CONTENTS

RESEARCH PAPERS

106-114 A FORTRAN Input Program Generator
G. SMITH

115-120 Construction of Quadtrees and Octtrees from Raster Data: A New Algorithm Based on Run-Encoding
D.M. MARK

121-127 Integrating The Structured Analysis and Design Models: an Initial Algebra Approach
T.H. TSE

TUTORIAL PAPERS

128-135 Simple Compiler Correctness — A Tutorial on the Algebraic Approach
P.A. COLLIER

136-145 A Survey of Control Facilities in Logic Programming
T. VASAK

SPECIAL FEATURES

105 Editorial
146 Book Reviews
Introducing NonStop VLX.

Tandem technology sets the new standard for large applications in on-line transaction processing.

More transactions per second at a lower cost per transaction than any system in the world.

THE CIRCUITRY'S FAST.
We designed the system in our own laboratory, right down to our own unique VLSI chips. The result is more circuitry in less space. With fewer components than our next largest system, the VLX delivers twice the performance and three times the reliability.

THE DATA EXPRESSWAY.
In a conventional database, I/O requests must be handled sequentially. This creates queues that slow response time. In the VLX system, there are multiple paths to multiple disks. Data enters and leaves the database simultaneously. No time is wasted, and all disk space gets used.

PROCESSORS WITH LARGE APPETITES.
The VLX processors move transactions in 32-bit chunks. They reach into main memory in 64-bit chunks. Because this happens in parallel, more work gets done in less time at a lower cost per transaction.

THE SYSTEM KNOWS THE SYMPTOMS.
Expert systems software, using fault analysis, directs the problem diagnosis systematically. It also allows us to analyse it and shorten service time even more.

THE SERVICE IS EASY.
All critical components are field replaceable. When service is required, it's faster. You don't even have to stop an operation to add or replace components.

DIAGNOSTICS FROM A DISTANCE.
An integrated microprocessor allows us to monitor the system environment from anywhere in the world. We can even run stress tests remotely. If a failure does occur, the VLX has the capability to automatically dial out to remote centres anywhere in our worldwide network.

SECRETS ARE SAFE.
We offer software that will protect the security of your data whether it's in the VLX, in another Tandem system or in transmission.

NO GROWING PAINS.
To add power, just add processors. You can grow from a base four-processor system to 16. From there, you can expand in whatever increments you choose, all the way to 255 systems. You never buy more than you need, and you'll never have to rewrite a line of applications code.

NO-FAULT INSURANCE.
Tandem systems achieve fault-tolerance with a unique, parallel processing architecture. There are no idle back-up components. Instead, multiple components share the workload. If one goes down, the others pick up the slack, and application processing is uninterrupted.

HERE TODAY. HERE TOMORROW.
The VLX is compatible with any Tandem system and with all major communications standards — SNA, X.25, MAP and O.S.I. And by acting as a gateway to other vendors' systems, the VLX can link them and enhance their value as well.

SECRETS ARE SAFE.
Editorial

We live in interesting times. People in the computing field expect new developments, progress, change, almost as a right. Progress in computing is often equated to progress in computer hardware, which is not entirely the same thing. There is a constant stream of new and improved computer product announcements from dozens of companies, constantly competing for our attention. Much of this activity does represent real progress, thanks to the miracle of microelectronic technology: the secret is that, in this technology, smaller almost always means better in every way: more efficient, more reliable, faster and cheaper! Hardware designers have clear goals to aim for, by and large, and we measure the progress of our industry by their success.

Progress in software is often much harder to achieve, to define and to assess. Usually better means bigger: more code, more intricate data structures, and more sophisticated interfaces. But size brings complexity, and complexity means less reliability, not more, and often less efficiency. Documentation, understanding and maintenance all become more difficult and expensive. The fact is that our magical progress in hardware is not complemented by equally magical progress in software.

Software designers cope with complexity by developing, reviewing and refining many intricate tools and procedures. Ab initio many of these look the same, and there is little to choose among them. Evidence gathers only slowly, and ponderously, that one way, one data-structuring technique, one approach or one convention may be superior to another, and should therefore be preferred. It is important that this evidence be packaged and presented for assessment and review—that is one of the main functions of Journals such as this. So, if one looks for a common theme for the five papers in this issue, they are all attempts, different in kind, to cope with ever-present complexity: in the first three research papers, Smith describes improved procedures for handling data for a long established language, Mark is concerned with better ways to create data structures to store geographic data, and Tse explores better ways to structure and design data processing systems. The remaining two papers are tutorial in intent: Collier shows how older mathematical ideas can be newly adapted for the formal specification of important programs such as compilers, and Vasak shows how the new generation of logic programming languages such as Prolog are not without their deficiencies.

It is now common-place to talk of the desk-top publishing revolution. This journal has been practising rather than preaching: this is the fourth successive issue in which a majority of the papers have been typeset using a computer-driven phototypesetter. In fact in this issue, all the papers, have been prepared this way, using machine-readable versions of texts provided by the authors. The book reviews and this editorial have been treated similarly, and ordinary typesetting services have not been used at all. The advantages for preserving the intricate details of the authors' intentions, and for editorial control of the final result are very considerable. The continuing support of Ross Nealon and the University of Wollongong has been vital to this experiment, and has been much appreciated. But it has been an experiment—and a successful one, since we have proved it can be done! During the past twelve months, the auxiliary tools that I use for handling electronic manuscripts have been developed, modified, assessed and refined. Unfortunately they are still not good enough for routine use. One day soon I plan to write a paper on my experiences and conclusions. It may be a suitable finale for my time as editor.

Most of the development of the computing field has occurred within the span of one human generation. Almost unbelievably, most of the giants of the field—for example, Amdahl, Cray, Dijkstra, Hoare, Knuth, Perlis and Wilkes, to name but a few—are alive today. I have been privileged during my career to meet and converse with many of these. Of course there are a few that I definitely have missed, such as Turing and von Neumann, but no mechanical engineer living today has had the opportunity to meet with Archimedes, Brunel, Carnot or Stephenson. On the other side of the coin, it is true that most of the pygmies of our field are also alive today—I refer to those of small capacity who have entered the field, prospered a little, and then closed their minds for ever more. Unfortunately some of these are members of the Australian Computer Society, receive but do not read this journal themselves, and Luddite-like, would seek to prevent other readers from profiting from the exertions of others. I recently received a communication in this vein from a Mr Jones of Hampton, Victoria. I have not replied to Mr Jones directly, but would suggest that if ever in his professional career he has done anything of technical merit, that he should share his experiences with the other readers of this journal. One last thought is that the Institution of Mechanical Engineers does not need to debate today whether George Stephenson would be a suitable professional member: it is free to demand a proper professional training for all its new members. In another generation, computing societies will be free also to do likewise.
A FORTRAN Input Program Generator

G. Smith†

Many FORTRAN programs have poor user-interfaces because input data streams have complex structures which cannot adequately be cast into rigid, fixed field format. This paper describes the ideas underlying INPROG—a FORTRAN input program generator which facilitates the use of more natural free-form data structures. From a simple description of the input data structure, the program generator constructs a program that will: input the data; check that data conform to the required structure; report errors when appropriate; and invoke specified FORTRAN statements when prescribed elements are recognised in the input data stream. INPROG is designed to be easy for FORTRAN programmers to use. A useful set of data element types is presented, and comprehensive error handling procedures are described.

Keywords and Phrases: code generator, finite automata, FORTRAN, input program, regular expressions, state transition tables, user interface

CR Categories: D. 2. 2, D. 3. 4

1. INTRODUCTION

Users of FORTRAN programs which read complex data streams must often endure unbearably rigid restrictions on the layout of the input data stream. One major reason for this is that it is difficult to input a record using FORTRAN unless the type (for example, REAL or CHARACTER) of each data element in the record is known in advance of the record being input.

Consider, for example, a record that specifies the height, width and depth of a box. The simplest data stream layout is to have a record containing the three numbers in consistent, known units (say, millimetres) representing respectively the three required dimensions. For example:

\[
55.9 \quad 121.1 \quad 27.3
\]

However, it may be convenient to use units other than millimetres for some dimensions, as illustrated below:

\[
55.9 \quad 12.11 \text{ cm} \quad 27.3
\]

Ideally either set of data should be acceptable to and correctly interpreted by the program. Note, however, that the third data element is now alphabetic rather than numeric, and that it is not possible to know what type of datum is in this field until after the record has been read.

It is not unreasonable for the user to expect that the program can accept and process correctly all of the following variants of the data:

\[
\begin{align*}
55.9 & \quad 121.1 & \quad 27.3 \\
55.9 & \quad 12.11 \text{ cm} & \quad 27.3 \\
55.9 & \quad 12.11 \text{ cm} & \quad 27.3 \\
\text{Height}=5.59 \text{ cm} & \quad \text{Width}=0.1211 \text{ m} & \quad \text{Depth}=27.3 \\
\text{DEPTH} & \quad 27.3, \text{ WIDTH} & \quad 121.1, \text{ HEIGHT}=55.9 \\
\text{and even} & \\
\text{HEIGHT} & \quad 5.59 \text{ cm} & \\
\text{Depth} & \quad 27.3 & \\
\text{width} & \quad 121.1 & \\
\end{align*}
\]

For users to write programs by hand which provide this kind of flexibility for all of their input data is an expensive and error-prone process. It is far better to generate them by computer. Many users, in particular scientists and engineers, regard writing computer programs as a secondary activity—the main business is the science or engineering. The work required for the occasional programmer to learn how to use complex tools in order to perform a secondary activity is not perceived to be cost effective. This is particularly so if the occasional programmer uses this knowledge rarely and must therefore relearn it every time it is used. The obvious lesson is that software tools for occasional programmers must be easy to learn and relearn. Furthermore, such tools should not require more than a passing knowledge of any aspect of computing science for their effective use.

Copyright © 1986, Australian Computer Society Inc.

General permission to republish, but not for profit, all or part of this material is granted, provided that the ACJ's copyright notice is given and that reference is made to the publication, its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

†Dr Graham Smith, School of Mechanical and Industrial Engineering, The University of New South Wales, P.O. Box 1, Kensington, 2033, Australia. Manuscript received February, 1986; revised June, 1986.

The Australian Computer Journal, Vol. 18, No. 3, August 1986
This paper deals with INPROG, a simple system for generating FORTRAN (that is, conforming to ANSI, 1978) input programs, which was originally developed to generate the user interface for a simulation language but has been found useful in a wide variety of applications including a linear programming matrix generator, a statistical distribution input routine, and several command interpreters.

1.1. Published Methods for Enhancing FORTRAN Input

In this section, three classes of published methods of improving FORTRAN input are discussed:
— language extensions provided by compilers;
— subprograms for partitioning an input data record into individual data elements; and
— input subprogram generators.

Compiler-provided language extensions

Almost all FORTRAN compilers provide language extensions. These extensions are often unique to that compiler and their use therefore restricts portability. There is, however, one class of language extension which has been adopted so enthusiastically by FORTRAN programmers that it has been incorporated into many different compilers (unfortunately, in differing ways). This is the NAMELIST facility.

The NAMELIST facility requires data values to be identified by a keyword which is actually the variable name used in the program. Data describing the dimensions of a box might appear as follows:

\[
\begin{align*}
  &\$BOX \\
  &\text{HEIGHT}=55.9, \\
  &\text{WIDTH}=121.1, \\
  &\text{DEPTH}=27.3 \\
  &\$END
\end{align*}
\]

The values can be specified in any order, and not all values need occur. This allows default values for variables to be used unless overridden by NAMELIST input.

Lexical analysis

FORTRAN programs manipulate a restricted set of types of data, namely: LOGICAL, REAL, INTEGER, DOUBLE PRECISION and CHARACTER. The input data stream read by FORTRAN programs, on the other hand, is comprised of characters grouped into records. A lexical analyser assembles the characters in the input data stream into data elements of types which can be manipulated by the program.

The lexical analyser can be implemented as a subprogram which returns both the value of the next data element and its type, thus providing FORTRAN programs with the capability to process an input data stream which contains elements having types in a sequence that is not known in advance.

There have been many attempts to enhance FORTRAN by providing lexical analysis subprograms. These subprograms are generally claimed to provide free-format input. Surprisingly few of these have been reported in the literature.

Hitchcock (1974) describes a comprehensive free-form input package with subprograms that partition the input data into elements of type INTEGER, REAL or CHARACTER and recognise the special structure

\[
\text{KEYWORD = REAL}
\]

in the data. Logical functions identify the type of data element and recognise whether or not particular character strings have been encountered.

Temple (1980) provides a subroutine for free-form input. This subroutine inputs a line of data and partitions it into individual data elements using blanks as delimiters. For example, the subroutine processes the line

\[
\text{TEMPERATURE} 5 \text{ MILLI KELVIN}
\]

to produce two arrays D and T containing

\[
\begin{align*}
  &T(1) = 'TEMP' \\
  &T(2) = 'MILL' \\
  &T(3) = 'KELV' \\
  &D(1) = 5.0
\end{align*}
\]

User-written code must interpret the contents of these two arrays. Data which are neither numeric nor alphabetic are ignored, and no error handling is provided. A more extensive version of this subroutine forms part of the Harwell Subroutine Library (Hopper, 1980).

Input subprogram generators

Lexical analysers can partition the data stream into data elements but cannot check that data elements occur in an acceptable sequence. These checks are usually performed by user-written code which implicitly specifies acceptable sequences of data elements. However, this code can also be produced by a program generator if the acceptable sequences of data elements are specified explicitly.

One such subprogram generator is YACC (Johnson, 1975). YACC uses a Backus-Naur Form (BNF) description of the input data stream to generate a RATFOR (Kernighan, 1975) subprogram for reading the input data stream. The generated code invokes a user-provided lexical analyser. Lexical analysers can be generated by the complementary software tool LEX (Lesk, 1975).

The BNF description consists of a number of rules, each of which specifies a small portion of an acceptable input data stream. Actions, consisting of fragments of RATFOR code, can be specified for each rule. These actions are performed (that is, the code is executed) when input data conforming to this rule are recognised.
YACC is an extremely versatile software tool which is capable of generating excellent input subprograms, yet FORTRAN programmers seem to use it only rarely. This is probably because of the perceived complexity of YACC. Not only is there a lot to learn (some of it quite complicated) about using YACC and LEX, but there are many design decisions to be made when using them. Furthermore, although YACC can produce RATFOR subprograms, the FORTRAN programmer has to contend with the fact that it more naturally operates with C as the host language.

Kovacic (1983) describes a comparatively simple system which is directed primarily towards menu-driven input. The system recognises many types of data element in the input data stream, and the allowable syntax of the input data stream can be partly specified in terms of these types. The system has several useful features, including allowing abbreviated or even non-existent keywords. It seems well suited to interactive work where the statement names correspond to command names. However, for batch input, the system has some drawbacks. For example: the statement name must be specified at the beginning of each logical record even when it is not logically required; the user has no control over error handling; and, although the syntax of data in each statement is checked, the syntax of the statements themselves does not seem to be.

1.2. Objectives of INPROG

INPROG became necessary when a simulation language was transported to a different computer system. The user interface, which was still being developed, had been generated by YACC, but YACC was not available on the new computer.

It was important that INPROG be able to generate the input subprograms for the simulation language and also that portability problems of the type that gave rise to the need for INPROG should not arise again. It was evident that developing INPROG would not be a trivial programming task, and therefore an attempt was made to make it generally useful—as much for the future benefit of the author as well as for other potential users. This meant that INPROG had to be simple to learn and use. Ideally, it should have the capability of YACC and the simplicity of use of the system of Kovacic. More realistically, it would lie somewhere between the two.

The objectives used in the design of INPROG were:

— INPROG should be a portable FORTRAN program;
— Programs generated by INPROG should be portable FORTRAN programs;
— Straightforward error-handling procedures must be provided in generated programs;
— A suitable balance must be struck between simplicity of use and the complexity of data structures which can be handled;
— No specialist knowledge of the theory underlying the parsing of input data should be required in order to use INPROG effectively.

2. OVERVIEW OF INPROG AND ITS LEXICAL ANALYSER

INPROG generates FORTRAN subroutines for reading input data. In this document it is often necessary to distinguish between the input data read by the INPROG-generated subroutine, and the data which INPROG itself reads. Two terms are defined below to identify these two kinds of input.

Data description: refers to the input data for INPROG.

Input data stream: refers to the input data for the INPROG-generated subroutine.

The data description may contain up to four major sections in the following order:

MODULES DEFINITIONS ACTIONS

The two MODULES sections are optional and, if present, contain FORTRAN subprograms which are output unchanged by INPROG.

The DEFINITIONS section contains the rules which define valid forms of the input data stream and the points at which groups of FORTRAN statements are to be executed.

The ACTIONS section contains groups of user-written FORTRAN statements which are to be executed by the generated program at certain points in the input data stream.

INPROG-generated subprograms consist of: a lexical analyser; procedures to check that data elements conform to the DEFINITIONS section; and procedures to invoke the user-written code in the ACTIONS section.

The lexical analyser in programs generated by INPROG recognises six types of data elements in the input data stream. These data elements, which are described below, are predefined and cannot be changed by the user.

INTEGER A string of characters which contains no decimal point and is acceptable to FORTRAN programs as an integer constant.

REAL A string of characters which is not an INTEGER, and is acceptable to FORTRAN programs as a real or double precision constant.

CHAR Any single character other than a letter of the alphabet, a digit 0 to 9 or a special INPROG punctuation character (these are: a minus which is immediately followed by one of the digits 0 to 9; a comma; a blank; a carriage return;
A FORTRAN Input Program Generator

NAME

A string of contiguous characters made up of letters of the alphabet, digits 0 to nine and underscore characters, the first of which is a letter of the alphabet.

STRING

A string of characters enclosed by either single or double quotes. A single quote within a string delimited by single quotes is represented by two adjacent single quotes. A double quote within a string delimited by double quotes is represented by two adjacent double quotes.

EOF

A virtual element which matches the end of file.

These definitions were extremely important in the design of INPROG because they influence both the power and the ease of use of the system. These data elements are generally similar to FORTRAN constants or identifiers, and so will be already familiar to FORTRAN users. Furthermore they are rich in descriptive power.

A separator is a set of contiguous characters containing only: end-of-line indications, and (if they are not part of a STRING) commas, blanks and tabs. A side effect is that commas, tabs, blanks and single and double quote characters can only be recognised in the data stream if they are a STRING or part of a STRING.

Each time the lexical analyser is invoked it returns the type and value of the next data element in the input data stream. The values returned by the lexical analyser and the variable names by which they can be accessed in user-written code are listed below:

<table>
<thead>
<tr>
<th>TOK</th>
<th>STRING</th>
<th>RVAL</th>
<th>IVAL</th>
<th>LINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>WIDTH</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHAR</td>
<td>=</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>REAL</td>
<td>52.8</td>
<td>52.8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NAME</td>
<td>Depth</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>INTEGER</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>CHAR</td>
<td>#</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHAR</td>
<td>/</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>REAL</td>
<td>15.3E2</td>
<td>1530.0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>STRING</td>
<td>DEPTH</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CHAR</td>
<td>=</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INTEGER</td>
<td>5</td>
<td>5.0</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>NAME</td>
<td>cm</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>EOF</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

3. A SIMPLE EXAMPLE

The purpose of this example is to illustrate a data description for the dimensions of any number of boxes. An input data stream may contain

HEIGH 55.9, WIDT 121.1, DEPTH 27.3

A full data description for this input data stream is developed in following subsections.

3.1. The DEFINITIONS Section

A DEFINITIONS section, which describes the structure of the input data stream for boxes (with INPROG keywords italicised) is

getbox (boxes: ('HEIGHT' REAL 'WIDTH' REAL 'DEPTH' REAL) * EOF ;)

We examine first two elements

getbox: This is used by INPROG as the name of the subroutine it generates for reading the input data stream.

boxes: This identifies a definition of the input data stream and is called a definition name.

The braces {...} delimit the definitions for getbox, and the colon semi-colon sequence :...; delimits a single definition body. In this case the definition body consists of

('HEIGHT' REAL 'WIDTH' REAL 'DEPTH' REAL) * EOF

Note that the structure is

(...*) EOF

where * signifies that whatever is defined within the parentheses can occur any number of times. EOF means the end of the file. Thus the definition specifies dimensions for any number of boxes, the list being terminated by the end of file. For each box the dimensions are specified as real numbers, each of which is preceded by an appropriate keyword.

3.2. The ACTIONS Section

The user-written code in the ACTIONS section is assembled by INPROG into a subroutine with a default name (which can be changed) of action. The
A FORTRAN Input Program Generator

statements between \textit{HEADER} \{ \} and the matching \{ \} are positioned at the beginning of the subroutine. These are generally non-executable statements (such as declaration and COMMON statements). The individual groups of action-coding statements are included in the subroutine in a cascade of tests which selects the specific group of statements to be executed.

The action-coding interacts with program units in the MODULES sections through COMMON memory. Suppose the data are to be stored in the COMMON block

\begin{verbatim}
COMMON /BOXDEF/ HEIGHT(99), WIDTH(99),
    1 DEPTH(99), NBOX
\end{verbatim}

so that they are available for later processing, and that the variable \texttt{NBOX}, which initially has the value zero, holds the number of boxes for which dimensions have been input so far.

The lexical analyser provides the action coding with information from the input data stream by assigning values to the variables \texttt{IVAL}, \texttt{RVAL} and \texttt{STRING}. Data can be recorded in memory by executing statements such as

\begin{verbatim}
DEPTH(NBOX) = RVAL
\end{verbatim}

immediately after the value of a depth has been input. The code is contained within braces and preceded by an identifying action-coding name in the ACTIONS section. This same action coding name is positioned within square brackets in the definition to signify when the code is to be executed.

The reserved action code name \texttt{START} can be used to identify statements which are to be executed before any data have been read, and the reserved action code name \texttt{END} can be used to identify statements which are to be executed after all data have been read.

\section*{3.3. A Data Description for Box Dimensions}

In order to generate a subroutine for reading the input data stream, the user prepares a file containing the data description, and submits this file as data to INPROG. An example of a data description for a program which inputs the dimensions of boxes is given in Figure 1. The following points should be noted:

1. The main program is in the MODULES section.
2. The data are simply input and printed.
3. In the main program, the statement \texttt{CALL GETBOX(5)} invokes the INPROG-generated input-subroutine. The argument is the logical unit number from which the input data stream is to be read. If necessary, this file should be opened before the generated input-subroutine is invoked.
4. In the DEFINITIONS section, points at which groups of FORTRAN statements are to be executed are identified by a name in square brackets

\begin{verbatim}
Figure 1. Version 1 of INPROG Data For Box Dimensions
\end{verbatim}

\begin{verbatim}
[...]. This same name identifies the code in the ACTIONS section. Thus the statements
\begin{verbatim}
NBOX = NBOX + 1
HEIGHT(NBOX) = RVAL
\end{verbatim}
\end{verbatim}

are executed immediately after a value for \texttt{HEIGHT} appears in the input data stream.

\section*{4. THE INPROG DEFINITIONS SECTION}

We have already seen an example of the DEFINITIONS section of the data description. However, INPROG provides many facilities which have not been used in this example. In this section a brief description of some of these facilities is given. Further information can be found in Smith, 1985.

\subsection*{4.1. Operators in the DEFINITIONS Section}

INPROG provides five types of operator, namely \begin{verbatim} ( . . . ) parentheses | | alternative ? | option * | repetition (but need not occur) + | repetition (must occur at least once) \end{verbatim}

These operators are commonly (but not universally) used in regular expressions with the same meanings. Use of some of these is illustrated by the following fragment of a definition which defines an optional unit of length. (Action-coding names in square
brackets are omitted for clarity.)

('<m' | 'cm' | 'mm')?
The ? signifies that the data element matching the contents of the parentheses is optional. If the data element does occur it can be any one of the three specified units. In the following definition fragment an identifying keyword is optional.

'HEIGHT'? REAL ('m' | 'cm' | 'mm')?

Some examples of data matching this definition fragment are

55.9
5.59cm
HEIGHT 55.9 mm

4.2. Subdividing the Definitions

Although the example given in Figure 1 defines the syntax in one definition, this definition could have been split up into several. A definition may contain, as an operand, another definition name. For example, the DEFINITIONS section for the box dimensions could have been written as:

gobox { boxes: (box)* EOF ;
box: height width depth ;
height: 'HEIGHT' REAL [h] ;
width: 'WIDTH' REAL [w] ;
depth: 'DEPTH' REAL [d] ;
}

Although this may seem to allow recursive definitions, these were purposely prohibited, but not without a great deal of thought since one of the strengths of comprehensive program generators, such as YACC, is that they allow this type of construction. Nevertheless, the use of recursive definitions leads to ambiguous grammars and/or complex parsers, and requires some knowledge of the theory of parsing, of precedence and association. However, it is fortunate that, whereas those who write compilers cannot do without recursive definitions, most scientists and engineers can. INPROG has been used in a variety of applications, and the only problem posed by the lack of recursive definitions has been in the DEFINITIONS section from which INPROG generates itself.

Even though recursive definitions are not allowed, there are still advantages to be gained by providing for subdivision of definitions, such as the following.

— The user may prefer to isolate parts of definitions to keep each definition simple.
— Repetition can be avoided.
— Error-handling procedures may be specified as local to individual definitions. This point will be dealt with in more detail later in §5.

4.3. Operands in the DEFINITIONS Section

The INPROG DEFINITIONS operands are

— Key words representing the input data stream data element types, namely INTEGER, REAL, CHAR, NAME, STRING, EOF.
— Definition names.
— Literal characters.
— Literal names.

4.4. Literal Characters

Any single character in the data description which satisfies the definition of a CHAR and which has a single quote character immediately before and after it, is called a literal character. A literal character (such as ‘;’) commonly specifies punctuation in the input data stream.

4.5. Literal Names

Just as there are literal characters, there can be literal names, which satisfy the specification of a NAME but are enclosed in either single or double quote characters. They often define keywords. For example, the data description

('<WIDTH' | "DEPTH")

allows either of the names WIDTH or DEPTH at the corresponding point in the input data stream.

