREMENTS ENGINEERING ● SOFTWARE MAINTENANCE AND EVOLUTION ● SOFTWARE RELIABILITY ● SOFTWARE TESTING ● SYSTEM THEORY ● WWW (WORLD-WIDE WEB)

BIBLIOGRAPHY

Boehm, B. W. (1981). "Software Engineering Economics," Prentice-Hall, Englewood Cliffs, N.J.

Brooks, F. P. Jr. (1995). "The Mythical Man-Month: Essays on Software Engineering," second edition, Addison-Wesley, Reading, MA.

Ghezzi, C., and Jazayeri, M. (1997). "Programming Language Concepts," third edition, Wiley, New York.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002). "Fundamentals of Software Engineering," second edition, Prentice Hall, Englewood-Cliffs, NJ.

Jazayeri, M., Ran, A., and van der Linden, A. (2000). "Software Architecture for Product Families: Principles and Practice," Addison-Wesley, Reading, MA.

Leveson, N. (1995). "Safeware: System Safety and Computers," Addison-Wesley, Reading, MA.

Neumann, P. G. (1995). "Computer-Related Risks," Addison-Wesley, Reading, MA.

Software Maintenance and Evolution

Elizabeth Burd
Malcolm Munro
Research Institute in Software Evolution

GLOSSARY

Change request A request for a change to a piece of software.
Program comprehension The process of understanding how a software system works.
Repository A data storage used for storing information about a software system.
Software engineering The discipline of producing and maintaining software.
Software system A collection of computer code and data that runs on a computer.

SOFTWARE is the central resource of many companies; it is the software that drives businesses and it is often the only true description of their processes. However, as businesses change, then it is necessary to drive these modifications through the software. Software change is notoriously difficult, and as the age of the software increases so the process becomes harder.

Software Maintenance is a costly activity consuming 50% of all computer and IT resources. It has been shown that maintenance costs can be up to 10 times those of an initial development.

Software maintenance has been defined within the ANSI standard as

the modification of software products after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.

It has also been defined as:

Software Maintenance is the set of activities (both technical and managerial) necessary to ensure that software continues to meet organizational needs.

The first definition can be considered as a technical definition and states that there are certain technical activities that we have to perform on software after it is delivered. The second definition is much more business oriented and states that software has to change as business changes.

Many different terms have been used to describe the activity of software maintenance. These include bug fixing, enhancement, support, further development, current engineering, postdeployment software support, postrelease development, and many more. Coupled with these terms and the previously cited definitions it is important to recognize the different types of maintenance. In most organizations software maintenance is only seen as bug fixing, whereas continued change and evolution are seen either as development or redevelopment. This is a mistake, as changing an existing system requires a number of specialized techniques that do not apply to green field development. The main technique required is that of program comprehension, that is, understanding how the current system is constructed and works so that changes can be made. Underestimating this task can lead to system degradation and further problems.

Software maintenance processes are initiated for a number of reasons. These differing reasons result in four categories of maintenance activities. These are

- Perfective maintenance—This involves improving functionality of software in response to user-defined changes.
- Adaptive maintenance—This process involves the alteration of the software which is due to changes within the software environment.
- Corrective maintenance—This process involves the correction of errors that have been identified within the software.
- Preventative maintenance—This involves updating the software in order to improve upon its future maintainability without changing its current functionality.

The costs of the maintenance processes are not distributed evenly across all categories. Studies show that 50% of the total maintenance costs can be attributed to perfective maintenance, 25% for adaptive maintenance, whereas only 21% of the total costs are attributed to corrective maintenance and 4% for preventive maintenance.

Despite the cost implications, software maintenance is generally perceived as having a low profile within the software community. Management teams have often in the past placed little emphasis on maintenance-related activities. Small advances have been made in combating these problems, and high-profile maintenance projects such as the year 2000 problem have been successful at highlighting the issues.

Software maintenance is made difficult by the age of the software requiring maintenance. The age of the software means that documentation has often been lost or is out of date. Furthermore, issues of staff turnover and constant demands for changes due to user enhancements or environmental changes exacerbate the problems. In addition, constant perfective and corrective maintenance, which is not supported by preventative maintenance, has a tendency to make the software more difficult to maintain in the future.

Software evolution is a term that is sometimes used interchangeably with software maintenance. In fact they are different, but related terms. For the purpose of this paper software evolution is considered to be the changes that occur to software throughout its lifetime to ensure it continues to support business processes. Software maintenance is the process of making these changes to the software thereby seeking to extend its evolutionary path.

Software evolution is defined as *the cumulative effect of the set of all changes made to software products after delivery.*

This chapter will investigate the existing approaches to maintenance and then describe how these processes are to be improved by studying the effect of software evolution. Within Section II there is a review of past and present maintenance practices. Section III evaluates the recent findings from studies of software evolution and evaluates these in terms of the implication to increasing complexity and the formation of the detrimental legacy properties within applications. It also makes some recommendations regarding best practice for software maintenance based on results of the evolution studies. Finally some conclusions are drawn.

I. SOFTWARE MAINTENANCE

Software maintenance is influenced by software engineering. As the software engineering fraternity strives for the perfect maintenance-free development method, the software maintainers have to pick up the pieces. For example, Object-Oriented designed systems were heralded as maintenance free. It is easy to state this when there are no Object-Oriented systems to maintain. It is only now that such systems have move into maintenance that all the problems manifest themselves. What should of course happen is that software maintenance should influence software engineering.

From the definitions given previously, it should be clear that all levels of an organization should be aware (and involved) of software maintenance. At the technical level the developers should not ignore it as they must address the issues of maintainability of the systems they develop; the maintainers cannot ignore it as they are carrying out the work; the users are involved as they want the systems to evolve and change to meet their ever-changing need; senior management cannot ignore it as its

business is more than likely dependent on the software it is running.

Why is there a maintenance problem?

- A large amount of existing software was produced prior to significant use of structured design and programming techniques.
- It is difficult to determine whether a change in code will affect something else.
- Documentation is inadequate, nonexistent, or out of date (or even there is too much of it);
- Maintenance is perceived as having a low profile within an organization.
- Software is not seen as a corporate resource and hence has no value.
- Maintenance programmers have not been involved in the development of a product and find it difficult to map actions to program source code.

Software maintenance has an important role to play in an organization. Many organizations do not recognize their software systems as a company asset and thus do not place a high price on its value. The risk is that if all types of maintenance are not carried out is that the business of an organization suffers because of inadequate software either because for example it is full of bugs or because it has not evolved to meet changing business needs. The benefits of carrying out maintenance in an organized and cost-efficient manner is that an organization can easily and continually adapt to new market opportunities.

A. Principles and Assumptions

The fact that software maintenance is inevitable and that systems must continually evolve to meet an organizations' needs is neatly encapsulated in a number of papers by Professor Manny Lehman. In particular he presents five laws of program evolution. The first two laws are relevant here.

1. Law 1: Continuing Change

A Program that is used and that, as an implementation of its specification, reflects some other reality, undergoes continuing change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the program with a recreated version.

2. Law 2: Increasing Complexity

As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it.

Both these laws state that software maintenance is inevitable and that organizations should be aware of this fact. There are no magic *silver bullets* that can be used in development to eliminate maintenance. The first law of Continuing Change says that the original business and organizational environment modeled in the software is modified by the introduction of the software system and thus the system must continually change to reflect this. Thus software must evolve. The second law says that extra resources must be devoted to preserving and simplifying the structure of the software system and that there is a cost associated with maintaining the quality of the system. Thus we should pay attention to preventive maintenance.

B. Models and Process in Software Maintenance

There have been a large number of models describing the software maintenance process. Probably the best models are those that have remained in-house and are a hybrid devised from several models and current in-house practice. The published models can be classified into four types.

Maintenance models can be classified into four types:

1. Modification Cycle Models. These models give a sequence of steps to be carried out—they are usually oriented for corrective maintenance. The steps are of the form *Problem Verification, Problem Diagnosis, Reprogramming and Rebuild, Baseline Verification, and Validation*.
2. Entity Models. These models enumerate the entities of a system and detail how for each step of the model the entities are generated or modified. The entities are items such as Requirement, Specification, Source Code, Change Requests, People, Tasks, Forms, and Knowledge. The classic model of this type is that developed for the IEEE in the IEEE Standard 1219-1993.
3. Process Improvement Models. These models concentrate on how to improve the maintenance process and are based on the SEI (Software Engineering Institute) five-layer model of process improvement.
4. Cost/Benefit Models. These models used the Entity models and applied cost/benefit analysis to them.

The major gap in software maintenance is a theory of software maintenance. There is not a formal theory nor is there a clear way forward to developing one. The nearest there is, is in the work of Manny Lehman and his laws of program evolution. These laws were formulated from empirical evidence but are difficult to validate for all systems. A way forward is to carry out some long-term research on

how systems evolve and how their structure changes over time. This must be linked with how business affects the evolving software. From this research a universal definition of maintainability may emerge.

C. Measurements and Metrics for Software Maintenance

Measurement and metrics for software maintenance is a difficult topic. There are a large number of product and process metrics for software. Product metrics tend to concentrate on complexity type metrics and there have been some attempts at correlating some of these metrics with maintenance effort. One of the problems here is there is no definitive and universal definition of maintainability. Another problem is that organizations are reluctant to publish data on their systems thus making any comparisons difficult.

Determination of the maintenance effort is again a difficult task. It will depend on factors such as the size of the system, the age since delivery, structure and type of system, quality of the documentation standards and the document update procedures in place, the number of reported bugs, the type and number of change requests that are mandatory and desirable, the use of change control procedures, the establishment of test procedures, the level of staff competence and training and staff turnover.

All these factors (and more) do not fit easily into a simple function that give the definitive answers to questions such as "Is my system maintainable," "How well is my maintenance operation doing," or "Should I continue with maintaining the system or throw it away and start again." Some of these factors can be combined into a simple function forming that can be called a "System Profile." From this profile judgments could be made as to how to improve the process and possibly make decisions on whether to continue maintenance or to redevelop. The system profile addressed the following factors:

1. Adequacy to the User. The values and judgments obtained here assess the extent to which the system currently meets the users requirements. Factors such as implementation of desirable changes, length of the changes backlog queue, and the effect of the business are assessed.
2. Risk to the Business. The values and judgments obtained here assess the risk of system failure and the degree to which this would be serious for the overall function of the organization. Factors such as staffing experience and retention, software change control procedures, number of known bugs, effect of errors on the business, and the state of the code are assessed.
3. System Support Effort. The values and judgments obtained here assess the amount of resources which are required, or would be required, to maintain the system adequately. Factors such as staffing levels on maintenance, and number of mandatory changes are assessed.

The importance of such an approach is that it attempts to make any assessment objective and that it forces managers to ask relevant questions about the systems they are maintaining.

II. EVOLUTION

The evolution of software is studied by investigating the changes that are made to software over successive versions. In the majority of cases this involved a historical study of past changes over the past versions that are available. Unfortunately since the benefit of evolutionary studies of software have yet to establish their significance and benefit within industry, the need to retain past versions and data regarding the environmental circumstances of the changes is not foreseen. While the authors have found very few applications with a record of versions from the entire lifetime of the software product they have been successful in collecting a great number of systems and their version upon which to base their analysis.

The approach adopted by the authors is to take successive versions of a software application and to investigate the changes that are occurring. Depending on the general data available regarding the environment of the changes, additional information may or may not be used. Where possible, analysis is performed using as much information as possible to supplement the analysis process. A number of different approaches are adopted but the fundamental approach replies on the appearance or removal of calls and data items.

A. Levels

Studies in software evolution have been conducted at three main levels. These are Level 1—the system level; Level 2—the function level; Level 3—the data level. These can broadly be viewed as at different levels of granularity ranging from the system level to studies of the underlying data. A more detailed description of each of the three levels is indicated in the following.

1. Level 1, The System Level

Software evolution research at the system level has been conducted, almost exclusively, by the Lehman team over a period of 30 years. The major contribution of this work was

the formation of the Lehman Laws of Evolution. These laws reveal general observations of the changing character of software systems and additionally indicate how the process of evolution should be best managed. For instance, the second law provides a high-level definition of increased complexity. It states that for a large program that is continuously changing, its complexity, which reflects deteriorating structure, increases unless work is performed to maintain or reduce it. In addition, law five, that of incremental growth limit, highlights that systems develop an average increment of safe growth which, if exceeded, causes quality and usage problems and time and cost overruns. These laws provide information upon which a high-level maintenance strategy can be composed, but in some instances it is necessary to be more specific regarding the nature of change especially when partial redevelopment is proposed to reduce legacy properties and improve future maintainability.

To gain a deeper understanding of software evolution at the system level it is interesting to study a software application as a whole and to investigate how applications change over time; however, for large applications a high level of abstraction for their representation is essential. Within Fig. 1 an application is represented at the file level. This represented application is the Gnu C compiler. In Fig. 1 each column represents a version of the software, moving from left to right so the age of the software increases. The rows each represent a different file within the application. Figure 1 highlights where changes have been made to one or more of the system files. Those files, which are changed within a specific version, are shaded. Those files, which remain unchanged within a version, are left unshaded.

Figure 1 has also been sorted based on the number of changes. Those files that are most frequently changed are at the top of the diagram. Those files changed least frequently are shown toward the bottom. From the diagram it is possible to see a number of characteristics of changes. Those changes in which columns are most heavily shaded, represent major changes which the software. Those columns with only a few changes may, for instance, represent the result of small bug corrections.

It is interesting to see how the majority of changes are made to relatively few of the files, especially when the major software changes are discounted. Specifically, 30 or 40 files seem to be changed in each software version. It is therefore likely that these files are in most need of preventative maintenance, as these either represent the core procedural units of the application or are hard to understand and therefore are a frequent source of misunderstandings, and so often requiring bug fixes. An investigation into these issues is currently an area of continued research.

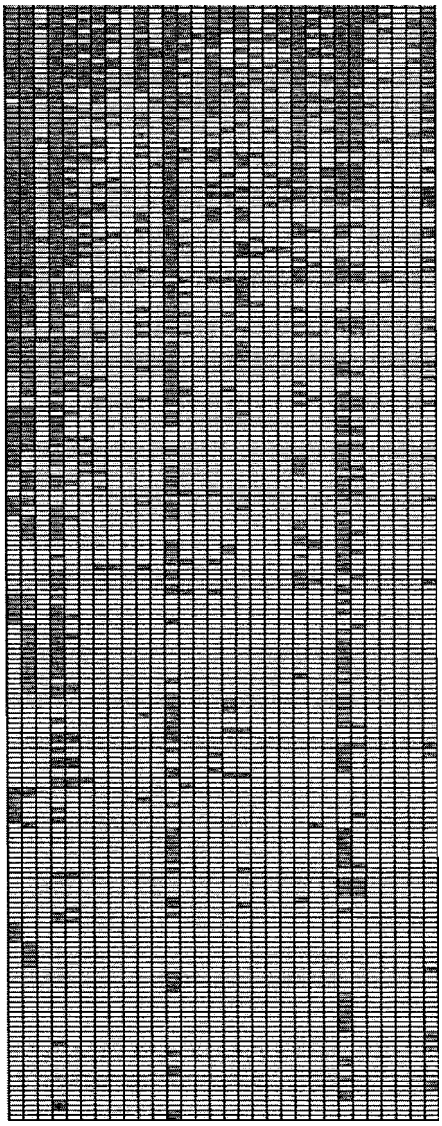


FIGURE 1 Versions for the GCC application.

The number of changes and the time to make these changes are mapped onto the graph in Fig. 2. A high number of changes should represent the large commitments in time, whereas the minor changes should represent much shorter time commitments. For applications, not showing this trend may indicate the presence of legacy properties. This is particularly likely to be the case when the trend of greater time commitments must to be allocated per change as the age of the application increases. From the graph within Fig. 2 it can be seen that within the early years of this application the time commitments are proportionally fewer than the number of changes. For instance, the sample versions 2 to 8 on Fig. 2 show this trend. During the later versions larger time commitments seem to be necessary. For instance, with changes 26 and onward

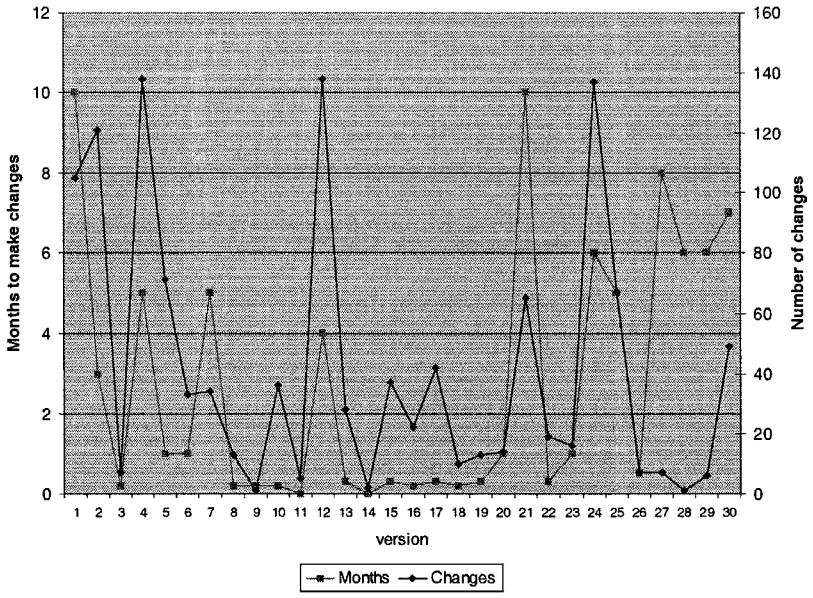


FIGURE 2 Changes to the GCC application and time to make them.

the months required for making the required changes are proportionally greater. When compared across the lifetime of the software a definite increase in time commitments per change can be seen. This may therefore imply that the software is developing legacy tendencies.

2. Level 2, The Function Level

Research at this level has concentrated on the evolution of the calling structure of code and to a lesser extent the control structure. Work at this level reveals more about the changes in complexity of individual source code modules and highlights the results and implications of specific change requests in a detailed manner. Such results are essential to fully understand the effects, detrimental or otherwise, of software evolution as a whole and how the change process will effect the future changes required. Specifically with studies at this level of granularity, it is possible to gain an understanding of the evolution process from the point of view of changes to the comprehensibility of a source code module. It is this change in comprehensibility that will directly effect the future maintainability of the module under investigation.

