CONTENTS

SPECIAL ISSUE ON PROGRAMMING LANGUAGES
2-8   LOFE: A Language for Virtual Relational Data Base
      J.K. DEBENHAM and G.M. McGRATH

9-16  Communications Policy for Composite Processes
      C.J. BARTER

17-27 A Comparison of the Concurrency Constructs and Module Facilities of
       CHILL and Ada
       C.J. FIDGE and R.S.V. PASCOE

RESEARCH PAPERS
28-32  Commercial Programming Productivity — An Empirical Look at
        Intuition
        M.J. LAWRENCE and D.R. JEFFREY

33-39  Student Selection for Computing Courses
        D. WOODHOUSE

SPECIAL FEATURES
1      Guest Editorial
16,40  Book Reviews
The National Management Contest

The Contest

The National Management Contest provides invaluable management training by putting competing teams, each in effect a "company," in a simulated boardroom environment and forcing them to make decisions to a deadline under the pressure of competition. Detailed business situations, with their attendant problems and risks, are presented by ICL computer. The computer program is interactive and the updated reports produced for each "company" include market statistics, sales analyses, profit and loss accounts, balance sheets, and cash statements that reflect the performance of the teams.

The team with the greatest accumulated profit at the end of a "year's" trading will be the winner. Any number of teams may be sponsored by a company, university, college or other institution for promotional purposes, management training, academic interest, or personnel evaluation. Elimination rounds will be conducted by post. The final round will be held live in Sydney. The winning team will receive a trip to England. Each finalist will receive a prize.

Timetable

<table>
<thead>
<tr>
<th>Period 1</th>
<th>Period 2</th>
<th>Period 3</th>
<th>Period 4</th>
<th>Period 5</th>
<th>Period 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 April</td>
<td>9 May</td>
<td>23 May</td>
<td>6 June</td>
<td>20 June</td>
<td>4 July</td>
</tr>
</tbody>
</table>

Postal deadline dates

1. A team must comprise no more than four members.
2. Teams may enter under the name of an organization or under a pseudonym, and any number of teams may be entered from one organization.
3. Where a team from a single organization uses a pseudonym, the administrators will fully respect the anonymity of the team until the final stages of the contest, when the name of the organization will be disclosed.
4. Up to two members of any team may be changed during the running of the competition, but only those participating in the final will be eligible for prizes.
5. Teams from anywhere in Australia may compete.
6. Finalists will be flown to Sydney to compete in the national final. Their travel and subsistence expenses will be borne by the sponsors.
7. The 1983 timetable is shown. Any team that has paid its entry fee and been registered as a competitor, no refund will be made if this timetable is found to be inconvenient.
8. Entry forms must reach The Administrator, National Management Contest, ICL Australia, 98 Arthur Street, North Sydney, NSW 2060, by Tuesday, 22nd March, 1983. Each entry must be accompanied by a remittance payable to National Management Contest, for $250.00 a team. This fee includes the cost of two copies of the participants' manuals. Extra copies of the manual may be purchased for $5.00 each.
9. Participants' manuals will be sent to competitors as soon as their entries are received. Briefing sessions will be held in capital cities during March/April (details in Timetable.) These will mainly be question and answer sessions and will give competitors the chance to meet other competitors.
10. The administrators act for the sponsors and their decision is final in disputes and all matters relating to the conduct of the competition, including the acceptance of entries. They reserve the right to vary the rules as they see fit.

The Rules Governing Entry

Entry Form

Complete this form and send to the Administrator, National Management Contest, ICL Australia, 98 Arthur Street, North Sydney, NSW 2060, by Tuesday, 22nd March, 1983.

Name of team

If your team is participating for the first time, please tick box.

Team members

<table>
<thead>
<tr>
<th>Name</th>
<th>Team Mailing Address</th>
<th>Phone No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Signed

I hereby apply to enter the National Management Contest 1983 and accept the rules governing entry. My remittance payable to National Management Contest is enclosed.
Wittgenstein asserted that "the limits of our world are the limits of our language". While we might debate this dictum as it applies to life at large, we would surely agree that it is very appropriate in the context of computing. Most of us interface with computers by using some sort of language and the language sets the limits on the things that we can do. What we see is not the raw facilities of the computer itself, but an abstraction — a Cobol machine, or a data base machine, or an accounting machine, or whatever. The language helps us by providing a world that is closer to the problem domain with which we are concerned. On the other hand, it limits us by restricting our activities and by requiring us to conform to certain notational conventions. Many are the love-hate relationships that ensue!

Because of these factors, programming languages have attracted a great deal of interest and attention. For example, the ACM Special Interest Group on Programming Languages (SIGPLAN) has a membership of over 10,000 and is the largest of the SIG's. The lure of the subject may be attributed not only to our innate dependence on language in general but also to the inadequacy of the languages that we actually encounter when programming. We would all like to shake off the limits that others have placed upon us and to create an ideal world in which to work. Unfortunately, experience suggests that such worlds are very elusive. Most languages turn out to be compromises between conflicting goals and the design of really good ones is an art that few people can claim to have mastered. Most of us would be doing well if we could design a single feature.

The popularity of the subject was evident in the wide range of papers that were submitted for this special issue of the Journal. At the same time, it is notable that only three of the papers were finally selected and that these are all fairly long ones. Their length does not reflect any editorial policy. More probably, it is related to the difficulty of writing anything significant on a subject that is thirty years old. Debenham is concerned with the world of data bases and proposes a language that is based on logic. Barter takes us into worlds where groups of processes need to communicate with each other. Fidge and Pascoe compare the more advanced features of two recent, large-scale languages, CHILL and Ada, for which implementations are now beginning to appear.