There is a special significance if a literal name is enclosed in double quotes in the data description. This denotes that the enclosed name can occur in any mixture of upper and lower case characters. The literal name

"DEPTH"

in the data description allows, for example,

Depth DEPTH depth etc.

Any literal name can contain a single hatch character ‘#’ which denotes that abbreviations at, or after, this point are permitted. For example,

"DEPT"

in the data description allows

D dep Dep DEPT etc.

in the input data stream.

4.6. Conflicts in the Data Description

INPROG executes, at most, one group of FORTRAN statements at any specific point in the input data stream. It is easy to write a definition such as

(REAL [COST] | REAL [PROFIT])

which contains a conflict between the code identified by COST and the code identified by PROFIT. INPROG reports on any such conflicts in the data description, and the user must eliminate them. There is potential for conflicts that are more subtle than that shown above and it was originally thought that users might have trouble both identifying the cause of such conflicts and redesigning the input data stream to remove them. However, this has not proved to be the case.
5. ERROR HANDLING

Error handling in input programs (for example, see Smith, 1969) poses many problems of which the two most prominent are:

1. realignment of the input data stream at a matching point of the data description after a syntax error is encountered; and
2. inconsistencies between the data that have been accepted, and statements that have been executed.

Because these problems are hard to solve in any general way, error handling in generated input programs is usually left to the programmer. Nonetheless, it was felt that INPROG should have some default error-handling procedure and that this should be as useful as possible.

The concept on which INPROG’s default error handling is based is that the sequences of data elements that are acceptable are likely to be the same at the beginning of consecutive lines of the input data stream. Whether or not this is true depends on how the user submits data to the program.

This concept leads to the following technique for aligning the input data stream and the data description. The internal state of the input subprogram is recorded at the beginning of each new line of the input data stream. When a syntax error is encountered, the input data stream is aligned at the beginning of the next line, and the input subprogram is reset to the state at the beginning of the line containing the error.

Consider, for example, the data description in Figure 2. If the program generated from this data description were to input the following data

```
1 m 2m 3m
100 cm 200 cm 300 cm
1000 2000 3000
```

then an error would be detected on line 2. The error would be reported, but the second line would have been partly recorded in memory.

It would be nice if the values of all variables could also be restored to their values at the beginning of the line containing the error. However this was too difficult to implement in any general way. Of course, the option still exists for the programmer to implement this restoration as is illustrated by the data description displayed in Figure 3. In the program generated from this data description, none of the data comprising line two of the input data given earlier would be recorded in memory. This is achieved by requiring that the dimensions of each box be terminated by a semicolon, and by maintaining the logical variable DONE which is .FALSE. at the end of an input line if an error was detected. If this is the case, then the next set of dimensions overwrite the partly read set from the previous line. Note that it is necessary for the user to submit the data for one box on each line in order for the error procedures to work nicely. It is shown below that this restriction is unnecessary when the user takes control of error handling.

The user may wish to execute different groups of error-handling statements depending on where the error occurs in the input data stream. One way of providing for this is for the user to set flags during the processing of acceptable data, which direct selection of error-handling code when required. This solution was rejected in favour of specifying the groups of error-handling statements in the DEFINITIONS section of the data description.

Each definition name can be followed by a name enclosed between ‘<’ and ‘>’. If no errors occur then this does not affect the processing in any way. However, if an error occurs when the generated program is reading data specified by that description, then the default procedures are not followed and error action-coding, identified by the name enclosed between ‘<’ and ‘>’ immediately following the definition name, is invoked. It is up to the user to

---

```
MODULES {
  PROGRAM BOX
  COMMON /BOXDEF/ HEIGHT(99), WIDTH(99), 1 DEPTH(99), NBOX = 0
  CALL GETBOX(S)
  PRINT *, ((HEIGHT(I), WIDTH(I), DEPTH(I))), 1 =1,NBOX)
  STOP
END

getbox (boxes: (box)* EOF ;
  box: height width depth ;
  height: ("HEIGHT" '='?? REAL[hl]
    ("m"[hm] | "cm"[hc] | "mm"[hm])? ;
  width: ("WIDTH" '='?? REAL[lw]
    ("m"[wm] | "cm"[we1] | "mm"[we1])? ;
  depth: ("DEPTH" '='?? REAL[ld]
    ("m"[dm] | "cm"[dc] | "mm"[dc])? ;

ACTIONS {
  HEADER {
    COMMON /BOXDEF/ HEIGHT(99), WIDTH(99), 1 DEPTH(99), NBOX
  }
  h {
    NBOX = NBOX + 1
    HEIGHT(NBOX) = RVAL
  }
  hm ( HEIGHT(NBOX) = HEIGHT(NBOX)*1000.0)
  hc ( HEIGHT(NBOX) = HEIGHT(NBOX)*10.0)
  w ( WIDTH(NBOX) = RVAL)
  wm ( WIDTH(NBOX) = WIDTH(NBOX)*1000.0)
  wc ( WIDTH(NBOX) = WIDTH(NBOX)*10.0)
  d ( DEPTH(NBOX) = RVAL)
  dm ( DEPTH(NBOX) = DEPTH(NBOX)*1000.0)
  dc ( DEPTH(NBOX) = DEPTH(NBOX)*10.0)
}
```

---

Figure 2. Version 2 of INPROG Data For Box Dimensions

The Australian Computer Journal, Vol. 18, No. 3, August 1986
moduleBox

common /boxdef/ height(99), width(99),
1 depth(99), nbox

nbox = 0

call getbox(5)

print *, ((height(i), width(i), depth(i)),
1 i=1,nbox)

stop
end

getbox { boxes: (box)*
EOF
;
box: height width depth 1 ; ' [finish] ;
height: ("height" =)? REAL h
(m"| cm" | "mm")?
;
width: ("width" =)? REAL w
(m"| cm" | "mm")?
;
depth: ("depth" =)? REAL d
(m"| cm" | "mm")?
}

Actions

header

common /boxdef/ height(99), width(99),
1 depth(99), nbox

logical done

start

if(done) nbox = nbox + 1

height(nbox) = rval

done = .false.

hm { height(nbox) = height(nbox)*1000.0)

hc { height(nbox) = height(nbox)*10.0)

w { width(nbox) = rval

wm { width(nbox) = width(nbox)*1000.0)

wc { width(nbox) = width(nbox)*10.0)

d { depth(nbox) = rval

dm { depth(nbox) = depth(nbox)*1000.0)

dc { depth(nbox) = depth(nbox)*10.0)

finish { done = .true.

end { if(not. done) nbox = nbox -1)

Figure 3. Version 3 of INPROG Data For Box Dimensions

issue whatever reports are desired and to align the
input data stream after the end of data specified by
that description. An error action-coding name
applies not only to the description in which it is speci-
\nified, but also to all nested descriptions appearing
within that description unless they, in turn, have their
own error action-coding name.

The data description in Figure 4 includes user-
written error-handling code identified by the action-
coding name realign. In the error action-coding
an error report is generated, the input data stream is
aligned just past the next semi-colon, and the flushed
data displayed in a WARNING message. In this code
the subroutine ALIGN aligns the input data stream,
the subroutine ERMSG displays error and warning
messages, and the function ITOA transforms an
integer to a character representation. These sub-
programs are part of an extensive library of
subprograms forming part of the INPROG system.

6. IMPLEMENTATION

Programs generated by INPROG are driven by two
state transition tables. (For a tutorial discussion on
these, see Hext and Hirst, 1982.) One state transition
table drives a lexical analyser which partitions the
data stream into data elements. At a higher level, the
second state transition table performs the syntactical
analysis. This second state transition table is con-
structed by INPROG from the user's description of

Figure 4. Version 4 of INPROG Data For Box Dimensions
A FORTRAN Input Program Generator

the input data stream, and thus reflects the structure of the input data stream.

Using techniques similar to those described in Aho and Ullman (1977), INPROG constructs a deterministic finite automaton (DFA) from the DEFINITIONS section of the data description. The second state transition table is prepared by modifying this DFA so that, for example, INTEGERs are acceptable when REALs are specified.

Both INPROG and the generated subprograms conform to ANSI(1978) except that lower case letters are used in FORTRAN statements. This has not yet caused any portability problems. The objective of portability has not been completely achieved. When transporting INPROG to a new computer system it may be necessary to change the value of up to thirteen constants in one BLOCK DATA subprogram used by both INPROG and the generated subprograms. These constants specify logical unit numbers and integer values of certain characters.

7. CONCLUDING REMARKS
The foregoing is a description of what has proved to be a useful tool with which occasional programmers can generate FORTRAN input programs. INPROG-generated input subprograms provide users with the kind of freedom for data-entry which is rarely available in FORTRAN programs. The use of INPROG encourages a decoupling of input from other functions of the program which allows extensive changes to be made to the user-interface without changing handwritten code.

It is quite in order for an INPROG-generated subprogram to invoke another INPROG-generated subprogram. For example, a command interpreter generated by INPROG may interpret a command which inputs a file using another INPROG-generated subprogram.

In order that INPROG be easy to use, some rarely needed facilities have been purposely omitted. In practice, occasional programmers seem to do very nicely without these facilities, and the payoff is a very simple system.

Some of the subprograms comprising INPROG have proved to be useful in their own right. These include the lexical analyser and a subprogram for converting non-deterministic finite automata to equivalent DFAs.

References

Biographical Note
Graham Smith is a Senior Lecturer in Operations Research at the University of New South Wales. His research interests include discrete event simulation languages and tools for their development.

Before joining the University in 1969, he worked with LEO Computers Limited and with Tubemakers of Australia Limited on the design and implementation of large scale information processing systems.
This paper presents a new algorithm for constructing linear quadtrees from raster data. The quadtree representation method termed two-dimensional run-encoding is used to define a raster dummy image for each individual raster. The raster dummy image has the actual image values (colours) for pixels on the raster, but assigns the value NULL to all pixels outside the current raster. Individual raster dummy images are then combined using a straightforward Boolean overlay procedure, with all values having priority over NULL. The complexity of this algorithm is linear in both the number of image pixels and in the number of quadtree leaves. The linear behaviour is achieved by preserving the spatial order inherent in the raster image during the execution of the algorithm. The extension of the algorithm to octtrees for three-dimensional data is straightforward.

**Keywords and Phrases:** quadtree, linear quadtree, octtree, raster data, image data, geographic information systems.

**CR Categories:** E.1, I.4

1. **INTRODUCTION**

Quadtrees have been shown to be an effective way to represent and process two-dimensional data (see Samet, 1984, for a review). Because of their wide applicability in image analysis and geographic information processing, the construction of quadtrees from other types of spatial data is an important topic. This paper describes and analyzes a new algorithm for the construction of quadtrees from raster data. This algorithm is linear in either the number of image pixels or the number of quadtree leaves, provided that the other of these is held constant. The linear behaviour is achieved because the quadtree leaves need never be sorted; sorting is avoided by preserving the spatial order inherent in the raster image during the execution of the algorithm. This algorithm can readily be extended to three dimensions for the construction of octtrees.

2. **QUADTREES**

Briefly, a quadtree is based on the recursive partition of a square region into quadrants and subquadrants, until all subquadrants are uniform with respect to image value (colour) or until some pre-determined lower level of resolution is reached. If the region consists of $2^n$ by $2^n$ pixels (cells), then the entire region is of level $n$, and individual pixels are of level zero. A quadrant of level $L$ ($0 < L < n$) is $2^L$ by $2^L$ pixels, and contains a total of $4^L$ pixels. Although quadtrees are known to be a space-efficient way to represent 'blocky' or coherent images (Dyer, 1982; Mark and Lauzon, 1985), their advantages over vector representations for many spatial algorithms may be even greater in terms of time-efficiency.

Early work on quadtrees represented relations among quadrants and subquadrants as a tree of out-degree four. In such a structure, relations between parents and children are implemented as explicit pointers in the system. All nodes in the tree except the root have exactly one parent, whereas all except the terminal ('leaf') nodes have exactly four children. Pointer-based representations are very fast if the entire quadtree can be held in core. For larger quadtrees, however, recent work has suggested that linear quadtrees may be more appropriate (Gargantini, 1982; Abel, 1984; Mark and Lauzon, 1984; Samet, Rosenfeld, Shaffer, and Webber, 1984; Lauzon, Mark, Kikuchi, and Guevara, 1985; Samet and Tamminen, 1985). In a linear quadtree, each leaf node is represented by a linear key number, which is based on an ordered list of the node's ancestors. Tree traversals are then accomplished using bit-level manipulations or modular arithmetic on the keys. Samet and
Tamminen (1985) outline a number of quadtree algorithms which exploit the properties of linear quadtrees. Linear key systems also provide an effective link between spatial data structures and algorithms for such computational geometry problems as rectangle retrieval (Abel and Smith, 1984a) and nearest-neighbour determination (Abel and Smith, 1984b).

Recently, Lauzon and Mark (Lauzon, 1983; Mark and Lauzon, 1984; Lauzon et al., 1985) have shown that certain forms of linear quadtrees can be run-encoded, that is, if there are several key-consecutive leaves of the same value, only the key of the last such leaf need be stored. For some algorithms, the runs are later decoded into their constituent leaves; other operations can be performed directly on these run-encoded structures without decoding. The Morton sequence numbers (Morton, 1966) form suitable keys for linear quadtrees; the Morton number of a pixel is obtained by interleaving the bits of the row and column coordinates of the pixel (coordinates are numbered from zero; see Lauzon et al., 1985). Lauzon and Mark represent a quadtree leaf by the largest Morton number enclosed within it. Then, the run-encoded linear quadtree file is exactly equivalent to a run-encoding of the individual pixels of the image in Morton sequence. Because the Morton sequence orders the pixels in two-dimensions simultaneously, Lauzon (1983; Lauzon et al.) has termed this approach two-dimensional run-encoding. Two-dimensional run-encoding (2DRE) is a space-efficient way to store linear quadtrees (Mark and Lauzon, 1985), but more importantly, is particularly suited to the superposition (overlay) operations which are central to geographic information systems (Lauzon et al., 1985; Mark and Lauzon, 1984; see also Samet, 1980b, p. 203).

3. RASTER TO QUADTREE CONVERSION

Although direct collection of quadtree data from analogue sources is possible, spatial data are rarely if ever collected in this way. Thus, for any quadtree-based image analysis or Geographic Information System (GIS), the construction of quadtrees from other types of spatial data is a very important topic. In geographic information systems applications, data are often digitized in vector format; algorithms for conversion of vector and boundary code data to quadtrees have been presented by Samet (1980a) and by Mark and Abel (1984; 1985). In most image-processing and some GIS applications, data are collected and distributed in raster form, that is, row-by-row or column-by-column. If the entire image can be held in core or in a random-access image buffer, conversion to quadtrees is straightforward (Samet, 1980b; Lauzon et al., 1985). Otherwise, the quadtree must be constructed as image data are examined row-by-row.

An algorithm to accomplish this task for pointer-based quadtrees has been presented by Samet (1981), and an algorithm for linear quadtrees has been developed by Abel (1986). Both of these algorithms are based on the concept of holding a list of candidate quadtree nodes. For example, if the first eleven pixels in the first raster are white, with the twelfth being black, then the first eight pixels might be part of a level three (eight by eight) white leaf. However, if any of the first eight pixels of any of the next seven rasters is black, then the level three white leaf does not exist. If each raster is 2^n long, then there can be at most 2^n^2 candidate leaves (level zero leaves are not considered to be 'candidates', since they are known to be leaves, and can be inserted into the quadtree).

As each new raster is examined, there are three possibilities for each candidate leaf: it can be confirmed as a complete leaf (and inserted in the quadtree); it can be found not to be a valid leaf (in which case its descendants are either inserted into the quadtree or added to the list of candidate leaves); or the candidate remains a candidate while the next raster is examined. The leaves are not generated in order, and may be inserted into the quadtree dynamically. However, if the linear quadtree is held as a list, then the list must be sorted eventually.

In this paper, a new raster to quadtree algorithm is described. This algorithm is based on two-dimensional run-encoding (2DRE).

4. THE ALGORITHM

The algorithm presented in this paper is extremely simple in concept. Basically, each raster is embedded in its own image (which will be referred to as the raster dummy image; see Figure 1A), which has the raster values along the raster itself, and a NULL value elsewhere. Each raster dummy image is represented as a 2DRE file. Then, the raster dummy images are simply superimposed (see Figure 1C), with any other value taking precedence over NULL.

One property of the Morton sequencing used in the 2DRE representation is that, if pixels are taken in pairs (left-key, right-key) counting from the beginning of the raster, left-key and right-key must differ by exactly one (see Figure 1A); furthermore, the next two keys Morton order must lie in a different raster (Figure 1A). Next, recall that 2DRE uses only the keys and values of pixels which represent the ends of runs of consecutive pixels all of the same value. Then, for each pair, a NULL run must end at (left-key - 1), while a run of some value must end at right-key. The pixel at left-key will be a run-end if and only if the values of left-key and right-key are different.

As an example, consider the raster dummy image for the fourth raster (Y=3) illustrated in Figure 1D. The pixels in the first pair (11, 12) have different colours, and thus are represented by three run-ends: (9, NULL), (10, W), and (11, B). The next pair (14, 15) is of one colour, and so only two run-ends are needed: (13, NULL) and (15, B).
Construction of Quadtrees and Octtrees

(A) Morton numbering scheme for a 2³ by 2³ image, with the origin in the upper
left corner.

(B) A binary image represented as a
2DRE linear quadtree (the keys of the
run-ends are given on the diagram; the
2DRE list appears to the right, with
W = white and B = black).

(C) The growing image of B after the
raster dummy images of the first three
rasters have been combined; NULL areas
of the image are shaded in grey; the
asterisk (*) denotes a coloured run-end
which will not be part of the final
image.

(D) The raster dummy image of the
fourth input raster (y = 3); the next step
in the algorithm will combine this with
the image in Figure 1C.

Figure 1.

The essential algorithms are described in three
procedures presented the appendix in a generic procedural
language. Procedure rastquad simply establishes the
appropriate quadtree structures and calls the other
two procedures. Procedure line_to_quad constructs
the raster dummy image of the current raster (line),
and procedure combine overlays this new raster
dummy image with the growing final 2DRE file.

5. AN EXAMPLE
An example may clarify the algorithm. Figure 1B
depicts an 8 × 8 binary image (n = 3); Morton keys
are shown in those cells which are run-ends, and the
records of the 2DRE representation are shown in
Table 1, column 1. Figure 1C (and Table 1, column
2) shows the result of combining the raster dummy
images for the first three rasters, and Figure 1D (and
Table 1, column 3) shows the raster dummy image for
the fourth raster. The next step in the execution of
the algorithm would combine the images in Figure 1C
and 1D through an overlay operation.

6. ALGORITHM ANALYSIS

6.1 Space Requirements
As noted above, each pair of pixels in the raster yields
either two (if they have the same value) or three (if
they differ) records in the raster dummy image.
Thus, since each raster contains 2ⁿ pixels, the
minimum number of records in a raster dummy image
is 2ⁿ, and the maximum number is 1.5×2ⁿ. When

<table>
<thead>
<tr>
<th>Key</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>W</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
</tr>
<tr>
<td>8</td>
<td>W</td>
</tr>
<tr>
<td>9</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>NULL</td>
</tr>
<tr>
<td>13</td>
<td>B*</td>
</tr>
<tr>
<td>15</td>
<td>NULL</td>
</tr>
<tr>
<td>17</td>
<td>W</td>
</tr>
<tr>
<td>18</td>
<td>B</td>
</tr>
<tr>
<td>23</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>B</td>
</tr>
<tr>
<td>25</td>
<td>W</td>
</tr>
<tr>
<td>27</td>
<td>NULL</td>
</tr>
<tr>
<td>29</td>
<td>W</td>
</tr>
<tr>
<td>30</td>
<td>B</td>
</tr>
<tr>
<td>31</td>
<td>W</td>
</tr>
<tr>
<td>39</td>
<td>B</td>
</tr>
<tr>
<td>47</td>
<td>W</td>
</tr>
<tr>
<td>52</td>
<td>B</td>
</tr>
<tr>
<td>53</td>
<td>W</td>
</tr>
<tr>
<td>54</td>
<td>B</td>
</tr>
<tr>
<td>63</td>
<td>W</td>
</tr>
</tbody>
</table>

Black or white run-end which will not be part of the final 2DRE file
because they are parts of longer runs.
The conversion of a raster to a raster dummy image

6.2 Time complexity

The conversion of a raster to a raster dummy image requires processing time which is proportional to the quadtree depth \( n \) times number of runs in the raster dummy image (which is at most \( 1.5 \times 2^n \); see above). The quadtree depth must be included because conversion of row-and-column values to the Morton number requires time proportional to \( n \). Next, combining the current raster dummy image with the growing image file requires time proportional to the sum of the number of records in the two files in question (Lauzon et al., 1985). An upper bound for this step is thus:

\[
(1.5 \times 2^n) + (N_R + 2^n) = (N_R + 2 \times 5 \times 2^n)
\]

For each raster, the two steps require at most time proportional to:

\[
(1.5n \times 2^n) + (N_R + 2.5 \times 2^n) = (N_R + (1.5n + 2.5) \times 2^n)
\]

Since there are \( 2^n \) rasters, an upper bound for the time complexity is thus \( 2^n \times (N_R + (1.5n + 2.5) \times 2^n) \). Multiplying through by \( 2^n \) and substituting \( N_p = 2^{2n} \) then gives the upper bound as:

\[
2^n \times N_R + 3.75n \times N_p
\]

For a fixed image size (i.e., constant \( n \)), the algorithm is linear in the final number of image runs (and, indirectly, of quadtree leaves, \( N_i \); see Lauzon et al., 1985); likewise, for a fixed number of runs (or leaves) in the final image, the algorithm is linear in the number of pixels. Previous algorithms for raster to quadtree conversion have asymptotic complexity of order \( (N_i \log N_i) \). This \( n \log n \) behaviour arises because the quadtree leaves must at some time be sorted. The current algorithm exploits the fact that raster data are already spatially ordered; the quadtree ordering is derived from this raster ordering directly, without the need for sorting. Other raster-to-quadtree algorithms disturb this inherent ordering and must later sort the quadtree leaves.

7. EXTENSION TO OCTTREES

Yau and Srihari (1981) have discussed the general problem of constructing a \( 2^k \)-tree in \( k \) dimensions from the \( 2^{k-1} \)-trees which, in effect, represent slices through it. The extension of the current algorithm from quadtrees to octtrees is very straightforward. The concept of Morton numbers generalizes directly to more than two dimensions (see Mark and Cebrian, 1986). For three-dimensional referencing, bits from the three coordinates are interleaved in threes. As in two dimensions, these pixels can also be run-encoded on the attributes, and later decoded ('three-dimensional run-encoding', or 3DRE; see Mark and Cebrian, 1986). If the direction along each incoming raster is considered to be the \( x \) direction (lowest-order bit in each triple), then the algorithm works without modification, except that procedure Morton has three arguments, and the algorithm must also keep track of the z-dimension as layers of rasters are included in the octtree.

8. SUMMARY

A new algorithm for constructing linear quadtrees from raster data has been presented. The algorithm uses a quadtree representation method termed two-dimensional run-encoding, and defines a raster dummy image for each individual raster. The raster dummy image sets all pixels outside the current raster to value NULL. The individual raster dummy images are then combined using a straightforward Boolean overlay procedure, with all other values having priority over NULL. In contrast with other algorithms for raster to quadtree conversion, the complexity of the current algorithm is linear in either the number of image pixels or the number of quadtree leaves, provided that the other of these is held constant. The linear behaviour is achieved because the quadtree leaves need never be sorted; sorting is avoided by preserving the spatial order inherent in the raster image during the execution of the algorithm. The extension of the algorithm to octtrees for three-dimensional data is straightforward, and has been presented.