Studies by Burd and Munro have identified some of the effects of software. Within Fig. 3, a call structure change is represented. The representation of the software on the left of the diagram shows a single module from an early version of the software. The figure shows that three modules result from the single module due to the evolution process. This is a commercial COBOL application. The cause of this splitting process is due to the addition of new functions added to the module. In the case of Fig. 3 these new units are represented as the shaded nodes. The result of process of evolution shows a requirement for existing modules to be designed in a way that allows them to split over time.

Further studies into this phenomenon have indicated that it may be possible to predict the likely places where additional functionality will be added. From studies it has been identified that where splitting of the modules occurs, it occurs in a specific location of the tree structure. Specifically, this usually occurs when the module is represented as a tree based on dominance relations, at a position in the tree where there are a number of branches occurring from a node (i.e., the node has a high fan-out to other nodes). In terms of the calling structure this equates to a function that calls many other functions. Examples of likely candidate locations for the module splitting are highlighted with arrows within Fig. 4.

With this knowledge, precautions can be taken in the identified areas to enhance comprehensibility therefore increasing potential adaptability. This kind of revelation regarding the evolution of software applications can be used to direct the software development process. Furthermore, it assists the knowledge of the cost benefit process for change management by indicating areas where localized redevelopment may enhance adaptability and thereby reduce some of the legacy properties of software.

Examples of the splitting process have also been identified within applications written in C. However, other interesting properties have also been found within C applications. In particular, this is the feature of increasing depth of the call structure over time. In general, it has been found

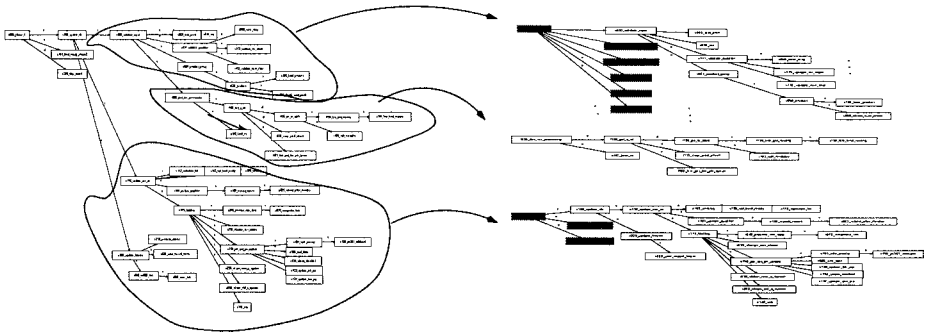


FIGURE 3 Changes in calling structure.

that the COBOL applications studied had a greater call depth than the C applications. In most cases, the increase is by approximately two levels. One example of the process of increasing call depth to software over time is the addition of a new call. An example is shown in Fig. 5, where a new call is placed between the two nodes to the graph on the left. The result of the addition of the call is shown in the graph to the right of Fig. 5, which shows an increased depth of the call structure in the later version of the software. Thus, the studies at the calling structure level, once again seem to show an increase in complexity as a result of the process of software change and hence the possible inclusion of legacy properties.

3. Level 3, The Data Level

Burd and Munro have also conducted some studies at Level 3. Specifically these studies have focused around changes in data usage across versions of a single software module. The results of the findings have been varied, but

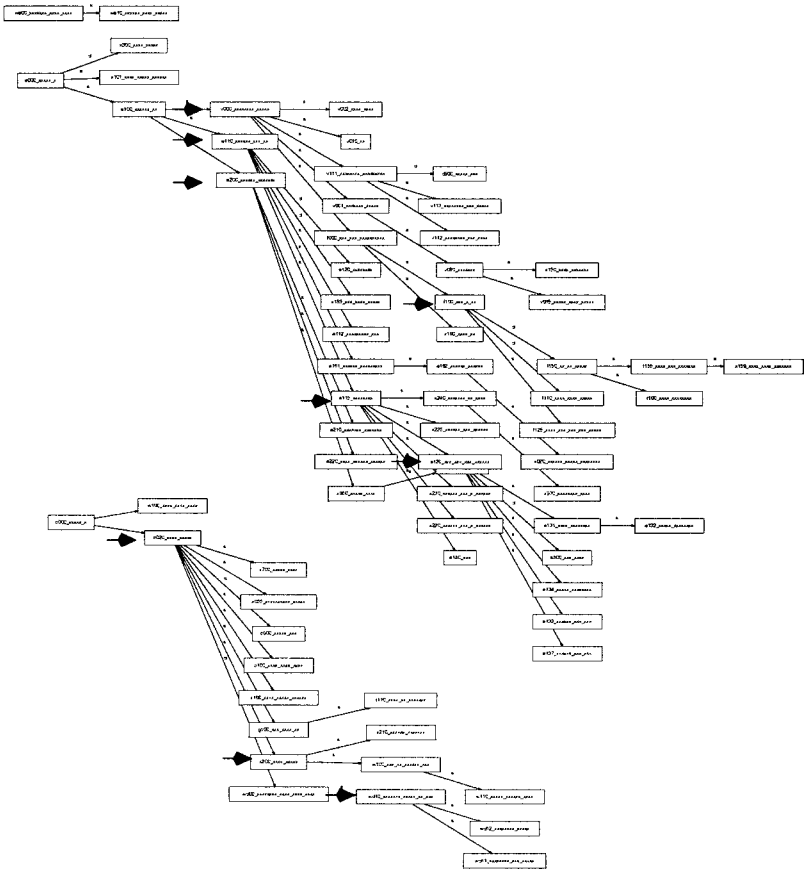


FIGURE 4 Potential portions of dominance tree where splitting is possible.

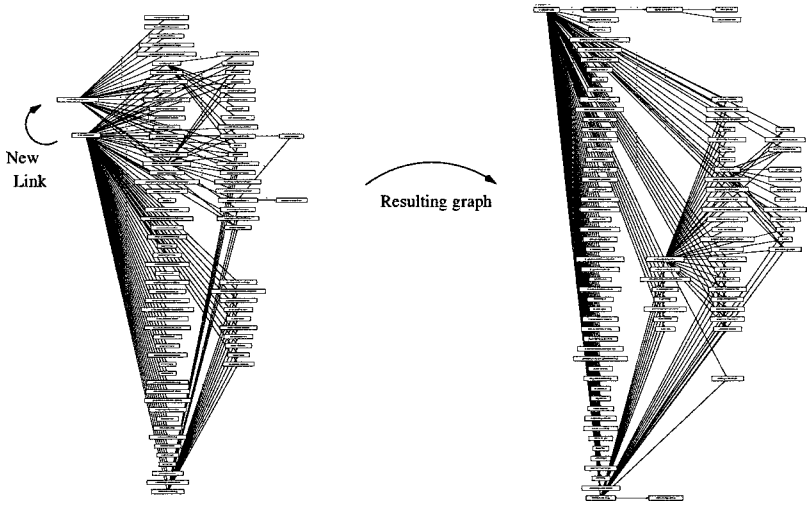


FIGURE 5 The addition of a new call and resulting increase in calling structure.

some of the more revealing about the process of software evolution are described here.

The first of these results is the process of de-localization of data due to the change process. Within Fig. 6 a COBOL module is represented. The rectangles represent the SECTION within the software module. The figure identifies the change in the usage of a single data item. Where a particular SECTION uses the data item, the SECTION is shaded. It can be seen from Fig. 6 that initially the data item is local to one SECTION but over time (moving left to right) the data item is introduced into one third of the SECTIONs of the software module.

The consequence of this to comprehension and change are great. Change ripple effects will be greatly increased, as will the amount of information that maintainers need to comprehend in order to gain an understanding of the change.

Figure 7 shows an example from the COBOL code where the process of increasing data complexity can be identified. The figure shows the changes that are occurring within the data usages for each SECTION. Comparisons are made between the data items within a specific SECTION in the earliest version of the software and compared with the data usage of the identical SECTION, but in the later version of the software. Within the COBOL application all data items are global, thus usages of the same data item within a number of SECTIONs means each one must be consulted when a change is applied. The graph in Fig. 8 shows an overall change in the number of SECTIONs for a specific data item.

Within Fig. 7 half of the graph shows data items which are in fewer SECTIONs (those to the left and labelled "Removal of data items"), whereas the other half of the graph represents the addition of data items. For instance, it can be seen that from the left-hand side, 5 data items have been removed from 4 SECTIONs. Thus, in this case the complexity of the relationships between SECTIONs can be said to be decreasing for these specific data items. However, the majority of the changes appear in the half of the graph that relates to the addition of data items. To the right-hand side it can be seen that over 20 data items have been added to a further SECTION, but in addition 6 data items have been added to more than 10 SECTIONs. Thus, the graph shows a definite increase in data complexity of the COBOL software due to the addition of data items.

Other increases in complexity, at least partly resulting from this phenomenon have also been identified. One of these is an increased complexity in the data interface

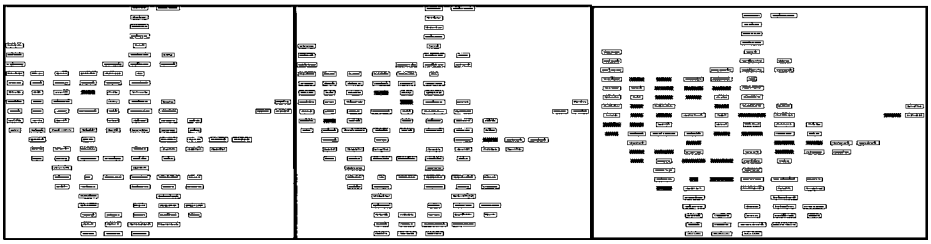


FIGURE 6 Changes to a local data item over time.

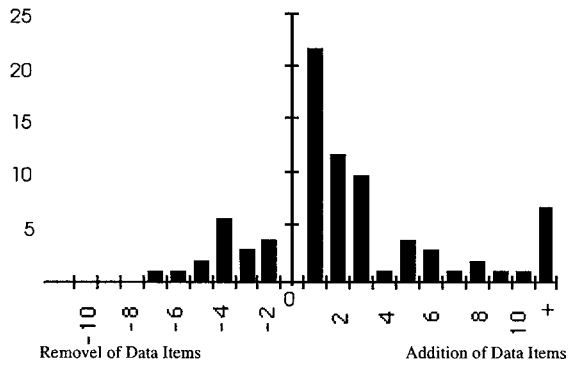


FIGURE 7 Showing the changes in localization and de-localization of data.

between subsystems within a software module. An example of this finding is shown within Fig. 8.

This figure represents the clear interface of data and subsystems within the initial version of the software (to the left) but shows how this structure is corrupted due to the evolution process. This result has major implications on the comprehensibility and future adaptability of the software module.

4. Comparing Levels

In order to gain an even greater understanding of the different maintenance trends of applications the results of call and data analysis can be compared. The approach adopted is to compare the proportion of data items modified and function call changes within each of the applications for each available version. Thus to compare the results of the analysis of Level 2 and Level 3.

The results of this analysis process are shown within Fig. 9. This graph represents the proportion of data items modified per call change for each of the applications. This graph would seem to indicate that within Sample Version S2 (the GCC application) revealed a considerably higher proportion of data per call modifications than was necessary with the other versions. In addition, it is interesting to investigate the rise and fall of these proportions.

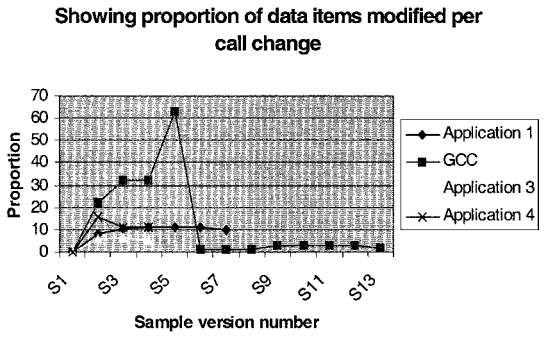


FIGURE 9 Comparing the results of analysis of Levels 2 and 3.

A very steep rise would indicate that a software application is quickly gaining legacy properties; whereas a steep fall may indicate that a preventative maintenance approach had been adopted. Slight falls within the later trends of the software's evolution can be observed within Application 1 and Application 3. From Fig. 9, it can be observed that the GCC application promotes a steep rise between Sample Version S1 to Sample Version S5. Likewise the same observation can be made with Application 4 between Sample Versions S1 and S2.

B. General Recommendations

From conducting this analysis process a number of factors for successful maintenance have been identified from early recommendations that the authors aim to extend and verify within later studies. However, in order that industry can see the benefits of such research it is necessary to make some early recommendations. In summary,

- *Study entire applications*—By studying the entire changes that occur to files within the application, a more specific investigation can be made as to the likelihood of where further changes will need to be directed.
- *Data seems to be less well understood than calling structure*—When making changes to software

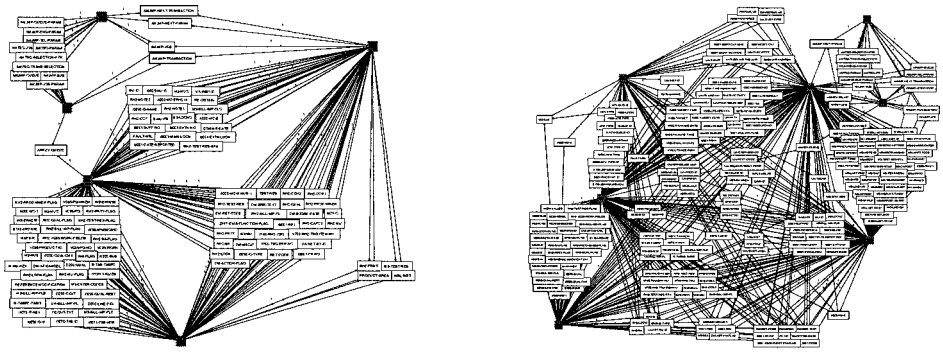


FIGURE 8 Overlap of data usage between subsystems.

applications it seems that frequently the data are modified in a less than optimal way. More effort should be applied when making a change to ensure that wherever possible data cohesion is not adversely effected. Representing data cluster changes is one way of highlighting such a problem.

- *Fewer software releases tend of lead to slower increases in data complexity*—A strategy that tends to batch change requests and issue releases at set periodic time-scales has the opportunity to develop a more considered overall maintenance strategy and optimize and integrate the design of requests.
- *Best people should be assigned to maintenance*—This research seems to highlight that when some of the best programmers were assigned to the maintenance tasks the overall quality of the code tended to improve. This is a complete reversal of the standard evolutionary path of software under maintenance where usually a steady increase in software data complexity is identifiable.
- *Preventative maintenance needs to be continuous theme*—Preventative maintenance is not something that can be performed once and then forgotten; rather it must either be a task that is carried out in detail at specific time periods or more appropriately as a continuing theme.

Additional work in this area will try to gain further insights into properties of specific maintenance changes and how these changes effect the evolution of software applications. From this it is hoped that other insights into appropriate strategies for maintenance providers will emerge.

III. CONCLUSIONS

This chapter has investigated the often-confused relationship between maintenance and evolution. It has defined software evolution as the change that occurs to software throughout its lifetime to ensure it continues to support business processes. Software maintenance is the process of making these changes to the software thereby seeking to extend its evolutionary path. This chapter has reviewed recent findings on studies of both the maintenance and the evolution process.

Software Maintenance is concerned with the way in which software is changed and how those changes are managed. Software will have to change to meet the ever-changing needs of a business. If these changes are not well managed then the software will become scrambled and prevent the business from achieving it full potential.

The chapter has reviewed studies at each of the three levels of evolution study; the system, function and data levels. It is shown how legacy properties are introduced into software applications through the change process and what are the long-term implications of these changes. Predictions of the consequence of software gaining legacy properties are made and also an indication of the low-level effect of these legacy properties have been shown diagrammatically. On the basis of these observations a number of recommendations, are made to assist software maintainers to prevent the further introduction of these legacy properties within future maintenance interventions. It is hoped from these recommendations that the maintainability of software will be improved within the future, therefore making the process of evolution easier and cheaper.

The authors hope that those involved within the maintenance and development of software applications will see the benefit of retaining information regarding the development and maintenance process. In the future more data must be retained, which will lead to studies of the evolution process to make even further observations, and therefore continue to strive for the improvement of the maintainability of software.

SEE ALSO THE FOLLOWING ARTICLES

COMPUTER ARCHITECTURE • COMPUTER NETWORKS • DATABASES • DATA STRUCTURES • OPERATING SYSTEMS • REQUIREMENTS ENGINEERING • SOFTWARE ENGINEERING • SOFTWARE RELIABILITY • SOFTWARE TESTING

BIBLIOGRAPHY

Boehm, B. W. (1995). The high cost of software. *In* "Practical Strategies for Developing Large Software Systems" (E. Horowitz, ed.), Addison Wesley, Reading, MA.

Burd, E. L., and Munro, M. (1999). "Characterizing the Process of Software Change," Proceedings of the Workshop on Principles of Software Change and Evolution: SCE'1999, ICSE.

Burd, E. L., and Munro, M. (2000). "Supporting program comprehension using dominance trees" (Invited Paper), *Special Issue on Software Maintenance for the Annals of Software Engineering* **9,** 193–213.

Burd, E. L., and Munro, M. (2000). "Using evolution to evaluate reverse engineering technologies: mapping the process of software change," *J. Software Systems* **53**(1), 43–51.

Glass, R. L., and Noiseux, R. A. (1981). "Software Maintenance Guidebook," Prentice Hall, Englewood Cliffs, NJ.

IEEE Standard for Software Maintenance, TEEE Std 1219-1998.

Information Technology—Software Maintenance—BS ISO/IEC 14764:1999.

Lehman, M. M. (1980). "On understanding laws, evolution, and conservation in the large-program life cycle," *J. Systems Software* **1,** 213–221.

Lehman, M. M. (1989). "Uncertainty in Computer Applications and its Control through the Engineering of Software," *J. Software Main.* **1**(1).

Lientz, B., and Swanson, E. B. (1980). "Software Maintenance," Addison-Wesley, Reading, MA.

Parikh, G., and Zvegintzov, N. (1993). "Tutorial on Software Maintenance," IEEE Computer Society Press, Silver Spring, MD.

Pigiski, T. (1996). "Practical Software Maintenance," Wiley, New York.

Software Reliability

Claes Wohlin
Blekinge Institute of Technology

Martin Höst
Per Runeson
Anders Wesslén
Lund University

GLOSSARY

Software error A mistake made by a human being, resulting in a fault in the software.
Software failure A dynamic problem with a piece of software.
Software fault A defect in the software, which may cause a failure if being executed.
Software reliability A software quality aspect that is measured in terms of mean time to failure or failure intensity of the software.
Software reliability certification To formally demonstrate system acceptability to obtain authorization to use the system operationally. In terms of software reliability, it means to evaluate whether the reliability requirement is met or not.
Software reliability estimation An assessment of the current value of the reliability attribute.
Software reliability prediction A forecast of the value of the reliability attribute at a future stage or point of time.

SOFTWARE RELIABILITY is defined as "the probability for failure-free operation of a program for a specified time under a specified set of operating conditions." It is one of the key attributes when discussing software quality. Software quality may be divided into quality aspects in many ways, but mostly software reliability is viewed as one of the key attributes of software quality.

The area of software reliability covers methods, models, and metrics of how to estimate and predict software reliability. This includes models for both the operational profile, to capture the intended usage of the software, and the operational failure behavior. The latter type of models is then also used to predict the future behavior in terms of failures.

Before going deeper into the area of software reliability, it is necessary to define a set of terms. Already in the definition, the word *failure* occurs, which has to be defined and in particular differentiated from *error* and *fault*.

Failure is a dynamic description of a deviation from the expectation. In other words, a failure is a departure

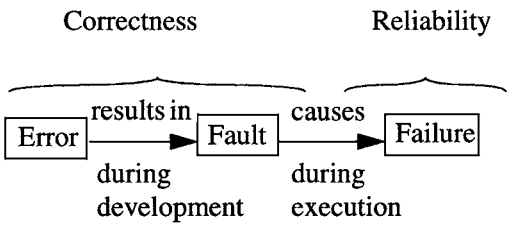


FIGURE 1 Relations between terms.

from the requirements of the externally visible results of program execution. Thus, the program has to be executed for a failure to occur. A fault is the source of a failure, statically residing in the program, which under certain conditions results in a failure. The term defect is often used as a synonym for fault. The fault in the software is caused by an error, where an error is a human action.

These definitions imply that the reliability depends not only on product attributes, such as number of faults, but also on how the product is used during operation, i.e., the operational profile. This also implies that software reliability is different from software correctness. Software correctness is a static attribute, i.e., number of faults, while reliability is a dynamic attribute, i.e., number of failures during execution.

The relations between the terms are summarized in Fig. 1.

Reliability is a probability quantity as stated in the definition. Probabilities are quantities hard to capture and to give adequate meanings for those whom are not used to them. It is often hard to interpret statements such as "the probability for a failure-free execution of the software is 0.92 for a period of 10 CPU hours." Thus, software reliability is often measured in terms of failure intensity and mean time to failure (MTTF), since they are more intuitive quantities for assessing the reliability. Then we state that "the failure intensity is 8 failures per 1000 CPU hours" or "the MTTF is 125 CPU hours," which are quantities easier to interpret.

I. RELIABILITY MEASUREMENT AND MODELING: AN INTRODUCTION

A. Usage and Reliability Modeling

The reliability attribute is a complex one, as indicated by the definitions above. The reliability depends on the number of *remaining* faults that can cause a failure and how these faults are exposed during execution. This implies two problems.

- The product has to be executed in order to enable measurement of the reliability. Furthermore, the execution must be operational or resemble the conditions under which the software is operated. It is preferable for the reliability to be estimated before the software is put into operation.
- During execution, failures are detected and may be corrected. Generally, it is assumed that the faults causing the failures are removed.

In order to solve these problems, two different types of models have to be introduced.

- A usage specification. This specification, consisting of a usage model and a usage profile, specifies the intended software usage. The possible use of the system (usage model) and the usage quantities in terms of probabilities or frequencies (usage profile) should be specified. Test cases to be run during the software test are generated from the usage specification. The specification may be constructed based on data from real usage of similar systems or on application knowledge. If the reliability is measured during real operation, this specification is not needed. The usage-based testing is further discussed in Section II.
- A reliability model. The sequence of failures is modeled as a stochastic process. This model specifies the failure behavior process. The model parameters are determined by fitting a curve to failure data. This implies also a need for an inference procedure to fit the curve to data. The reliability model can then be used to estimate or predict the reliability (see Section IV).

The principal flow of deriving a reliability estimate during testing is presented in Fig. 2.

As mentioned above, failure intensity is an easier quantity to understand than reliability. Failure intensity can, in most cases, be derived from the reliability estimate, but often the failure intensity is used as the parameter in the reliability model.

As indicated by Fig. 2, the measurement of reliability involves a series of activities. The process related to software reliabililty consists of four major steps.

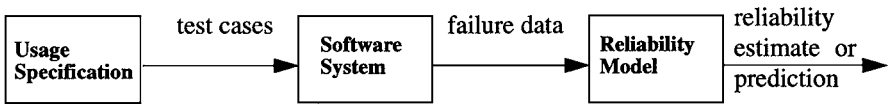


FIGURE 2 Reliability estimation from failure data.

1. Create usage specification. This step includes collecting information about the intended usage and creation of a usage specification.
2. Generate test cases and execute. From the usage specification, test cases are generated and applied to the system under test.
3. Evaluate outcome and collect failure data. For each test case, the outcome is evaluated to identify whether a failure occurred or not. Failure data is collected as required by the reliability model.
4. Calculate reliability. The inference procedure is applied on the failure data and the reliability model. Thus, a reliability estimate is produced.

If the process is applied during testing, then process steps 2–4 are iterated until the software reliability requirement is met.

Additionally, it is possible to use attribute models to estimate or predict software reliability. This means that software reliability is predicted from attributes other than failure data. For example, it may be estimated from different complexity metrics, particularly in early phases of a project. Then the estimates are based on experience from earlier projects, collected in a reliability reference model as outlined in Fig. 3.

Attributes used in the reference model can be of different types, such as project characteristics: project size, complexity, designers' experience etc.; or early process data, for example, inspection measurements. Software reliability estimation using this type of model is similar to determining other attributes through software measurement, and hence, attribute models are not specific for software reliability.

B. Application of Reliability Measurement

The reliability measurement can be used for different purposes in software project management. First of all, we differentiate between *reliability estimation* and *reliability prediction*:

- Reliability estimation means assessment of the current value of the reliability attribute.

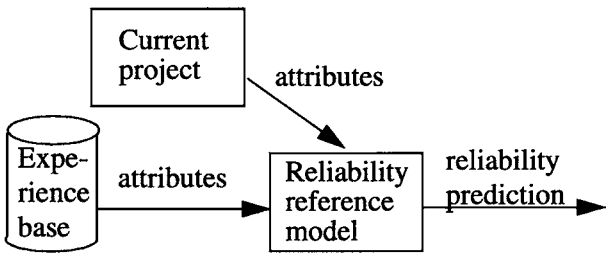


FIGURE 3 Reliability prediction from other attributes.

- Reliability prediction means forecasting the value of the reliability attribute at a future stage or point of time.

Reliability measurements can be used for different purposes. One of the most important is certification:

- Certification means to formally demonstrate system acceptability to obtain authorization to use the system operationally. In terms of software reliability, it means to evaluate whether the reliability requirement is met or not.

The certification object can be either a complete product or components in a product or in a component library. The certification can be used for internal development purposes, such as controlling the test process by relating the test stopping criteria to a specific reliability level, as well as externally as a basis for acceptance.

- Reliability predictions can be used for planning purposes. The prediction can be used to judge how much time remains until the required reliability requirement is met.
- Both predictions and estimations can be used for reliability allocation purposes. A reliability requirement can be allocated over different components of the system, which means that the reliability requirement is broken down and different requirements are set on different system components.

Hence, there are many areas for which reliability estimations and predictions are of great importance to control the software processes.

II. USAGE-BASED TESTING

A. Purpose

Testing may be defined as any activity focusing on assessing an attribute of capability of a system or program, with the objective of determining whether it meets its required results. Another important aspect of testing is to make quality visible. Here, the attribute in focus is the reliability of the system and the purpose of the testing is to make the reliability visible. The reliability attribute is not directly measurable and must therefore be derived from other measurements. These other measurements must be collected during operation or during the test that resembles the operation to be representative for the reliability.

The difficulty of the reliability attribute is that it only has a meaning if it is related to a specific user of the system. Different users experience different reliability,

because they use the system in different ways. If we are to estimate, predict, or certify the reliability, we must relate this to the usage of the system.

One way of relating the reliability to the usage is to apply *usage-based testing*. This type of testing is a statistical testing method and includes:

- A characterization of the intended use of the software, and the ability to sample test cases randomly from the usage environment
- The ability to know whether the obtained outputs are right or wrong
- A reliability model

This approach has the benefits of validating the requirements and accomplishing this in a testing environment that is statistically representative of the real operational environment.

Modeling the usage in a usage specification makes the characterization of the intended usage. This specification includes both how the users can use the system, i.e., the usage model, and the probabilities for different use, i.e., the usage profile. From the usage specification, test cases are generated according to the usage profile. If the profile has the same distribution of probabilities as if the system is used during operation, we can get a reliability estimate that is related to the way the system is used (see Fig. 4).

To evaluate whether the system responses from the system for a test case are right or wrong, an oracle is used. The oracle uses the requirements specification to determine the right responses. A failure is defined as a deviation of the system responses from its requirements. During the test, failure data is collected and used in the reliability model for the estimation, prediction, or certification of the system's reliability.

The generation of test cases and the decision on whether the system responses are right or wrong are not simple matters. The generation is done by "running through" the model, and every decision is made as a random choice according to the profile. The matter of determining the correct system responses is to examine the sequence of user input and from the requirements determine what the responses should be.

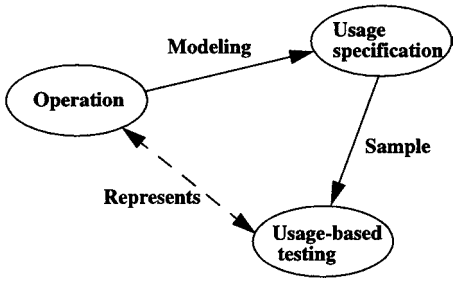


FIGURE 4 Relationship between operation, usage specification, and usage-based testing.

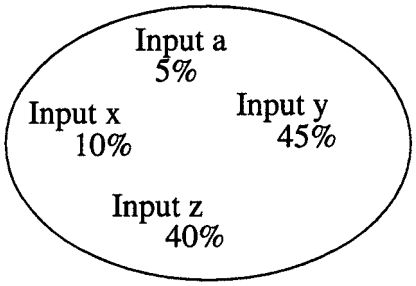


FIGURE 5 The domain-based model.

B. Usage Specifications Overview

In order to specify the usage in *usage-based testing*, there is a need for a modeling technique. Several techniques have been proposed for the usage specification. In this section, the most referred usage specification models are introduced.

1. Domain-Based Model

These models describe the usage in terms of inputs to the system. The inputs can be viewed as balls in an urn, where drawing balls from the urn generates the usage. The proportion of balls corresponding to a specific input to the system is determined by the profile. The test cases are generated by repeatedly drawing balls from the urn, usually with replacement (see Fig. 5).

The advantage of this model is that the inputs are assumed to be independent of each other. This is required for some types of reliability models.

The disadvantage is that the history of the inputs is not captured, and this model can only model the usage of a batch-type system, where the inputs are treated as a separate run and the run is independent of other runs. The model is too simple to capture the complex usage of software systems. The input history has, in most cases, a large impact on the next input.

2. Algorithmic Model

The algorithmic model is a refinement of the domain-based model. The refinement is that the algorithmic model takes the input history into account when selecting the next input. The model may be viewed as drawing balls from an urn, where the distribution of balls is changed by the input history.

To define the usage profile for the algorithmic model, the input history must be partitioned into a set of classes.

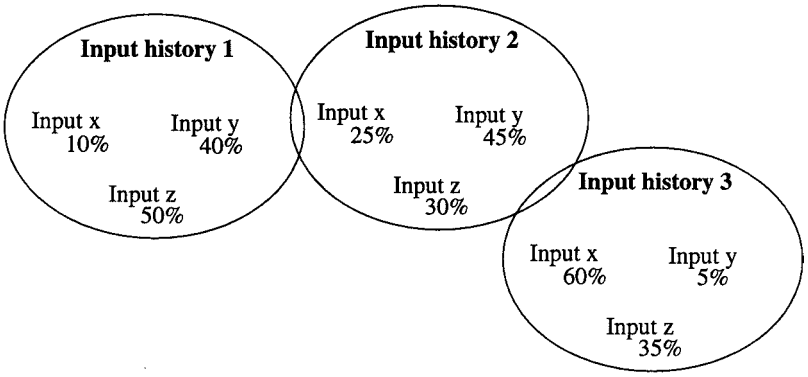


FIGURE 6 The algorithmic model.

For each class, the distribution of the inputs is determined. If there are m classes and n different inputs, the usage profile is described in an $m * n$ matrix. The elements in the matrix are the probabilities for the different inputs given the input history class.

The advantages are that this model takes the input history into account when generating the test cases and that it is easy to implement for automatic generation. The drawback is that there is a need for the information, which is not in the usage profile, on how to change from one input history class to another (see Fig. 6).

3. Operational Profile

A way of characterizing the environment is to divide the execution into a set of runs, where a run is the execution of a function in the system. If runs are identical repetitions of each other, these runs form a run type. Variations of a system function are captured in different run types. The specification of the environment using run types is called the operational profile. The operational profile is a set of relative frequencies of occurrence of operations, where an operation is the set of run types associated with a system function. To simplify the identification of the operational profile, a hierarchy of profiles is established, each making a refinement of the operational environment.

The development of operational profiles is made in five steps.

- Identify the customer profile, i.e., determine different types of customers, for example, private subscribers and companies (for a telephony exchange).
- Define the user profile, i.e., determine if different types of users use the software in different ways, for example, subscribers and mainenance personnel.
- Define the system modes, i.e., determine if the system may be operating in different modes.
- Define the functional profile, i.e., determine the different functions of the different system modes, for example, different services available to a subscriber.
- Define the operational profile, i.e., determine the probabilities for different operations making up a function.

This hierarchy of profiles is used if there is a need for specifying more than one operational profile. If there is only a need for specifying, for example, an average user, one operational profile is developed, see Fig. 7.

Choosing operations according to the operational profile generates test cases.

The operational profile includes capabilities to handle large systems, but does not support the detailed behavior of a user. It does not specify a strict external view, but takes some software internal structures into account. This is because the derivation of the operational profile needs information from the design or, in some cases, from the implementation to make the testing more efficient.

4. Grammar Model

The objective of the grammar model is to organize the descriptions of the software functions, the inputs, and the distributions of usage into a structural database from which test cases can be generated. The model has a defined

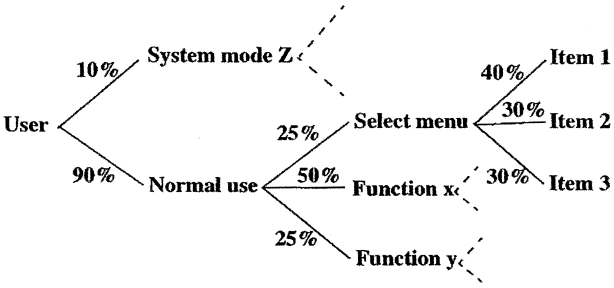


FIGURE 7 The operational profile.

```
<test_case> ::=        <no_commands> @ <command> <select>;

<no_commands> ::=      ( <unif_int>(0,1) [prob(0.8)]
                       | <unif_int>(2,4) [prob(0.2)]);

<command> ::=          (<up> [prob(0.5)]
                       | <down> [prob(0.5)]);
```

FIGURE 8 The grammar model.

grammar to describe the information of the database. The grammar defines how a test case looks, such as in length, used functions, and inputs and their distributions.

The grammar model is illustrated in Fig. 8 with the example of selecting an item in a menu containing three items. A test case in the illustration is made up of a number of commands ending with a selection. A command is either up or down with equal probability. First, the number of commands is determined. The number of commands is uniformly distributed in the range of 0–1 with the probability 0.8 and in the range of 2–4 with the probability 0.2. The command is either "up" or "down," each with a probability of 0.5. After the command, a selection is made and the test case is ended.

The outcome of a test case can be derived as the grammar gives both the initial software conditions and the inputs to it. The grammar is very easy to implement, and test cases can be generated automatically. The drawback is that the grammar tends to be rather complex for a large system, which makes it hard to get an overview of how the system is used.

5. Markov Model

The Markov model is an approach to usage modeling based on stochastic processes. The stochastic process that is used for this model is a Markov chain. The construction of the model is divided into two phases: the structural phase and the statistical phase.

During the structural phase, the chain is constructed with its states and transitions. The transitions represent the input to the system, and the state holds the necessary information about the input history. The structural model is illustrated in Fig. 9, with the example of selecting an item in a menu containing three items.

The statistical phase completes the Markov chain by assigning probabilities to the transitions in the chain. The probabilities represent the expected usage in terms of relative frequencies. Test cases are then selected by "running through" the Markov model.

The benefits of Markov models are that the model is completely general and the generated sequences look like a sample of the real usage as long as the model captures

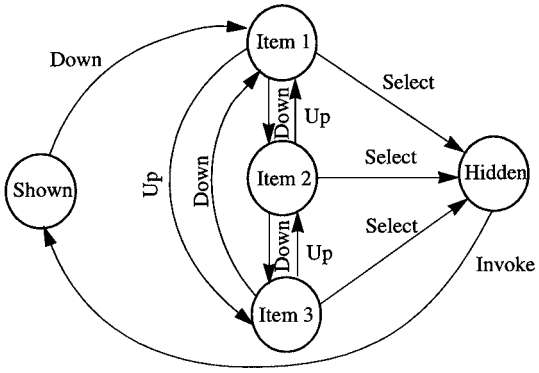


FIGURE 9 The Markov model.

the operational behavior. Another benefit is that the model is based on a formal stochastic process, for which an analytical theory is available.

The drawback is that the number of states for a complex system tends to grow very large.

6. State Hierarchy Model

The state hierarchy (SHY) model was introduced to cope with modeling of complex systems with several user types and numerous different users. The objective of the model is to divide the usage modeling problem into different levels, hence focusing on one aspect at the time. The number of levels in the model can easily be adapted to the needs when modeling (see Fig. 10). The usage levels in the figure represent all usage of the system, the user type level represents users with the same structural usage, and the usage subtype level represents all users with the same structural and statistical usage. The user level represents the users of the system, and the service level describes which services a particular user can use. The structural description of a service is described in the behavior level.

The hierarchy means that a service used by several users is only modeled once and then instantiated for all users using that particular service. The generation of test cases according to the anticipated software usage is made by "running through" the state hierarchy. The next event to be added to the test case is generated by first choosing a particular user type, then a user subtype, then a specific user of the chosen type, and, finally, a service is chosen. Based on the state of the chosen service, a transition is made in the behavior level and an event is added to the test case.