ACKNOWLEDGEMENTS

This paper is part of a project funded by (United States) National Science Foundation Grant SES-8420789 to D.M. Mark; this support is gratefully acknowledged. David J. Abel sent the writer a copy of his manuscript on raster-to-quadtree conversion (cited below); Dr. Abel also read an earlier draft of this paper and provided detailed comments, especially on the analysis section; comments of the reviewers were very useful. I also wish to thank Juan A. Cebrian and J.P. Lauzon for their comments on the
Construction of Quadtrees and Octtrees

References

Biographical Note
David M. Mark received the degrees of B.A. (1970) and Ph.D. (1977) from Simon Fraser University, and M.A. (1974) from the University of British Columbia; these degrees were all in Geography. Since 1981 he has been a member of the Department of Geography, State University of New York at Buffalo, where he is currently Associate Professor and Director of the Cartographic Laboratory. In July and August, 1983, he was a Visiting Scientist with the CSIRO Division of Computing Research in both Canberra and Townsville. His current research interests include quadtrees, digital elevation models, geographic information systems, artificial intelligence, and theoretical geomorphology.
Construction of Quadtrees and Octtrees

procedure rastquad()
/* procedure to convert raster data to a quadtree, based on run­encoding; it assumes the existence of procedures: 'create_qtree' which creates a quadtree structure (opens a file, or sets up a B-tree, etc.); 'get_raster' which obtains the next raster; and 'transfer' which transfers a quadtree from one representation to another; */
begin
integer n_rasters, line, rast_length, raster[256];
quadtree_structure quadtree__1, quadtree__2, quadtree__3;
max_key := 65535; /* for 256 by 256 image */
rast_length := 256;
n_rasters := 256;
create_qtree (quadtree__1); /* initialize three quadtree */
create_qtree (quadtree__2);
create_qtree (quadtree__3);
for line := 0 to (n_rasters - 1) do
begin
get_raster (raster);
line_to_quad (raster, rast_length, line, quadtree__2);
/* invoke procedure line_to_quad (listed below) to convert the current raster to a 2DRE file (quadtree_2) with value NULL for all pixels outside the raster */
combine (quadtree_1, quadtree__2, quadtree__3);
/* Boolean overlay of quadtree_1 and quadtree__2, to produce quadtree__3, with the higher value taking priority, and all values having priority over NULL; the combine procedure is listed below */
transfer (quadtree__3, quadtree__1);
/* transfer the growing 2DRE representation from quadtree__3 to quadtree_1 */
end
end

procedure line_to_quad (raster, rast_length, line, quadtree)
/* this procedure creates a 2DRE linear quadtree structure of an image with the current raster (line) filled and the remainder of the image NULL; line_to_quad assumes the existence of procedure output, which inserts a Morton key and the associated value into structure 'quadtree' */
integer raster[], rast_length, line;
quadtree_structure quadtree;
begin
integer column, NULL;
unsigned integer key, morton();
NULL := -1; /* NULL must be a number less than any value which actually occurs in the image */
for column := 0 to (rast_length - 2) by 2 do
begin
key := morton (column, line);
if key = 0 then /* no pixels before the one with key = 0 */
output (quadtree, key, NULL);
if raster [column] != raster [column + 1] then
output (quadtree, key, raster[column]);
output (quadtree, key + 1, raster[column + 1]);
end

procedure combine (quadtree_1, quadtree__2, quadtree__3)
/* this is a modified version of procedure overlay (see Lauzon et al., 1985; Mark and Lauzon, 1984) to overlay two 2DRE structures. It overlays quadtree_1 and quadtree__2, with a rule that the higher value takes priority, and produces quadtree__3. This procedure assumes the existence of a procedure 'get_key' which gets the next key and associated value from a quadtree structure. It also returns a value of 'true' (T) through the function name if the end of the quadtree has been reached. */
quadtree_structure quadtree_1, quadtree__2, quadtree__3;
begin
integer value, old_value, value_1, value_2;
unsigned integer old_key, key_1, key_2;
boolean end_1, end_2;
old_key := 0;
value_1 := 0;
value_2 := 0;
end_1 := get_key (quadtree_1, key_1, value_1);
end_2 := get_key (quadtree__2, key_2, value_2);
if value_1 > value_2 then
old_value := value_1;
else
old_value := value_2;
while end_1 != 'T' and end_2 != 'T' do
begin
/* define value as the larger of value_1 and value_2; recall that NULL was set to a value lower than any valid image value */
if value_1 > value_2 then
value := value_1;
else
value := value_2;
/* if a value run has ended, output the value and key at which it ended */
if value = old_value then
begin
output (quadtree_3, old_key, old_value);
old_value := value;
end
/* whether or not anything was sent to output, get a new key and value from whichever structure contained the lower key (one from each if keys equal) */
if key_1 < key_2 then
begin
old_key := key_1;
end_1 := get_key (quadtree_1, key_1, value_1);
end
else if key_1 > key_2 then
begin
old_key := key_2;
end_2 := get_key (quadtree__2, key_2, value_2);
end
else if key_1 = key_2 then
begin
old_key := key_1;
end_1 := get_key (quadtree_1, key_1, value_1);
end_2 := get_key (quadtree__2, key_2, value_2);
end
/* end of while loop */
end
Integrating the Structured Analysis and Design Models: an Initial Algebra Approach

T.H. Tse†

Numerous models have been proposed under the name of structured systems analysis and design. Because of the lack of a common theoretical framework, the transition from one model to another is arbitrary and can only be done manually. An initial algebra approach is proposed to integrate the structured models. The algebra defined can be mapped through homomorphisms to Yourdon structure charts and DeMarco data flow diagrams. It can also be linked to Jackson structure text through equations.

Keywords and Phrases: algebraic approach, structured analysis, structured design

CR Categories: D. 2. 1, F. 3. 2, K. 6. 3

1. INTRODUCTION
Numerous models have been proposed under the name of structured analysis and design. Examples are data flow diagrams (DeMarco, 1978; Gane and Sarson, 1979; Weinberg, 1980), Jackson structure diagrams, Jackson structure text (Jackson, 1975), system specification diagrams, system implementation diagrams (Jackson, 1983), Warnier/Orr diagrams (Orr, 1977) and structure charts (Yourdon and Constantine, 1979). They are widely accepted by practicing systems analysts and designers through the simplicity of use and the ease of communication with users. But because of the lack of a common theoretical framework, the transition from one model to another is arbitrary and can only be done manually. Users tend to stick to a particular model not because of its superiority but because of familiarity. Automatic development aids tend to be ad hoc and model-dependent.

To solve the problem, there is a need to provide a formal link for the structured models. The initial algebra approach is proposed. It has a rich mathematical linkage with category and algebraic theories. But at the same time, the concepts can be simply stated for those who do not want to be involved with elaborate theories.

In this paper we will define the algebra and illustrate how it can be related to Yourdon structure charts, DeMarco data flow diagrams and Jackson structure text by means of homomorphisms and equations. The three models have been chosen for illustration because they represent three distinct classes of structured models.

We shall concentrate on the conceptual framework rather than on formal proofs. Only a knowledge of elementary set theory will be assumed. Readers who are interested in a deeper understanding of algebraic formalism may refer to Cohn (1981) for a general algebraic introduction, to Burstall and Goguen (1982), Goguen et al. (1978), Wagner (1981) and Zilles et al. (1982) for a computer science oriented treatment, and to Goguen et al. (1975) and Wagner et al. (1977) for a category-theoretic treatment.

2. ADVANTAGES OF INTEGRATION
An integration of the structured models is useful for several reasons:

a. Specifications can be transformed from one form to another through homomorphisms and equations, as illustrated in the subsequent sections of the paper.

b. Different structured models are suitable for different situations depending on the environment (Shigo et al., 1980), emphasis (Colter, 1982) and stage of development (Lauber, 1982). But it has been found that individual models may not be used in some installations because the users are not familiar with them (Beck and Perkins, 1983). Through a transformation system, the most suitable model can be used independently of user
Structured Analysis and Design Models

task (process-sales; get (); put ());
seuq (task (get-valid-order; get (); put (outdata valid-order));
   sequ (task (get-order; get (source customer)); put (outdata order); elem);
   task (validate-order; get (indata order); put (outdata valid-order); elem));
task (process-order; get (indata valid-order); put (outdata invoice-info);
   seq (task (prepare-local-invoice;
             get (indata lc-order); put (outdata invoice-info)); elem);
   task (compute-tax;
             get (indata pre-tax-info); put (outdata invoice-info); elem));
task (prepare-overseas-invoice;
   get (indata os-order); put (outdata invoice-info); elem));
task (put-invoice; get (indata invoice-info); put (outfile invoice); elem))

Figure 1. Sample Term in Initial Algebra

familiarity.

c. Automatic development aids for one structured methodology can be applied to another through transformations. The development aids described, for example, in Delisle et al. (1982), DeMarco and Soceneantu (1984) and Tse (1985) may therefore be extended to other models.
d. In recent years the initial algebra approach has been used extensively in the specification of abstract data types. Examples are Clear (Burstall and Goguen, 1980, 1981; Sannella, 1984) and OBJ (Futatsugi et al., 1985; Goguen 1984; Goguen and Tardo, 1979; Goguen et al., 1983). Interpreters for abstract data types are already available. Although such interpreters are not originally intended for structured analysis and design models, they can nevertheless be adapted to suit our needs, e.g. for validating our specifications.
e. In informal specifications such as DeMarco data flow diagrams, a certain degree of omission or ‘mutual understanding’ is permitted. This often leads, however, to ambiguity and misunderstanding. If a formal specification is used, we can enforce predefined standards more easily.

3. ALGEBRAS

Intuitively, an algebra is a family of objects that satisfy a formal structure. To define an algebra \( A \), we must first of all define the formal structure through the concept of a signature. A signature consists of a set \( S \) of object types, known as sorts, together with a family \( E \) of sets, each set containing operation symbols (or simply symbols) that connect the sorts. We will use \( \Sigma \langle S_1, \ldots, S_n; S > \) to denote the set of operation symbols that connect the sorts \( S_1, \ldots, S_n \) to the sort \( S \).

Given the skeleton structure, we then complete the definition by relating it to real objects. Each sort \( s \) is mapped to a set \( A < s > \), which is called the carrier of \( s \). Each symbol \( q \) in \( \Sigma \langle S_1, \ldots, S_n; S > \) is mapped to a function

\[
q_A : A < s_1 > \times \ldots \times A < s_n > \rightarrow A < s >
\]

which is called an operation.

Let us apply the algebraic fundamentals to structured systems. Conceptually, a structured system is specified by a hierarchy of tasks. Each task consists of a name, a structure, together with the input and output. The structure determines whether the task is elementary, or is made up of subtasks in the form of sequence, selection, iteration or parallelism. The input and output are in the form of data flows related with other tasks, files and the environment.

The signature for structured systems, then, consists of a set \( S \) of seven sorts: task, name, struct, input, output, dataflow and flowname, and a family \( \Sigma \) of sets of operation symbols:

\[
\Sigma \langle \text{name}, \text{input}, \text{output}, \text{struct}, \text{task} \rangle = \{ \text{task} \}
\]

\[
\Sigma \langle \text{task}^n, \text{struct} \rangle = \{ \text{sequ}, \text{seln}, \text{para} \}
\]

\[
\Sigma \langle \text{struct}, \text{struct} \rangle = \{ \text{iter} \}
\]

\[
\Sigma \langle \Lambda, \text{struct} \rangle = \{ \text{elem} \}
\]

\[
\Sigma \langle \text{dataflow}^n, \text{input} \rangle = \{ \text{get} \}
\]

\[
\Sigma \langle \text{dataflow}^n, \text{output} \rangle = \{ \text{put} \}
\]

\[
\Sigma \langle \text{flowname}, \text{dataflow} \rangle = \{ \text{indata}, \text{inflag}, \text{inflie}, \text{source}, \text{outdata}, \text{outputflag}, \text{outfile}, \text{sink} \}
\]

for any positive integer \( n \) and where \( \Lambda \) is the empty string.

<table>
<thead>
<tr>
<th>Sort</th>
<th>Carrier</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>set of tasks</td>
</tr>
<tr>
<td>name</td>
<td>set of task names</td>
</tr>
<tr>
<td>struct</td>
<td>set of structures</td>
</tr>
<tr>
<td>input</td>
<td>set of inputs</td>
</tr>
<tr>
<td>output</td>
<td>set of outputs</td>
</tr>
<tr>
<td>dataflow</td>
<td>set of data flows</td>
</tr>
<tr>
<td>flowname</td>
<td>set of data flow names</td>
</tr>
</tbody>
</table>

The sorts of the signature are mapped to the carriers as shown in Table 1. The symbols are mapped to the following operations:

a. The operation \( \text{task}_A \) specifies the name, structure, input and output of a task.

b. The operations \( \text{sequ}_A \), \( \text{seln}_A \), \( \text{iter}_A \) and \( \text{para}_A \) link up a number of subtasks to form a structure.
c. The operation \text{elem}_A indicates that a structure is elementary, i.e. it does not consist of subtasks.

d. The operations \text{get}_A and \text{put}_A specify the input and output flows for a task.

e. The operations \text{indata}_A, \text{inflag}_A, \text{infile}_A and \text{source}_A specify the name of each input data flow. They also denote, respectively, that the data flow consists of data from some other task, a flag from some other task, data from a file, and data from the environment.

f. The operations \text{outdata}_A, \text{outflag}_A, \text{outfile}_A and \text{sink}_A are similarly used for output data flows.

Different algebras can be defined over the same signature. Homomorphisms, or functions preserving the signature, can be defined from one algebra to another. Such homomorphisms enable us to forget about minor syntactical differences in various specification methods and concentrate on the major issues. For example, suppose

\[
\text{task}_A(p, \text{get}_A(\text{indata}_A(f_1)), \text{put}_A(\text{outdata}_A(f_2)), \text{sequ}_A(t_1, t_2)) =
\]

\[
\begin{array}{c}
\text{f}_1 \Downarrow \\
\text{f}_2 \\
\end{array}
\]

\[
\begin{array}{c}
\text{p} \\
\text{t}_1 \\
\text{t}_2 \\
\end{array}
\]

\[
\ldots (1)
\]

in algebra \( A \), and

\[
\text{task}_B(p, \text{get}_B(\text{indata}_B(f_1)), \text{put}_B(\text{outdata}_B(f_2)), \text{sequ}_B(t_1, t_2)) =
\]

\[
\begin{array}{c}
\text{procedure } p(f_1, f_2): \text{begin } t_1; t_2 \text{ end.} \ldots (2)
\end{array}
\]

in algebra \( B \). Then a homomorphism mapping the variables \( p, f_1, f_2, t_1 \) and \( t_2 \) in \( A \) to the corresponding variables in \( B \) will automatically map (1) to (2).

4. INITIAL ALGEBRA

The algebra that has the richest context is called an initial algebra, satisfying the following property:

An algebra \( A \) is initial if, for any algebra \( B \) over the same signature, there exists a unique homomorphism mapping \( A \) to \( B \).

We would like to construct an initial algebra for structured systems using the concept of term algebras. A term algebra \( T_\Sigma \) for structured systems is defined as follows:

4.1 Carriers

We regard task names and data names as more fundamental than other variables in our algebra, because these names appear unaltered in a final specification. They are like terminals in the theory of formal languages. We will enlarge the signature by putting in task names and data names as 'symbols'. Thus two more sets of symbols are defined:

\[
\Sigma < \wedge, \text{name} > = \text{the set of task names},
\Sigma < \wedge, \text{flowname} > = \text{the set of names of input/output data flows}.
\]

Although also known as ‘operation symbols’, these symbols are actually not operating on any sort.

Let \( X \) denote the enlarged set of symbols, together with three delimiter symbols: ‘(’, ‘;’ and ‘)’. The carriers of \( T_\Sigma \) are made up of \textit{terms} in \( X \), i.e. strings of symbols from \( X \). We define the carriers \( T_\Sigma < s > \) by induction as follows:

a. For any symbol \( q \) in \( \Sigma < \wedge, s > \), we let the term ‘\( q \)’ be in \( T_\Sigma < s > \).

b. For any operation symbol \( q \) in \( \Sigma < s_1 \ldots s_n, s > \), and for any terms ‘\( u_1 \)' in \( T_\Sigma < s_1 > \), ‘\( u_n \)' in \( T_\Sigma < s_n > \), we let the term ‘\( q(u_1; \ldots; u_n) \)' be in \( T_\Sigma < s > \).

4.2 Operations

Operations \( q_T \) in \( T_\Sigma \) are induced from the symbols \( q \) as follows:

a. For any symbol \( q \) in \( \Sigma < \wedge, s > \), we define \( q_T \) to be the term ‘\( q \)’.

b. For any operation symbol \( q \) in \( \Sigma < s_1 \ldots s_n, s > \), and for any terms ‘\( u_1 \)' in \( T_\Sigma < s_1 > \), ‘\( u_n \)' in \( T_\Sigma < s_n > \), we define \( q_T(u_1, \ldots, u_n) \) to be the term ‘\( q(u_1; \ldots; u_n) \)'.

It can be shown that the term algebra \( T_\Sigma \) thus defined is an initial algebra. It can be mapped by homomorphisms to other algebras over the same signature. For example, we will illustrate how the sample term shown in Figure 1 can be related to a Yourdon structure chart, a DeMarco data flow diagram and Jackson structure text.

5. YOURDON STRUCTURE CHARTS

To illustrate how the terms in our initial algebra can be mapped to structure charts, we must first of all define an algebra \( Y \) of Yourdon structure charts (which we will call \textit{Yourdon algebra} for short). The carriers are defined similarly to Table 1. The operations \( q_T \) are defined as shown in Figure 2. Then the obvious homomorphism will map our initial algebra to the Yourdon algebra. The term in Figure 1, for example, will be mapped to the structure chart of Figure 3.

Can we do the reverse? That is to say, can we define a unique reverse homomorphism from the Yourdon algebra to the initial algebra, and hence get back our term? We find that even though reverse homomorphisms can be created, they are not unique. One reason is that some operations, such as \text{infl}_{\text{Y}}, \text{outfl}_{\text{Y}}, \text{source}_{\text{Y}} and \text{sink}_{\text{Y}}, are effectively not used
structured analysis and design models

such that

Figure 2. Operations in Yourdon Algebra

Figure 3. Sample Structure Charts in Yourdon Algebra

in the Yourdon algebra. Another reason is that other operations, although distinct in the initial algebra, may overlap in the Yourdon algebra. For example, both sequ_y and para_y give identical results. The Yourdon algebra must be extended to cater for these operations before a unique reverse homomorphism can be defined. The algebra of extended Yourdon structure charts will then be regarded as isomorphic, or equivalent, to our term algebra.

6. DeMARCO DATA FLOW DIAGRAMS

We can similarly define an algebra D of DeMarco data flow diagrams (or DeMarco algebra for short). The carriers are defined similarly to Table 1, and the operations are as shown in Figure 4. The obvious homomorphism will map the terms in our initial algebra to data flow diagrams. The term in Figure 1, for instance, can be mapped to the DeMarco data flow diagram of Figure 5. Furthermore, if we forget about the intermediate task names in the original term, then a flattened data flow diagram can be obtained, as shown in Figure 6.

The transformation system under study (see Section 5) will also accept terms in the initial algebra and generate DeMarco data flow diagrams automatically. Conversely, given a data flow diagram, the system will enquire the user about extensions to the DeMarco algebra and then generate automatically a term of the initial algebra. The system will therefore help the user to convert DeMarco data flow diagrams into Yourdon structure charts.

7. JACKSON STRUCTURE TEXT

We also want to define an algebra J of Jackson structure text (or Jackson algebra). This is done through the algebraic concept of equations, and involves the
Structured Analysis and Design Models

such that

\[

task_D(p, i, o, s) = \begin{cases} 
  & \text{subject to three further conditions:} \\
  & (a) \text{If } p = \land, \text{ then } task_D(p, i, o, s) \\
  & = \begin{cases} 
    & \text{(b) If } s = \land, \text{ then } task_D(p, i, o, s) \\
    & = \begin{cases} 
      & \text{(c) } \\
      & = \end{cases}
  \end{cases}
\end{cases}
\]

following steps:

a. We define a preliminary Jackson algebra which has the same carrier and operations as our initial algebra.

b. We define four new operations

\[
\begin{align*}
seq_D : & \quad J < name > \times J < task > \to J < struct > \\
seq : & \quad J < name > \times J < task > \to J < struct > \\
\end{align*}
\]

The functions of these operations are shown in Figure 7.

c. Since input and output are not included in Jackson structure text, we must define two equations, or relationships that connect different elements of the algebra together. Thus we have

\[
\begin{align*}
eg & \quad get_D(d_1, \ldots, d_n) = get_D(d_1, \ldots, d_n) \\
& \quad put_D(d_1, \ldots, d_n) = put_D(d_1, \ldots, d_n) \\
& \quad sink_D(f) = f \\
& \quad source_D(f) = f \\
\end{align*}
\]

Figure 4. Operations in DeMarco Algebra
structured analysis and design models

$$\text{sel}_j(p, t_1, \ldots, t_n) = \text{task}_j(p, \text{get}_j(\lambda), \text{put}_j(\lambda), \text{sel}_j(t_1, \ldots, t_n))$$

$$\text{itr}_j(p, t) = \text{task}_j(p, \text{get}_j(\lambda), \text{put}_j(\lambda), \text{itr}_j(t))$$

$$\text{elm}_j(p) = \text{task}_j(p, \text{get}_j(\lambda), \text{put}_j(\lambda), \text{elm}_j)$$

for any task names $p$ and tasks $t_1, t_1, \ldots, t_n$.

In this way, the obvious homomorphism will enable us to map our terms into Jackson text. For example, the term in Figure 1 will be mapped to the Jackson structure text of Figure 8. Furthermore, the transformation system under study can also be enhanced to accept or generate Jackson structure text.

8. CONCLUSION

Structured analysis and design models can be integrated algebraically. A term algebra has been defined and can be mapped through homomorphisms to Yourdon structure charts and DeMarco data flow diagrams. It can also be linked to Jackson structure text through equations.

As a result, specifications can be transformed from one form to another. The most suitable model can be chosen for a target system independently of user familiarity. Algebraic interpreters may be adapted to validate the specifications. Automatic development aids for one methodology may be applied to another.

ACKNOWLEDGEMENTS

Part of this research was done at the London School of Economics, University of London under a Commonwealth Academic Staff Scholarship. The author is indebted to Ronald Stamper, Haya Freedman and Professor Frank Land of LSE for some invaluable suggestions. He is also grateful to Professors Joseph Goguen of Stanford, Jim Emery of Pennsylvania, Blake Ives of Dartmouth and Joe Davis of Indiana for the most encouraging comments on the project.

Get-valid-order:

Get-order:

Process-order:

Process-local-order:

Figure 5. Sample Data Flow Diagram in DeMarco Algebra

Process-sales:

Figure 6. Further Data Flow Diagram in DeMarco Algebra

$$\text{seq}_j(p, t_1, \ldots, t_n) = p \ \text{seq}$$

$$t_1; \ldots; t_n;$$

$$p \ \text{end}$$

$$\text{sel}_j(p, t_1, \ldots, t_n) = p \ \text{sel}$$

$$t_1; \ldots;$$

$$p \ \text{alt}$$

$$t_n;$$

$$p \ \text{end}$$

$$\text{itr}_j(p, t) = p \ \text{itr}$$

$$t$$

$$p \ \text{end}$$

$$\text{elm}_j = p$$

Figure 7. New Operations in Jackson Algebra

The Australian Computer Journal, Vol. 18, No. 3, August 1986
Structured Analysis and Design Models

References


Biographical Note

T. H. Tse received his B.Sc. degree from the University of Hong Kong in 1970, and his M.Sc. degree from the University of London in 1979. He is currently a lecturer in computer science at the University of Hong Kong. His research interests include software engineering and formal methods in information systems.

Mr Tse is a member of the British Computer Society, the British Institute of Management and the Institute of Data Processing Management. He is a council member of the Vocational Training Council in Hong Kong. He was awarded an MBE by the Queen in 1982.
Simple Compiler Correctness—A Tutorial on the Algebraic Approach

P. A. Collier†

A well motivated introduction to compiler correctness using algebraic methods is presented, with emphasis placed on the intuitive notions underlying the algebraic concepts.

Keywords and Phrases: compiler, semantics, algebra
CR Categories: F. 3. 2

1. Introduction

It is now more than 15 years since the paper by Burstall and Landin (1969) introduced the idea of compiler correctness in terms of (universal) algebraic concepts. Since this pioneering work much effort has been expended in refining the appropriate style of algebra to be used in this task, and increasing the size of language whose compiler may be proved correct. For examples of this work, see Morris (1973) and various contributions to the Aarhus workshop on Semantics-Directed Compiler Generation (Jones, 1980), especially by Thatcher et al. (1980), Wand (1980), and Henson (1983). Most of this work, though, is expressed very algebraically, and tends to be inaccessible to a non-mathematically oriented computer scientist. This is a pity, since many of the concepts and ideas underlying the algebraic description method are interesting, and relevant. Furthermore, it turns out that really only two key algebraic results are being used in a proof of simple compiler correctness.

The purpose of this tutorial paper is to attempt to remove the aura of mysticism from an algebraic description of the compiler correctness problem. We have assumed only a little knowledge of programming language semantics, and our language is sufficiently simple that this knowledge should not be absolutely necessary, certainly for anyone with an understanding of compiling issues.

It is hoped that this article will not only serve as an introduction to algebraic specification of compiler correctness, but will also broaden awareness of the fashionable program specification issue. One main theme currently in program specification work is algebraic specification. See, for example, Burstall and Goguen (1981), many contributions to TAPSOFT (Ehrig et al., 1985a, 1985b), and the CIP (1985) project.

A few tutorial style papers in the area of algebra applied to computer science have appeared. Probably the best is by Burstall and Goguen (1982). Indeed that paper complements the present one by adding more formality and detail to the algebraic issues we touch on here. For a presentation biased towards abstract data types see Goguen (1978), while the survey paper by Goguen and Meseguer (1983) covers many of the significant results in algebra relevant to computer science. This tutorial paper has a different focus from all the above in that the algebra is kept to a minimum, while the intuitive understanding of any algebra we are forced to introduce is highlighted. We are only focusing on the simple compiler correctness problem we set ourselves. Readers whose appetite for more algebra is whetted by this example will find plenty of material in the references.

2. Background

The problem of showing that a compiler is correct is currently unsolved in computer science for any real programming language. We can ‘validate’ compilers and show that they behave ‘correctly’ on a large class of example programs. But this does not prove the compiler correct for all programs.

What does it mean for a compiler to be correct? Quite simply the meaning of a source program should be equivalent to the meaning of the target program produced by the compiler. For this purpose we can view the meaning of a program as a function from the program’s input to its output. The problems in establishing an equivalence of this sort come in several forms: semantic complexity of many programming languages in current usage, and the difficulty of capturing this formally; the difference between the source code and target code is hard to bridge, and the fact that there are many different ways of expressing the same ‘meaning’ algorithmically; and finally, in practice, machines are finite and do not correspond with our abstract semantic domains.

†Department of Information Science, University of Tasmania, Hobart, Tasmania. Manuscript received August, 1985; revised June, 1986.

Copyright © 1986, Australian Computer Society Inc.

General permission to republish, but not for profit, all or part of this material is granted, provided that the ACP's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.
Methodologies for proving compilers correct are currently being researched actively. The algebraic approach is one of them. A complete proof will be presented here by avoiding the above problems in the following ways:

- considering a very simple example language
- working with a small difference between source and target code.
- defining a compiler on an abstract machine with no finitary limitations.

Informal description of compiler correctness

When contemplating a problem such as compiler correctness it is easy to be overawed by the enormity of the problem. An example compilation of a reasonably large program illustrates this. A program in a source language like Pascal appears to be a large piece of text and the compiled program in a target language may appear as a very large number of instructions, or even as a different large text. The key contribution from the algebraic approach to compiler correctness is to reject the simple functional view of a compiler as above, and to impose structure on the programs involved. We shall see that the structure used in the algebraic approach is already present in a grammar describing the language, so no new structure is being added, it is simply being accorded significance. Algebraically this structure is referred to as a signature.