The SHY model divides the usage profile into two parts, namely, individual profile and hierarchical profile. The individual profile describes the usage for a single service, i.e., how a user behaves when using the available services. All users of a specific user type have the same individual profile. This profile refers to the transition probabilities on the behavior level.

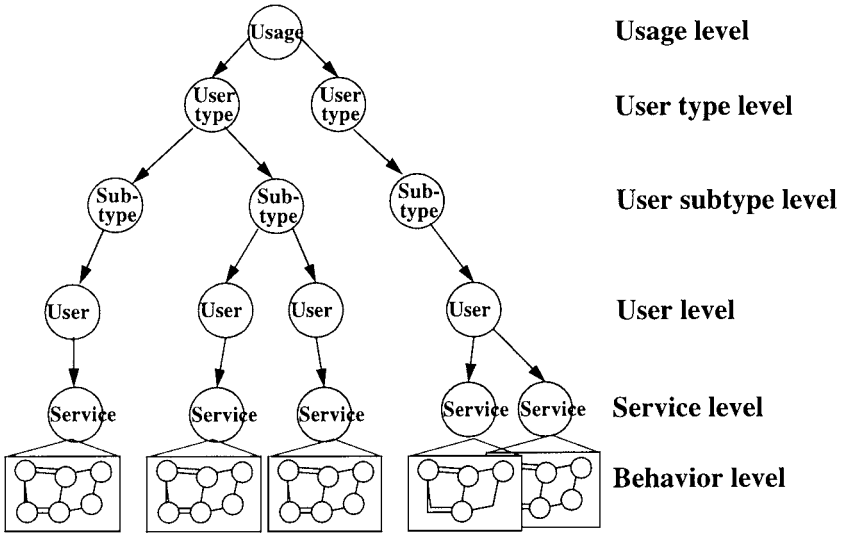


FIGURE 10 Illustration of the SHY model.

The hierarchical profile is one of the major advantages with the SHY model, as it allows for dynamic probabilities. It is obvious that it is more probable that a subscriber, connected to a telecommunication system, who has recently lifted the receiver, dials a digit than another user lifts the receiver. This means that the choice of a specific user to generate the next event depends on the actual state of the user and the states of its services. This is handled by introducing state weights, which model the relative probability of generating the next event compared to the other states of the service. Thus, the state weights are introduced on the behavior level to capture that the probability of the next event depends on the state in which the services of the different users are. The state weights are the basis for deriving the probabilities in the hierarchy.

One of the drawbacks of this model is that it is a complicated model, and it can be hard to find a suitable hierarchy and to define the state weights. Another drawback is that since both services and users are dependent of each other, and the model tries to take this into account, the model becomes fairly complex, although realistic.

7. Summary

The usage models presented here have their different advantages and disadvantages. The choice of model depends on the application characteristics and how important the accuracy is.

If the system is a batch system, then either the domain-based model or the algorithmic model is suitable, but not if the input history is important. If the input history is important, there are the grammar, the Markov, or the SHY models. These models take the input history into account, and the input can be described in detail if necessary. If the system is complex and has a large number of users, the grammar model becomes very complex, and the number of states in the Markov model grows very large. The models that can model the usage of these systems are the operational profile and the SHY models. The operational profile is the most widely used model.

Before the testing can start, test cases must be generated from the usage specification. This can be done by running through the usage specification and logging test cases. Basically, transforming the usage specification into an executable representation generates test cases and then they are executed with an oracle. The oracle determines the expected response from the system under the generated usage conditions. Another opportunity is that the oracle determines the correct output during testing, although this makes the testing less efficient in terms of calendar time.

C. Derivation of Usage Data

Reliability has only a meaning if it is related to the usage of the system. By applying usage-based testing, the system is tested as if being in operation, and the failure data is representative of the operational failures. Usage-based testing includes a usage profile, which is a statistical characterization of the intended usage. The characterization is made in terms of a distribution of the inputs to the system during operation.

The usage profile is not easy to derive. When a system is developed it is either a completely new system or a redesign or modification of an existing system. If there is an older system, the usage profile can be derived from measuring the usage of the old system. On completely new systems there is nothing to measure, and the derivation must be based on application knowledge and market analysis.

There are three ways to assign the probabilities in the usage profile.

1. Measuring the usage of an old system. The usage is measured during operation of an old system that the new system shall replace or modify. The statistics are collected, the new functions are analyzed, and their usage is estimated based on the collected statistics.

2. Estimate the intended usage. When there are no old or similar systems to measure on, the usage profile must be estimated. Based on data from previous projects and on interviews with the end users, an estimate on the intended usage is made. The end users can usually make a good profile in terms of relating the different functions to each other. The function can be placed in different classes, depending on how often a function is used. Each class is then related to the other by, for example, saying that one class is used twice as much as one other class. When all functions are assigned a relation, the profile is set according to these relations.

3. Uniform distribution. If there is no information available for estimating the usage profile, one can use a uniform distribution. This approach is sometimes called the uninformed approach.

III. DATA COLLECTION

A. Purpose

The data collection provides the basis for reliability estimations. Thus, a good data collection procedure is crucial to ensure that the reliability estimate is trustworthy. A prediction is never better than the data on which it is based. Thus, it is important to ensure the quality of the data collection. Quality of data collection involves:

- Collection consistency. Data shall be collected and reported in the same way all the time, for example, the time for failure occurrence has to be reported with enough accuracy.
- Completeness. All data has to be collected, for example, even failures for which the tester corrects the causing fault.
- Measurement system consistency. The measurement system itself must as a whole be consistent, for example, faults shall not be counted as failures, since they are different attributes.

B. Measurement Program

Measurement programs can be set up for a project, an organizational unit, or a whole company. The cost is, of course, higher for a more ambitious program, but the gains are also higher for more experience collected within a consistent measurement program.

Involving people in data collection implies, in particular, two aspects:

- Motivation. Explain why the data shall be collected and for what purposes it is used.
- Feedback. Report the measurements and analysis results back to the data providers.

Setting up a measurement program must be driven by specific goals. This is a means for finding and spreading the motivation, as well as ensuring the consistency of the program, i.e., for example, that data are collected in the same way throughout a company. The Goal-Question-Metric (GQM) approach provides means for deriving goal-oriented measurements. Typical goals when measuring reliability are to achieve a certain level of reliability; to get measurable criteria for deciding when to stop testing; or to identify software components, which contribute the most to reliability problems. The goals determine which data to collect for software reliability estimation and prediction.

C. Procedures

To achieve data of high quality, as much as possible shall be collected automatically. Automatic collection is consistent—not depending on human errors—and complete, as far as it is specified and implemented. However, automatic collection is not generally applicable, since some measurements include judgements, for example, failure classification. Manual data collection is based on templates and forms, either on paper or electronically.

D. Failure and Test Data

The main focus here is on software reliability models based on failure data. From the reliability perspective, failure data has to answer two questions:

- When did the failure occur?
- Which type of failure occurred?

The failure time can be measured in terms of calendar time, execution time, or the number of failures per time interval (calendar or execution). Different models require different time data. Generally, it can be stated that using execution time increases the accuracy of the predictions, but requires a transformation into calendar time in order to be useful for some purposes. Planning of the test period is, for example, performed in terms of calendar time and not in execution time. Thus, there is a need for mapping between execution time and calendar time. Keeping track of actual test time,

instead of only measuring calendar time, can also be a means of improving the prediction accuracy.

When different failure severity categories are used, every failure has to be classified to fit into either of the categories. Reliability estimations can be performed for each category or for all failures. For example, it is possible to derive a reliability measure in general or for critical failures in particular.

IV. SOFTWARE RELIABILITY MODELING

A. Purpose

As stated in opening, software reliability can be defined as the probability of failure-free operation of a computer program in a specified environment for a specified time. This definition is straightforward, but, when the reliability is expressed in this way, it is hard to interpret.

Some reasonable questions to ask concerning software reliability of software systems are:

- What is the level of reliability of a software system?
- How much more development effort must be spent to reach a certain reliability of a software system in a software development project?
- When should the testing stop? That is, can we be convinced that the reliability objective is fulfilled, and how convinced are we?

Here, the first question seems to be the easiest to answer. It is, however, not possible to directly measure the reliability of a system. This has to be derived as an indirect measure from some directly measurable attributes of the software system. To derive the indirect measures of reliability from the directly measurable attributes, software reliability models are used. Examples of directly measurable attributes are the time between failures and the number of failures in a certain time period (see Fig. 11).

The main objective of a software reliability model is to provide an opportunity to estimate software reliability, which means that Fig. 4 may be complemented as shown in Fig. 12.

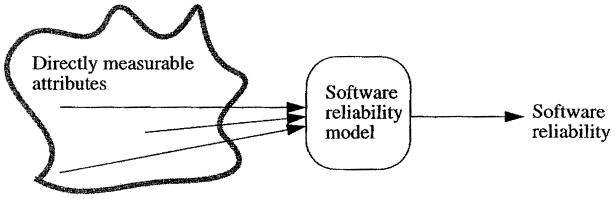


FIGURE 11 The reliability can be derived from directly measurable attributes via a software reliability model.

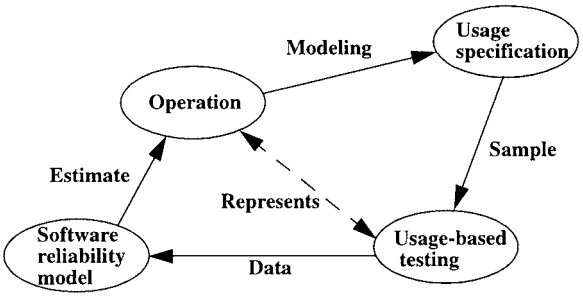


FIGURE 12 Relationship between operation, usage specification, usage-based testing, and software reliability models.

B. Definitions

As a starting point, we introduce some basic reliability theory definitions. Let X be a stochastic variable representing time to failure. Then the failure probability $F(t)$ is defined as the probability that X is less than or equal to t. We also define the survival function as $R(t) = 1 - F(t)$.

Some important mean value terms are displayed in Fig. 13. Here, the state of the system is simply modeled as alternating between two states: when the system is executing, a failure can occur and the system is repaired; and when the system is being repaired, it will after a while, when the fault is corrected, be executed again. This is iterated for the entire life cycle of the system.

The expected value of the time from a failure until the system can be executed again is denoted MTTR (mean time to repair). This term is not dependent on the number of remaining faults in the system.

The expected time that the system is being executed after a repair activity until a new failure occurs is denoted MTTF[1] (mean time to failure), and the expected time between two consecutive failures is denoted MTBF (mean time between failures). The two last terms (MTTF and MTBF) are dependent on the remaining number of software faults in the system.

The above three terms are standard terms used in reliability theory in general. In hardware theory, however, the last two terms are often modeled as being independent of the age of the system. This cannot, in most cases, be done for software systems.

Two simple but important relationships are

$$\text{MTBF} = \text{MTTF} + \text{MTTR},$$

$$\text{Availability} = \text{MTTF}/\text{MTBF}.$$

When modeling software reliability, the repair times do not have any meaning. Instead, only the times between consecutive failures are considered and therefore measured. In this case, the only term of the above three that

[1]Sometimes the term is defined as the time from a randomly chosen time to the next failure.

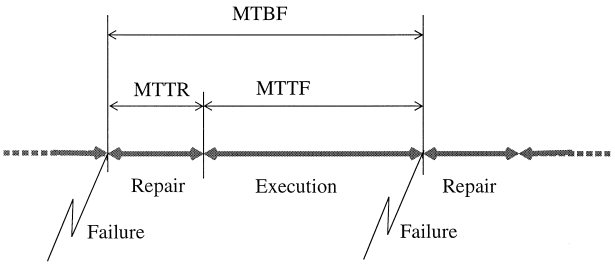


FIGURE 13 Three important mean value terms: MTTF, MTBF, and MTTR.

can be determined is the MTTF and the availability cannot be determined.

C. Principles

As stated in the previous section, the reliability must be derived as an indirect measure from directly measurable attributes of the software system. The directly measurable attributes are typically the times of failures, i.e., at what different times the different failures have occurred, or the number of failures in different time intervals.

These attributes can be measured in typically two different situations:

- When the software is under development and being tested. In this case, it is assumed that faults resulting in failures are corrected immediately. It could be the case that the faults are not removed immediately, but it can be some time from when the failure is detected until the fault is located and corrected. Most software reliability models do, however, not account for this time.
- When the software is in use and is being maintained. In this case, the faults are, in most cases, not corrected directly. This is instead done for the next release of the software system. In this case, the MTBF and MTTF are constant between two consecutive releases of the system.

The second situation is the simplest one, and it is similar to basic hardware reliability theory. In this case, the failure intensity can be modeled as constant for every release of the software. In the first case, however, the failure intensity cannot be modeled as being constant. Here, it is a function of how many failures that have been removed. This is a major difference compared to basic hardware reliability theory, where components are not improved every time they are replaced.

The situation where the failure intensity is reduced for every fault that is corrected can be modeled in a number of different ways, and a number of different models have been proposed. This section concentrates on the case when faults are directly corrected when their related failures occur.

The majority of all software reliability models are based on Markovian stochastic processes. This means that the future behavior after a time, say, t, is only dependent on the state of the process at time t and not on the history about how the state was reached. This assumption is a reasonable way to get a manageable model, and it is made in many other engineering fields.

Regardless of the chosen model and data collection strategy, the model contains a number of parameters. These parameters must be estimated from the collected data. There are three different major estimation techniques for doing this:

- The maximum likelihood technique
- The least square technique
- The Bayesian technique

The first two are the most used, while the last is the least used because of its high level of complexity.

The application of reliability models is summarized in an example in Fig. 14. In this example, the failure intensity is modeled with a reliability model. It could, however, be some other reliability attribute, such as MTBF. First, the real values of the times between failures in one realization are measured (1). Then the parameters of the model are estimated (2) with an inference method such as the maximum likelihood method. When this is done, the model can be used, for example, for prediction (3) of future behavior (see Fig. 14.)

D. Model Overview

Reliability models can be classified into four different classes:

1. Time between failure models
2. Failure count models
3. Fault seeding models
4. Input domain-based models

Since the first two classes of models are most common, they are described in some more detail, while the latter two classes only are described briefly.

1. Time between Failure Models

Time between failure models concentrate on, as the name indicates, modeling the times between occurred failures. The first developed time between failure model was the Jelinski-Moranda model from 1972, where it is assumed that the times between failures are independently exponentially distributed. This means that, if X_i denotes the

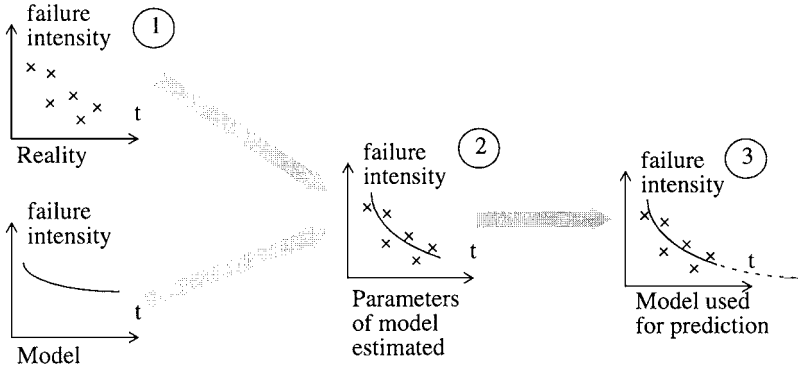


FIGURE 14 The application of reliability models. In this example the model is used for prediction.

time between the $(i-1)^{\text{th}}$ and i^{th} failure, then the probability density function of X_i is defined as in Eq. (1).

$$f_{X_i}(t) = \lambda_i e^{-\lambda_i t}, \tag{1}$$

where λ_i is the failure intensity after the $(i-1)^{\text{th}}$ failure has occurred (and before the i^{th} failure has occurred). In the Jelinski-Moranda model, λ_i is assumed to be a function of the remaining number of failures and is derived as in Eq. (2).

$$\lambda_i = \phi(N-(i-1)), \tag{2}$$

where N is the initial number of faults in the program and ϕ is a constant. The parameters in the model can be interpreted as follows: let N be the initial number of faults in the program and ϕ be a constant representing the per fault failure intensity.

The above formulas are together a model of the behavior of the software with respect to failures. It is not exactly representing the real behavior, merely a simplification of the real behavior. To be able to use this model, for example, for prediction, N and ϕ must be estimated from the measured data. It is possible to make a maximum likelihood estimate of the parameters. The likelihood function that can be used to estimate N and ϕ is found in Eq. (3).

$$L(t_1,\ldots,t_n;N,\phi) = \prod_{i=1}^{n} f_{X_i}(t_i) = \prod_{i=1}^{n} \phi(N-(i-1)) \times e^{-\phi(N-(i-1))t_i}, \tag{3}$$

where t_i is the measured values of X_i, i.e., the measured times between failures, and n is the number of measured times. By taking the natural logarithm of the likelihood function and simplifying, we obtain Eq. (4).

$$\ln L = n\ln\phi + \sum_{i=1}^{n}\ln(N-i+1) - \phi\sum_{i=1}^{n}(N-i+1)t_i \tag{4}$$

This function should be maximized with respect to N and ϕ. To do this, the first derivative with respect to N and ϕ can be taken. The $\hat{N}$ and $\hat{\phi}$, which satisfy that both the derivatives equals 0, are the estimates we are looking for.

After the Jelinski-Moranda model was published, a number of different variations of the model were suggested. Examples are:

- Failures do not have to be corrected until a major failure has occurred.
- The failure intensity does not have to be constant between successive failures. One proposal in the literature is to introduce an increasing failure rate (IFR) derived as $\lambda_i = \phi(N-(i-1))t$, where t is the time elapsed since the last failure occurred.
- A variant of the Jelinski-Moranda model, which accounts for the probability of imperfect debugging, i.e., the probability that a fault is not removed in a repair activity, has been developed. With this model, the failure intensity can be expressed as $\lambda_i = \phi(N-p(i-1))$, where p is the probability of imperfect debugging.

The Jelinski-Moranda model (with no variants) is presented here in some more detail, since it is an intuitive and illustrative model. In another situation when the main objective is not to explain how to use reliability models, it may be appropriate to use one of the variants of the model or a completely different model.

2. Failure Count Models

Failure count models are based on the number of failures that occur in different time intervals. The number of failures that occur is, with this type of model, modeled as a stochastic process, where $N(t)$ denotes the number of failures that have occurred at time t.

Goel and Okomoto (1979) have proposed a failure count model where $N(t)$ is described by a nonhomogenous Poisson process. The fact that the Poisson process is nonhomogenous means that the failure intensity is not

constant, which means that the expected number of faults found at time t cannot be described as a function linear in time (which is the case for an ordinary Poisson process). This is a reasonable assumption since the failure intensity decreases for every fault that is removed from the code. Goel and Okomoto proposed that the expected number of faults found at time t could be described by Eq. (5).

$$m(t) = N(1 - e^{-bt}), \tag{5}$$

where N is the total number of faults in the program and b is a constant. The probability function for $N(t)$ can be expressed as in Eq. (6).

$$P(N(t) = n) = \frac{m(t)^n}{n!} e^{-m(t)} \tag{6}$$

The Goel and Okomoto model can be seen as the basic failure count model, and as with the Jelinski-Moranda model, a number of variants of the model have been proposed.

3. Fault Seeding Models

Fault seeding models are primarily used to estimate the total number of faults in the program. The basic idea is to introduce a number of representative failures in the program and to let the testers find the failures that these faults result in. If the seeded faults are representative, i.e., they are equally failure prone as the "real" faults, the number of real faults can be estimated by a simple reasoning.

If N_s faults have been seeded, F_s seeded faults have been found, and F_r real faults have been found, then the total number of real faults can be estimated through Eq. (7).

$$N = N_s \cdot \frac{F_r}{F_s} \tag{7}$$

A major problem with fault seeding models is seeding the code with representative faults. This problem is elegantly solved with a related type of models based on the capture-recapture technique. With this type of model, a number of testers are working independently and separately to find a number of faults. Based on the number of testers that find each fault, the number of faults in the code can be estimated. The more testers that find each fault, the larger share of the faults can be expected to be found, and the fewer testers that find each faults, the fewer of the total number of faults can be expected to be found.

4. Input Domain-Based Models

By using these types of models, the input domain is divided into a set of equivalent classes, and then the software is tested with a small number of test cases from each class. An example of an input domain-based model is the Nelson model.

E. Reliability Demonstration

When the parameters of the reliability model have been estimated, the reliability model can be used for prediction of the time to the next failure and the extra development time required until a certain objective is reached. The reliability of the software can be certified via interval estimations of the parameters of the model, i.e., confidence intervals are created for the model parameters. But often, another approach, which is described in this section, is chosen.

A method for reliability certification is to demonstrate the reliability in a reliability demonstration chart. This method is based on faults that are not corrected when failures are found, but if faults were corrected, this would only mean that the actual reliability is even better than what the certification says. This type of chart is shown in Fig. 15.

To use this method, start in the origin of the diagram. For each observed failure, draw a line to the right and one step up. The distance to the right is equal to the normalized time (time $*$ failure intensity objective). For example, the objective may be that the mean time to failure should be 100 (failure intensity objective is equal to $1/100$) and the measured time is 80, then the normalized time is 0.8. This means that when the normalized time is less than 1, then the plot comes closer to the reject line; on the other hand, if it is larger than 1 then it comes closer to the accept line.

If the reached point has passed the accept line, the objective is met with the desired certainty, but if the reject line is passed, it is with a desired certainty clear that the objective is not met.

The functions for the two lines (accept line and reject line) are described by Eq. (8).

$$x(n) = \frac{A - n \ln \gamma}{1 - \gamma}, \tag{8}$$

where γ is the discrimination ratio (usually set to 2) and n is the number of observed failures. For the reject line A is determined by Eq. (9).

$$A_{rej} = \ln \frac{1 - \beta}{\alpha}, \tag{9}$$

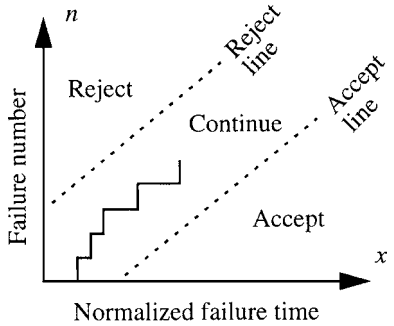


FIGURE 15 Software reliability demonstration chart, which is based on sequential sampling.

where α is the risk of saying that the objective is not met when it is and β is the risk of saying that the objective is met when it is not. For the accept line A is determined by Eq. (10).

$$A_{acc} = \ln\frac{\beta}{1-\alpha} \tag{10}$$

F. Accuracy of Reliability Predictions

The accuracy of the existing reliability models depends on the data that the prediction is based on. One model can get an accurate prediction for one set of test data, but can get an inaccurate prediction for another set of data. The problem is that it is impossible to tell which model has an accurate prediction for a particular set of test data.

When predicting the future growth of the reliability, one method to evaluate the accuracy of the model is to use a u-plot. A u-plot is used to determine if the predicted distribution function is, on average, close to the true distribution function. The distribution function for the time between failures is defined by Eq. (11).

$$F_{X_i}(t) \equiv P(X_i < t) = \int_0^t f_{X_i}(\tau)\, d\tau \tag{11}$$

The predicted distribution function is denoted $\hat{F}_{X_i}(t)$. If T_i truly had the distribution $\hat{F}_{X_i}(t)$, then the random variable $U_i = \hat{F}_{X_i}(X_i)$ is uniformly distributed in (0,1). Let t_i be the realization of T_i, and calculate $u_i = \hat{F}_{X_i}(x_i)$, then u_i is a realization of the random variable U_i. Calculating this for a sequence of predictions gives a sequence of $\{u_i\}$. This sequence should look like a random sample from a uniform distribution. If there is a deviation from the uniform distribution, this indicates a difference between $\hat{F}_{X_i}(t)$ and $F_{X_i}(t)$. The sequence of u_i consists of n values. If the n u_i's are placed on the horizontal axis in a diagram, and for each of these points a step function is increased with $1/(n+1)$, the result is a u-plot (see Fig. 16). The u-plot is compared with the uniform distribution, which is the line with unit slope through the origin. The distance between the unit line and the step function is then a measure of the accuracy of the predicted distribution function. In Fig. 16, we see that predictions for short and long times are accurate, but the predictions in between are a bit too optimistic; that is, the plot is above the unit line.

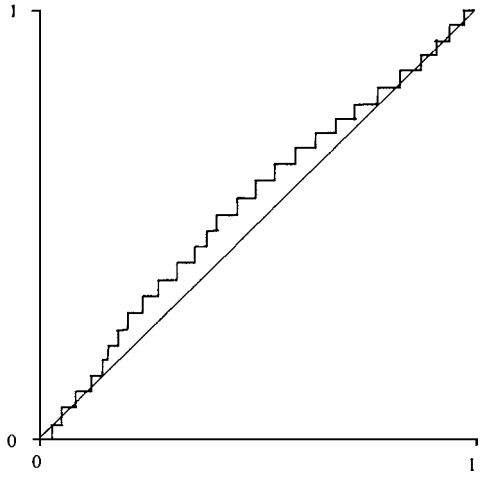


FIGURE 16 An example of a u-plot.

V. EXPERIENCE PACKAGING

A. Purpose

In all measurement programs, collected experience is necessary to make full use of the potential in software product and process measurement and control. The experience base is a storage place for collected measurements, predictions, and their interpretations. Furthermore, the models with parameters used are stored in the experience base.

The experience is used for different purposes:

- Constitute a baseline. Reference values for the attributes to be measured and a long-term trend, for example, in an improvement program
- Validate predictions. To judge whether predictions are reasonable by comparing data and predictions to the experience base
- Improve predictions. The accuracy of predictions can be improved by using data from earlier projects
- Enable earlier and more confident predictions. To predict product and process attributes in the early phases requires a solid experience base

All these purposes are valid for measurement of product and process attributes in general and reliability in particular. In the reliability area, we focus on two types of models in the experience base.

- The usage model and profile applied in usage-based testing have to be stored with the predictions made since a reliability prediction always is based on a usage profile. There is also a reuse potential of the usage model and profile between different projects.
- The reliability model and its parameters are stored to constitute a basis for early predictions of the reliability for forthcoming similar projects.

Experience packaging, related to these two model types, is further elaborated in Sections V.B and V.C.

B. Usage Model and Profile

The usage model and profile are descriptions of the intended operational usage of the software. A reliability prediction is conducted based on test cases generated from a usage profile and is always related to that profile. Hence, a reliability prediction must always be stored in the experience base together with the usage profile used.

Comparing the prediction to the outcome in operational usage can validate a reliability prediction. If there is a considerable difference between predicted and experienced reliability, one of the causes may be discrepancies between the usage profile and the real operational profile. This has to be fed back, analyzed, and stored as experience.

Continuous measurements on the operational usage of a product are the most essential experience for improving the usage profile. A reliability prediction derived in usage-based testing is never more accurate than the usage profile on which it is based.

The usage models and profiles as such contain a lot of information and represent values invested in the derivation of the models and profiles. The models can be reused, thus utilizing the investments better. Different cases can be identified.

- Reliability prediction for a product in a new environment. The usage model can be reused, and the usage profile can be changed.
- Reliability prediction for an upgraded product. The usage model and profile can be reused, but have to be extended and updated to capture the usage of added features of the product.
- Reliability prediction for a new product in a known domain. Components of the usage model can be reused. Some usage model types support this better than others.

C. Reliability Models

Experience related to the use of reliability models is just one type of experience that should be stored by an organization. Like other experiences, projects can be helped if experience concerning the reliability models is available.

In the first stages of testing, the estimations of the model parameters are very uncertain due to too few data points. Therefore, it is very hard to estimate the values of the parameters, and experience would be valuable. If, for example, another project prior to the current project has developed a product similar to the currently developed product, then a good first value for the parameters would be to take the values of the prior project.

Another problem is to decide what model to choose for the project. As seen in the previous sections, a number of different reliability models are available, and it is almost never obvious which one to choose. Therefore, it could be beneficial to look at previously conducted projects and compare these projects with the current one and also to evaluate the choice of models in previous projects. If similar projects have successfully used one specific reliability model, then this reliability model could be a good choice for the current project. On the other hand, if previously conducted similar projects have found a specific model to be problematic, this model should probably not be chosen.

Experience of reliability from previous projects can also be of use early in projects when reliability models cannot yet be used, for example, in the early planning phases. Experience can, for example, answer how much testing effort will be required to meet a certain reliability objective.

As with any other type of reuse, special actions must be taken to provide for later reuse. When the experience is needed, it is not possible to just look into old projects and hope to find the right information and conclusions from those old projects. Experience must be collected systematically and stored in the previous projects. This means, for example, that

- Measurements should be collected for the purpose of evaluation of the prediction models. Storing the choice of reliability model together with actual results can, for example, do this. This can be used to evaluate the reliability in, for example, a u-plot.
- Measurements should be collected for the purpose of understanding the model independent parameters such as initial time between failures and the fraction of found faults in different phases of the development.

The above-mentioned measurements are just examples of measurements that can be collected to obtain experience. The intention is not to provide a complete set of measures that should be collected with respect to reliability.

VI. SUMMARY

When you use a software product, you want it to have the highest quality possible. But how do you define the quality of a software product? In the ISO standard 9126, the product quality is defined as “the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs.” The focus here has been on one important quality aspect: the reliability of the software product.

Software reliability is a measure of how the software is capable of maintaining its level of performance under stated conditions for a stated period of time and is often expressed as a probability. To measure the reliability, the

software has to be run under the stated conditions, which are the environment and the usage of the software.

As the reliability is related to the usage, it cannot be measured directly. Instead, it must be calculated from other measurements on the software. A measure often used is the failure occurrence, or more precisely the time between them, of the software which is related to the usage of the software.

To calculate the reliability from the failure data, the data must be collected during operation or from testing that resembles operation. The testing method that is presented here is the usage-based testing method. In usage-based testing, the software is tested with samples from the intended usage. These samples are generated from a characterization of the intended usage and are representations of the operation. The characterization is made with a statistical model that describes how the software is used.

After the system is tested with usage-based testing, failure data that can be used for reliability calculations are available. The failure data are put into a statistical model to calculate the reliability. The calculations that are of interest are to estimate the current reliability, to predict how the reliability will change, or to certify with certain significance that the required reliability is achieved.

Software reliability is not only an important aspect for the end user, but it can also be used for planning and controlling the development process. Reliability predictions can be used to judge how much time is remaining before the required reliability is obtained. Estimations or certifications can be used to certify if we have obtained the reliability requirement and as a criterion to stop testing.

SEE ALSO THE FOLLOWING ARTICLES

COMPUTER ALGORITHMS • COMPUTER ARCHITECTURE • DATA STRUCTURES • OPERATING SYSTEMS • REQUIREMENTS ENGINEERING • SOFTWARE ENGINEERING • SOFTWARE MAINTENANCE AND EVOLUTION • SOFTWARE TESTING • STOCHASTIC PROCESSES

BIBLIOGRAPHY

Fenton, N., and Pfleeger, S. L. (1996). "Software Metrics: A Rigorous & Practical Approach," 2nd ed., International Thomson Computer Press, London, UK.

Goel, A., and Okumoto, K. (1979). "Time-dependent error-detection rate model for software reliability and other performance measures," *IEEE Trans. Reliab.* **28**(3), 206–211.

Jelinski, Z., and Moranda, P. (1972). Software Reliability Research. "Proceedings of Statistical Methods for the Evaluation of Computer System Performance," 465–484, Academic Press, New York.

Lyu, M. R., ed. (1996). "Handbook of Software Reliability Engineering," McGraw-Hill, New York.

Musa, J. D., Iannino, A., and Okumoto, K. (1987). "Software Reliability: Measurement, Prediction, Application," McGraw-Hill, New York.

Musa, J. D. (1993). "Operational profiles in software reliability engineering," *IEEE Software* **March,** 14–32.

Musa, J. D. (1998). "Software Reliability Engineering: More Reliable Software, Faster Development and Testing," McGraw-Hill, New York.

van Solingen, R., and Berghout, E. (1999). "The Goal/Question/Metric Method: A Practical Guide for Quality Improvement and Software Development," McGraw-Hill International, London, UK.

Xie, M. (1991). "Software Reliability Modelling," World Scientific, Singapore.

Software Testing

Marc Roper
Strathclyde University

GLOSSARY

Error A (typically human) mistake which results in the introduction of a fault into the software.

Failure The manifestation of a fault as it is executed. A failure is a deviation from the expected behavior, that is, some aspect of behavior that is different from that specified. This covers a large range of potential scenarios including, but by no means limited to, interface behavior, computational correctness, and timing performance, and may range from a simple erroneous calculation or output to a catastrophic outcome.

Fault Also known as a bug or defect, a fault is some mistake in the software which will result in a failure.

Test case A set of test data and expected results related to a particular piece of software.

Testing technique A mechanism for the generation of test data based on some properties of the software under test.

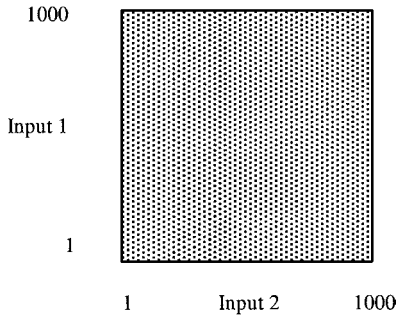
TESTING is essentially the dynamic execution of the software with a view to finding faults and consequently gaining confidence in the behavior of the software. The emphasis on dynamic execution is important. Testing is one of the primary ways of establishing that the software "does what it should do" (there are limitations to this that will be explored later), but it is by no means the only way. Alternative strategies ranging from verification to inspections all have a role to play in this. However, these alternatives inevitably do not involve executing the software and are hence described as static approaches. Put succinctly, testing involves cost-effectively choosing, executing, and analyzing some subset of a program's input data.

I. FUNDAMENTAL LIMITATIONS OF TESTING

A. The Input Domain

In simple terms testing is about sampling data from the input space of a system. The naïve reaction to this is to consider testing as involving running the program on every possible input. Unfortunately, this is impossible as the input space is prohibitively large. Consider, for example, a well-known program (a simple program is used for illustrative purposes here, but the results are equally applicable to any piece of software) which finds the greatest common divisor of two integers. Assume further that in addition to

constraining the input to deal with positive integers it is also limited to a maximum value of 1000. The input domain of the program can be visualised as follows:



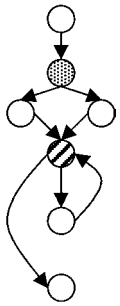
To exhaustively test this program it is necessary to choose *every* combination of Input1 and Input2—that is (1,1), (1,2), . . . , (234,789), . . . , (1000,1000). Even though it might be tempting to assume that after perhaps trying 10, or 50, or 100 pairs of inputs successfully that the program works, it is not safe to do this. There always exists the possibility that the next untried combination will be the one that causes a failure. So, the input domain for this simple program contains one million (1000×1000) possible distinct input combinations. Obviously, if the constraints on this program are relaxed slightly to allow input in the range 1 to 1,000,000, and include a further input (so it is now finding the greatest common divisor of three numbers) then the input domain grows to $10^{18}(1{,}000{,}000 \times 1{,}000{,}000 \times 1{,}000{,}000)$ distinct input combinations. To put this into perspective, if it was possible to create, run, and check the output of each test in one-hundredth of a second, then it would take over 317 million years to exhaustively test this program. This is obviously not a tractable proposition and this is still a very simple program with only three (constrained) integer inputs. Most nonartificial programs have input domains which are far larger and more complex than this.