The structure that is seen explicitly through algebraic eyes generalises the structure referred to in the 'structured programming' design methodology. Algebraically there is structure in a sequence of statements while for a program to be described as structured it needs to use procedures, blocks, while loops etc. These are also structures algebraically, so it can be seen that high level programming languages have a rich structure when viewed algebraically. Low level languages have a more barren structure.

Algebras have a rich taxonomy. Because they accord structure importance it makes sense to view algebras with the same sense of structure as being more or less closely related. Algebras with a different underlying notion of structure are not related at all. This appears to be very serious for algebraic compiler correctness, but not so. Algebra makes another feature of compiling explicit: a particular construct in the source language will generate a particular sequence of instructions in the target language. If this sequence of instructions is accorded a significance for each structural part of the source language then the target language can be viewed as having the same structure as the source language. Imposing such additional structure on the component instruction sequences means that some instruction sequences in the full target language will not be expressible. This confirms the intuition that a compiler will not normally be able to generate every target language program, from all legal source programs. Often only a small proportion of them will be generable.

Within the family of algebras with the same underlying structure, or signature, there are a number of useful theorems which aid the compiler correctness proof. A function from one of these algebras to another is called a homomorphism if it respects the structuredness of the two algebras. Homomorphisms play an important part in the compiler correctness proof expressed algebraically.

As noted earlier the target language can be structured to mirror the structure of the source language. This enables the compiler to be expressed as a homomorphism. Additionally the meanings given to the language, which are function domains, can also be structured, and the semantic functions will also be homomorphisms.

A compiler correctness proof using algebra becomes much easier when it is seen that the algebra representing the source language has an important property. The property is that each object generated by the underlying structure of the algebra represents a different program, and all the different programs are characterised in this way. Part of the motivation for giving meaning to programs is to determine which programs have the same meaning. For example functional equivalence or operational equivalence would give different classes of equivalent programs. Initially, at the syntactic level, programs are different if they have different structure. This distinguished algebra, where each program is represented by its structure, is a good candidate to be viewed as the abstract syntax for the programming language. We will call this algebra the word algebra.

Two key results aid in the compiler correctness proof expressed algebraically. The first is that there is a unique homomorphism from the word algebra to any other algebra with the same structure. So to define a homomorphism it is only necessary to define another algebra with the same structure. Consequently imposing a source language structure on top of the target language is tantamount to defining a compiler. And imposing the source language structure on a collection of functions implicitly defines a semantics.

The second key result is that the composition of two homomorphisms on a particular structure is also a homomorphism on that structure. This is useful when we see a diagram of the objects and homomorphisms we have described in this section (Figure 1).

Each of the four vertices have the same underlying structure, although both target language vertices have had this structure imposed on them. Each arrow is a homomorphism on this structure. In the diagram consecutive arrows represent composition of homomorphisms. This pair also represent a homomorphism by our second result. The first result gives us that any homomorphism from the source
3. Algebraic Prerequisites

This section consists of prerequisites for the development of a compiler for expressions involving only addition. This is a language traditionally used in treatments of semantics for expository purposes. The treatment will not avoid the symbolism found in other treatments of this material, but it will be related to the earlier informal discussion. In fact we will keep the symbolism consistent throughout eschewing the frequent habit of introducing shorthand notation, with disambiguation ‘clear from the context’. This will lead to a proliferation of subscripts and parentheses at times, but these have a consistent meaning. A glossary for our symbols will be found as an appendix.

The first definition captures the property of an algebra that we called its ‘structure’ in the last subsection. Structure is characterised by the ‘signature’ of the algebra.

Definition: A collection of operator symbols \( \Sigma \) (read as ‘sigma’), is called a signature. A signature, \( \Sigma \), can be viewed as an infinite union:

\[
S = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup \ldots
\]

where members of:

\( \Sigma_0 \) are function symbols of zero arguments (i.e., constant symbols)

\( \Sigma_1 \) are function symbols of one argument

\( \Sigma_2 \) are function symbols of two arguments etc.

The names of the symbols in each \( \Sigma_i \) \((0 \leq i)\) are not important, the signature simply characterises abstract structure. For purposes of exposition we shall always fix upon a name for the symbols of a signature.

Definition: An algebra, \( A(\Sigma) \), with signature \( \Sigma \) consists of a set \( A \), called the carrier, together with a function \( \sigma_{A(\Sigma)} \) for each symbol \( \sigma \) in the signature \( \Sigma \). The functions \( \sigma_{A(\Sigma)} \) are total and closed on the carrier and the number of arguments to \( \sigma_{A(\Sigma)} \) must equal that of the \( \sigma \) in the signature to which it corresponds.

A function is total and closed on a set if it gives a result for all combinations of arguments from the set, and the result is also contained in the set. Note that we will denote a symbol from a signature by some name, \( \sigma \) say. When these are given meaning for an algebra \( A(\Sigma) \) we will subscript the symbol name with algebra names, so \( \sigma_{A(\Sigma)} \) will denote a function on the carrier of \( A(\Sigma) \). This convention is not normally rigorously applied in the literature, \( \sigma \) and \( \sigma_{A(\Sigma)} \) are generally conflated, with disambiguation possible from the context.

Examples: Consider the signature defined as

\[
\Sigma = \bigcup_{i=0}^{\infty} \Sigma_i
\]

where \( \Sigma_2 = \{ \ast \} \) and for all other \( i \in N, \Sigma_i = \{ \} \). That is \( \Sigma \) contains one function symbol which requires two arguments. The following are two algebras with this signature:

1. \( N(\Omega) \): Carrier \( N \) (the natural numbers), and operation \( \ast_{N(\Omega)}(p_1, p_2) = p_1 + p_2 \)
2. \( T(\Omega) \): Carrier \{true, false\} and operation \( \ast_{T(\Omega)}(t_1, t_2) = t_1 t_2 \)

All algebras having the same signature have the same structure. Notice that the signature in this case is relatively barren, so it is easy to ‘naturally’ see many classes of objects with this structure.

Definition: Given a signature, \( \Sigma \), with at least one operation symbol contained in \( \Sigma_0 \) we can form a \( \Sigma \)-word algebra. There are many ways of writing down a \( \Sigma \)-word algebra for any particular \( \Sigma \). All of them are isomorphic, or indistinguishable in an abstract sense. One way of writing down a \( \Sigma \)-word algebra is as follows:

The carrier of a \( \Sigma \)-word algebra is the set of terms \( \Omega \) defined as follows:

1. The operation symbols in \( \Sigma_0 \) are all terms.
2. If \( \sigma \in \Sigma_i \) and \( t_1, \ldots, t_i \) are terms then \( \sigma(t_1, \ldots, t_i) \) is a term.
3. Nothing else is a term.

We will formally define a \( \Sigma \)-word algebra \( W(\Sigma) \), as follows: the signature is \( \Sigma \), the carrier, \( W \), and the operations are:

---

The Australian Computer Journal, Vol. 18, No. 3, August 1986
Simple Compiler Correctness

1. If \( \sigma \in \Sigma_0 \) then \( \sigma_{W(\Sigma)} = \text{"o( "} \)
2. If \( \sigma \in \Sigma_i \) and \( t_1, \ldots, t_i \) are terms (which are therefore elements of the carrier of \( W(\Sigma) \)) then \( \sigma_{W(\Sigma)}(t_1, \ldots, t_i) = \text{"o( "} t_1 \text{", "} t_2 \text{", \ldots, "} t_i \text{")} \)

This definition may appear circular, but this is the way it should be. A \( \Sigma \)-algebra consists of a carrier and functions for the operation symbols in \( \Sigma \). The function symbols have to be closed on the carrier, so the carrier has to contain all the finite words constructed from operation symbols. Then the meaning of the operation symbols applied to \( i \) arguments is the operation symbol applied to \( i \) arguments. Of course this is the algebra we use to characterise the syntax of a programming language. If a signature is chosen to reflect the BNF description of a programming language then the carrier of the word algebra for that signature is an abstract realisation of all possible programs in the language. (Often this requires a many-sorted signature, generalising the notion of signature here.)

Example: A definition of the syntax of a simple expression language can be described in BNF as follows:

\[
\langle \text{expression} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{numeral} \rangle \mid (\langle \text{expression} \rangle + \langle \text{expression} \rangle)
\]

Formal definitions of \( \langle \text{identifier} \rangle \) and \( \langle \text{numeral} \rangle \) are not provided, they are assumed to be standard.

Algebraically this syntax is represented as a signature, which we will call \( \Omega \) (read as 'omega'). Recall that

\[
\Omega = \bigcup_{i=0}^{\infty} \Omega_i
\]

and each \( \Omega_i \) \( (i \geq 0) \) needs to be defined:

\[
\begin{align*}
\Omega_0 &= \{ i \mid i \text{ is an } \langle \text{identifier} \rangle \} \\
\Omega_1 &= \{ n \mid n \text{ is a } \langle \text{numeral} \rangle \} \\
\Omega_2 &= \{ + \} \\
\Omega_3 &= \Omega_4 = \ldots = \{ \}.
\end{align*}
\]

\( \Omega_0 \) has operation symbols for each identifier and each numeral. These have no arguments. The only other operation symbol, \( + \), has two arguments. Given the signature \( \Omega \) the language described by \( \Omega \) is the \( \Omega \)-word algebra. This is an abstract object, but according to the constructive definition given earlier the expression

\[
((3 + j) + k)
\]

is represented as:

\[
+(+(3(),j()),k())
\]

in the \( \Omega \)-word algebra. Clearly the exact textual form of a program is not important except for the pragmatics of using the language.

This section on algebraic prerequisites is completed with the definition of homomorphism, together with the two algebraic results required for a simple compiler correctness proof.

Definition: A \( \Sigma \)-homomorphism, \( h \), from a \( \Sigma \)-algebra \( A(\Sigma) \) to another \( \Sigma \)-algebra \( B(\Sigma) \) is a function which preserves the structure of the algebras in the following sense:

\[
\begin{align*}
\text{for all } \sigma \in \Sigma_0: \quad h(\sigma_{A(\Sigma)}) &= \sigma_{B(\Sigma)} \\
\text{and for all } \sigma \in \Sigma_i, \text{ with } i \geq 1: \quad h(\sigma_{A(\Sigma)}(a_1, \ldots, a_i)) &= \sigma_{B(\Sigma)}(h(a_1), \ldots, h(a_i))
\end{align*}
\]

Recall that \( \sigma_{A(\Sigma)} \) denotes a function on the carrier of \( A(\Sigma) \) represented by \( \sigma \in \Sigma_j \), and \( \sigma_{B(\Sigma)} \) denotes that function on the carrier of \( B(\Sigma) \) represented by \( \sigma \). This information can be extracted from the definitions of how \( A(\Sigma) \) and \( B(\Sigma) \) are in fact \( \Sigma \)-algebras.

Because of the way a homomorphism preserves structure, and because a word algebra consists solely of this structure it turns out that there is only one way to define a homomorphism from a word algebra to any other algebra with the same signature:

**Theorem 1:** For any signature \( \Sigma \), and any \( \Sigma \)-algebra \( A(\Sigma) \), there is a unique \( \Sigma \)-homomorphism from a \( \Sigma \)-word algebra to \( A(\Sigma) \).

This turns out to be a very powerful and useful result. The second result is perhaps much less surprising than the last, but is still most useful.

**Theorem 2:** The composition of two homomorphisms, \( f : A(\Sigma) \rightarrow B(\Sigma) \), and \( g : B(\Sigma) \rightarrow C(\Sigma) \) is also a homomorphism:

\[
(f \circ g) : A(\Sigma) \rightarrow C(\Sigma).
\]

4. Source and Target Languages

This section describes the source and target language for the simple compiler to be proved correct.

4.1 Source Language, \( S(\Omega) \)

This has been given as an example in Section 3. It is the language of addition expressions with identifiers and numbers as primitive expressions. Algebraically the syntax is described by \( \Omega \), and the full language is realised as the carrier of a \( \Omega \)-word algebra.

4.2 Target Language, \( T(\Sigma) \)

As might be expected this will be defined as the carrier of a word algebra. The language is simple, with operations designed for use on a simple stack machine. With a single instruction it is possible to load a numeral, or the value of an identifier on the stack, or add the top two elements of the stack. Instructions are programs which may be pasted together to form a program using the concatenate operation we will denote by \( \ast \). All these objects are captured in the definition of signature \( \Sigma \):

\[
\Sigma = \bigcup_{i=0}^{\infty} \Sigma_i
\]
Simple Compiler Correctness

where

\[ \Sigma_0 = \{ \text{load } i \mid i \text{ is an } < \text{identifier}> \} \]
\[ \cup \{ \text{li } n \mid n \text{ is a } < \text{numeral}> \} \]
\[ \cup \{ \text{add} \} \]
\[ \Sigma_1 = \{ \} \]
\[ \Sigma_2 = \{ * \} \]
\[ \Sigma_3 = \Sigma_4 = \ldots = \{ \} \]

The example expression

\[ ((3 + j) + k) \]

will be compiled to the target language program:

```plaintext
li 3
load j
add
load k
add
```

which will be represented as

\[ \bullet (\bullet (\text{li } 3(), \text{add}())), \text{add}()) \]

in the formulation of the \( \Sigma \)-word algebra according to the rules given earlier. The exact bracketting of operands of \( \bullet \) is irrelevant to the meaning of the target language programs, since the operation is associative. A further development of the algebraic concept is required to specify this formally.

In the informal treatment of the compiler correctness it was noted that the signatures of the source and target language would usually be different. In this case they are the same, as operation symbols are arbitrarily named, but they still do not correspond. This is because the operation requiring two arguments in the source is to model the addition operation, while in the target it concatenates programs. Consequently the target language has to be remodelled to conform to the signature \( \Omega \). Or, more precisely, the part of the target language required by the compiler has to be remodelled. There are a number of programs in the target language that the compiler will never create, such as the program:

```plaintext
load j
li n
```

4.3 Target Language with signature \( \Omega \), T(\( \Omega \))

To describe a \( \Omega \) algebra we need a carrier, and definition for the operation symbols in \( \Omega \) which are closed on the carrier.

Carrier of T(\( \Omega \)): Subset of carrier of T(\( \Sigma \)) being all the terms from the carrier of T(\( \Sigma \)) that can be generated by the following operations:

For each \( n \) which is a \( < \text{numeral}> \): \( n_{T(\Omega)} = \text{li } n \)

For each \( i \) which is an \( < \text{identifier}> \):

\[ i_{T(\Omega)} = \text{load } i \]

\[ +_{T(\Omega)} (p_1, p_2) = \bullet (p_1, \bullet (p_2, \text{add})) \]

\[ = p_1 \cdot p_2 \cdot \text{add} \]

Now we have defined T(\( \Omega \)) it is possible to talk precisely about the compiler's target language. This is the carrier of T(\( \Omega \)), which contains all the programs that may be generated by the compiler and only those. It can be seen that these programs either consist of a single instruction or consist of two programs pasted together followed by an add instruction.

Defining T(\( \Omega \)), the target language, also implicitly defines the compiler homomorphism, as the source language is represented by a \( \Omega \)-word algebra (by Theorem 1). The compiler will be denoted by C. An explicit definition for C can by written as follows, as an aid to the intuition:

\[ C[i] = \text{load } i \quad \text{if } i \text{ is an } < \text{identifier}> \]
\[ C[n] = \text{load } n \quad \text{if } n \text{ is a } < \text{numeral}> \]
\[ C[e_1 + e_2] = \bullet (C[e_1], \bullet (C[e_2], \text{add})) \]

The brackets ']' and '[' are used to avoid confusion with the parentheses which are an important part of our formal algebraic representation of a target language program.

5. Source and Target semantics

To be able to discuss compiler correctness a notion of meaning for programs is required. The correctness will be relative to the chosen notion of meaning, so agreement must be reached that this meaning is appropriate. A well known algebraic trick of having a 'one point algebra' to represent meaning will give a compiler correctness proof, but not a generally acceptable one.

The languages in this paper are so simple that there can be little argument about the meaning that should be given.

5.1 Source language meaning, SM(\( \Omega \))

The meaning of a source language expression will be a number. There is only one slight complication in calculating this, that is what value is to be given to identifiers appearing in an expression. A traditional way to treat this issue is to assume the existence of an environment being a mapping from identifiers to their numeric values. An environment models a symbol table in a compiler. The meaning of an expression will be a function which given an environment will yield the numeric value of an expression. Formally this is expressed as:

\[ \text{Carrier of SM(\( \Omega \)) : ENV } \rightarrow \text{ N} \]

(where ENV : \( < \text{identifier} > \rightarrow \text{ N} )

Definitions for the operation symbols of \( \Omega \) for this carrier follow:

For each \( n \) which is a \( < \text{numeral}> \):

\[ n_{SM(\( \Omega \)} = \text{val } n \in \text{ ENV } \rightarrow \text{ N} \]

where \( \text{val } n \rho \) is the number represented by the numeral \( n \). The environment argument, \( \rho \), is ignored by val.

The Australian Computer Journal, Vol. 18, No. 3, August 1986
Simple Compiler Correctness

For each i which is an <identifier>:

\[ i_{SM(0)} = \text{find } i \in \text{ENV} \rightarrow \text{N} \]

where find \( i \) ‘looks up’ the value of \( i \) in the environment \( \rho \).

\[ +_{SM(0)}(m_1, m_2) = m_1(\rho) + m_2(\rho) \]

where the ‘+’ on the right hand side denotes the addition function on numbers.

Note that

\[ +_{SM(0)} \in ([\text{ENV} \rightarrow \text{N}] \times [\text{ENV} \rightarrow \text{N}]) \rightarrow [\text{ENV} \rightarrow \text{N}] \]

is a fairly complex function. It takes first an argument which is a pair of program values, then an argument which is an environment. Given these it returns a numeric value found by applying the two program values to the environment and summing the respective numeric results.

Again defining \( SM(0) \) is tantamount to explicitly providing a semantics for the source language, by theorem 1. An explicit realisation of the semantic function is provided, which is denoted by \( E \):

\[
\begin{align*}
E[i]\rho &= \text{val } i \rho \\
E[e_1 + e_2]\rho &= +_{SM(0)}(E[e_1], E[e_2])\rho \\
&= E[e_1]\rho + E[e_2]\rho
\end{align*}
\]

5.2 Target language semantics, \( TM(\Sigma) \)

To give the semantics for the compiler target language \( T(0) \) an indirect route is taken. Firstly the semantics for the full target language is provided. This language is designed as a stack language, so the carrier of the semantic algebra will model a stack of numbers. Such a stack is modelled mathematically by the domain \( \text{N}^* \) which may have numbers added to it by the operation push:

\[ \text{push} : [\text{N} \times \text{N}^*] \rightarrow \text{N}^* \]

The meaning of programs in the target language is a transformation on stacks, modelled by \( \text{N}^* \rightarrow \text{N}^* \). This still needs to be in the context of an environment to give values for identifiers.

So we define

\[ \text{Carrier of } TM(\Sigma) : \text{ENV} \rightarrow [\text{N}^* \rightarrow \text{N}^*] \]

Given that \( \xi \) is a stack we define operation symbols of \( \Sigma \) as follows:

For each \( li \) \( n \):

\[ li_{TM(\Sigma)} = \text{stack } n \in \text{ENV} \rightarrow \text{N}^* \rightarrow \text{N}^* \]

where

\[ \text{stack } n \rho \xi = \text{push} (\text{val } n \rho, \xi) \]

For each load \( i \):

\[ \text{load }_{TM(\Sigma)} = \text{stack } i \in \text{ENV} \rightarrow \text{N}^* \rightarrow \text{N}^* \]

where

\[ \text{stack } i \rho \xi = \text{push} (\text{find } i \rho, \xi) \]

\[ \text{add}_{TM(\Sigma)} = \text{addst } \in \text{ENV} \rightarrow \text{N}^* \rightarrow \text{N}^* \]

where

\[ \text{addst } \rho (\text{push } (v_2, \text{push } (v_1, \xi))) = \text{push } (v_1 + v_2, \xi) \]

and

\[ \text{addst } \rho (\xi) = \xi \]

if \( \xi \) does not have at least two numbers pushed onto it.

Note that the second clause in the definition of \( \text{addst} \) is given to make \( \text{addst} \) total. Its arbitrary definition does not affect the compiler correctness as the compiler generates sensible target language programs.

\[ \bullet_{TM(\Sigma)}(m_1, m_2) \rho \xi = m_2 \rho (m_1 \rho \xi), \quad \text{so} \]

\[ \bullet_{TM(\Sigma)} \in ([\text{ENV} \rightarrow \text{N}^* \rightarrow \text{N}^*] \times [\text{ENV} \rightarrow \text{N}^* \rightarrow \text{N}^*]) \rightarrow [\text{ENV} \rightarrow \text{N}^* \rightarrow \text{N}^*] \]

We have used \( m_1 \) and \( m_2 \) as arguments to \( \bullet_{TM(\Sigma)} \) to remind readers that the arguments arise as the meanings of ‘sequences’ of machine code instructions.

6. The correctness proof

A diagram of the algebraic expression of our simple compiler and its correctness is shown in Figure 2.

So far details have been given of the left hand and right hand down arrows and the two arrows across the top, together with all relevant source and target algebras. The solid arrows denote homomorphisms, while the dashed arrow, labelled \( \pi \), is a method by which the \( \Omega \)-algebra \( T(\Omega) \) is derived from the \( \Sigma \)-algebra \( T(\Sigma) \).

The parts of the above diagram not so far described are those required to perform the compiler proof. Only one more algebra needs to be defined: that being \( TM(\Omega) \), the target language semantics. As the diagram indicates this is derived from \( TM(\Sigma) \) in the same way that the target language algebra \( T(\Omega) \) is derived from \( T(\Sigma) \).

![Figure 2. Correctness Proof Diagram](image-url)
Carrier of TM(Ω) = ENV → N* → N*

(Note: We have been somewhat informal about the nature of our carriers when they have been function domains. If we assume the carriers consist of all the ‘continuous’ functions on the component domains we get all the functions we require, but also a lot of ‘junk’ functions which the homomorphism from the term algebra does not require. Our carrier of TM(Ω) looks the same as that we used for SM(Ω) just as T(Ω) had a carrier whose domain was similar to T(Σ). However programs in T(Ω) will have meanings in a ‘smaller’ part of the domain than those of T(Σ). This is the part of N* → N* which only allows the addition of one extra number to the input stack.)

The operator symbols in Ω, for the algebra TM(Ω) are defined:

For each n : nTM(Ω) = stack n
For each i : iTM(Ω) = stack i
+TM(Ω)(m₁, m₂) = •TM(Ω)(m₁, •TM(Ω)(m₂, addst))

Definitions of the objects on the right hand side of these definitions are to be found in section 5.

Having now defined T(Ω) and TM(Ω), being the target language and its meanings, we now require the homomorphism Mx, which will provide a meaning for any target language program. The most elegant approach to this is to derive Mx from M, using the π derivation, and this is possible. A different approach is adopted, since one aim here is to keep to a get language and its meanings, we now require the homomorphism , which will provide a meaning for any target language program. The most elegant approach to this is to derive M, from M, which will provide a meaning for any target language program. The most elegant approach to this is to derive M, from

To increase confidence in its reasonableness, its definition is given below:

For each n : nT(Ω) = li n
Mx[li n] = stack n
For each i : iT(Ω) = load i
Mx[load i] = stack i
+T(Ω)(i₁*, i₂*) = •(i₁*, •(i₂*, add))
Mx•(i₁*, •(i₂*, add))ρ = addst ρ(M[i₁*]ρ (M[i₂*]ρ i))

At this stage we have nearly filled in all the details of our earlier diagram. We are left with the most difficult problem of showing the existence of a Ω-homomorphism U from MS(Ω) to MT(Ω). Unfortunately there are no short cuts in this case. We will actually have to prove that our function U is a Ω-homomorphism. Notice that U has to relate the domain ENV → N with ENV → N* → N*. So given an environment, U must relate a number with a stack transformation. We have already commented that the part of N* → N* which interests us in the algebra MT(Ω) is the part which only adds one number to the argument stack. If we ignore what the argument stack is, we can relate v with the function which adds v to its stack argument.

Definition: U : [ENV → N] → [ENV → N* → N*] is defined as U(m) ρ = push (m ρ, ρ)

This is a perfectly acceptable definition of a (complex) function. But, is it a Ω-homomorphism between the algebras MS(Ω) and MT(Ω), whose carriers are the domain and codomain of U respectively? The answer is yes, which we now prove:

Theorem 3: U is a Ω-homomorphism.

Proof: We have to show that U satisfies the Ω-homomorphism property given in section 3.

1. For n ∈ Num :
   U(nSM(Ω)) ρ = push (val n ρ, ρ)
   = stack n ρ = nTM(Ω)ρ

2. For i ∈ Ide :
   U(iSM(Ω)) ρ = push (find i ρ, ρ)
   = stack i ρ = iTM(Ω)ρ

3. U(+(SM(Ω)(m₁, m₂)) ρ = push(m₁ ρ + m₂ ρ, ρ)
   = addst ρ push (m₂ ρ, push (m₁ ρ, ρ))
   = addst ρ U(m₂)ρ push(m₁ ρ, ρ)
   = addst ρ U(m₂)ρ U(m₁ ρ, ρ)
   = addst ρ (•TM(Ω)(U(m₁), U(m₂)) ρ)
   = •TM(Ω)(•TM(Ω)(U(m₁), U(m₂)), addst ρ)

The Australian Computer Journal, Vol. 18, No. 3, August 1986
Simple Compiler Correctness

\[ = +_{T_H(\Omega)}(U(m_1), U(m_2)) \rho \]

The above proof is purely mechanical, using only definitions of the elements of the signature \( \Omega \), in the various algebras we have defined. Now we have homomorphisms from \( S(\Omega) \) to \( MT(\Omega) \) by two routes, referring back to our diagram. Theorem one tells us that these two must be the same hence our compiler is correct with respect to our semantic embedding function \( U \).