B. The Internal Structure

It is clear that in the previous example, some of the problems might be alleviated by examining the internal structure of the program. This might make it possible to reason about how certain groups or ranges of data are treated and hence reduce the number of potential tests. The internal structure of a program can conveniently be represented by a directed graph (essentially a form of flowchart). To illustrate the concepts involved consider a simple program-like representation of the GCD calculation using Euclid's algorithm:

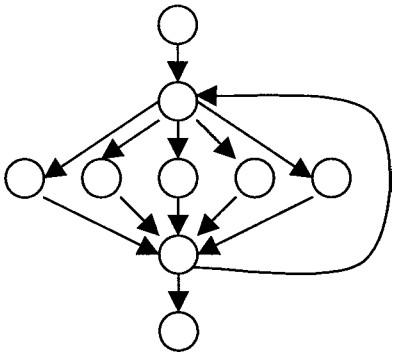
```
print("Input two integers");
read x;
read y;
if (x > y)
  { a = x; b = y;}
else
  {a = y; b = x;}
while (remainder (a,b) != 0)
  {
    r = remainder(a,b);
    a = b;
    b = r;
   }
print(b "is the GCD");

// remainder(a,b) returns the
// remainder after dividing a by b
```



The directed graph on the right illustrates the structure of the algorithm on the left. The nodes represent blocks of code and the edges represent transfer of control. In this example, the shaded node corresponds to the if condition (hence there are two possible exits) and the striped one to the while condition. The blank nodes correspond to the blocks of assignment statements and the final print statement. Note that this notation is not conventional but is merely used here for illustrative purposes—normally all nodes are blank. When testing a program it is valuable to consider the number of potential paths, or routes, through the program as this provides a different focus from the input domain. If it is possible to test all the paths through the program, then this can provide a level of confidence in the behavior of the program as every possible route will have been tried. In the GCD example, it might be tempting to think that there are two paths—one which takes one of the branches from the if-statement and then iterates around the while loop before finishing, and the other which takes the previously unexplored other if-statement branch (what it does after that is irrelevant). However, this is not the case. The key to the problem is the while-loop. The values of a and b are updated on every iteration of the while loop and so their values after one iteration are going to be different to their values after two iterations, or three. . . . For this reason, paths which differ only in the number of times they iterate around a loop are considered to be distinct. So, in the GCD example the number of paths is not 2, but constrained only by the range of the input values.

To illustrate how easily this number of paths becomes unmanageable, consider the simple example of a program to process a test mark for up to 10 students and categorize the result as one of A, B, C, D, or E. This could be represented by the flowchart-like directed graph below. One branch is required for each category of result and the return branch for the loop is executed up to nine times.



The number of paths through this program is calculated by:

$$\sum_{i=1}^{n+1} x^i,$$

where n is the maximum number of loop iterations (9) and x is the number of branches or decisions within the loop (5). This gives approximately 12.4 million paths! Once again this is an intractable problem.

II. DEVELOPMENTS IN TESTING

A. The Problem of Extrapolating from Observations

Given the problems described, it should be clear that testing software is an extraordinarily difficult task and one which is quite unrelated from the role that testing takes in other disciplines. To take a trivial example, a new mixture of concrete might be tested by subjecting blocks of it to various extremes of pressure, temperature etc. and its behavior at these limits allows for extrapolation over a range of other, less stressful, conditions. Enough is known about the material and the way in which it is constructed to be able to reason about these extrapolations with a degree of confidence. Software is very different in this respect. Not enough is known about the way the software is constructed and executed to be able to generalize from a test case—the GCD program might produce the correct (or incorrect!) answers for the values (12,34) but it is not possible to conclude *anything* about what it might produce for the values (11,34), (12,35), or (85920,84053). Admittedly, formal verification is the one exception to this, but this requires languages to be constrained and the presence of trusted compilers and execution platforms before it can be applied. Additionally, it is an expensive process, which makes it only feasible for small portions of safety-critical code.

The implications previously described are very serious since it means that looking for faults in software really is like searching for a needle in a haystack. A significant amount of effort can be put into the construction and execution of test cases, but it is all too easy for these to leave faults unexposed because the particular combination of test data that would have revealed them was not chosen.

B. The Development of Testing Methods

This problem led to the rise of the testing methods and techniques as ways of trying to make the selection of test data more likely to detect faults. For example, the technique of equivalence partitioning (see Section IV.C) deliberately tries to improve the extrapolation of results by requiring the tester to identify a set of data which, according to the specification, is treated identically. Once this set is identified then it is necessary to choose only one element from this set because the results of this test can be extrapolated over the whole set. This strategy helps to reduce the overall number of tests, prevents the tester from choosing redundant tests, and allows them to conclude more from the results. However, the identification of the input partition is informal and based purely on the specification and there is no formal analysis of the implementation of this partition. Consequently, there is always a chance that the hypothesis behind the technique is flawed. Other techniques such as boundary-value analysis or cause-effect graphing are based upon similar reasoning (that faults might typically lie at the boundaries of ranges or be revealed by particular combinations of data).

The approaches described here all fall into the general category of black box or functional testing. In other words, the software under test is treated like a black box, supplied with inputs, and its outputs observed. No attention is paid to the way the software has been constructed and so large portions of the software might go untested and be implemented in a way that is quite different from what the tester might expect from reading the specification. To complement this a set of techniques based upon the structure of the code was developed (and termed structural or white/glass box). These techniques focus on structural elements which are not so much good things to test (in the sense that in so doing is likely to reveal faults), but *bad things not to test*. For example, one of the most modest and intuitive is statement testing which requires that every statement in the source code be executed at least once. The rationale behind this is that leaving a statement *untested* is a very uncomfortable thing to do. Of course, the mere act of executing a statement is no guarantee that any fault contained within it is going to be revealed. It also has

to be executed with the data that are going to reveal the fault. To pick a simple example, the statement `x = x + x` is indistinguishable from the statement `x = x*x` if only tested on the values 0 and 2 for `x`. It is only when other data such as 3 or -1 are used that they calculate different results and it becomes clear that maybe one should have been used in place of the other. It is also easy to see other limitations of statement testing. For example, if there is a compound condition such as `if (x < 3 and y == 0.1)` statement testing will not require that the subconditions within this are thoroughly tested. It is sufficient just to choose values that will cause any subsequent dependent statements to be executed (such as $x = 2$ and $y = 0.1$, and $x = 4$ and $y = 0.1$). For reasons such as this stronger structural coverage techniques were introduced based on elements such as compound conditions or the flow of data within the program (see Section IV.B, which also explains the important development of perceiving coverage techniques as adequacy criteria). However, the stronger techniques were devised to address loopholes in the weaker techniques, but still have loopholes themselves.

There is another important factor to note at this point. It is naturally tempting to think of the more demanding testing techniques as being more likely to reveal faults. This is only the case if the data used for the more demanding technique are built upon and subsume that of the less demanding technique (for example, a set of data might be created to achieve statement coverage which is then augmented to achieve a higher level of coverage). If, however, the data sets are created independently then the situation can arise where, by chance rather than design, the data to achieve statement testing might reveal a fault which the data to achieve the higher level of coverage did not. This raises the issue of the relationship between techniques and external attributes such as reliability or, more nebulously, quality. For the reasons just discussed this relationship is by no means simple or well defined. Work in this area has produced some interesting results. For example, comparisons between partition testing (any criteria that divides up the input domain in some way—such as statement testing or branch testing) and random testing found partition testing to be generally preferable, but also discovered some situations where random testing was superior! This is an active area of research.

C. The Impact of Technologies

This general overview has indicated that there is quite some way to go before a general, effective, and predictable testing strategy is in place for even relatively straightforward imperative single-user systems. Given that software development technology is moving at an astonishing pace, what is the lesson for developers of, for example, distributed systems, real-time systems, e-commerce systems, web sites etc.? And what about the impact of the dazzling array of implementation technologies that can be used? Most research and development in the area of testing has concentrated on vanilla, imperative systems, with an occasional nod in the direction of the other technologies mentioned. Recently there has been a much greater focus on object-oriented systems, but this has appeared *long* after the technology itself became popular.

A paradigm shift such as the significant move to object-oriented programming as the primary means of developing software has a significant impact on what is known and assumed about testing strategies. For example, object-oriented systems consist of networks of communicating objects, each with their own encapsulated data that are accessed and manipulated by (typically) quite small methods. The structure of an object-oriented system is very different from that of an imperative system and consequently the structurally based testing strategies are not so obviously applicable. Traditional imperative systems are typically composed of modules that contain procedures or functions of reasonable structural complexity. Methods in object-oriented software on the other hand tend to be much simpler (and hence focusing on structural tests is not likely to have the same benefits) but usually exhibit a much greater interaction with other objects. For this reason the focus of testing has to shift and new strategies have to be developed. For example, in object-oriented systems there is a much greater emphasis on state-based testing than might tend to appear in imperative systems.

III. A GENERAL STRATEGY FOR TESTING

Obviously it is unreasonable to suggest that, just because what is formally known about testing is currently scant, software should not be tested. Pragmatically this cannot be allowed to happen—software *must* be tested. Although there is not one recommended over-arching strategy, there is general agreement that testing which is focused on particular properties of the software is likely to be far more effective than that which tries to test the software with no particular goal in mind (other than the general behavior of the software). The testing strategy is then a reflection of the property being tested. For example, timing properties might be tested by focusing on data which exercised particular hot-spots within the software, whereas reliability properties would use data which were derived from likely usage patterns of the software. A consequence of this approach, and acknowledging the limitations of testing itself, is that some (many!) tests are going to be omitted. This omission should be done in a calculated risk-based

fashion, as opposed to arbitrarily missing out those tests for which there was not enough time. A final note should be the emphatic statement that testing should not be relied on as the only means of ensuring good quality software—the complexity of the software construction activity means that every stage of development has a vital role to play and a poor product cannot be rescued by rigorous testing.

IV. TERMINOLOGY AND TECHNIQUES

There are a plethora of terms and techniques used within the software testing domain. This section aims to introduce the primary ones categorized according to techniques (structural or functional) or area of application.

A. Testing in the Software Development Life Cycle

There are distinct strategies used within the development life cycle that reflect the focus of testing at that particular stage.

1. **Unit Testing.** The act of testing the smallest piece of software in the system. This is typically the smallest independently compilable component and may correspond to elements such as a module or a class, depending on the type of software being tested. This is very often the focus of the structural and functional techniques described later.
2. **Integration Testing.** The process of joining together individual modules or components to form a complete system and determining that they communicate correctly and do not demonstrate any adverse collective behavior (for example, in the form of undesirable side-effects as a result of manipulating global data or interacting incorrectly). There are a number of different strategies to integration testing:
   - Top-Down—Which concentrates on building the top-level shell of the system first and moving toward integrating the lower-level modules later. Until they are integrated, lower-level modules have to be replaced by stubs—modules which take the place of the called modules to allow the higher level ones to be tested. These stubs have the same interface as the lower level modules but only a fragment of their functionality and are replaced with the real modules as the integration process progresses.
   - Bottom-Up—The opposite of top-down, which pulls together the lower-level modules first and works its way up to the outer layer of the system. This strategy requires the construction of drivers (the opposite of stubs)—modules that take the place of the calling modules and which have the role of invoking the lower-level modules, but again without the full range of functionality. Similarly to stubs, these are replaced by the real modules as the integration process works its way up the system.
   - Sandwich—An "outside-in" approach that combines the top-down and bottom-up strategies.
   - Big Bang—The simultaneous integration of all modules in one go.
   - Builds—Modules are integrated according to threads of functionality. That is, all the modules in a system which implement one distinct requirement (or use-case) are pulled together. This proceeds through the set of requirements until they have all been tested. In this approach a module might find itself being integrated several times if it participates in many requirements.

   A strategic approach is valuable in integration as a means of identifying any faults that appear. At this stage, faults are going to be the result of subtle interactions between components and are often difficult to isolate. A strategy assists this by focusing the tests and limiting the number of modules that are integrated at any one phase. The big-bang strategy is the obvious exception to this and is only a feasible strategy when a system is composed of a small number of modules. The actual strategy chosen will often be a function of the system. For example, in a an object-oriented system the notion of "top" and "bottom" often does not exist as such systems tend to have network rather than hierarchical structures and so a threaded strategy based upon builds is a more natural choice. In contrast, systems designed using more traditional structured analysis techniques according to their flow of data will often display a distinct hierarchy. Such systems are more amenable to the application of top-down (if the priority is on establishing the outward appearance of the system) or bottom-up (if the behavior of the lower-level data gathering modules is considered important) strategies.
3. **System Testing.** A level of testing that is geared at establishing that the functionality of the system is in place. This would often be based around a high-level design of the system. System testing will often involve analysis of other properties of the system by using techniques such as:
   - Performance Testing—Which examines the system's ability to deal efficiently with the demands placed upon it, for example, by focusing on the response time of the system under various loads or operating conditions
   - Stress Testing—The activity of determining how the system deals with periods of excessive demand

(in terms of transactions, for example) by overloading it for short periods of time
- Volume Testing—The operation of the system at maximum capacity for a sustained period of time
- Configuration Testing—The execution of the system on different target platforms and environments

4. **Acceptance Testing.** The highest level of tests carried out to determine if the product is acceptable to the customer. These would typically be based around the requirements of the system and usually involves the user. If there were many possible users, for example when the product is being built for a mass market, then the product would also be subjected to alpha and beta testing:
   - Alpha Testing—The idea of inviting a "typical" customer to try the product at the developer site. The ways in which the customer uses the product is observed, and errors and problems found are noted by the developers.
   - Beta Testing—Usually performed subsequent to alpha testing, a number of typical customers receive the product, use it in their own environment and then problems and errors are reported to the developer.
5. **Regression Testing.** The process of retesting software after changes have been made to ensure that the change is correct and has not introduced any undesirable side effects. Very often, the difficulty in regression testing is in identifying the scope of the change.

B. Structural Testing Techniques

Structural testing (or coverage) techniques are those that, as the name suggests, are based upon the internal structure of the software being tested (or at the very least, take into consideration the way that the software has been built). Structural testing approaches are often referred to as white box or glass box. They are also sometimes referred to as *Adequacy Criteria* which reflects the perception of test data in terms of structural coverage criteria. For example, for a given program, if a set of test data exercises all statements then the set is described as being statement testing, or statement coverage, adequate.

1. **Statement Testing/Coverage.** A level of test coverage that requires every statement in the software to have been executed
2. **Branch Testing/Coverage.** A level of test coverage that requires the true and false outcomes of every branch in the software to have been executed
3. **Multiple Condition Coverage.** A level of test coverage that requires every combination of the outcomes of sub-conditions within a compound condition to be tested. For example, a compound condition might take the form of $((a == b)$ and $(x < y))$. Multiple condition coverage would required four tests—one where both $(a == b)$ and $(x < y)$ are true, one where $(a == b)$ is true and $(x < y)$ is false, and so on. In between multiple condition coverage and branch coverage is a variety of other coverage levels which focus on the individual subconditions but do not exercise every combination and so might fail to achieve branch coverage.
4. **Mutation Testing.** A testing strategy based around deliberately introducing faults into a system and then determining the effectiveness of test data by measuring how many of these faults it detects. The faults introduced are typically small (changes to operators, variables, or constants), and mutation testing is based on the assumption that data that detect these small faults are also going to be effective at detecting bigger ones. In practice a large number of mutants are created automatically, each containing one fault.
5. **Data Flow Testing.** A testing strategy that is based on the way that data contained within variables is manipulated within a program. In contrast to other structural testing strategies which use control flow information to determine test criteria, data flow testing looks at the way that variables are used. Variables are categorised as being defined (assigned some value) or used (referenced in some way). The essence of data flow testing is to exercise all possible pairs of definition and usage (i.e. ways in which variables can be given values which can be subsequently referenced). Within the broader strategy of data flow testing are a number of less demanding strategies that, for example, focus on a subset of variable contexts, but still follow the same principles. The rationale for the strategy is that it mirrors the likely data usage patterns within the program.
6. **State-Based Testing.** A technique that focuses on identifying all the possible distinct states within a module. It is often employed when testing individual objects (the localized maintenance of state being one of the central tenets of object-orientation) or other systems that implement state machines.

C. Functional Testing Techniques

Functional testing techniques ignore the way that the software has been constructed and support the generation of

data that is based upon the specification, or some other external description of the software. As the name suggests, their focus is to help establish that the software correctly supports the intended functions. They are frequently referred to as black box techniques.

1. **Equivalence Partitioning.** The mechanism whereby classes of input data are identified from the specification on the basis that everything within this partition is treated identically according to the specification. Having identified such partitions it is only necessary to choose one test for each partition since the underlying assumption is that all values are treated the same. It is also recommended that data falling outside the partition also be chosen. In addition, the partitioning of the output domain should also be considered and treated in a similar way.
2. **Boundary Value Analysis.** Having identified the equivalence partitions, boundary value analysis is a technique that encourages the selection of data from the boundary of the partition on the basis that it is more likely that errors will have been made at this point. Data values should be chosen as close as possible to each side of the boundary to ensure that it has been correctly identified.
3. **Cause-Effect Graphing.** A technique that attempts to develop tests that exercise the combinations of input data. All inputs and outputs to a program are identified and the way in which they are related is defined using a Boolean graph so that the result resembles an electrical circuit (the technique has its roots in hardware testing). This graph is translated into a decision table in which each entry represents a possible combination of inputs and their corresponding output.
4. **Category-Partition Testing.** This attempts to combine elements of equivalence partitioning, boundary value analysis, and cause-effect graphing by exercising all combinations of distinct groups of values of input parameters. The method identifies the parameters of each "function" and, for each parameter, identifies distinct characteristics termed "categories." The categories are further subdivided into "choices" in the same way to equivalence partitioning, and any constraints operating between choices are then identified. Finally, test cases are generated which consist of the allowable combinations of choices in the categories.

SEE ALSO THE FOLLOWING ARTICLES

DATA STRUCTURES • REQUIREMENTS ENGINEERING • SOFTWARE ENGINEERING • SOFTWARE MAINTENANCE AND EVOLUTION • SOFTWARE RELIABILITY

BIBLIOGRAPHY

Adrion, W. R., Branstad, M. A., and Cherniavsky, J. C. (1982). "Validation, verification and testing of computer software," *ACM Computing Surv.* **14**(2), 159–192.

Beizer, B. (1990). "Software Testing Techniques," Van Nostrand Reinhold.

Hamlet, R. (1988). "Special section on software testing (Guest Editorial)," *Commun. ACM* **31**(6), 662–667.

Hamlet, R. (1992). "Are We Testing for True Reliability?" *IEEE Software* **9**(4), 21–27.

Harrold, M. J. (2000). Testing: A roadmap. *In* (A. Finkelstein, ed.), "The Future of Software Engineering," ACM Press, New York.

Ould, M. A., and Unwin, C. (eds.) (1986). "Testing in Software Development," Cambridge University Press.

Roper, M. (1993). "Software testing: A selected annotated bibliography," *Software—Testing, Verification and Reliability* **3**(2), 135–157.

Roper, M. (1994). "Software Testing," McGraw-Hill.

Software Testing Online Resources (STORM MTSU)http://www.mtsu.edu/∼storm/

Zhu, H., Hall, P. A. V., and May, J. H. R. (1997). "Software unit test coverage and adequacy," *ACM Comput. Surv.* **29**(4), 366–427.

WWW (World Wide Web)

Mike Jackson
University of Wolverhampton

GLOSSARY

Client A device that requests a service from a remote computer.

Internet An internet is a collection of networks connected together. The Internet is a global collection of interlinked networks. Computers connected to the Internet communicate via the Internet Protocol (IP).