7. Conclusions
We have shown the proof of correctness of a compiler for a very simple language. No claim is being made about the extensibility of the method to more complex programming languages. Indeed adding any looping facility would cause a large increase in the complexity of the proof (Thatcher et al., 1980). Another issue which is far from resolved is the nature of the homomorphism \( U \). In our simple example it clearly amounts to a compiler correctness, but simply having a \( U \) in a more general setting does not. There have been suggestions that \( U \) should 'go the other way', or it should be more than just a homomorphism. It is clear that the final form of algebraic ideas applied to computing science has yet to be formulated, but currently it seems that these ideas may become very influential.

8. Acknowledgments
Advice regarding presentation from several sources, including the referees, is gratefully acknowledged.

Appendix: Glossary of Symbolic Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Omega, S )</td>
<td>Refer to a signature which consists of a continued union, for example: ( \Omega = \Omega_0 \cup \Omega_1 \cup \Omega_2 \cup \ldots )</td>
</tr>
<tr>
<td>( \Omega_i, \Sigma_i, i \geq 0 )</td>
<td>Refer to that part of a signature consisting of all the operator symbols of ( i ) arguments. (These refer to names which will later be associated with functions of ( i ) arguments.)</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>Refers to an arbitrary member of a signature.</td>
</tr>
<tr>
<td>( \sigma_A(\Sigma) )</td>
<td>Refers to a particular function associated with operator symbol ( \omega ) from signature ( \Sigma ) by the definition of the algebra ( A(\Sigma) ).</td>
</tr>
<tr>
<td>( A(\Sigma) )</td>
<td>Refers to a particular algebra with signature ( \Sigma ). The definition of the algebra must pair appropriate functions with the operator symbols in ( \Sigma ), and give a carrier, ( A ), for the algebra.</td>
</tr>
</tbody>
</table>

A Refers to the carrier of an algebra, \( A(\Sigma) \), say. This is the set (or class) upon which the functions associated with each operator symbol in \( \Sigma \) are defined, and have their range.

References

Biographical Note
Phil Collier gained a B.Sc. from Hull University and an M.Sc. from Essex University. He worked at Stirling University before moving to Tasmania. His research interests are currently in applications of functional programming to relational database management, typing schemes in functional programming, and implicit typing.
A Survey of Control Facilities in Logic Programming

T. Vasak†

One of the major problems in logic programming that needs to be overcome is the provision of control facilities more powerful than those currently available yet which preserve the declarative semantics of programs. The topic has been the subject of considerable research in recent years with many diverse approaches being taken. This survey provides a critical overview of these various attempts, highlighting their strengths and weaknesses.

Keywords and Phrases: logic programming, PROLOG, control mechanisms
CR Categories: D.3, I.1, I.2

1. Introduction

The control facilities available in early logic programming languages such as PROLOG are too naive to allow programmers to solve problems in a clear yet efficient manner. The depth-first search strategy used by PROLOG has been recognised as inadequate and many diverse attempts have been made to invent more sophisticated control facilities. This paper briefly introduces some of the more interesting approaches to the problem highlighting their strengths and weaknesses. It is not intended to present an exhaustive literature survey of the field; a more complete bibliography is given by Vasak (1986).

Before focussing on the details of existing attempts at remedying this deficiency, it will be useful to describe two examples that illustrate some of the control problems that can occur with PROLOG.

A. Appending Three Lists

This example is referred to by Naish (1982) and was apparently introduced by Darlington (1981). The problem is to implement a predicate

\[
\text{append3}(A, B, C, D).
\]

which is true if the list \(D\) is list \(C\) appended to list \(B\) appended to list \(A\). The required program expressed in the syntax of UNSW-PROLOG (Sammut, 1983; Sammut and Sammut, 1983) is:

\[
\begin{align*}
\text{append3}(A, B, C, D) & : - \\
\text{append}(A, B, E), \\
\text{append}(E, C, D).
\end{align*}
\]

Now, evaluating the goal

\[
\text{append3}([f, r], [e], [d], X).
\]

the interpreter will execute the first call on \text{append}, binding the temporary variable \(E\) to \([f, r, e]\). The second call to \text{append} will then bind \(X\) to \([f, r, e, d]\) as expected. Subsequent backtracking will yield no further solutions. However, it is also possible to invoke the \text{append3} predicate in other modes, for instance, to determine a prefix list \(X\) of a list \([f, r, e, d]\) given the suffix lists \([e]\) and \([d]\):

\[
\text{append3}(X, [e], [d], [f, r, e, d]).
\]

The first call to \text{append} will match the first clause, binding \(X\) to \([\].\). The second \text{append} call will fail immediately and backtracking occurs. Retrying the first call will result in \(X\) being bound to a list of one element (the element being uninstantiated). Subsequently, the second \text{append} call will fail, causing \(X\) to be bound to a list of two unbound elements by retrying the first call. On this occasion, the second \text{append} call will end in success with \(X\) being bound to \([f, r]\).

As Naish (1982) points out, not only is the time complexity of the algorithm proportional to the square of the length of the list but more seriously, the algorithm will not even terminate. Following the success of the second \text{append} predicate, backtracking will attempt to retry the first call. Rather than failing, \(X\) will be successfully bound to a list of three uninstantiated elements. Clearly the second call must fail! But now \(X\) is re-bound by the first predicate to a list of length four and so on. Consequently, invocation of \text{append3} in this manner will not terminate because

Copyright © 1986, Australian Computer Society Inc.

General permission to republish, but not for profit, all or part of this material is granted, provided that the ACJ's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

†Department of Computer Science, University of New South Wales, Kensington, NSW, 2033. Manuscript received March, 1986; revised June, 1986.

The Australian Computer Journal, Vol. 18, No. 3, August 1986
the first `append` call in the `append3` predicate is satisfied by infinitely many substitutions.

Within the scope of the given depth-first computation rule are a number of possible solutions to this problem. One technique that can be used to force termination is to reorder the clause literals in the definition of the `append3` clause. This will have the desired effect in solving (A.2), but will now fail to terminate for (A.1). Unfortunately, no reordering satisfies both joining and splitting lists.

Another possibility is to place a `cut` after the second `append` call in the `append3` clause. This has the effect of saying that there is at most one solution to any invocation of the `append3` predicate. While that solves the problem of (A.2), it destroys correctness of the predicate when invoked in certain modes; for example, the following query has eight solutions:

```
?- append3([i, n], L1, L2, [i, n, c, o, r, e, c, t]).
```

Yet another is to duplicate the logic of `append3` and embed non-logical control predicates such as `var`, `non-var` and `cut`. Regrettably, this obscures the declarative semantics of the program whilst introducing an increase in program size (and consequently decrease in execution speed) that is exponential in the number of clause variables.

**B. Generating Infix Binary Tree Traversals and Permutation Trees**

Consider the task of writing a predicate `inorder(T, L)` which given a binary tree `T` will compute the list `L` of node values in infix order. The following program is a straightforward translation of the specification into PROLOG:

```
inorder(T, []).
inorder(T, [N | RI, LI]):- 
inorder(LI, RI),
inorder(R, RI),
append(LI, [N | RI], L).
```

where `0` is used to denote an empty tree. The above has a dual problem: given a list `L` of elements, find all ordered binary trees `T` that can be constructed by order-preserving insertions of elements in `L` taken in all possible permutations. Careful examination of this problem shows that it is specifiable in the same way as the inorder binary tree traversal and, hence, should be solvable by the above program. The query

```
?- inorder(X, [1, 2, 3]).
```

should generate a set of five bindings for `X`. Follow the execution of query (B.1). Firstly, the goal will match the second `inorder` clause definition. The left subtree will be instantiated to `0` followed by the right, thus corresponding to the tree `t(0, _, _)`. However, the `append` call will fail and backtracking will reinitialize the right subtree to a non-empty node. This binding will again fail in the `append` call. This process of growing the tree continues until the tree `t(0, _, t(0, _, t(0, _, 0)))` is constructed. The `append` call here succeeds and yields the first solution. In an attempt to discover other solutions, control will backtrack into the most recent `inorder(R, RI)` call and resatisfy it with a non-empty node. It is clear that the result tree is too large and hence the `append` call will not succeed, yet quite senselessly the computation constructs ever increasing trees. Other queries such as

```
?- inorder(X, [1]).
```

yield the complete set of solutions but do not terminate.

**2. Broad Overview of Current Methods**

Both of above examples are typical instances of the `generate and test` syndrome. In short, this occurs where a subgoal `Lj` which has an infinite number of solutions lies to the left of another subgoal `Lk` which repeated rejects the bindings generated by `Lj`.

```
C := L1, L2, · · · , Lj, · · · , Lk, · · · , Lm.
```

The left-to-right depth-first control strategy disallows any form of communication from literal `Lk` to `Lj` for `1 ≤ j < k ≤ m`. Thus when control backtracks into `Lj`, it is not known whether retrying `Lk` may lead to the satisfaction of `C`. As was evident from the preceding examples, this gives rise to some quite primitive behaviour in certain cases resulting in inefficiency, and may lead to non-termination. In some instances, execution will not terminate although all solutions have been discovered while on other occasions, some solutions may not be computed.

This behaviour can be explained using some of the theoretical work of Lassez and Maher (1984) on computation rules. They show that SLD resolution is complete with respect to finite failure assuming a fairness condition. In other words, any query not consistent with the program is guaranteed to fail as long as the fairness criterion is met. A computation rule is defined to be fair if and only if it does not ignore any subgoal for an infinite number of execution steps. Unfortunately, the depth-first rule employed in the above examples is not fair. One example of a fair rule is the breadth-first rule which cycles through the subgoals in a regular fashion. Regrettfully, the breadth-first rule is not particularly efficient in detecting failure. This means that it may explore very large search spaces before failing or discovering a refutation. Coupled with its poor space complexity, this renders the breadth-first rule as an unsuitable alternative to the depth-first.

Both the depth-first and breadth-first rules are examples of `static` computation rules, that is, the selection of subgoals to be evaluated is independent of program content. For example, the depth-first rule will always select the leftmost atom whereas the breadth-first rule will cycle through the goals.
An alternative is to employ a dynamic rule which responds in some way to the requirements of the program. It may do so in different ways, either by itself analysing the state of the computation or by utilising some programmer-supplied information to aid its decision on which atom to select. There has been much research into investigating such control rules.

Just as Kowalski’s procedural interpretation of logic programs executing under a depth-first rule (Kowalski, 1974) provided an operational model very similar to the model upon which the conventional programming framework is founded, it is also possible to develop procedural interpretations of the breadth-first rule and of dynamic computation rules.

A breadth-first environment produces a timeshared or multiplexed execution of subgoals where processing time is shared equitably between the constituent parts of the computation. An analogy is process management in a timeshared operating system.

A procedural interpretation of dynamic computation rules is that of coroutined execution where the execution of various subgoals is effectively interleaved. This interpretation has led to such control rules being described as coroutining rules.

The remainder of this section will provide a very brief overview of the existing attempts at providing more sophisticated control strategies; succeeding sections will address these methods in detail.

It is very difficult to settle on an appropriate taxonomy for classifying nearly the various approaches. There are numerous criteria that could be used to categorise individual strategies; for example, they could be classified on the basis of whether they provide local or global control facilities or alternatively, as Naish (1985) does, according to certain properties that they possess. The strategies have been broadly grouped under five headings, namely, coroutined execution, intelligent backtracking, higher order control specification, program transformation and miscellaneous. While this classification is adequate, several strategies may span one or more of these categories.

2.1. Coroutined Execution
Considerable effort has been expended in attempting to produce a useful coroutined execution facility. Basically, these have all tried to find methods in which to express which predicate calls should be suspended and when they should be suspended. Most have attempted to embed the coroutining facility in a depth-first system.

The most prominent systems that have emerged have been IC-PROLOG (Clark and McCabe, 1980), EPILOG (Porto, 1982), PROLOG-II (Colmerauer, 1982), Two-Level PROLOG (Porto, 1984), METALOG (Dincbas, 1980), PROLOG-M (Babb, 1983) and MU-PROLOG (Naish, 1982, 1984). There are considerable differences between these systems which will be demonstrated later. Some have attempted more than the provision of an effective coroutining facility. EPILOG and Two-Level PROLOG have tried to provide general control facilities that may be specified in logic while METALOG has realised an extremely useful general purpose tool for constructing logic interpreters as well as allowing localised control within clauses.

All of the above systems have had to find methods of specifying which predicate calls will be suspended and when. The recent work originated by Ullman (1984) and developed further by Sagiv and Ullman (1984) and Ullman and Van Gelder (1985) has focussed on this problem. Their approach, based on data flow analysis, proposes the notion of capture rules as a means of planning query evaluation.

2.2. Intelligent Backtracking
A second major stream of research has concentrated on improving the techniques used to traverse the search space of the program. The process of logic program execution can be viewed as a derivation of various states in a deduction (Bruynooghe and Pereira, 1984). For each forward derivation step, one of the operators applicable to the current state is used to derive the next state. This forward execution is repeated until either a solution state is reached and success reported or the set of unused operators applicable to the current state is empty; that is, failure is discovered. At this point, the search backtracks. The current state is abandoned, its predecessor is reinstated as the current state and forward execution recommences.

It is clear that this approach does not exploit any relationships between successive states. After encountering a failed state, the execution simply returns to the previous state. However, continuing from this point will not necessarily be successful: the execution may be performing an exhaustive search over a subspace which is irrelevant to the failure. The goal of intelligent backtracking is to deduce the source of failure in a particular state and to unwind the execution back to that point.

There are two major approaches to intelligent backtracking documented in the literature, one due to Cox (1977, 1978) and the other to Bruynooghe (1981). Considerable differences exist between these approaches and they will be discussed later.

Both of these methods rely on a dynamic analysis of unification conflicts, that is, conflicts are determined at run-time. An alternative is to consider static analysis of a program at compile time to determine which conflicts are likely to arise when the program is executed. Using this strategy, a program may be compiled into efficient code which incorporates a form of intelligent backtracking. Chang and Despain (1985) have investigated this approach. The major problem appears to be that there is very little information available at compile time since the query is...
unknown. This leads to the method finding only quite obvious conflicts thus missing out on the substantial savings possible with more radical backtracking.

In addition to the work of the groups above, many of the PROLOG systems mentioned in the earlier section on coroutining have implemented forms of intelligent backtracking, for example, IC-PROLOG, CML (Gallaire and Lasserre, 1979) and METALOG.

2.3. Higher Level Control Specification
Some attempts have been made to improve on coroutined execution of programs and/or provision of intelligent backtracking facilities, by devising generalised meta-languages for controlling the derivation process. Using these powerful meta-languages, the programmer is able to specify the behaviour of the interpreter, in some cases either globally or locally. Examples of such systems are CML, METALOG, EPILOG and Two-Level PROLOG.

2.4. Program Transformation
Rather than providing explicit mechanisms for controlling execution of logic programs, it is possible to adopt quite a different approach and consider transforming given programs into equivalent programs that possess considerably improved efficiency and termination characteristics. Various groups have studied the transformation of logic programs. A summary may be found in Parths and Steinbruggen (1981). The motivation behind this work has been one of enhancing the efficiency of programs rather than investigating the usefulness of these transformation techniques in deriving terminating programs. The most interesting work is the unfold/fold methodology developed by Darlington (1981) and further pursued by Tamaki and Sato (1984).

2.5. Other Approaches
Amongst other research undertaken into the provision of better control facilities was Brough's investigation (1979) of the detection of run-time loops in logic programs. His techniques were based on an analysis of differences between subgoals and program assertions, and incorporated into a Horn clause problem-solving system developed at Imperial College.

Another technique for obtaining more refined control over program execution was suggested by Vasak and Potter (1985). The method is based on the specification of clausal annotations which are simply logical constraints on bindings. For each goal, there is an notion of an environment which is a set of annotations constraining the solutions to the goal. This allows the exclusion of infinite subcomputations whilst preserving a program's declarative semantics. Vasak (1986) describes the hybridisation of these annotations with a coroutining strategy to obtain a very powerful control mechanism.

Considerable research has also been undertaken into the provision of parallelism in logic languages. Some of the languages to emerge from these investigations have been PARLOG (Clark and Gregory 1983), Concurrent PROLOG (Shapiro, 1984) and EPILOG (Wise, 1984) (this is not related to Porto's EPILOG). Whilst they have indisputably provided new powerful and flexible control techniques, they have also brought with them their own problems. This paper will confine itself to surveying purely sequential control mechanisms.

3. Detailed Analysis of Existing Approaches
Having presented an overview of the various approaches, the following section shall examine them more closely and evaluate their usefulness in solving the control problems afflicting logic programming.

3.1. IC-PROLOG
The first PROLOG system to actually propose and implement coroutining as a general purpose control facility was IC-PROLOG which was developed at Imperial College by Clark and McCabe (1980). With quite simple syntactic additions to the basic Horn clause syntax, the programmer could specify a computation rule that would result in a coroutining interaction between subcomputations that is triggered by the flow of data through shared variables. This was done by attaching the annotations, and ?, to terms in the heads of program clauses. These annotations specify extra constraints that must be satisfied by the call to a clause before that clause may be used. Intuitively, the annotated term t? indicates that t must be used as an input template. More formally, it signals the requirement that each of the variables in t must have matched against a non-variable term. The annotation signals the opposite use. A term t, annotated as t, must be used as an output template. Formally, this requires t to be uninstantiated except for a variable renaming. The unification process will match the first clause matching the given modes. If none match, an error is signalled. This allows a relation computing the grandparent x of a person y to be expressed as:

\[
\text{grandparent}(x?, z): - \\
\text{parent}(x, y), \\
\text{parent}(y, z), \\
\text{grandparent}(x, z?): - \\
\text{parent}(y, z), \\
\text{parent}(x, y).
\]

The intermediary parent is to be found by looking up the children of the grandparent when the grandparent is known, and by looking up the parents of the grandchild when the grandchild is given and the grandparent is to be found.

Apart from control annotations on the clause heads in IC-PROLOG, it is also possible to attach ? or to any variable appearing in the clause body. Consider
with $A_i$ (for some $1 \leq i \leq n$) possessing a variable $v$ that is annotated. The effect of the annotation is to make $v$ a data channel for $A_i$. If the ? annotation is used, $A_i$ is an eager consumer of the data that will be sent down the channel by the evaluation of $A_1, \ldots, A_{i-1}$. If the annotation is employed, $A_i$ is a lazy producer of the data that it sends down the channel. These are implemented by coroutining the execution of the clause literals. An eager consumer will continue executing as soon as the partial result is available in the data channel. It will consume it and then suspend and wait for the next result. A lazy producer on the other hand suspends as soon as it generates a partial result into a data channel.

The mechanism provided by lazy producers and eager consumers provides a powerful control facility. The basic problems with IC-PROLOG were that in spite of the potential power of the system, it still lacked a certain element of control at times, for example, the ability to specify which literal is to be suspended. Furthermore, the annotations of IC-PROLOG, despite their fairly simple interpretations, did obscure the declarative semantics of the program. Understanding a moderately complex IC-PROLOG program is a complicated task and reduces to simulating, in some informal way, the operational semantics of the program. The ability to annotate clause heads and thus affect the selection of different clauses based on different modes of invocation is simply an alternative to expressing a number of variants of the same clause spiced with non-logical control primitives such as var, nonvar and cut, as was suggested before. However, this suffers from the problem of markedly increasing the space complexity of the program.

3.2. PROLOG-II

One of the major benefits of co-routining the execution of subgoals is to avoid the creation of failure branches when there is insufficient information (if there no/few bindings from other subgoals, it is generally preferable to pursue another goal with a smaller search space). In PROLOG-II, Colmerauer (1982) proposed an alternative method of delaying calls by employing a built-in predicate named geler (this translates into English as freeze). The freeze is used to delay a subgoal until a particular variable is bound to a non-variable. For example, consider the following program that determines whether a list is ordered.

\[
\text{ordered}([\]).
\]
\[
\text{ordered}(\{A\}).
\]
\[
\text{ordered}(\{A, B | C\}) :-
\]
\[
A \leq B, \\
\text{freeze}(C, \text{ordered}(\{B | C\})).
\]

Freeze has the effect of delaying, until $C$ is bound to a non-variable term, the call to $\text{ordered}(\{B | C\})$.

In some cases, it is possible to write more efficient programs with freeze than with IC-PROLOG’s eager consumers. However, there are several disadvantages to the system. It lacks sufficient control which may lead to infinite computations in certain cases where clause heads have more than one level of functor. Naish (1985) gives an example. Also, because freeze only waits for one variable, it is less useful in more complicated examples.

3.3. EPILOG

Spurred by the attempts of IC-PROLOG and PROLOG-II to extend standard PROLOG in a suitable manner, Porto (1982) developed the language EPILOG. The motivation for designing EPILOG was the realisation that these systems were not really adequate for reasons ranging from insufficient power through inefficiency to lack of clarity. The main idea behind EPILOG is to provide the logic programmer with several connectives for conjunction, all declaratively equivalent to the logical AND but procedurally distinct, defining different time constraints on the execution of the goals they apply to. Porto takes the view that the problem of controlling the execution of a logic program should be regarded as the problem of enforcing some suitable partial order among production and consumption of some instantiations by certain goals. He criticises IC-PROLOG for assuming that there are pre-determined production/consumption relationships since, as he correctly argues, these may change with different modes of invocation and may thus only be correctly determined at run-time. The use of freeze in PROLOG-II amounts to predefining eager consumers and thus also lacks generality.

What EPILOG does is to provide a more high level specification of control. For example, coroutining between two literals $L_1$ and $L_2$ is represented simply as $L_1 \setminus L_2$, with no reference to specific producer/consumer variables. EPILOG’s metalogical control can also be used to obtain enhancements in efficiency.

EPILOG recognised some of the problems inherent in earlier attempts at coroutining in particular, and control specification in general, and proposed a very interesting concept of logic control. Its major contribution was its recognition that logic and control ought to be separated in order to retain the declarative semantics of logic programs.

3.4. Two-Level PROLOG

Based on his experiences with EPILOG, Porto designed a new language named Two-Level PROLOG (Porto, 1984). Substantially a refinement and to some degree a simplification of EPILOG, Two-Level PROLOG questions the adequacy of implication and conjunction as the sole logical connectives, and advocates enriching the language of Horn clause logic with many other connectives and operators. These have fairly simple interpretations in standard PROLOG.
between its left and right operands solely in terms of infix operator ‘/’ denoting coroutined execution.

Porto showed that it is possible to define the his extended set of primitive connectives:

- used extra-logical expressions into higher level opera­
cation of control for logic programs. It is very
argue that the primitives selected do really have
higher level of control than is seen in traditional Horn

3.5. METALOG

Motivated by principles similar to Porto's EPILOG and Two-Level PROLOG was the language META­
log control information, specified in another logic program, is viewed as meta-knowledge. The meta-level expression of control information gives a programmer power to intervene in the deduction process, to define his stra­
tegies and to specify his own interpreter. This is done by adding meta-clauses to the logic language. These are statements involving various primitives used to affect the selection of literals, clauses or even implement a form of intelligent backtracking by being able to choose backtrack points.

Because METALOG can obtain such fine control over a deduction and specify global control strategies, it is well-suited to applications such as expert systems construction or the building of specialised problem-solving systems. The main disadvantage of METALOG is that its meta-clauses are typically very compi­
cated rules which tend to promote the operational semantics of the underlying programs over their declarative semantics. Another drawback is that this approach may be computationally expensive. It appears to be a very useful tool for specifying global control of deduction (that is, specifying interpreters) but less effective for local control.

3.6. CML

CML (abbreviation for control meta-language) is a meta-language for controlling the derivation process in Horn clause logic programs (Gallaire and Lasserre, 1979). The aims of CML are essentially the same as those of METALOG in that it seeks to provide a metalogical level of control for logic programs. Rather than mixing up control directives with the pro­gram logic, both CML and METALOG separate the two components and devise a language for specifying the logic component. However, there is a distinctive difference between CML and METALOG: whilst METALOG allows control to be specified in terms of the problem domain, CML reasons about control only in terms of interpreter behaviour. Control information that may be specified using CML's meta-rules include which clause should be selected first when more than one clause is applicable, or which literal should be resolved upon. An example of a clause selection rule that can be specified with CML is that clauses are to be selected according to their number of literals, the shorter clauses first. In selecting which literal should be solved, CML provides control over many facets of the execution by allowing the relative selection priorities of literals to be expressed, providing amongst other features a limit on the depth of recursion and allowing the specification of backtrack points.

As was the case with METALOG, this approach makes the language more suited to specialised applications rather than general purpose programming. Although the system is quite powerful, the ability to express control only in terms of interpreter behaviour is limiting. Furthermore, the meta-rules tend to be fairly operational specifications because they deal with details of the underlying inference mechanism.

3.7. MU-PROLOG

Another PROLOG system to have implemented a coroutining facility is MU-PROLOG developed at Melbourne University (Naish, 1982, 1984). MUPROLOG allows each user-defined predicate to have its own control information through the specification.
of a set of \textit{wait declarations} for that predicate. In order to explain these wait declarations, it is essential to define the auxiliary concept of \textit{construction} of arguments. A term \( t \) is constructed if either any variable in \( t \) becomes bound to a non-variable term, or any variable in \( t \) becomes bound to a structure (possibly trivial) containing another variable in \( t \). Consider the \texttt{append} example. Suitable wait declarations for this program are

\begin{verbatim}
?- wait append(1, 1, 0).
?- wait append(0, 0, 1).
\end{verbatim}

The effect of the wait declarations is to slightly restrict the way in which \texttt{append} may be called. If a call to \texttt{append} constructs either of the first two arguments, the third may not be constructed. Should this condition be violated, the call will delay. A 1 in a wait declaration means that the corresponding argument in the call may be constructed. A 0 indicates that the corresponding argument must not be constructed. Multiple wait declarations simply provide alternative ways of invoking predicates.

Using wait declarations to specify when goals should delay in a coroutined execution of a query has some advantages over the other approaches described. Apart from the fact that wait declarations have a fairly intuitive interpretation, the ability to specify multiple wait declarations for a predicate gives a much more flexible control mechanism than with the producer/consumer annotations of IC-PROLOG, for example. Furthermore, the declarations are separate statements and do not cloud the declarative semantics of the original program.