Internet Protocol A protocol that was designed to provide a mechanism for transmitting blocks of data called datagrams from sources to destinations, where sources and destinations are hosts identified by fixed length addresses.

Protocol A description of the messages and rules for interchanging messages in intercomputer communication.

Server A device (normally a computer) that provides a service when it receives a remote request.

TCP Transmission control protocol. A protocol that provides reliable connections across the Internet. Protocols other than TCP may be used to send messages across the Internet.

TCP/IP Transmission control protocol implemented on top of the Internet protocol. The most commonly used protocol combination on the Internet.

FOR MANY PEOPLE the World Wide Web is the public face of the Internet. Although the majority of network traffic is still devoted to electronic mail, most people visualize a Web page when they try to portray the Internet. The Web has done much to popularize the use of the Internet, and it continues to be one of the enabling technologies in the world of e-commerce.

The World Wide Web (WWW) is a mechanism for making information stored in different formats available, across a computer network, on hardware from a variety of hardware manufacturers. At the present time (Fall 2000) the hardware is normally a computer; however, in future devices such as mobile phones will be increasingly used for displaying information obtained via the Web. It is in the spirit of the Web that the information can be accessed and displayed by software that can be supplied by a variety of software suppliers, and that such software should be available for a variety of operating systems (ideally every operating system one can name). A further aim is that the information can be presented in a form that makes it suitable for the recipient. For example, whereas information might appear to a sighted person in the form of a text document, a visually impaired person might be able to have the same information read to them by the receiving device. A similar mechanism could be used by someone working in

an environment that required them to observe something other than a display screen.

I. THE HISTORY OF THE WEB

The content of the Web consists of the contributions of anyone who creates a page and mounts it on a Web server; however, the ideas behind the architecture that makes this possible are generally accredited to a single person, Tim Berners-Lee. Berners-Lee is an Englishman who graduated in 1976 from Queens College Oxford with a degree in physics. In 1980 he began to work as a software consultant at CERN, the European Particle Physics Laboratory in Geneva. During this spell at CERN he wrote a program called Enquire. The purpose of Enquire was to track the software that existed at CERN: who wrote which program, what machine a program ran on, and so on. Information in Enquire was stored on pages, and pages were connected by bidirectional links. The software associated with Enquire made it possible to browse the information by following links from page to page. Links could address information within the same file as well as in external files. Enquire ran on a stand-alone computer (a Norsk Data) and had no facilities for exploiting network links. When Berners-Lee left CERN after 6 months, the original version of Enquire was lost.

After working on intelligent printer software, Berners-Lee applied for a fellowship with CERN and became part of its data acquisition and control group. CERN at this time had a huge collection of disparate hardware and software. Despite this disparity within the organizational infrastructure, scientists working there still needed to collaborate. Attempts at collaboration that required everyone to standardize on particular platforms failed dismally. Berners-Lee identified the need for a system similar to Enquire, which could operate on any of the available platforms within a networked environment. This software would enable the external hypertext links in the original Enquire program to span network connections. Such a system would need to be completely decentralized so that its users could obtain information without having to obtain passwords or permissions.

Berners-Lee approached his manager Mike Sendall for permission to develop his idea. He was told to write a proposal for the project. This was finished in March 1989 and was entitled "Information Management: A Proposal." The proposal was not initially greeted with great enthusiasm, but eventually Berners-Lee was given permission to develop the system, and a NeXT computer was purchased to assist in its development. Work commenced in October 1990. At this point the system was christened "The World Wide Web." At around the same time the original proposal document was rewritten with the help of Robert Cailliau. The revised proposal was entitled "Proposal for a HyperText Project."

Berners-Lee and a number of others began to develop software that demonstrated the basic principles of the World Wide Web. By May 1991 the Web was on general release across CERN. In August 1991 the software was made publicly available on the Internet. Many people outside of CERN began developing the software tools necessary to construct and deliver Web pages. A major breakthrough occurred in February 1993 with the release of the alpha version of Mosaic, a piece of software for viewing Web pages (a browser) written by NCSA's Marc Andreessen. Initially this was only available for the Unix operating system, but by September 1993 versions for Microsoft Windows and the Apple Macintosh were downloadable across the Internet. It was the availability of this program that helped to popularize the Web.

The first international Web conference was held in May 1994, and the World Wide Web Consortium (W3C) was founded in December of the same year. W3C continues to coordinate the development of new standards for the Web with Berners-Lee as its director.

Since 1994 the Web has experienced phenomenal growth. It would be foolish in a description of the Web such as this to estimate its size. Any estimation would almost certainly turn out to be too low and would anyway be hopelessly out of date within 6 months. The Web is now truly worldwide. Software for viewing Web content is freely available and normally supplied as standard with most computers. Software for serving Web pages is also generally available either as a free download or at a price from major software suppliers. The Web features in the plans of most commercial companies either as a platform for advertising or as a mechanism for direct selling. Its existence has fueled significant interest in what has come to be known as e-commerce, and a number of multimillion-dollar companies have come into existence simply because the Web is there to be exploited.

II. THE WAY THE WEB WORKS

The prerequisite that enabled the creation of the Web was the existence of the Internet. It was Tim Berners-Lee's aim in creating the Web to place information on any computer in a network and make it available on any other computer connected to that network. The computers could have different hardware architectures and different operating systems. In 1989 when the Web was first conceived, it was clear that the Internet with its TCP/IP protocols was capable of connecting heterogeneous hardware executing heterogeneous software. The Web was therefore built on

top of facilities provided by Internet software. The additional software that is necessary for the creation of a web falls into two categories: *servers* and *browsers*.

In the Web any number of computers may run server software. These machines will have access to the information that is to be made available to users of the Web. Web server software normally executes permanently on server machines (it is only possible to request information while server software is executing). The purpose of such server software is to listen for requests for information and, when those requests arrive, to assemble that information from the resources available to the server. The information is then delivered to the requestor. A machine that is connected to a network via the Internet protocol TCP (Transmission Control Protocol) sees the network as a collection of numbered ports. A message may arrive on any of these ports. Programs may attach themselves to a port (by specifying its number) and then listen for requests. The port normally used by a Web server is port 80 (although it is possible to use other ports). Requests for Web pages are therefore normally directed to port 80 of a machine known to be operating Web server software. The owners of Web sites (Webmasters) typically register an alias for their machines that makes it clear that they are open to requests for Web pages. Such aliases often begin with the character string www, as in www.scit.wlv.ac.uk. Numerous different types of Web server software are available. The most popular Web server software, such as Apache and Microsoft's Internet Information Server (IIS), is either freely downloadable from the Internet or else bundled with an operating system. Web servers that perform specialized tasks in addition to delivering Web-based information can be purchased from a variety of software suppliers. Most hardware/software platforms have at least one Web server product available for them.

Information available via the Web is delivered to the user via a browser. Browser software runs on client machines. A browser makes a request to a server and, when the relevant information arrives, presents it to the user (Fig. 1). The common conception of a browser is of a piece of software with a graphical user interface (GUI) capable of displaying pictures and playing sound. In fact the original concept of a browser was never intended to be so limited. A browser is simply a software product that can request information from a Web server and then present that information in a way that is suited to the recipient of that information. The original browser developed by Tim Berners-Lee made use of features of the NeXT platform, which were quite advanced (in 1990), and was indeed GUI based. At the same time, however, it was also recognized that not all Web client machines would be so advanced, and therefore a text-only browser was developed. From the beginning, the Web was intended to be available to the

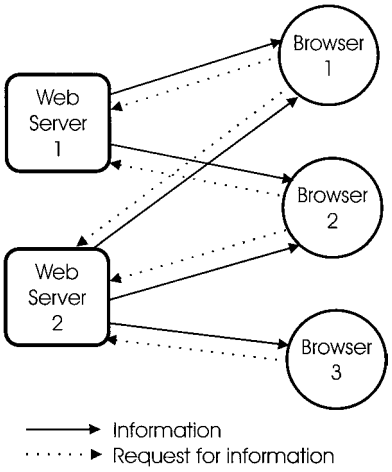


FIGURE 1 The way the Web works.

world, and therefore browsers for the visually and aurally challenged were envisaged. W3C continues to reflect this concern via its Web accessibility initiative. This vision of the use of a browser has resulted in many misunderstandings by those people who wish to provide information via the Web. Many people assume that the Web works in a WYSIWG (What You See Is What You Get) fashion. Collators of Web information therefore sometimes believe that the information transmitted will be received in exactly the same form in which it was constructed. This is not, and was never intended to be, the case. Another original aim for browser software was that it should be capable of creating Web content. This aim was not realized in the early commonly available browsers (although it was a feature of Berners-Lee's original browser on the NeXT platform); however, some later browser software has incorporated page editing functionality. Like servers, browsers are available for a wide variety of hardware/software combinations. The most widely used are Microsoft's Internet Explorer and Netscape Communication's Communicator. Some browsers (for example, Lynx) cater for computer terminals that do not provide graphical output. There are also browsers designed for hand-held computers.

The role of a browser is to request information from a server. This request can take two forms. The user of a browser can explicitly request a given piece of information on a given Web server. Alternatively, the user of the browser may request information that is referenced in the information they have already obtained from the same or another server. This second type of access makes use of a hypertext link. Ted Nelson invented the term "hypertext" in 1965. It refers to a document that contains links to related documents in appropriate places in the text. If a link is followed, it will lead to another document that expands on the issues raised by the document containing the link.

This allows a reader to approach a topic in a nonsequential manner that is said to be similar to normal human thought patterns. Web documents may contain links to other documents, and by selecting these links a user can access (or browse) other related material. In this second type of access each link contains information about identifying a particular piece of information on a particular server. As with explicit requests, it is the task of the browser to contact the server (usually on port 80) and request the information that has been cited.

III. TECHNOLOGIES THAT MAKE THE WEB WORK

In order to make the Web work in the way described above, Berners-Lee needed to develop three key elements that would build on top of the features already provided by the Internet. In his book *Weaving the Web*, he ranks these in the following order: Universal Resource Identifiers (URIs), HyperText Transfer Protocol (HTTP), and HyperText Mark-up Language (HTML).

If the Web is to function at all, there must be some way to uniquely specify a document located anywhere on the Internet. This is achieved through a URI. In essence, the URI naming scheme makes it possible for every item on the Web to have a unique name. Such a naming scheme must also include a technique for identifying the server on which a document is located. Given this, the user of a browser obtains documents by either explicitly entering a URI or by invoking a link that has a URI built into it.

All communication relies on the existence of sets of rules or protocols that govern what form the communication may take and how it will be understood. Computer communication generally requires a very tightly defined rule set. The transmission control protocol (TCP), which is universal across the Internet, defines how data is reliably transmitted and received by Internet hosts. It does not define how those hosts manipulate the data they send or receive. The Web requires a mechanism whereby a browser can request a document and a server can deliver it. This type of interchange will also require a server to reply that a document (for a variety of reasons) is unavailable. The protocol that enables such interchanges is HTTP. This was built on top of the Internet's TCP protocol (it was, however, designed to be compatible with other similar protocols).

The original concept of the Web was that all types of documents should be available on it. This is achievable, but it presents some problems. It is clearly unreasonable to expect a browser to be able to display all the different file formats that already exist and all the new formats that might exist in the future. In addition, the majority of the documents on the Internet will not contain links to other documents, i.e., they will not be hypertext documents. In order to realize the aims of the Web, it was therefore necessary to create a type of document that could be rendered by every browser (and hence become the lingua franca of the Web). It was also important that such a document could contain hypertext links to other documents, in a sense gluing the Web together. The language created by Berners-Lee to achieve these goals was called HTML.

These three innovations are described in detail in the next sections.

A. Universal Resource Identifier (URI)

What is currently meant by the term URI is a matter of debate. Rather than stating what a URI is, it is easier to say what it was originally meant to be and what it has become. The purpose of a URI is to uniquely identify any document on the Web. For this reason Berners-Lee originally called it the Universal *Document* Identifier. The idea was presented to the Internet Engineering Task Force (IETF) with a view to standardization. It was suggested during discussions that the word uniform be used instead of universal. In addition, the IETF argued that Berners-Lee's implementation of a URI specified an address that might be transient, so that the word locator was more appropriate than identifier. Thus the term URL or Uniform Resource Locator came into existence. A discussion of what a locator is can be found in IETF Request For Comments (RFC) 1736. (All Internet standards are defined in IETF Requests for Comments.)

A statement of what is meant by a URI in the context of the Web can be found in RFC 1630. This does not define a standard for URIs, but it does say how they work on the Web. It defines two types of URIs: Uniform Resource Locators (URLs), which reference Web documents using existing Internet protocols, and Uniform Resource Names (URNs), which attach unique persistent names to Web documents. The standard for URLs appears in RFC 2396. A URN is assigned without reference to a resource's Internet address, URLs, in contrast, contain the address of the Internet host on which the resource resides. Consequently, a URL may become invalid if the resource to which it refers is moved.

The idea of an URN is appealing as documents on the Internet are often relocated to new servers. An organization may legitimately move its data from one server to another. If it were possible to maintain a directory of names that could associate a name with its location on the Internet, then no Web links would be broken when data was moved. In the URL mechanism, as we shall see, Internet host addresses form part of the document address, and therefore when a document is moved any links that reference that document become invalid. In practice, however,

a satisfactory mechanism for implementing URNs has yet to be implemented, and therefore for all practical purposes all URIs in use are URLs. This description of the Web will therefore concentrate on describing the nature of URLs.

URIs were designed to be extensible, complete, and printable. The Web was created in order to be able to integrate all the ways in which the Internet has been traditionally used; for example, it is able to deal with connectionsl to electronic mail and file transfer facilities. URIs had to be capable of encompassing these and any future Internet facilities. The mechanism for encoding a URI must be capable of providing a representation for every legal Internet address. Any URI may be represented as a set of 7-bit ASCII characters, making it possible to exchange all URIs as written text on pieces of paper.

URIs consist of two parts, a *scheme* and a *path*. A colon separates the scheme and the path in the written URI. In the URI http://www.scit.wlv.ac.uk, the scheme is http (hypertext transfer protocol) and the path is //www.scit.wlv.ac.uk. The way in which the path is interpreted depends on the scheme in use. The most commonly used URI schemes are listed in Table I. All the schemes shown in Table I are normally referred to as URLs; the prefix urn is reserved for future use when a Internet resource naming mechanism comes into operation.

The way in which URLs reference the Internet can best be seen by considering some examples of their use. For example, the URL

```
mailto:someone@somehost.com
```

when expressed as a hyperlink and invoked by a user of a browser, will initiate the sending of an electronic mail message to the email address `someone@somehost.com`.

The http scheme is more complex. Again it can be explained by considering some examples. In these examples the word "page" has been used to refer to a Web document.

The URL

```
http://www.scit.wlv.ac.uk
```

refers to the "home" page of the Web server with the Internet address `www.scit.wlv.ac.uk`. As a port number is not explicitly specified, it will be assumed to be listening on the standard Web server port, port 80.

The URL

```
http://www.scit.wlv.ac.uk:8000
```

refers to the "home" page of a Web server located with the Internet address `www.scit.wlv.ac.uk`, which is listening on port 8000.

The URL

```
http://www.scit.wlv.ac.uk/myfile.html
```

refers to something called myfile.html that is accessible by the Web server listening on port 80 with the address `www.scit.wlv.ac.uk`. Most likely this will be a file containing HTML, but it may in fact turn out to be a directory or something else. The exact nature of what is returned by the server if this link is followed will depend of what myfile.html turns out to be and the way the server is configured.

The URL

```
http://www.scit.wlv.ac.uk/~cm1914
```

refers to the home page of user cm1914 on the machine with address `www.scit.wlv.ac.uk`. The full file name of such a page will depend on the server configuration.

The URL

```
http://www.scit.wlv.ac.uk/mypage.
    html#para4
```

refers to a fragment within the page referenced by `http://www.scit.wlv.ac.uk/mypage.html` called `para4`.

The URL

```
http://www.scit.wlv.ac.uk/phonebook?
    Smith
```

references a resource (in this case a database) named phonebook and passes the query string "Smith" to that resource. The actual document referenced by the URL is the document produced when the query "Smith" is applied to the phonebook database.

One important aspect of the definition of URIs (which featured in the original RFC) is a mechanism for specifying a partial URI. This allows a document to reference

TABLE I URI Schemes and Their Associated Protocol Types

Scheme prefix	Type of access
http	Hypertext transfer protocol
ftp	File transfer protocol
gopher	Gopher protocol
mailto	Electronic mail address
news	Usenet news
telnet, rlogin, tn3270	Interactive terminal session
wais	Wide area information servers
file	Local file access

another document without stating its full URI. For example, when two documents are on the same server, it is not necessary to specify the server name in a link. This makes it possible to move an entire group of interconnected documents without breaking all the links in those documents.

The examples just given have demonstrated how URIs can be used to provide links to any resource on the Internet. As new resources become available, new schemes are proposed and the debate over the implementation of URNs continues. W3C operates a mailing list on the topics of URIs in order to support these developments.

B. HyperText Transfer Protocol (HTTP)

Tha basic architecture of the Web consists of browsers that act as clients requesting information from Web servers. Computer-to-computer communications are described in terms of protocols, and Web interactions are no exception to this rule. In order to implement the prototype Web, Berners-Lee had to define the interactions that were permissible between server and client (normally a browser, but it could in fact be any program). The HyperText Transfer Protocol (HTTP) describes these interactions. The basis of the design of HTTP was that it should make the time taken to retrieve a document from a server as short as possible. The essential interchange is "Give me this document" and "Here it is."

The first version of HTTP was very much a proof of concept and did little more than the basic interchange. This has become known as HTTP/0.9. The function of HTTP is, given a URI, retrieve a document corresponding to that URI. As all URIs in the prototype Web were URLs, they contained the Internet address of the server that held the document being requested. It is therefore possible to open a standard Internet connection to the server using TCP (or some other similar protocol). The connection is always made to port 80 of the server unless the URI address specifies a different port. HTTP then defines the messages that can be sent across that connection.

In HTTP/0.9 the range of messages is very limited. The only request a client may make is GET. The GET request will contain the address of the required document. The response of the server is to deliver the requested document and then close the connection. In HTTP 0.9 all documents are assumed to be HTML documents.

HTPP/0.9 is clearly not sufficient to meet the aspirations of the Web. It was quickly enhanced to produce HTTP/1.0. This protocol is described in RFC 1945. For the first few years of the Web's existence HTTP/1.0 was the protocol in use. In July 1999 the definition HTPP/1.1 became an IETF draft standard and the majority of commonly used Web servers now implement it. HTTP/1.1 is described in RFC 2616.