What makes this approach so attractive, though, is that there are techniques for deriving wait declarations automatically. Naish (1985) outlines such an algorithm for generating wait declarations.

An alternative to wait declarations called \textit{when declarations} was proposed by Naish (1985). As well as increasing expressiveness, the when declarations further improve the semantics by replacing the 0's and 1's with symbolic variables. For example,

\begin{verbatim}
?- merge(A, B, C) when A and B or C.
\end{verbatim}

indicates that \texttt{merge} may be called if the first two arguments, or the last argument are non-variables.

Despite its apparent success at solving many of the control problems occurring in logic programming, MU-PROLOG does suffer some notable disadvantages. In particular, wait declarations often tend to be too conservative and delay a call that is quite safe to continue with. This may produce deadlocks, that is, all the current goals may be suspended waiting for each other. Replacing wait declarations by when declarations should lead to better performance by providing a finer level of control. Another problem with a coroutining mechanism based on wait declarations is that the ordering of literals within a body is significant. In some cases, one ordering may cause a goal to deadlock while another may yield a quite satisfactory performance. A detailed discussion revealing some of the problems with MU-PROLOG can be found in Vasak (1986).

PROLOG M described by Babb (1983) implements a coroutining mechanism based on a similar delaying heuristic to that employed by MU-PROLOG. All in-built system predicates delay when they have an infinite number of solutions. The basic difference is that delays in PROLOG M are implemented in predefined system predicates such as equality; delays in calling user-defined predicates cannot be specified directly.

3.8. Capture Rules

Capture rules were introduced by Ullman (1984) as a method for planning query evaluation in function free deductive databases with recursively defined rules. It is claimed by Ullman that a number of strategies suggested previously can be effectively subsumed by the theory of capture rules. Central to the concept is the construction of a rule/goal graph that expresses variable binding relationships between instances of rules and subgoals, these instances being parameterised by tuples denoting which variables are free and which are bound. A capture rule is a law that states that under which circumstances certain nodes in the rule/goal graph may be \textit{captured} provided some other nodes have already been captured. This yields a goal selection strategy.

Unfortunately, several restrictions were placed on programs that this theory could cover, the most constraining being that programs do not have function symbols. Sagiv and Ullman (1984) investigated removing this restriction while Ullman and Van Gelder (1985) sought to extend the theory by analysing inequalities in goal argument sizes.

Captures rules are closely related to the work of Naish on wait declarations and their automatic generation. While capture rules are based on a sounder theoretical foundation than Naish's work, they are actually less powerful in certain cases (see Sagiv and Ullman, 1984).

The theory of capture rules can also be viewed as an extension of database query optimisation techniques that have been the subject of considerable study. For example, see Smith and Chang (1975).

3.9. Intelligent Backtracking: a Comparative Analysis

As was outlined earlier, there are two main approaches to intelligent backtracking, that pioneered by Cox and that by Bruynooghe. Considerable confusion exists, even amongst these two groups as to how these methods differ. This is highlighted by Chen, Lassez and Port (1984) who succeed in providing a theoretical interpretation for the two approaches.
The best way to explain the difference is to refer to a small example due to Bruynooghe and Pereira (1984):

\[
\begin{align*}
p(a). \\
p(X). \\
s(b). \\
r(U, U).
\end{align*}
\]

with the following query

\[
p(X), q(Y), r(X, Y)? \quad (C)
\]

Ferguson (1977) introduced a representation for proof trees that is particularly useful in considering intelligent backtracking and has been widely used by Bruynooghe. Ferguson’s deduction tree is simply a potential proof tree for a goal. Each node $N$ in the tree is divided into an upper and lower half. The upper half represents a goal to be solved while the lower denotes the head of some clause that will unify with that goal. For every goal $G$ in the body of the particular clause selected, there will be a subtree emanating from node $N$ that represents a deduction tree for $G$. Thus any lower half of a node together with all the upper halves of nodes adjacent are a copy of a clause. Each leaf node of a deduction tree can either be closed, if the upper half can match a unit clause and thus has a lower half, or open if not. Any deduction tree with an open node is unsolvable. Otherwise, if the proposed unifications at all nodes of a closed deduction tree are consistent, the deduction tree is a proof tree. A deduction fails if it is inconsistent or unsolvable.

Using the above representation, the deduction tree for the above query is shown in Figure 2. In order for the tree to be a valid proof tree, the four unifications between the upper and lower terms of each node must be consistent. Notice that this is not so here. The Bruynooghe approach of analysing failure is based on the notion of finding minimal inconsistent deduction trees, that is, minimal subtrees such that unification is impossible. By identifying such trees, it is possible to remove that part of the derivation that caused unification conflict. In the above example, there is a single minimal inconsistent deduction tree corresponding to the entire tree.

The method of Cox (1977, 1978) is based on the complementary notion of maximal consistent deduction trees, that is, maximal subtrees such that unification is possible. Each of them is obtained by removing a derivation step from a reduced conflict set (Matwin and Pietrzykowski, 1982). Referring to the above example, these are

- the tree consisting of nodes $p, q$ and $s$
- the tree consisting of nodes $q, s$ and $r$
- the tree consisting of nodes $p, q$ and $r$

Each of these deduction trees now serves as a starting point for the continuation of the search process.

A basic difference discernable from the consideration of the above example is that the Bruynooghe method has no effect on the SLD-trees computed whereas Cox’s method actively determines them. This suggests that the latter approach is really a hybrid strategy of coroutining and intelligent backtracking since it has effects on the shape of the space actually being searched. Bruynooghe’s technique is a purer form of intelligent backtracking whose role is to determine the optimal method in traversing given search spaces.

Chen, Lassez and Port (1984) develop the theory of maximal unifiable subsets and minimal non-unifiable subsets which gives a solid formal foundation to these approaches. They also apply this theory to breadth-first search and discuss the role of intelligent backtracking in improving the behaviour of the breadth-first computation rule.

Tønisson (1986a, 1986b) points out that in actually implementing his technique, Bruynooghe has had to weaken his intelligent backtracking system since he could not compute the entire conflict set efficiently. In particular, difficulties arise with aliasing of variables. Tønisson addresses the problem of calculating complete conflict sets efficiently, and also demonstrates that an intelligent backtracking system incorporating this extended method is quasi-complete with respect to finite failure.

The primary disadvantage of intelligent backtracking methods is that the termination properties of programs run under such systems are not clear. If a programmer wishes to determine whether a particular query will terminate for a given program, he is forced to undertake an elaborate operational analysis of his program. There are also doubts whether intelligent backtracking systems with considerable power can be implemented efficiently. To a certain extent, the very recent work of Tønisson appears to indicate that such systems may be efficiently realisable, perhaps in hardware.
3.10. Unfold/Fold Transformations

The work of Darlington (1981) and later Tamaki and Sato (1984) on program transformation has been included since it provides a means of modifying control by altering the program itself. This is achieved by a series of nested unfold, simplify and fold operations that effectively evaluate the program statically. Unfolding refers to the replacement of a predicate with a sequence of goals by treating the logic program as a set of textual rewriting rules. Where a sequence of goals is an instance of a right-hand side of a rule, it is possible to fold these goals to the appropriate of goals is an instance of a right-hand side of a rule, as a set of textual rewriting rules. Where a sequence with a sequence of goals by treating the logic program

Unfolding refers to the replacement of a predicate with a sequence of goals by treating the logic program as a set of textual rewriting rules. Where a sequence of goals is an instance of a right-hand side of a rule, it is possible to fold these goals to the appropriate instantiation of the head of this rule. For example, the append3 predicate could be transformed into the following program:

```
dappend([], A, B, C): -
aappend(A, B, C).
daappend([X | A], B, C, [X | D]): -
daappend(A, B, C, D).
```

In this form, the program would not only terminate correctly for queries such as (A.1) and (A.2), but would also do so in linear time. Darlington’s semi-automatic programming aid is capable of effecting the above transformation with minimal user intervention. A benefit of this approach is that the intended semantics of the initial program are unaltered by the transformation. However, it is not clear whether this transformation technique would be of much use for examples of non-trivial complexity.

3.11. Annotations

Vasak and Potter (1985) proposed an alternative computational model based on logical annotations for predicates. These annotations, which are simply theorems about terms that satisfy a particular clause, can be used to generate run-time environments within which it is possible to exclude some infinite computations whilst using a depth-first computation rule. The major advantage of such a model is that it is possible to write conceptually transparent programs which otherwise would not have terminated under a naive control mechanism. Annotations are expressions of the form

\[ f(A_1, \ldots, A_m) \leq g(B_1, \ldots, B_n) \]

where \( f \) is a left-admissible annotation function and \( g \) a right-admissible annotation function (both map program terms into the integers); \( A_i, B_j \) are terms involving variables in the clause.

Both equality and inequality constraints may be used. Vasak and Potter define a partial ordering on terms based on the primitive concept of instantiation. A left-admissible annotation function is defined to be monotonically increasing under this partial order whereas a right-admissible annotation function is monotonically decreasing. These annotation functions are generalised metrics over the abstract data type used by the clause. For lists, \( \text{length} \) will suffice while for trees \( \text{nodes} \) or \( \text{height} \) could be used. The inorder example introduced earlier can be written as:

```
inorder([], []).
inorder((L, N, R), I): -
inorder(L, LI),
inorder(R, RI),
append(LI, [N | I], I)
with nodes((L, N, R)) = length(I).
```

Although the primitive annotation concept is quite inefficient, Vasak (1986) presents an implementation by embedding them in a coroutined system. This hybrid strategy benefits from the power of annotations while retaining the efficiency of coroutining.

4. Summary

All of the various approaches providing more sophisticated control for logic programs that have been discussed in this survey go some distance toward solving problems highlighted by the earlier examples. Some methods appear stronger than others. Two of the most common problems are that

— the declarative semantics of the program are sacrificed in order to obtain more finely grained control;
— some strategies are too ad hoc to allow programmers to reason effectively about program behaviour, particularly termination.

There have been some encouraging developments. In particular, investigation of intelligent backtracking under a depth-first computation rule has shown that it is potentially very powerful. Some other approaches have concentrated on maintaining the declarative semantics and tended to relegate efficiency concerns. Currently, we appear to have the option of either an efficient non-declarative control mechanism or a less efficient but more logical one. Considerable investigation is still required in this area to capture the best of both worlds.

Acknowledgements

The author would like to thank the referees for their constructive criticisms which have undoubtedly improved the presentation of this material.

References


The Australian Computer Journal, Vol. 18, No. 3, August 1986
Control Facilities in Logic Programming


Biographical Note

Tom Vasak received his B.Sc. degree in Computer Science from the University of New South Wales in 1983. Supported by an Australian Computer Research Board Scholarship, he completed his Ph.D. dissertation earlier this year in the field of logic programming at the same university. Since then he has joined Ausonics Pty Ltd in Sydney, a division of the Australian biomedical company Nucleus. Apart from logic programming, his interests also include expert systems, programming language design, programming environments and methodologies.

This is an important book for computer professionals, because computer professionals so rarely see the computer systems they build as they really are, but rather as they know they can be. Vallee, on the other hand, tends to see them as the end user sees them. For example, an end-user will ask why the power cord can’t be rolled up inside the microcomputer like a vacuum cleaner. Do you know why it isn’t? Or why the on/off switch is not labelled as such (it wasn’t on any of the five micros I just checked!).

I am sure every reader will, at one time or another, have received a letter addressed something like ‘Dear Mr Dr’. What was your reaction to this? Perhaps you smiled ruefully to yourself about the stupidity of so many computer systems in operation in our world. It is very likely that you gave it no further though, other than perhaps to resolve that no computer system you designed would ever be so simplistic.

The computer industry certainly is rife with such ‘computer errors’, and we have become quite blase about them. Accordingly, Jacques Vallee takes the time to ask if there is any more sinister implication to this and other similar seemingly innocuous computer ‘funnies’. It is not just that it is very hard to see such ‘computer mistakes’ as genuine progress (computers do represent one type of ‘progress’ in its highest form); nor that it brings a certain amount of ridicule onto the computer profession. Rather it is because it is symptomatic of a corrupt technology—how can we solve the world’s problems, when we cannot even solve our own? These are the signs of an unthinking and blind application of technology which will end up destroying the fabric of our civilisation, he concludes. These are very powerful thoughts. But then, this is a very powerful book. Its title and its general approach seem quite harmless, full as it is of amusing anecdotes from Vallee’s own experience. Its main impact is made almost in passing. This increases the force of the point being made, namely that, as computer professionals, we are changing the nature of society in ways whose impact we cannot predict, much less control.

Vallee starts with an incident which took place in France in 1979, in which a young man was shot by police acting on false information which was supplied by a computer. For the remainder of the book, Vallee seems to be asking whether he and others like him in the computer business should accept any responsibility for this and similar incidents. He does this by drawing on a series of incidents in his own experience.

In chapter 1, he looks briefly at the origins of the computer industry, reflecting on his earliest days with an IBM 650. In chapter 2, he reviews several cases of oversell in the industry, including the early expectations held for computer-aided instruction, management information systems, and machine translation. In chapter 3, he looks more in greater depth at information retrieval, pointing out that, at the very least, this is a misnomer. One cannot store information, only data - ‘information resides mostly in the questions [asked]’ of a data base. He also indicates that information systems are, first and foremost, social systems, and that computer scientists almost always ignore this fact.

By chapter 4, Vallee has concluded that technology in general is out of control, and that computers represent a leading example of that phenomenon. He also dwells on the problems of predicting technological innovation, and points out that a fundamental difference between the Old and New Worlds, and the reason that the New (America) has been making faster progress in computing, is that the Old tends not to move forward until it has a philosophical base from which to do so, whereas the New takes action first, seeking to explain it afterwards. He does not, however, come to the obvious conclusion that we can blame all our ills on the New World philosophy.

In chapter 5, we are introduced to an example of New World philosophy - the counter-culture hippies of the sixties and seventies in California. Their influence, he concludes, resulted in attempts to build a New Society using computer communications technologies. If you have ever read James Martin’s The Wired Society, you will recognise there an example of the aspirations characteristic of this breed of computer person. Vallee believes their modern-day descendents, the attempts to create office automation, are doomed because they are built to screen their owners from the real world, rather than to facilitate interaction.

From this perspective, the next logical step is taken in chapter 6, where Vallee tries to demonstrate that computer people deliberately make their systems difficult for ordinary people to understand. He points out that, when constructing systems, we can change either the people or the system; the fact that the computer industry now has in high power to start changing people has clearly dangerous implications. After a seeming digression in chapter 7 to talk about hackers and the strange social uses to which Bulletin Boards are being put, he returns to this theme in chapter 8, concluding that the prospects for a crisis are increasing. For example, a good information system, he claims, is also a good dis-information system - i.e. one that can be used effectively for entering and promulgating false information.

However, in chapter 9 Vallee suggests that the very technology that is precipitating human crises may be the technology that will ultimately solve them. This theme is further developed in chapter 10, where he introduces the concept of an electronic grapevine, an informal computer network combined with human resources. The solution to the dangers of too much reliance on computer systems lies in consulting the computer database, but then reviewing its decisions in a human conference. Perhaps here, too, is the reason behind the main title of the book, the choice of which otherwise escapes me.

Vallee concludes the book in chapter 11 with an allegorical conversation with a Washington funding agency representative. Perhaps the Government is aware of what is going on, and perhaps they are just stringing us computer scientists along. Or do they all have as much potency as a Whispy breeze?

Overall, a really thought-provoking book, which is highly recommended reading for all computer professionals. You may decide that Vallee is talking nonsense; but if you come to that conclusion, then you have a lot of explaining to do! More likely, you will have your perceptions sharpened, and you may be a little more circumspect in the next computer you sell, application you build, or lectures you deliver on the merits of computing. I believe the book would make excellent background material to accompany a series of lectures on the social implications of computers, especially if the lectures are accompanied by a series of tutorials (as they must, if the subject is to be done justice to).

Alex Reid
W. A. Regional Computing Centre


This Prime Ministerial advisory group has assessed ‘electronic funds transfers and their potential effects on Australian society’. Its report commences with 20 pages of executive summary and background to the study. Four chapters then provide a succinct introduction to recent developments in finance and retailing, to EFTPOS systems, and to developments overseas and in Australia (pp. 21-72). The misleadingly titled central chapter deals with a range of consumer protection concerns. One conclusion is that operators of EFTPOS systems are providing inadequate information to the public about the terms of contract, and that ‘the rights and responsibilities of card-users were unclear, difficult to obtain and inconsistent’ (pp. 4-5, 75-6). Moreover, EFTPOS system operators are seeking to avoid liability for system malfunction and error, in ways not available to banks under the conventional cheque-based payment system (pp. 5-6, 76-78). ‘ASTEC believes that there is a need for

The Australian Computer Journal, Vol. 18, No. 3, August 1986
This book consists of papers presented at the First International papers even if it meant lower quality elsewhere. This analysis is rather trite, relying heavily on the A.L.R.C.'s already dated Report (1976-83): EFT offers ' . . . the opportunity to trace information about individuals with or without their consent' (p. 86), and 'the combination of computers, telecommunications and information processing equipment was capable of producing consequences not seen before' (p. 88). However the Report does highlight the existence of Commonwealth legislation (e.g. the Social Security Act) empowering uncontrolled access to information held in private sector financial systems (p. 89). It concludes that 'the advent of EFT and EFTPOS makes it even more certain that there is inadequate protection for the recording, storage and use of personal information' (p. 90), and that 'it is urgent that such protection be offered now to personal information held in all data bases, both public and private sector' (p. 91).

The report's low level of technical appreciation reflects the absence of industry involvement. The Australian Computer Society is not represented on either ASTEC itself or the working group, and only one of those 26 people appears to be from the computer industry. The Australian Information Industry Association (the industry lobby group, dominated by the local subsidiaries of hardware suppliers) met with the ASTEC working group, but the A.C.S. did not. The Society has made frequent submissions to Government on the privacy issue since it arose in the early 1970s, and appears to be dispirited by the total absence of action to date.

The report also deals with industrial relations issues (pp. 10-12 and 97-107). Less predictably, it raises the need for public participation (pp. 15-6 and 115-6). This is after all one of the first applications of information technology which impinges quite directly on the general public.

This is a useful report on an important issue, refreshingly free of misunderstanding, and drawing many strings together, but lacking deep insight.

Roger Clarke
Australian National University


This book is a collection of papers from the First International Workshop held in September 1984. In contrast to many workshop/conference proceedings this is well bound, on good quality paper, and all contributions are legible. Despite this quality, the book is fairly slim—neither panel discussions nor four of the five invited papers are included, although the promotional sheet claims they are. Personally, I would have preferred the extra papers even if it meant lower quality elsewhere.

Running underneath this Workshop is of course Prolog, as a language to implement logic-based systems. A few papers appear merely to describe systems written in Prolog, much as others might be in Pascal or Assembler or whatever, but many of the seventeen papers included are involved in linguistic issues coupled to formal bases of programming.

An introductory paper by Dahl gives a rapid survey of the logic approach to NLU. Basically this describes abstract versions of logic grammars which can be 'compiled' into logic programs. Papers by Miyoshi et al. (from ICOT), Uehara et al., and Sabatier continue this abstraction approach although in different directions. Some of the 'compilation' issues are addressed by Matsumoto et al. and Sabatier wherein logic grammars are targeted to various search mechanisms.

Many papers deal with general AI and linguistic problems: Uehara et al. view parsing via Actors; Miyoshi et al. base theirs on object orientated parsing; Frey compares Lexical Function Grammars to the logical based Extra-Position Grammars; Boyer generates sentences from semantic networks; Schwind analyses dialogue using frames. All of these attempt to ground their work to predicate logic.

Despite the grounding, this is not a totally cohesive set of papers. Much argument is visible in this still developing field, particularly from papers by Daladier and Kay.

Certainly libraries should acquire this book. It could also find a place as a high-level introduction to logic and NLU for those who might wish to venture into this field.

J. Newmarch
Canberra College of Advanced Education


Can Artificial Intelligence be applied to commercial business? In an effort to explore the level to which AI can be applied to Industry, MIT's Industrial Liaison Program brought together four groups of people for a colloquium in this area. These four groups represent the following perspectives:

— academic
— hard core financial
— industrial research and development
— solutions orientated.

This book consists of papers presented at the colloquium, and transcripts of the discussion sessions. The book focuses on Expert Systems and Robotics, but also includes various other areas in lesser detail, for example natural language understanding. Authors of the various papers are Frederick Adler, James Baker, Michael Brady, John Brown, Alan Kay, Arnold Kraft, Marvin Minsky, Harry Popl Jr, Karen Prendergast, Charles Rich, Paul Russo, Roger Schank, Philippe Villers, and Patrick Winston.

The technical research papers contain little information not previously published. Other papers such as Adler's 'An Investment Opportunity?' and Janeway's 'Financing the Future' analyse the business aspects of AI. Several other papers discuss the philosophy of AI. The discussion segments make up almost 20% of the book. They are possibly the most interesting aspect, containing histories of various projects and the panelists' point of view about directions, possibilities and problems. These discussions clearly show an MIT bias, for example in the PROLOG/LISP argument. However the discussions contain interesting examples of what can be done now, and possible means of improving the transfer of AI technology thus making better use of the AI techniques. The lack of qualified AI staff is considered the major worry, together with the risk that basic AI research is suffering as the researchers attempt to solve real business problems. This is a long term fear, as it seems to take about 15 years for basic research to be applied in any general manner to business.

This book should be of greatest interest to organisations wanting to know whether AI is applicable to business. It also discusses the pitfalls involved in the application of these techniques. It is of less appeal to AI researchers who will find they already know most of the technical content.

Rea Fodell
Science Computing


This is a status report at the end of the first pilot year of the ten year ESPRIT programme. The report comprises an overall summary of this ambitious European project, together with thirty seven papers describing the individual status of the original projects. At first glance the value of working papers published in the raw, almost two years after the September 1984 workshop, appears limited. There are, however, many facets of the papers that make...
extremely relevant reading for Australia today. The first is that there is a well defined strategic objective: 'The achievement of technical parity with, if not superiority over, world competitors within 10 years'.

The second is the chosen development realm, the pre-competitive stages that lie between research and final product production. This encourages the almost mandatory grouping of industrial, research and academic contributors within each project.

As a result the papers have the unique flavour of a blend of pragmatism and academic rigour, quite unlike the normal range of conference proceedings. In many cases the projects have had to go back to the basics of their area and construct an overall viable approach that can direct the later implementation stages. A typical example is the paper by Potton and Klittich, on Integrated Microelectronics Subsystems for Plant Automation, that identifies requirements for VLSI designs and parallel computers for the robotics area.

This paper is in one of the three application orientated areas, Computer Integrated Manufacturing; the other two areas being Advanced Information Processing and Office Systems. These areas are complemented by two enabling areas, Advanced Microelectronics and Software Technology.

Within the thirty seven papers there are a number that simply enumerate issues, and one that admits that the task has not been fulfilled. By and large, however, each paper represents a solid start to the EEC approved initial five year segment of the ten year plan. As such, each paper is worthwhile as its placement in the pre-competitive stage, and the project team composition means that it addresses some of the most difficult aspects of product development, such as turning good research ideas into marketable products.

One important area that has been included retrospectively is that of common communication facilities and documentation standards within the ESPRIT community, a similar discovery to the Alvey programme. Two papers cover information exchange systems in a special section of the book and a number of papers in the Office Systems section tackle the document standards issue. Future cooperative projects will hopefully regard these two areas as intrinsically of high priority.

The most difficult question to answer is the relevance of the ESPRIT approach to the current Australian environment. In some ways the ESPRIT program's task is simpler due to market size and industrial base, in other ways more complex due to a variety of communication authorities.

On balance there is a lot to learn from the successful ESPRIT approach overall, and from the individual approaches made in the projects, making this a worthwhile book to at least borrow from the library.

Fergus O'Brien
Computer Power, Sydney


This is an account of the Proceedings of the Second IFIP International Conference on Computer Security, held in Toronto, Canada, 10-12 September, 1984. It is a very easy book to find your way around, as the invited papers are presented at the front (and can mainly be ignored) and the accepted papers are grouped under their various subject headings. As always, the set of papers is rather like the curate's egg—very good in parts. The papers that took my fancy are:

I. L. Auerbach, Professional Responsibility for Information Privacy
A clear statement of the privacy issues, and a charge to us, as DP professionals to carry out our responsibilities.

D. B. Parker, Safeguard Selection Principles
A statement of twenty principles for providing security of information, and two appendices to help in assessing proposed safeguards.

C. R. Symons and J. A. Schweitzer, A Proposal for an Automated Logical Access Control Standard
The appendix to this paper contains a set of basic requirements for a logical access package; it should be useful in assessing the offerings in the market.

R. L. Brown, Computer System Access Control Using Passwords
A comprehensive paper on the selection and storage of passwords, and a summary of the effectiveness of the various methods against brute-force attack.

M. I. Svanks, Integrity Analysis: a Methodology for EDP Audit and Data Quality Assurance
This seems to be a new approach to verifying the consistency of records in a data base by checking the data against an ordered set of constraints.

J. Miguel, A Composite Cost/Benefit/Risk Analysis Methodology
A simple rating method using consequences, exposure, and likelihood of occurrence of threats, and cost and degree of protection provided by counter-measures.

R. Wrede, The SBA Method; a Method for Testing Vulnerability
An eight-step method developed in Sweden for assessing vulnerability to system interruptions, unauthorised data disclosure and poor data quality, with suggestions for setting up action plans to improve matters.

A. Mustonen, Security Threats and Planning of Computer Centers
A very comprehensive paper, written by an architect who has done research on disturbances at 150 data centres. A must for anyone planning a new data centre.

J. F. Donovan, Industrial Relations and Contingency Planning
A series of recommendations to help a DP centre to combat a strike. The paper includes suggestions to help in the recovery after the strike.

H. B. Becker, Security Considerations in the Small Systems Environment
These two papers give, respectively, a good account of the threats in a small system environment, and a survey of a number of products available to protect data in microcomputers and the communications systems to which they are attached.

A visit to your local library to read the papers that interest you would be time well spent.

Des Bright
IBM, West Pennant Hills, N. S. W.


This year has seen a big increase in the demand in Australia for AI technology transfer to industry; curiously parallel to a demand for someone to tell us what the technology is and why it should be transferred. With the exception of one outstanding paper, this book won't help much, even though it results from an attempt to get researchers and planners into the same bed—at a symposium last year in Zurich.

The first section is on the technology. Raulefs (University of Kaiserslautern) gives an authoritative and honest overview, but in ten pages and without references, who benefits? Wahlster (University of Saarland) has a fine account of natural language interface techniques. The come two shorter papers that describe robotics and vision. The next section, on applications, is even more disappointing: rather journalistic case studies by industrial researchers and managers.

So far, the curious reader would do better to look up IJCAI conference proceedings at the local university library. However in the remaining two sections, 'Economic Impact' and 'Infrastructure, Research and Training', one finds a type of paper that is more relevant to assessment of technology transfer and, perhaps, harder to find elsewhere. Easily the most important paper here, which
This book came with a diskette, but it could have been sold as an application in three stages of increased user-friendliness and utility. About Japan, Littles predict that they will soon start an AI business (at present it is virtually nil), penetrating to 50% in 11 years and thus overtaking the American lead of 20 to 1 at present; with a total market potential of 50% of the American. No mention of the European contribution, even at a European conference. A commercial seminar company in Australia is offering a two-day seminar introduction to expert systems for over $400 per head, at which each attendee will receive this package and be taught to build applications using it. If you can’t afford the seminar, but can spare a weekend to teach yourself the basics of expert systems, this package may suit you.

The tutorial concludes with a discussion of features available in some large-scale shells, but not in MICRO-PS, such as inexact reasoning and Bayesian inferences. Much of this discussion is by way of comparison to KES (Knowledge Engineering System), a shell which sells for over $US20,000, and from which MICRO-PS is derived, and is in effect an advertisement for KES. The novice could easily gain the mistaken impression from parts of this book that all expert systems are based on production rules. As a shell which is primarily suitable as a teaching and prototyping tool, which is all the authors claim, MICRO-PS is interesting and good value. Although it is based on backward-chaining production rules, it also includes ‘actions’, procedural commands which give the writer considerable control over the order in which rules fire and over the user interface. The commands ‘ask for’ and ‘obtain’ allow the writer to specify the order in which values of attributes will be obtained. ‘If-then-endif’ and ‘if-then-else-endif’ structures within actions may be used in much the same way as rules. User-activated commands allow conclusions to be justified and rules displayed, giving the necessary transparency of an expert system. There are minor irritations such that all user commands be in lower case.

It is reasonably easy to write applications and have them run. I had a trivial application, which advised whether or not the Copyright Act was applicable to books published on certain dates, running within a couple of hours, utilising a reasonable assortment of MICRO-PS’s facilities.

MICRO-PS is limited so that you can have no more than 20 attributes with 15 values (and a total of no more than 50), and 20 rules with no more than 15 antecedent or consequent conditions, plus various other limitations. It is clearly only valuable for prototypes and not for applications of any size. Although the syntax is tricky for beginners, it is not too frustrating.

The MICRO-PS manual which comprises the second half of the book comprehensively explains and illustrates the syntax and commands. The many useful examples are marred by the occasional running together of examples of code and user sessions. Although there are appendices covering reserved words, illegal characters, size limitations, and even a helpful glossary, there is no summary of commands and syntax. To use MICRO-PS effectively you need to construct your own summary, or hunt through over 100 pages of manual.

A commercial seminar company in Australia is offering a two-day seminar introduction to expert systems for over $400 per head, at which each attendee will receive this package and be taught to build applications using it. If you can’t afford the seminar, but can spare a weekend to teach yourself the basics of expert systems, this package may suit you.


This book came with a diskette, but it could have been sold as software, MICRO-PS, an expert system shell, with a manual that includes an extensive tutorial. Instead, the emphasis is on the book, an introduction to rule-based expert systems for absolute beginners. The tutorial part of the book explains the rationales for expert systems, and the components of an expert system (inference engine, knowledge base etc.). It illustrates how to create an expert system using MICRO-PS by extensive reference to an application which advises on problems concerning PC file operations. The tutorial concludes with a discussion of features available in some large-scale shells, but not in MICRO-PS, such as inexact reasoning and Bayesian inferences. Much of this discussion is by way of comparison to KES (Knowledge Engineering System), a shell which sells for over $US20,000, and from which MICRO-PS is derived, and is in effect an advertisement for KES. The novice could easily gain the mistaken impression from parts of this book that all expert systems are based on production rules.

As a shell which is primarily suitable as a teaching and prototyping tool, which is all the authors claim, MICRO-PS is interesting and good value. Although it is based on backward-chaining production rules, it also includes ‘actions’, procedural commands which give the writer considerable control over the order in which rules fire and over the user interface. The commands ‘ask for’ and ‘obtain’ allow the writer to specify the order in which values of attributes will be obtained. ‘If-then-endif’ and ‘if-then-else-endif’ structures within actions may be used in much the same way as rules. User-activated commands allow conclusions to be justified and rules displayed, giving the necessary transparency of an expert system. There are minor irritations such that all user commands be in lower case.

It is reasonably easy to write applications and have them run. I had a trivial application, which advised whether or not the Copyright Act was applicable to books published on certain dates, running within a couple of hours, utilising a reasonable assortment of MICRO-PS’s facilities.

MICRO-PS is limited so that you can have no more than 20 attributes with 15 values (and a total of no more than 50), and 20 rules with no more than 15 antecedent or consequent conditions, plus various other limitations. It is clearly only valuable for prototypes and not for applications of any size. Although the syntax is tricky for beginners, it is not too frustrating.

The MICRO-PS manual which comprises the second half of the book comprehensively explains and illustrates the syntax and commands. The many useful examples are marred by the occasional running together of examples of code and user sessions. Although there are appendices covering reserved words, illegal characters, size limitations, and even a helpful glossary, there is no summary of commands and syntax. To use MICRO-PS effectively you need to construct your own summary, or hunt through over 100 pages of manual.

A commercial seminar company in Australia is offering a two-day seminar introduction to expert systems for over $400 per head, at which each attendee will receive this package and be taught to build applications using it. If you can’t afford the seminar, but can spare a weekend to teach yourself the basics of expert systems, this package may suit you.


This book came with a diskette, but it could have been sold as software, MICRO-PS, an expert system shell, with a manual that includes an extensive tutorial. Instead, the emphasis is on the book, an introduction to rule-based expert systems for absolute beginners.

The tutorial part of the book explains the rationales for expert systems, and the components of an expert system (inference engine, knowledge base etc.). It illustrates how to create an expert system using MICRO-PS by extensive reference to an application which advises on problems concerning PC file operations. This application was so dull that I was not sure all users can be expected to have some knowledge of it—or if they don’t, then they need the advice! The tutorial leads you through building an application in three stages of increased user-friendliness and utility. Along the way, it provides useful suggestions on approaches to analysis and design.

The tutorial concludes with a discussion of features available in some large-scale shells, but not in MICRO-PS, such as inexact reasoning and Bayesian inferences. Much of this discussion is by way of comparison to KES (Knowledge Engineering System), a shell which sells for over $US20,000, and from which MICRO-PS is derived, and is in effect an advertisement for KES. The novice could easily gain the mistaken impression from parts of this book that all expert systems are based on production rules.

As a shell which is primarily suitable as a teaching and prototyping tool, which is all the authors claim, MICRO-PS is interesting and good value. Although it is based on backward-chaining production rules, it also includes ‘actions’, procedural commands which give the writer considerable control over the order in which rules fire and over the user interface. The commands ‘ask for’ and ‘obtain’ allow the writer to specify the order in which values of attributes will be obtained. ‘If-then-endif’ and ‘if-then-else-endif’ structures within actions may be used in much the same way as rules. User-activated commands allow conclusions to be justified and rules displayed, giving the necessary transparency of an expert system. There are minor irritations such that all user commands be in lower case.

It is reasonably easy to write applications and have them run. I had a trivial application, which advised whether or not the Copyright Act was applicable to books published on certain dates, running within a couple of hours, utilising a reasonable assortment of MICRO-PS’s facilities.

MICRO-PS is limited so that you can have no more than 20 attributes with 15 values (and a total of no more than 50), and 20 rules with no more than 15 antecedent or consequent conditions, plus various other limitations. It is clearly only valuable for prototypes and not for applications of any size. Although the syntax is tricky for beginners, it is not too frustrating.

The MICRO-PS manual which comprises the second half of the book comprehensively explains and illustrates the syntax and commands. The many useful examples are marred by the occasional running together of examples of code and user sessions. Although there are appendices covering reserved words, illegal characters, size limitations, and even a helpful glossary, there is no summary of commands and syntax. To use MICRO-PS effectively you need to construct your own summary, or hunt through over 100 pages of manual.

A commercial seminar company in Australia is offering a two-day seminar introduction to expert systems for over $400 per head, at which each attendee will receive this package and be taught to build applications using it. If you can’t afford the seminar, but can spare a weekend to teach yourself the basics of expert systems, this package may suit you.
Book Reviews

Trends in survey data, financial models, budgets and sales forecasting.

Clearly the book is not intended to teach managers all about Lotus 1-2-3 but to provide basic introduction and overview of what can be achieved using spreadsheet software and specifically 1-2-3.

John Delerno
Morgan Guaranty Australia Limited


For those without extensive computing experience this book provides a thorough introduction to dBASE III, not only for interactive use but also as a basis for writing one’s own programs. It should be noted that the additional features of dBASE III Plus are not covered.

It is written in a readable style using a tutorial approach. There are many examples of screen layouts and sample dialogues, including those with error messages. As well as the wealth of information on how to do things correctly, there are a lot of hints on avoiding pitfalls—and recovering if a problem is encountered. I did however miss having a summary of commands for quick reference.

Following an initial discussion of how the user can best implement dBASE III on his computer, the book is divided into seven sections. The first section introduces databases and the second, using dBASE III interactively. After a brief discussion of database design in the third section, the fourth section considers ordering of files and generation of reports and labels. Section 5 deals with multiple file handling, and formatting screen displays and printed output. Section 6 introduces programming in dBASE III, including program design and the use of word processors to edit code. The final section discusses briefly more advanced programming techniques such as parameters and procedures; also tucked away here are some important comments on the importance of making backup copies of files and how to set about debugging programs.

The diskette included with the book contains a database management system for a hypothetical zoo. Two datafiles are used for the multiple enquiry, report and update programs. The databases and program files are drawn on extensively to provide examples in the second half of the book. The software is subject to a personal licence for the owner’s use only.

At $68.90, this package is not cheap. The book is useful for frequent reference initially, and could be a worthwhile investment, but there are many other books available. I would borrow this one from the library—even without the diskette, it makes an excellent introduction to dBASE III.

Jennifer Hawkins
National Circuit, Canberra


This book is the Proceedings of the Conference on System Description Methodologies held at Kecskemet, Hungary in May, 1983. The conference was sponsored by IFIP Technical Committee TC2. The 35 papers are divided into sections titled The System Development Process, System Development Technology, Support Systems, Specification Systems, Hardware and Software Systems, Conceptual Models, Maintenance, Work Done and to be Done, and Examples and Test Studies. I confess to being a little bewildered by some of the section titles and, after a more detailed reading, could find no rationale behind the classification of many of the papers.

The majority of the papers are rather disappointing. Irrespective of their titles and abstracts, they lapse into a description of the authors’ pet system development tool or process, complete with the obligatory inane acronym. Their value seems to lie in investigating the range of applicability of particular methodologies and their underlying philosophies rather than contributing anything more fundamental to the topic. Other papers of a more theoretical nature are included, based on Petri nets or clausal logic. However, they are poorly motivated and again the absence of a need to be more interested in basing a particular methodology on theoretical foundations rather than addressing the more basic question of the appropriateness of the theory.

The papers by Lehmann and Mason are among the few that seem to maintain any real sense of perspective of the system development process. Lehmann’s detailed survey identifies the concept of program evolution and argues that it is intrinsic to the nature of computer usage and not due to shortcomings in current programming practice. Given that premise, he contends that program development tools should be integrated in such a way that they reflect the evolutionary nature of the process that they support. Mason attempts to develop a taxonomy of programming as an aid to communication between those interested in software system development by providing a standard framework for investigating and comparing various methodologies. On a more detailed note, readers interested in PSL/PSA will find a number of relevant contributions, including papers dealing with test cases, modifications for particular applications and the use of the tool in teaching software engineering.

An unusual inclusion in the proceedings is the reports of three working group sessions held during the conference, the discussion topics being Toward a Conceptual Framework for System Methodologies, Specifications to Code Mapping and Systems Education. While these reports are rather short, they do offer some insight into the current state of development in those areas and are written in a clear and direct style.

In his Turing Award Lecture in 1972, E.W. Dijkstra predicted: well before the seventies have run to completion, we shall be able to design and implement the kind of systems that are now straining programming ability, at the expense of only a few percent in man-years of what they cost us now, and that besides, the systems will be virtually free of bugs.

In comparison, one of the contributors to this volume identifies a serious problem in software engineering: the difficulty of understanding whether progress is actually being made in this field! Perhaps the lack of any fundamental coherence among the papers from this proceedings can provide some hint as to why Dijkstra’s vision of the future has not come to pass.

Clem Baker-Finch
University of Tasmania


In the Australian context this is a text-book designed for the use of students in universities and colleges of advanced education. It aims to provide a methodology which enables the definition of an information system to be achieved with proper weight being given to the environment in which it exists.

The exposition of a non-mathematical account of systems analysis and design working from the probabilistic world of the organisation through to the technical design. The so-called Multiview Methodology propounded thus provides a guide through the analysis of human activity systems, the analysis of entities and functions, i.e. information modelling, analysis and design of the socio-technical system, design of the human user-computer interface and finally technical design.

In a book of about 160 pages, the treatment is both elementary and condensed. However, the authors have abstracted a surprising amount of material on sound methods and procedures and presented it in a coherent and readable text. The exposition develops from a useful working definition of a system and the question of drawing the boundaries to systems and sub-systems. The indispensable idea of identifying their structure is stressed and the concept of root definitions to aid in this is explained and illustrated. Throughout the main text the authors make use of a case study which, for ease of identification, is printed in italics. This...
enables the techniques used in the stages of multiview approach to be demonstrated effectively. Combined with the good references to current bibliography the reader can gain an understanding of the subject.

A very real advantage of the approach is that it relates analysis and design to the 'real-world' and is not just another exposition of the abstract techniques to be used. Students who have studied the book should find the systems they must deal with when they enter the workforce much less confusing.

Perhaps the major weakness of the book is that, while user oriented, the methods to be used for improving flexibility in design are not sufficiently stressed. Indeed, the sections on strategies for design, design of the technical sub-system and acceptance, maintain­ance and development of the computerised information system comprise only about 40 pages. Collateral reading of the references to prototyping, analysis and design techniques and systems specification will be needed to supplement the text.

By concentrating on the analysis of the system in relation to its environment at an introductory level the authors have provided a very useful text and one which should prove of value to students, teachers and practising analysts alike, although the price is somewhat expensive at $29.95.

H. D. Pridmore
Computer Sciences of Australia


This book outlines an approach (methodology) for the development of information systems which concentrates on data analysis (modeling) and data base design. This is in contrast to the more conventional approaches to systems analysis which concentrate on processes and outputs. Given the large scope of the book and its relatively short length the treatment of individual topics is fairly brief. Fortunately the author has a succinct style and also makes good use of diagrams in his exposition so clarity does not suffer. He also assumes some knowledge of the business/organizational environment in which information systems are typically located. Nevertheless this is not a reference manual, it is an introduction to this subject. Selected, up-to-date, references are provided for readers wishing to pursue individual topics in greater depth.

The book was primarily written for computer studies and business studies students who have already completed at least one course in data processing. It should also be of use to data processing professionals looking for a succinct treatment of a data base oriented approach to information systems development. Managers in other functional areas such as finance, production and marketing who are considering the use of data base systems would also benefit from reading this book.

The book contains ten chapters, a 15 page appendix describing a practical application of the methodology, a bibliography and an index. In summary, these contain:

1. A critique of conventional systems analysis and an introduction of the ideas, tools and techniques of the methodology.
2. Business analysis of the organization’s goals, structure and information requirements.
3. Data analysis including the entity-relationship approach, relational modelling and normalisation.
4. The Logical Schema including the relational, hierarchical and network approaches.
5. The Physical Schema including lists and inverted files.
6. An overview of the structured systems analysis and design, participation and prototyping approaches.
7. An introduction to DBMSs including an overview of IMS, DB2, IDMS, ADABAS and INGRES.
8. An overview of microcomputer DBMSs and of distributed data bases.
9. An introduction to data dictionary systems including an overview of MSP DATAMANAGER and the ICL DDS.
10. A discussion of the role of, benefits of, and position of the Data Base Administrator in the organization.

This book provides a very readable introduction to a representa­tive data base oriented approach to information systems develop­ment. Given its objectives the only criticisms I have are the existence of a fair number of typographical errors and the use of paper that on occasions was not sufficiently opaque to prevent print from the other side of the page showing through. Nevertheless I intend to adopt this text for our Systems Analysis course next year.

R. C. Reeve
University of New England


This is a collection of papers presented at a IFIP Working Group colloquium titled, ‘Information Systems Research - a doubtful Sci­ence?’, held in the Manchester Business School during September 1984.

The conference aim as stated in the opening paper by Fitzgerald was ‘to enable a concern about research methods in information systems to be aired’. It is fair to say that the conference achieved its aim of encouraging concern to be aired. Information systems, being the wide field that it is, does attract a wide range of topics and opinion. This conference was certainly no exception. In line with the conference aim, the majority of papers deal with the more abstract and fundamental issues of information systems research or research in general. As an example, two papers one by H. E. Nissen and another by A. M. Pettigrew cover the inherent problems of acquiring knowledge.

An important question raised by the conference was whether the scientific approach to research is appropriate for information systems research or whether alternative approaches are needed. Two papers, one by H. K. Klein and K. Lyytinen and the other by R. D. Galliers specifically address this issue. However, it is also raised in a number of other papers.

A number of papers propose research methods appropriate to information systems research. These include action research, specifically described by T. Wood-Harper but also raised by other authors. Other research methods discussed are phenomenology by R. J. Boland and a method called STROBE, which is described by K. Kendall and J. Kendall.

In another paper, K. J. Lyytinen and H. K. Klein propose a new theory for information systems. Of the papers, only that by A. Milton Jenkins defines specific topics for research, these being user interface design, prototyping, 4GL operational efficiencies and critical success factors for information systems projects. This paper also defines the research process in terms of a set of steps, which some people may find attractive. There are also papers concerned with socio-technical design.

This series of papers cannot be viewed as easy reading. It is not intended for practitioners nor for those who are looking for specific research topics, but more for those who are questioning the nature of information systems research in general.

I. T. Hawrysziakiewcz
Canberra C. A. E.


This book contains a collection of seventeen papers with an accom­panying introduction. The papers were presented at a formal course conducted at the University of East Anglia in September 1982 but have been 'extensively revised and supplemented since the course'. The references in the papers give some clue as to the era of thinking represented therein: most works referenced have publication dates in the late '70s, with the latest citation being of a 1983 (unpublished) paper.
I don't have space to list the names of all the papers and their authors and to outline and comment upon the relevance of the issues raised in the papers to today's situation, so I'll concentrate on the last matter only.

Put simply, the papers are mostly about the deficiencies of existing Data Base Management systems. These inadequacies are (paraphrasing and generalising considerably):

1. There is still no universally agreed and user-convenient abstract model which can be used to represent a slice of reality and which maps into a simple physical structure which can support efficient, recoverable, inter-connectable, semantically valid data base operations. The book describes two of the most recent contenders for the ideal data model, the functional model and a 'semantic model'.

2. Existing database languages do not fit as natural extensions to extant programming languages. There is discussion of problems with the current situation and solutions to some of those problems.

3. There is a real need for a database system that allows interaction of users located at many different sites, each site having its own preferred DBMS; such a goal could be achieved if there were some common method of specifying views of the local data which views may be seen and manipulated by remote users.

4. The issues of using a CODASYL DBMS as the physical basis of implementation of a data base which shows a relational interface to its users, and still not fully worked out.

5. Many of the objectives originally sought to be addressed with the creation of centralised DBMS have not been achieved, but then, many sub-objectives have been achieved, and such achievement may well be more important than the original goals.

So this is a reference book for the advanced DB applications practitioner and for those currently involved in building database software and who wish to overcome some of the deficiencies of products already on the market. For this group, it offers a snapshot of the developing world of database management not documented elsewhere. In addition very committed and wealthy post-graduate students and academic staff (are there any?) will wish to purchase this book because it suggests a wealth of research projects and directions.

A. Y. Montgomery  
Royal Melbourne Institute of Technology


This reference guide is produced by the European offices of the international accounting firm of Coopers & Lybrand. They, and other large accounting firms, generate extensive documentation on the audit-relevant characteristics of different business computing systems for in-house use and this is the first (as far as I know) such work that has been publicly released. I understand that similar guides to other systems may be published in the future.

The guide has six brief sections: an introduction, descriptions of and commentaries on hardware characteristics, software characteristics, audit aspects, a short bibliography (mainly to IBM documentation) and an Appendix of Control Program Facility (CPF) object types.

The objective of the guide is to apprise auditors of the unique features of the System/38 so that they will not overlook any of the security features of the system and also so that they will be alert to those areas where extra caution and security may be needed.

My guess is that there are around 300 System/38's in Australia. This guide should be a required reference to the auditors (external and internal) who work with them, and to D.P. managers seeking to gain some leverage on the growing problem of security. The book would also be useful to those seeking a succinct summary of the System/38 (e.g. potential purchasers, system analysts) as the guide's first three sections provide a good, technical overview of the system which might not be obtained so readily or clearly from an examination of more comprehensive systems manuals.


The words of its preface best describe this book:

This book is one of the results of a 5-year research effort on multiprocessor architectures carried on by the authors in the frame of the MUMICRO project of the Italian National Research Council (C. N. R.) Computer Science Program.

Bearing this in mind and the fact that the book is printed directly from the apparent output of a standard personal computer word processing program using a normal daisy-wheel printer, I am afraid that the whole book could only be described as introductory in nature. This is far from what should be expected from a five year research program.

Another problem with the book is the lack of full reference citations. For example, chapter 4 on the 'Design of Multiprocessor Buses', a vital if not overwhelmingly important topic in this area, warrants only 29 references (and these include such titles as Intel's 1979 Multibus-1 specification and the 1978 Fairchild TTL data book).

In construction the book consists of seven chapters by ten authors as follows:

1. Multiprocessor System Architecture  
2. Performance Analysis of Multiprocessor Systems  
3. TOMP Software  
4. Design of Multiprocessor Buses  
5. Some Examples of Multiprocessor Buses  
6. Hardware Modules for Multiprocessor Systems  
7. Multiprocessor Benchmarks

Since the book is a collection of edited chapters by different authors, you naturally expect some parts to repeat or to be independent of the other parts. While all the above topics may be familiar it may be worthwhile to explain what the 'TOMP Software' chapter is about. Essentially it describes 'TOS, the Operating System of the TOMP prototype'. The 'TOMP' machine is elsewhere called the 'TOMP-80' and is described as a system of modules (boards) and bus structures. There are two basic boards, a processor board (Zilog Z8001 based) and a dual-port memory and local-to-global bus interface board. Against this background the software development (normally a crucial part of the whole computer engineering project) has been relegated largely to the background with the following summary from chapter 3, viz.

In summary, the software project for the TOMP-80 has been more conceived as a tool for the design and refinement of the architecture and as a research workbench than as a complete and engineered multiprocessor operating system for application development. This has been due to two severe constraints for the project. First the implementation had to follow and partly support the hardware development; the resources devoted to this task were about one person at full time. Secondly, only limited software development tools were available. In fact the first version of the software was developed using cross-development packages running on a 8-bit microcomputer system (MCZ 1/20) where assembly language only was available . . . prototype development deadline (9 months) . . .

Now, who in Australia can complain after that? But unfortunately that is the tone of the book and the project. Even the ill-fated Intel iAPX-432 processor gets a mention in chapter 1. Really the whole project is best described by the authors of chapter 1 from the Politecnico di Torino in Italy when they state,

The system had to be defined and designed in such a way to allow the use of the available 16-bit processors. . . .

The reader may be forgiven for feeling that the authors regret this throughout the book!
On the whole, various sections are useful summaries (e.g. chapter 2 on Performance Analysis) but unfortunately this does not warrant its purchase for the professional library. This can be clearly seen by the unfortunately dated statements given in the section of chapter 5 dedicated to the vital bus architecture for the computer (the so-called M3BUS). The authors put it this way:

Owing to cooperation with P896, some ideas are shared by the two designs; in particular M3BUS has many common features with P896 specification D4.1, developed in 1980/81. Since then, P896 has evolved into a 32-bit bus, while M3 stayed with 16-bit machines. M3 is now used in Italy by companies which develop systems for industrial automation and control ... a 32-bit version is being defined ... and is not described here.

Well now I know where the 'real-time' in the title came from!

William J. Caelli
ERACOM Pty. Ltd.


This is the report from an IFIP workshop held at the University of Kent, Canterbury, UK, in September, 1983. Ring technology is obviously here to stay, and this is an excellent reference for those wishing to understand it well. It contains a wealth of useful inside information on ring networks, especially the Cambridge, slotted ring style, which is installed at the host site.

In the Cambridge ring, a number of small, fixed-size 'carriers' circulate constantly, and may be used to transfer data between pairs of stations in one or two byte chunks at a time. While perhaps not as efficient overall as the token ring designs favoured by IBM, the Cambridge ring is well-suited to a community of highly-interactive computer users.