HTTP/1.0, like its predecessor, is a *stateless* protocol, meaning that no permanent connection is made between the client and the server. The sequence of events is always as follows:

The browser connects to the server
The browser requests a resource
The server responds
The server closes the connection

This means that the behavior of neither the server nor the browser is affected by previous requests that have been made. This feature has had profound implications for some types of Web access, for example, database access across the Web, where the history of a transaction is important.

The format of an HTTP/1.0 message (request or response) is as follows:

An initial line
Zero or more header lines
A blank line
An optional message body (e.g., a file, or query data, or query output)

As in all Internet protocols a line is terminated by the ASCII characters for carriage return (CR) and line feed (LF), hexadecimal 0D and 0A. Therefore the blank line is simply a CR followed by a LF.

The initial line specifies the nature of the request or response. The header lines describe the information contained in the initial line or in the message body. The message body is used to hold data to be sent to the server or received by the browser. If sent to a server it will typically contain query terms. If received by the browser it will typically contain the data requested.

The content of the initial line will depend on whether the message is a request or a response. If it is a request it has the following format:

Method Request-URI HTTP-version

For example:

```
GET index.html HTTP/1.0
```

This asks the server to return the resource corresponding to index.html and specifies that the protocol in use is HTTP/1.0. Note that only the part of the URI that specifies the resource is sent. The full URI would include the server address, but as this is an interchange between a given server and the browser it is not necessary to repeat this. Similarly, if the hyperlink specifies a fragment of a document (using the # notation described in the section on URIs), this part of the URI is not passed to the server.

TABLE II Methods Specified in the HTTP Protocol

Method name	Action
GET	Requests a resource specified by a URI from a server. The action of GET can be made conditional if certain headers are supplied. If successful it will result in a response which contains the requested resource in the message body.
HEAD	Similar to GET, except that if the request is successful only the header information is returned and not the resource. This could be used when a user requires to know whether a resource exists but is not interested in its content.
POST	This is used to send data to the server. A typical use of POST would be in a Web-enabled database system. In this type of system the browser might display a form in order to collect input data. The URI in this case would specify a program that could take data from the message body and insert it into the database. The introduction of the POST method has been extremely important in terms of making the Web interactive.

Where a fragment is requested, the browser acquires the complete document and then displays the part of the document corresponding to the fragment.

HTTP/0.9 supports the single method GET. HTTP/1.0 introduced two additional methods, HEAD and POST. These are described in Table II.

The initial line of a response message from a server (which will only be sent after a request has been made) has the format

HTTP-Version Status-Code Reason-Phrase

For example:

```
HTTP/1.0 404 Not found
```

Here the client is informed that the resource it has requested cannot be found on the server to which it has sent a request. The status code was designed to be machine readable and the reason phrase to be understandable by humans. The format of the status code is Nxx, where N indicates a general category of code. These categories are shown in Table III.

The header fields in HTTP allow additional information to be passed between the client and the server. The format of a header line is:

Header name:value

For example:

```
From: MyEmailAddress@myhost.com
```

TABLE III HTTP Status Code Categories

Status code category	Meaning
1xx	Informational
2xx	The request was successfully carried out
3xx	The request must be redirected in order to be satisfied
4xx	The request cannot be carried out because of client error
5xx	Server error; although the request appeared to be valid, the server was unable to carry it out

This header could be inserted into a request to indicate the email address of the user requesting the resource. Servers can use headers such as this to gather usage statistics.

The set of permitted headers is defined in the protocol. HTTP/1.0 defines 16 headers, whereas HTTP/1.1 defines 46 headers.

The most significant header to appear in HTTP/1.0 was the Content-Type header. This header indicates the media type of the message body. HTTP/0.9 made the assumption that every document on the Web would contain HTML. HTTP/1.0 makes provision for any type of document and for the document type to be indicated by the Content-Type header. A browser that receives a document from a server can examine the Content-Type header to determine how it should deal with the document. Browsers can normally deal with plain text and HTML directly. If instead they receive a file that has been produced by a word processor, they can handle it by launching an appropriate "helper" application. The Content-Type field will indicate which application should be launched.

The introduction of this header was a significant step in making it possible for a browser to successfully handle files of any type. There still remains the problem, however, of defining a set of values for the header that can identify the type of any Internet document. Fortunately, a naming scheme for document types was defined in RFC 1521, which describes a mechanism known as Multimedia Internet Mail Extensions (MIME). This proposal set up a central registry of document types for the Internet. This scheme is sufficient for indicating the type of any document transmitted using HTTP, and therefore the Content-Type header of a document is set to its MIME type. An example of a Content-Type header is:

```
Content-Type: text/html
```

This indicates that the document transmitted has type text and subtype html; in other words, it is an HTML document. If the Content-Type header is

```
Content-Type: application/msword
```

it indicates that the document received is in a format that can be handled by the application program Microsoft Word.

HTTP/1.0 was introduced to make the Web a reality. This was the major design goal, and aspects such as efficient use of network resources, use of caches, and proxy servers were largely ignored. A public draft version of HTTP that addresses these issues, HTTP/1.1, first appeared in January of 1996. At the time of writing it is nearing adoption as a formal Internet standard and is described in RFC 2616. Many recently released servers and browsers implement HTTP/1.1.

HTTP/1.1 is essentially a superset of HTTP/1.0. HTTP/1.1 servers must continue to support clients who communicate using HTTP/1.0. The most important additional features introduced in HTTP/1.1 are as follows:

- Clients may make a connection with a server and without opening a further connection may request multiple documents from that server. This gives a faster response time, as opening a TCP connection imposes a significant time overhead.
- Facilities are provided to enable clients to ensure that data is only transmitted when the copy of a document sent in response to a previous request is no longer the same as the version on the server.
- A server can begin to send a page before its total length is known. Many pages sent across the Web are generated on-the-fly. HTTP/1.1 allows faster response times by permitting part of a response to be sent before the whole response has been generated.
- A new mechanism allows multiple server names to operate at the same Internet address. This makes more efficient use of the finite number of Internet addresses.
- Additional methods include PUT, DELETE, TRACE and CONNECT.

The existence of URIs makes it possible to address every document on the Internet. HTTP provides a mechanism whereby if a URI is given, the document represented by that URI can be obtained from a remote server.

C. HyperText Markup Language (HTML)

The third technology necessary in order to construct the World Wide Web was a document description language in which URIs could be embedded. These then act as hyperlinks to local or remote documents. A number of possibilities existed at the time the Web was being created. Berners-Lee, however, considered that all the existing options were too complex for representing hyperlinked documents suitable for display on nongraphical terminals. He designed his own language, which was called HTML. This language was based on the Standard Generalized Mark-up Language (SGML) standard.

An important aspect of HTML is that it is readable by humans as well as machines. Given an HTML document, a human reader can make reasonable sense of it. A computer can take the same source and display it in a way suited to whatever output device is available. The outcome of this requirement was that HTML is based on plain text. A mechanism for describing the structure of plain text documents already existed in the ISO standard (ISO standard 8879:1986) for SGML. SGML was widely accepted by the Hypertext research community as an appropriate platform for the description of hyperlinked documents.

SGML provides a mechanism to define a mark-up language. Mark-up languages are plain text languages containing textual tags that assign meaning to different parts of the document. In languages developed from SGML, tags are always surrounded by angled brackets (< and >). Most tags occur in pairs, an opening tag and a closing tag. For example, in HTML the beginning of a document title is signified by the <TITLE> tag and the end of the title by the </TITLE> tag. The <TITLE> tag is one of the simplest types of tag; other tags may possess attributes. For example, the anchor tag (<A>) in HTML may have a URL as an attribute. One advantage of a mark-up language based on SGML is, as outlined previously, that humans can read it. In addition, SGML provides mechanisms whereby mark-up languages can be quickly parsed and understood by a computer.

A simple HTML example shown in Fig. 2 illustrates how the language is used. Figure 2 lists the HTML found in a simple Web page. Previously in this description of the Web, the word "document" has been used to describe a Web resource that may have been encoded in HTML or another scheme. Web documents encoded in HTML are generally referred to as pages. The terms will be used interchangeably from now on.

The first two lines of the file shown in Fig. 2,

```
<!DOCTYPE HTML PUBLIC"-//W3C//DTD HTML 4.0
Transitional//EN"
"http://www.w3.org/TR/
REC-html40/strict.dtd">
```

are required to meet the SGML requirement that files containing mark-up should either contain the definition of the mark-up scheme in use or indicate where the definition can be found. Every SGML-based mark-up scheme has a Document Type Definition (DTD). The DTD specifies which tags must be used, which tags can be used, and how those tags can be arranged (ordered and nested). The file in Fig. 2 has been encoded using the tags defined in the HTML 4.0 Transitional Recommendation from W3C.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/strict.dtd">
<HTML>
<HEAD>
<TITLE>Simple HTML Page</TITLE>
</HEAD>
<BODY>
<H1>Example HTML Page</H1>
<P>
This page contains a link to the home page of the
<A HREF="http://www.wlv.ac.uk">
University of Wolverhampton</A>.
</P>
<P>
<A HREF="http://validator.w3.org/check/referer">
<IMG BORDER=0
SRC="http://validator.w3.org/images/vh40"
ALT="Valid HTML 4.0!" HEIGHT=31 WIDTH=88>
</A>
</P>
</BODY>
</HTML>
```

FIGURE 2 The contents of an HTML file describing a Web page.

A URL (`http://www.w3.org/TR/REC-html40/strict.dtd`) that references the HTML 4.0 DTD is provided. In this particular case a version of the DTD that prohibits the use of deprecated tags has been used.

The `<HTML>` tag that begins the actual HTML document is matched at the end of the file by a `</HTML>` tag. SGML requires that documents consist of a single element enclosed within a pair of tags, and the `<HTML>` and `</HTML>` tags serve this purpose. In SGML mark-up schemes, wherever a tag pair is used to enclose some other text, the closing tag will have the same name as the opening tag prefixed by a '/'.

HTML documents have two sections: a head and a body. The head contains information about the document that is normally additional to that contained within the document. The `<HEAD>` tag indicates the beginning of the head section. This end of this section is indicated by the `</HEAD>` tag. In the page shown in Fig. 2 the only information given is the title of the document. This appears within the `<TITLE>` and `</TITLE>` tags. The most commonly used GUI browsers display the title of the page (which in this case is "Simple HTML Page") at the top of the browser window, but this is not a mandatory requirement. Strings within `<TITLE>` tags could, for example, be searched by Web applications that wish to find pages with a particular

keyword in the title (even if that title is never displayed). Other information that may be included in the head section is the name of the page author, keywords describing the page, the name of the editor used to produce the page, etc.

The body section of the page contains descriptions of the part of the document that should be displayed by a browser. This section is delineated by the <BODY> and </BODY> tags.

The line <H1>Example HTML Page</H1> defines a heading with the text "Example HTML Page." HTML 4.0 defines six levels of headings H1 through H6. H1 is considered to be a heading of the greatest importance and H6 a heading with the least importance. Browsers often display headings in bold and at a font size that distinguishes them from the body text. The lines

```
<P>
This page contains a link to the
home page of the
<A HREF= "http://www.wlv.ac.uk ">
University of Wolverhampton</A>.
</P>
```

define a paragraph. All the text between the <P> and </P> tags forms part of the paragraph. Browsers cannot guarantee the layout they will use for text, as they cannot know the screen space they will have to use in advance. The layout of the text in an HTML file has no influence on the way it will appear to someone viewing the text on the Web. Visual browsers normally indicate paragraphs by inserting white space between them, but this is not mandatory. The paragraph shown contains a hyperlink to another document. The link is declared in the anchor (A) tag. The anchor tag has an attribute HREF that is set to a URL (in this case http://www.wlv.ac.uk). Since the <A> and </A> tags surround the text "University of Wolverhampton," it is this text that will contain the link to the document located at http://www.wlv.ac.uk.
The second paragraph,

```
<P>
<A HREF="http://validator.w3.org
/check/referer">
<IMG  BORDER=0
SRC= "http://validator.w3.org
/images/vh40"
ALT="Valid HTML 4.0!" HEIGHT=31
WIDTH=88>
</A>
</P>
```

contains a hyperlink that is attached to an image rather than being associated with text. The link specifies the URL of an HTML validation service operated by W3C. The image is specified using the <IMG> tag. This is a tag that does not enclose text, and therefore a corresponding closing tag is not necessary. A number of attributes define how the image is to be displayed. The BORDER attribute controls how wide the border around the image should be: in this case no border is to be displayed. The SRC attribute specifies the URL that gives location of the image file. The HEIGHT and WIDTH attributes give the browser an indication of the scaling that should be applied to the image when it is displayed. The ALT attribute specifies what text should be displayed if the browser is unable to display images. It could also be used by Web page processing software to determine what the image represents.

Many other HTML tags beyond those shown in this example are defined in the HTML 4.0 recommendation.

When the HTML shown in Fig. 2 is displayed in one of the most commonly used Web browsers, it is rendered as shown in Fig. 3.

There are number of things to note about Fig. 3. The browser has chosen to put the text that appeared between the <TITLE> and </TITLE> tags into the title of the window. The text for the heading has been rendered in a bold with a large font size. In addition, the browser has separated the heading from the first paragraph with white space.

The text of the first paragraph has been reformatted to fit into the size of the browser window. The text between the <A> and the </A> tags has been underlined (seen in color it appears blue). This is this particular browser's default way of indicating that this text is a hyperlink. If the browser user clicks on the hyperlink, the browser will issue an HTTP request to the server indicated in the URL referenced in the HREF attribute of the anchor tag. This will request the document indicated by the URL and the browser will eventually display it.

As the image and the text are in separate paragraphs, the browser uses white space to separate them. The image

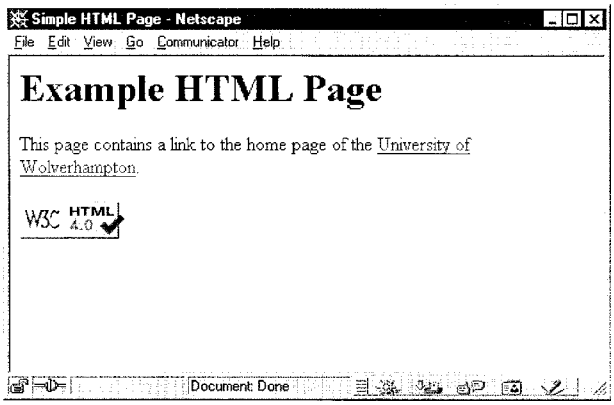


FIGURE 3 The HTML file in Fig. 2 displayed in a browser.

has been obtained from a server completely separate from the server from which the text was retrieved (its location is indicated by the URL in the IMG SRC attribute). Since the image is a hyperlink, if the user chooses to select it a new document will be retrieved. In this example the document will be one obtained from the W3C HTML validation service and will contain a report on the validity of the HTML describing the page (generated by checking it against the HTML 4.0 DTD).

IV. SUMMARY

This article has described the key Web technologies: URIs, HTTP, and HTML. Since the creation of the Web many other technical innovations (e.g., Active Server Pages, browser scripting languages, Java Applets, etc.) have contributed to ensuring its popularity and ubiquity. The three technologies discussed here, however, are the ones that provide the basis for the deployment of all Web-based software. They continue to develop as a result of commercial pressure or via the standardization efforts of W3C.

New schemes for URIs are devised as new media types are introduced on the Web. The HTTP/1.1 standardization effort is nearing completion. It is already clear, however, that HTTP/1.1 is not a lasting solution for the Web. A working group on HTTP Next Generation (HTTP-NG) has been set up by W3C. This group has already established a number of directions for the development of HTTP.

HTML has undergone the most changes since its creation. The original set of tags was quite small. These have been considerably expanded in later versions of HTML. Many changes have come about from the acceptance of what were at one time browser-specific tags. Finally, it has been recognized that HTML cannot continue to grow forever in this way and that at some point the set of tags supported must be frozen. At the same time it is admitted that the need for new tags will continue. A subset of SGML called Extensible Markup Language (XML) has been defined. This will enable purpose-specific mark-up languages to be defined, and an XML-based version of HTML called XHTML (Extensible HyperText Markup Language) has become the latest W3C recommendation.

SEE ALSO THE FOLLOWING ARTICLES

CRYPTOGRAPHY • DATABASES • INFORMATION THEORY • NETWORKS FOR DATA COMMUNICATION • SOFTWARE ENGINEERING • SOFTWARE MAINTENANCE AND EVOLUTION • WIRELESS COMMUNICATIONS

BIBLIOGRAPHY

Berners-Lee, T. (1989). "Information Management: A Proposal." [Online] 2 May 2000. <http://www.w3.org/History/1989/proposal.html>.

Berners-Lee, T. (1994). "RFC 1630: Universal Resource Identifiers in WWW." [Online] 2 May 2000. <http://www.ietf.org/rfc/rfc1630.txt>.

Berners-Lee, T. (1999). "Weaving the Web," Orion Business, London.

Berners-Lee, T., and Cailliau, R. (1990). "WorldWideWeb: Proposal for a HyperText Project." [Online] 2 May 2000. <http://www.w3.org/Proposal.html>.

Berners-Lee, T., Fielding, R., Irvine, U. C., and Frystyk, H. (1996). "RFC 1945: Hypertext Transfer Protocol—HTTP/1.0." [Online] 3 May 2000. <http://www.ietf.org/rfc/rfc1945.txt.>

Berners-Lee, T., Fielding, R., Irvine, U. C., and Mastiner L. (1998). "RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax." [Online] 2 May 2000. <http://www.ietf.org/rfc/rfc2396.txt>.

Brewer J., and Dardallier, D. (2000). "Web Accessibility Initiative (WAI)." [Online] 2 May 2000. <http://www.w3.org/WAI/>.

Fielding R., Irvine, U. C., Gettys, J., Mogul, J., Frystyck, H., Masinter, L., Leach, P. and Berners-Lee, T. (1999). "RFC 2616: Hypertext Transfer Protocol—HTTP/1.1." [Online] 3 May 2000. <http://www.ietf.org/rfc/rfc2616.txt>.

Kunze, J. (1995). "RFC 1736: Functional Recommendations for Internet Resource Locators. [Online] 2 May 2000," <http://www.ietf.org/rfc/rfc1736.txt>.

Raggett, D., Le Hors, A. and Jacobs I., eds. (1998). "HTML 4.0 Specification." [Online] 3 May 2000. <http://www.w3.org/TR/1998/REC-html40-19980424/>.

World Wide Web Consortium (W3C). [Online] <http://w3c.org>.