J. Lions
University of New South Wales


This book, which is published in Australia, contains a collection of twenty seven articles dealing with the application of microcomputers to local government planning and management. The book aims to give the technical professional concerned with contemporary research problems in local government planning an overview of recent developments in micro computer software solutions in these fields.

These problem areas are organised under seven headings:

1. Microcomputers in local government,
2. Data management and display,
3. Strategic planning,
4. Human factors in planning,
5. Infrastructure planning,
6. Planning and management for building,
7. Future directions.

The problems addressed under each heading are wide ranging and cover areas as diffuse as strategic land use planning, population forecasting, local area traffic management, waste collection and estimating resource requirements for land subdivision and house construction.

The articles vary in quality. Some attempt to present a 'primer' on the problems typically analysed under a general area such as local area traffic management, or financial planning and strategic analysis, and then add, almost as an afterthought, the microcomputer applications. Others give a more balanced discussion of the nature, implementation and advantages of particular microcomputer applications together with an explanation of their implementation and function. A few of the articles try to be more general and discuss wider issues of microcomputer use in particular technical area. Unfortunately, in this type of discussion most articles are too short to permit other than rather vague generalisations about microcomputer impact and advantages, that has little solid information to offer the intending practitioner.

The book is not really addressed to the local government administrator or council person interested in the introduction of microcomputers for local government financial or administrative tasks; or anxious to find out the pitfalls and problems of microcomputer introduction into local government offices. It addresses particular technical areas of local government planning and management which are more likely to be addressed by research and public sector bodies at the State or Commonwealth level of government. But then, as the authors say, the book is directed towards the professional who works in these areas, at whatever level of research or government.

The book will have a fairly select audience among those professionally interested in the areas of research covered by the articles. But for that audience it can provide a concise introduction to recent microcomputer solutions. The commendably complete referencing at the end of most articles also provides a good guide to further reading on microcomputer application.

Michael McCrae
Australian National University


This introductory book contains three parts: part one covers the computer system and programming; part two is concerned with systems software, and part three looks at commercial data processing.

Unlike many other introductory books pitched at the novice, this book fails to catch the reader's attention due to lack of 'white space', no highlighting of keywords, lack of figures and tables, and a large dose of verbosity. Several figures and sample programs in BASIC had to be reduced to fit into the space provided with the result that the annotations and programs became illegible. The sample programs are written in an unstructured Dartmouth BASIC and contain few, if any, comments and are not preceded by a design. The term documentation appears only in the glossary which means it is not pointed out to managers or users to insist on adequate internal and external documentation in the form of user's guides/programmer's reference manuals for either 'canned' or in-house produced software.

The author gives away his long association with the DP industry by extolling the virtues of core dumps on p. 77 in the five-page section on interactive programming (which is unbroken by any figures!). It is doubtful if a user or manager these days needs to know anything about core dumps. Users and managers would have benefited from an explicit classification of utilities into data set utilities, etc., instead they are treated to a list of IBM utilities, such as IEBCOMPR, IEBCOPY, etc., and their functions. This IBM bias is very evident in the book and extends to detailed explanations of IBM access methods from BSAM to VTAM in the main part of the book and in Appendix A.

Several computing terms are incorrectly described, e.g., on p. 130, even parity is explained as requiring each character to have an even number of bits'. Multitasking and multiprogramming are used interchangeably on p. 4, which is incorrect. COBOL is classed as a problem-oriented language instead of a procedure-oriented language. BASIC is considered to be a commercially-oriented language rather than a general-purpose language and the term BASIC compiler is used rather than the more common BASIC translator.

As little effort is made to quantify the attributes of parts of the computer system, the reader will be left in the dark as far as typical disk capacities and access times, print speeds and printer types, memory capacities and memory speeds are concerned. There is no section on systems development which must be a serious omission as managers communicate with system analysts and would therefore benefit from an exposure to the topic.
This book performs badly in comparison with four other recent introductory texts (at half the price) on the use of computer in business data processing by authors such as L. Long, S.I. Mandell, R.M. Stair or K.J. O'Brien. Because of its curantist price, the bias towards large-scale IBM installations and the lack of structure in the text and page layout, it is not recommended.

Dominic Wild
Pert Technical College


I opened this book with anticipation: many fascinating aspects of modern computing technology depend on magnetic surface recording—the only digital storage technology that predates the era of modern computing and that is still going strong. Just how much longer this technology may last is an open question—it is certainly not yet finished: there are still new techniques such as vertical recording that have hardly been exploited, and of course the ultimate possibilities for laser disks can only be guessed at today. I am hoping to find a book which portrays the principles, the history and the future prospects of disk storage technology, in an unemotional way suitable for the intelligent, scientifically-trained layman.

Well, unless you prefer thumbprint sketches, this is definitely not it. After an hour, I put the book aside with a strong sense of frustration—there is some useful information here, but not much. It is poorly-organised, poorly-researched, superficial and repetitive, and obviously put together by a journalist in a hurry to get on with his next assignment. It is worth browsing if you come across a copy, but it is not worth paying even a small fraction of the publisher's price.

J. Lions
University of New South Wales


What is there about a book on aspects of electronic music which makes it of potential interest to readers of this Journal? The answer, of course, is that this area offers great scope for applications of computers, and is still in its infancy. Much remains to be done in such areas as instrument design, human interfacing, input languages and notation, graphic and other aids, music composition, networking, etc.

One of the biggest obstacles to progress is the widespread lack of understanding of the true nature of electronic music. This lack is on all sides. Classical musicians mostly view electronics as no more than a new orchestration possibility, and think it intrinsically lacks 'human' attributes (I pass over their probably justified fears that it will displace musicians from jobs). For example, a recent extensive done in such areas as instrument design, human interfacing, input languages and notation, graphic and other aids, music composition, networking, etc.

Unfortunately, this book does not greatly help in spreading the true gospel on electronic music, although it clearly tries to. It has strengths, but they are buried. For those who can follow it, it does present a balanced overview of recent developments in the performance area, and it makes interesting and intelligent predictions for future movements. It is especially good on the way pop musicians view these developments, and their attitudes to them—this should be useful feedback to synthesiser designers, or others interested in modern music.

For readers of this journal who would like a more technical introduction to the area, the best place to start is still Chamberlin (1980).

References

University of New South Wales


'Resilient' is used as a synonym for 'fault tolerant'. The book is described in the flyleaf as a 'practical text' on the topic, and indeed would serve as an introduction to the state of the art in the early 80's. It comprises an introductory chapter, chapters surveying fault-handling techniques in hardware, system design (mostly software), communications, real-time systems and distributed systems. The hardware chapter, by W.C. Carter, is particularly comprehensive in its coverage of the literature on fault detection, testing and diagnosis, but rather light on the coverage of error correction techniques such as majority voting by redundant modules and error correcting codes. B. Littlewood contributes a useful chapter on the modelling and prediction of software failures.

Unfortunately but predictably, the above chapters would assist the would-be designer of a fault-tolerant system only to the extent of defining some concepts, introducing a few basic techniques, and providing a list of references. Even the definitions, which occupy a fair fraction of several chapters, must be used with caution, since the different authors of the various chapters do not always agree in their language. None of the chapters present much quantitative or critical analysis of the techniques described, and none actually contain the detailed information a designer would need to assess the probable success of the techniques. The book falls sadly short of being a practical text on fault-tolerant design, let alone a handbook for the designer. One's impression after reading these chapters is that there is still no sound body of proven theory relevant to fault-tolerant design outside the design of error detecting and correcting codes.

The remaining chapters of the book introduce and describe four commercially available fault-tolerant computers: Tandem, MOMENTUM, STRATUS and the industrial control computers of August Systems. These probably are more useful to a designer than the earlier chapters in that they address real system designs.

The Australian Computer Journal, Vol. 18, No. 3, August 1986
offering proven, if limited, solutions to real world failures. It is interesting that all rely on the replication of modules for larger than the optimum scale suggested by the elementary theory of reliability, and none seems to make much use of such theory as is covered in the survey chapters. Perhaps this reflects the sensible desire of designers to use standard components whose functional specification is not itself dependent on a particular fault model. Thus we see replication of entire CPU's, each of conventional design, being preferred to the design of a CPU from replicated modules of smaller size. Another factor for this preference may be a healthy scepticism about theories which assume that individual failures will affect single signals, whereas the not uncommon accident of a short between logically unrelated signals on a printed circuit can mimic the simultaneous failure of dozens of gates.

At well over $50, the book cannot be recommended for personal purchase. It is a useful source of references to hold in a library, but in five or six years will probably be obsolete in that role.

C. S. Wallace
Monash University


Eurographics '85 contains the 39 papers, by authors from 17 countries, presented at the Sixth European Graphics Conference and Exhibition held at Nice, France, September 1985. As with earlier such proceedings, it is published in book form, this time with 16 colour pages. Although publication was very prompt after the Conference, the number of typographical errors appears to have increased in this issue.

The main disappointment with the book is again the the absence of extended papers from the invited speakers—only 12-page abstracts are included from Allain (Thompson-CSF), Fuchs (University of North Carolina, Encarnacao (Technische Hochschule Darmstadt) and Braid (Shape Data Ltd). This omission severely diminishes its usefulness.

There is an increase in the number of sessions on scene simulation techniques—a total of 9 papers. Given the marginal contributions to the state of the art made by these papers, it is hoped that an imbalance towards such papers does not arise in future Conferences (as has occurred at ACM SIGGRAPH Conferences). Scene simulation techniques are very important but there is a danger that Conferences might promote the area because of its pictorial impact.

The other main topic concerned graphics standards. Six papers covered issues such as language bindings for graphics standards, GKS within a distributed computing environment, certification of GKS and the design of object-oriented languages on top of graphics standards. The papers give an insight to the wide scope of the standardisation efforts.

The other papers cover a wide range of topics—man-machine interface, algorithms, computational geometry, hardware, modelling and graphics applications. Readers are likely to find several papers of interest here, but probably insufficient to justify the book's purchase for private reference. Like earlier proceedings, the book is a worthwhile collection of papers on graphics and is therefore most suited to the libraries of research organisations.

John F. O'Callaghan
CSIRO Division of Information Technology


This book contains a selection of extended versions of papers presented at the IMACS European Simulation Meeting held in Eger (Hungary) during 27-30 August 1984. In all 32 papers are presented, apparently produced from camera-ready copy supplied by the authors judging from the typestyles.

Simulation may appear to be a very specialised field to some readers, but it is worth pointing out that applications for simulation are growing rapidly, not least in the schools where the simulation of dangerous, awkward or expensive situations can be usefully modelled in the class-room. This book, however, deals with simulation in R&D roles. One of the most rapidly growing uses of simulation in this field is the simulation of electronic circuits as used in integrated circuit designs. Vast numbers of software and hardware engineers are now exercising both logic and analogue circuit simulators in attempts to verify that a design is correct before it is committed to fabrication. This topic crops up in several papers, and Chapter IV is devoted to it. Relevant papers are entitled Multimode Simulation Concepts, Switch Level Extension of a Logic Simulator, A Theory and Algorithmic Frame for Switch Level Simulation, Modelling of Semiconductor Structures, Two-Dimensional Simulation of Semiconductor Technology and Devices, and Hardware Description Language for Design of Digital Equipment.

Another important area covered by the conference is the use of highly parallel systems to carry out simulation computations. The match between the problem, itself highly concurrent, and the computation mechanism is thereby much improved. The papers, however, do no more than sketch out research work in progress, and hint at applications for the future.

Two chapters are devoted to questions of general simulation interest: mathematical models, formalization, numerical methods, and languages and tools. Readers interested in languages may be interested in papers dealing with GPSS-FORTRAN Version-3, and with standardization and development of the SIMULA language. Standards for this latter language are, interestingly, in the hands of the implementors (one implementation—one vote) which is claimed to lead to implementability and commonality.

The remaining papers cover a wide range of applications, ranging from simulations of blood pressure through economy models to port traffic in Dover, England. The book should be of interest to technical and university libraries for its conference coverage and to persons actively engaged in work in simulation.

Arthur Sale
University of Tasmania


For some time the programming language Pascal has been the principal algorithmic language taught for programming education in Computer Science courses. While there are many reasons for this (fall outside the scope of this brief article), in the modern day context there are also many reasons why Pascal should now be considered outdated.

One of the principal failures of Pascal is any sincere attempt to address the modularity issue. All Pascal's module facilities (essentially the procedures) are directed towards run-time structures, there being no facility specifically for information hiding. Pascal has several other shortcomings, again outside the present scope.

Many computer science educators (including this writer) would wish to change this and move to a more modern language with the abstraction power of Pascal, but incorporating the additional lessons learned over the last fifteen years. There are few general candidates here—C is ruled out because of its abysmal abstraction facilities (though it does not claim to be that sort of language any way).

The two serious contenders are Modula-2 and Ada. The CSIRO and the University of Queensland are currently reviewing the position of Modula-2 as an educational language (Modula-2 is surely the computing David to the Goliath of Ada.)

Most Ada books thus far printed (and there are now many) have approached Ada as a second—or later—language. A book which takes an alternative view and suggests Ada as an introductory language is a welcome break in this tradition, and needs be taken seriously by Computer Science educators. 'Programming in Ada: a first course' is a well presented beginning programming text, covering all the facilities one would either like to use in a first program-.
Good human factors can ensure the success of systems and software, and a bad interface is inefficient and frustrating. Yet despite the ease with which we may distinguish between the good and the bad by direct experience, the principles of good human factors design are not obvious. Elucidating these principles is a multidisciplinary activity involving the collaborative efforts of psychologists, ergonomists, computer scientists and engineers. The multidisciplinary and youthful nature of the activity has resulted in a scattered literature and, to date, relatively few textbooks and monographs have been published. Conference publications, therefore, have the potential of consolidating the literature with contemporary studies.

This book is the published proceedings of the April 1985 Computer and Human Interaction Conference (CHI '85) held in San Francisco. The 30 papers are grouped into 10 sections and interspersed with introductions to 7 panel sessions. The material all has the form of conference pre-prints and there are no keynote papers, nor editorial introductions to its sections. The contributors are from the major industrial and university human factors laboratories in USA and Canada, together with a very small (but significant) European contingent.

The main value of this book is to be found in the papers. A wide range of human factors areas are covered, including novel computer interface. There are quantitative analyses of VDU text presentation rates on reading comprehension and reading speed (Tombaugh, Arkin and Dillon); depth vs. breadth in menu design (Tullis); and the reasons why synthetic speech is harder to remember than natural speech (Waterworth and Thomas). These and other authors discuss their experimental methods with candour, and thus give valuable insight into the hazards of conducting experimental human factors work. Papers such as ‘ADM: A Dialog Manager’ (Schulert, Rogers and Hamilton) provide a useful tutorial approach to user interface manager systems. Several authors discuss knowledge-based user assistance techniques. Almost all papers provide a full bibliography.

Interesting though the panel sessions at the conference must have been, their inclusion as pre-print introductions is largely unsuccessful, except as a record of a topic included. However, two sessions are worthy of note and make interesting reading. Firstly, the panel on ‘Psychological Research Methods in the Use of Computers’ prompted three prominent practitioners (Gould, Barnard and Landauer) to reflect on their approaches to human-factors work in a way which is unusual in the scientific literature. Secondly, the inclusion of the panel on ‘Interfaces in Organisations’ indicates that the human factors discipline is starting to look beyond its traditional individual-human individual-computer boundary.

This book is a useful collection of papers with value for academics, researchers and students interested in human factors research. Academics and industrial workers wishing to gain a speedy insight into a wide range of current human-factors activity would find value in many of the papers and their bibliography. A number of the papers could form the basis of case-study material for courses in human factors.

Robin W. King
University of New South Wales


Good human factors can ensure the success of systems and software, and a bad interface is inefficient and frustrating. Yet despite the ease with which we may distinguish between the good and the bad by direct experience, the principles of good human factors design are not obvious. Elucidating these principles is a multidisciplinary activity involving the collaborative efforts of psychologists, ergonomists, computer scientists and engineers. The multidisciplinary and youthful nature of the activity has resulted in a scattered literature and, to date, relatively few textbooks and monographs have been published. Conference publications, therefore, have the potential of consolidating the literature with contemporary studies.

This book is the published proceedings of the April 1985 Computer and Human Interaction Conference (CHI '85) held in San Francisco. The 30 papers are grouped into 10 sections and interspersed with introductions to 7 panel sessions. The material all has the form of conference pre-prints and there are no keynote papers, nor editorial introductions to its sections. The contributors are from the major industrial and university human factors laboratories in USA and Canada, together with a very small (but significant) European contingent.

The main value of this book is to be found in the papers. A wide range of human factors areas are covered, including novel computer interface. There are quantitative analyses of VDU text presentation rates on reading comprehension and reading speed (Tombaugh, Arkin and Dillon); depth vs. breadth in menu design (Tullis); and the reasons why synthetic speech is harder to remember than natural speech (Waterworth and Thomas). These and other authors discuss their experimental methods with candour, and thus give valuable insight into the hazards of conducting experimental human factors work. Papers such as ‘ADM: a Dialog Manager’ (Schulert, Rogers and Hamilton) provide a useful tutorial approach to user interface manager systems. Several authors discuss knowledge-based user assistance techniques. Almost all papers provide a full bibliography.

Interesting though the panel sessions at the conference must have been, their inclusion as pre-print introductions is largely unsuccessful, except as a record of a topic included. However, two sessions are worthy of note and make interesting reading. Firstly, the panel on ‘Psychological Research Methods in the Use of Computers’ prompted three prominent practitioners (Gould, Barnard and Landauer) to reflect on their approaches to human-factors work in a way which is unusual in the scientific literature. Secondly, the inclusion of the panel on ‘Interfaces in Organisations’ indicates that the human factors discipline is starting to look beyond its traditional individual-human individual-computer boundary.

This book is a useful collection of papers with value for academics, researchers and students interested in human factors research. Academics and industrial workers wishing to gain a speedy insight into a wide range of current human-factors activity would find value in many of the papers and their bibliography. A number of the papers could form the basis of case-study material for courses in human factors.

Robin W. King
University of New South Wales


This is the proceedings of an IFIP WG 6.5 conference held in Nottingham, England, in May, 1984. It deals with technical and user aspects of electronic mail and related communication services. This is a fast-moving, rapidly maturing area of computing technology, which will potentially connect all the computer users in the world, and their friends into one vast network, with an immediacy never achieved by the telephone network or the traditional postal system.

In this new world, finding the correct address for your intended recipient may take longer than delivering the message incorrectly to a dozen wrong ones. The problems with electronic junk mail already exist; the problems with doing what to do about wrongly addressed mail that you have just received are only just starting. Appropriately, the first part of this volume is given over to ‘Naming, Addressing and Directory Services’. The most interesting of the four papers here is, in my opinion, ‘The Domain Name System’, by P. V. Mockapetris.

Given that the message is correctly addressed, there are the questions of how does it get from here to there, and what may it contain? There are three papers in the second section on multimedia mail (i.e. messages that are much more interesting than mere telegrams, potentially containing pictures and sound, and colour graphics to boot). The third section, on ‘User Interface Architecture’ addresses the problems of what to do with it all when you’ve got it: the problems of filing and keeping electronic mail are just as bad as for the ordinary kind. J. Palm’s contribution, ‘You Have 134 Unread Mail! Do You Want to Read Them Now?’, emphasises one of the problems that occurs only too frequently in my experience.

The remaining sections address issues such as cost/benefit analysis, security, regulations, conferences and telex systems. As is commonplace for books from this publisher, the book is printed from ‘camera ready’ copy of very uneven standard provided by the author. Even so, this does not detract too much from its usefulness for any person actively engaged in the implementation of an electronic mail system.

J. Lions
University of New South Wales

156
FATHER OF AUSTRALIAN COMPUTER OPENS ENTRE'

The co-father of Australia's computer industry, Mr Maston Beard, declared open Australia Entre' Business Centres first retail centre at Moorabbin.

Mr Beard was co-designer of CSIRAC, Australia's first computer and regarded as the third oldest in the world. The computer ran its first test programs in late 1949. The main memory of the machine took the form of mercury filled ultrasonic delay tubes, at each end of which a quartz crystal could detect pulses of sound energy.

The memory capacity of CSIRAC was initially 1/2K which was later boosted to 1K.

Both Pearcey and Beard had been involved with the development of radar during the war and teamed up when they were both engaged in research at the CSIRO.

CSIRAC remained in service until 1964 where it had operated at Melbourne University, but was retired to the safe keeping of the Science Museum of Victoria.

CALL FOR PAPERS

The Tenth Australian Computer Conference will be held at Deakin University, Geelong, Victoria on 4-6 February, 1987. The aim of the conference is to stimulate the interchange of ideas and information among computer scientists working in academic and other institutions around Australia. New members of the profession and graduate students are particularly encouraged to submit papers and to attend the conference. To this end every effort is made to minimize the cost of attending.

Papers are sought that survey current research or report the results of new research in any area of Computer Science. Authors should follow the guidelines for papers in the Australian Computer Journal and should not exceed 5000 words in length. Following recent policy decisions, it is planned that a selection of the best papers will be published in the May, 1987 issue of the Australian Computer Journal.

Initial versions of papers for refereeing are needed by 22 September, 1986, and final versions of accepted papers will be needed by 15 December, 1986. Intending authors should seek further information as soon as possible from the Papers Chairman, ACSC-10, Division of Computing and Mathematics, Deakin University, Waurn Ponds, Victoria, 3217.
The Australian Computer Journal is an official publication of the Australian Computer Society Incorporated.

OFFICE BEARERS: President: R. Christie; Vice-Presidents: M.L. Cattermole, J. Goddard; Immediate Past President: A.W. Coulter; National Treasurer: R.G. Heinrich; Chief Executive Officer: A. Kelly; P.O. Box 319, Darlinghurst, NSW 2010 telephone (02) 211 5855.


SUBSCRIPTIONS: The annual subscription is $20.00. All subscriptions to the Journal are payable in advance and should be sent (in Australian currency) to the Australian Computer Society Inc., PO Box 319, Darlinghurst, 2010. A subscription form may be found below.

PRICE TO NON-MEMBERS: There are now four issues per annum. The price of individual copies of back issues still available is $2.00. Some are already out of print. Issues for the current year are available at $5.00 per copy. All of these may be obtained from the National Secretariat, P.O. Box 319, Darlinghurst, NSW 2010. No trade discounts are given, and agents should recover their own handing charges.

MEMBERS: The current issue of the Journal is supplied to personal members and to Corresponding Institutions. A member joining partway through a calendar year is entitled to receive one copy of each issue of the Journal published earlier in that calendar year. Back numbers are supplied to members while supplies last, for a charge of $2.00 per copy. To ensure receipt of all issues, members should advise the Branch Honorary Secretary concerned, or the National Secretariat, promptly, of any change of address.

MEMBERSHIP: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries. Canberra: PO Box 447, Fyshwick, ACT, 2609, NSW: 1st Floor, 72 Pitt Street, Sydney, NSW, 2000. Qld: Box 1484, GPO, Brisbane, Qld, 4001. SA: Box 2423, GPO, Adelaide, SA, 5001. WA: Box F320, GPO, Perth, WA, 6001. Vic: PO Box 98, East Melbourne, Vic, 3002. Tas: PO Box 216, Sandy Bay, Tas, 7005. NT: GPO Box 213, Darwin, NT, 5794.

CONTRIBUTIONS: All material for publication should be sent to: Associate Professor J. Lions, Editor, Australian Computer Journal, Department of Computer Science, University of New South Wales, Kensington, NSW 2033. Prospective authors may wish to consult manuscript preparation guidelines published in the February 1986 issue. The paragraphs below briefly summarise the essential details.

Types of Material: Four regular categories of material are published: Papers, Short Communications, Letters to the Editor and Book Reviews. Generally speaking, a paper will discuss significant new results of computing research and development, or provide a comprehensive summary of existing computing knowledge with the aim of broadening the outlook of journal readers, or describe important computing experience or insight. Short Communications are concise discussions of computing research or application. A letter to the Editor will briefly comment on material previously appearing in the Journal or discuss a computing topic of current interest. Descriptions of new software packages are also published to facilitate free distribution.

Refereeing: Papers and Short Communications are accepted if recommended by anonymous referees. Letters are published at the discretion of the Editor, and Book Reviews are written at the Editor's invitation upon receipt of review copies of published books. All accepted contributions may be subject to minor modifications to ensure uniformity of style. Referees may suggest major revisions to be performed by the author.

Proofs and Reprints: Page proofs of Papers and Short Communications are sent to the authors for correction prior to publication. Fifty copies of reprints will be supplied to authors without charge. Reprints of individual papers may be purchased from Associated Business Publications, PO Box 440, Broadway, NSW, 2007. Microfilm reprints are available from University Microfilms International, Ann Arbor/London.

Format: Papers, Short Communications and Book Reviews should be typed in double spacing on A4 size paper, with 2.5cm margins on all four sides. The original, plus two clear bond-paper copies, should be submitted. References should be cited in standard Journal form, and generally diagrams should be ink-drawn on tracing paper or board with stencil or Letraset lettering. Papers and Short Communications should have a brief Abstract. Key word list and CR categories on the leading page, with authors' affiliations as a footnote. The authors of an accepted paper will be asked to supply a brief biographical note for publication with the paper.

This Journal is Abstracted or Reviewed by the following services:

- ACM: Bibliography and Subject Index of Current Computing Literature.
- ACM: Computing Reviews.
- AMS: Mathematical Reviews.
- Data Processing Digest.
- ISI: Current Contents/Computer Science, Engineering, and Technology.
- ISI: Computer Science Citation Index.
- ELSEVIER: CompuMath Citation Index.
- GRENZGEBIETE: Zentralbibliot fur Mathematik und ihre Grenzgebiete.

Copyright © 1986, Australian Computer Society Inc.

Production Management: Associated Business Publications, Room 104, 3 Smail Street, Ultimo, NSW 2007 (PO Box 440, Broadway, NSW 2007). Tel: 212 2780, 212 3780.

All advertising enquiries should be referred to the above address.

Printed by: Ambassador Press Pty Ltd, Parramatta Road and Good Street, Granville, NSW 2142.