The question of size is worth considering a little further. We have noted the length of the papers, but what about the languages themselves? Ada, for example, is often criticised for being too large. Yet this is primarily the result of its attempt to extend "the limits of our world". In these days, when so many programmers are being reared on a diet of Basic, the question arises: what sort of world do we want to live in? What investment are we prepared to make in introducing larger languages and in learning to benefit from the worlds that they open up? Too often we settle for a few weeks training and, consequently, for a world of meagre resources. A larger language calls for a much greater educational commitment; but if it extends the limits of our world in directions that are useful and satisfying, the commitment should surely prove worthwhile.

Jan Hext,
Guest Editor,
Macquarie University
LOFE: A Language for Virtual Relational Data Base

By J. K. Debenham* and G. M. McGrath†

A powerful query language for virtual relational data base is described. It is shown that this language has a natural restriction which serves the needs of the low level user. Keywords and phrases: data base, knowledge information processing, logic programming, query languages.


INTRODUCTION

It is assumed that the reader will be familiar with logic as a data base language, as discussed for example by Gallaire, Minker and Nicholas (1978) or Dahl (1982).

In this paper we describe our front end language, LOFE, which is based on logic and designed for virtual relational data base. It is well-known that data base updates, deletes and integrity checks can be phrased in logic (Gallaire, Minker and Nicholas, 1978): we will not discuss them. We will concentrate solely on the query facilities in our language.

First, we discuss the role of front end languages in data base, and note criteria for their design. We then define LOFE, a powerful data base query language for the high level user. A declarative interpretation of LOFE is given; this is illustrated with examples. Then the imperative interpretation is discussed and comments are made on an experimental implementation. Finally we show that a natural restriction of LOFE provides an adequate and simple language for the low level user, and that this restriction will prevent inefficient usage.

DATA BASE FRONT END LANGUAGES

The role of the front end language in our approach to data base design is as follows. We assume that the data has been analysed, that the important and high frequency queries have been identified, and that the data is represented efficiently as a data base in some data model. We also assume that this representation is capable of servicing the high frequency queries at least. In general this will involve the use of the procedural facilities in the model. The front end language will then consist of two parts, a procedural component and a notation. The procedural component will extend the power of the data model to be able to process the less important, low frequency queries. The notation will be designed to represent all the required query types, hopefully, in a convenient and uniform way.

Thus the design of a data base front end language must take into account the semantic schema of the data model employed, the available procedural facility (traditionally a host language), the technique used to design the data base and the range of required query types not directly supported by the representation together with the procedural facilities available within the data model. Our problem was to design a front end language to be used with our logic data base; our data model and design methodology are described in (Debenham and McGrath, 1982a). An important principle in the design of our front end language was that its computational mechanism should be similar to that of logic programming.

OUR LOGIC DATA BASE IMPLEMENTATION

Our implementation consists of three modules:

(a) the front end
(b) the logic unit
(c) the data base engine.

We now describe these briefly (Debenham and McGrath, 1982a for greater detail).

The data base engine is some (possibly conventional) device for storage in which the “real” data is stored. In our experiments, this was a CODASYL data base management system on a large Honeywell mainframe. The use of an established technology for data storage has the advantage that restart and recovery, and other support functions, are provided and are reliable. Within the data base engine, links are realised to support transactions of secondary importance (Debenham and McGrath, 1982a). These links are made available to the logic unit using inbuilt predicates identified by the design process. For comments on the use of relational implementations see Gallaire (1981) and Chakrarthy et al. (1982).

The logic unit contains a representation in Horn clause logic of the data base rules, or, “virtual” data together with integrity checks and so on. The real data structures in the data base engine, including the links referred to above, are made available to the logic unit using predicates which operate like the inbuilt predicates in PROLOG. However, unlike the fixed set of inbuilt predicates in PROLOG, the inbuilt predicates here are determined by the design methodology. The logic unit also contains an automatic theorem prover. As we will see, the theorem prover is also used by the front end, but with an augmented search strategy. In fact, the language of the logic unit is a proper subset of the front end language. Thus our distinction between logic unit and front end is a theoretical one; in the implementation they use the same theorem prover, and have access to the same set of data structures.

Copyright © 1983, Australian Computer Society Inc.

General permission to republish, but not for profit, all or part of this material is granted, provided that 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.

*School of Computing Sciences, NSW Institute of Technology, PO Box 123, Broadway, NSW 2007; + Telecom Australia. Manuscript received August 1982, revised November 1982.

The front end is our language LOFE, (LOGic Front End). In other words, LOFE is a language which operates effectively on our logic standard form, (which is essentially a binary relation form derived from 3NF [Debenham and McGrath, 1982a]), is compatible with resolution and the PROLOG search procedure, can be incorporated in our design technique (McGrath, 1981) and, in rough terms, extends the capacity of the query types to that implied by QBE (Zloof, 1975) in a reasonably efficient way. LOFE consists of a notation which includes some metalogical devices and a procedural component which is realised by augmenting the logic search procedure. These will be described shortly. The notation used for LOFE is substantially a development of the QBE format. In line with our design principle of maintaining similarities with logic programming, the procedural component is intentionally similar in flavour to the "find all" mechanism in PROLOG.

THE USER SPECTRUM

We have shown (Debenham and McGrath, 1982b) that LOFE caters to a full user spectrum in a uniform way. The spectrum described there was found to be appropriate for controlling data base usage in a substantial data base design project. See McGrath (1981) for the specification and the initial design of this project.

The user spectrum referred to may be summarised as:

(1) The basic user who performs routine enquiries of an identified well defined type.
(2) The low level user who presents a variety of simple queries which are directly supported by the data structures available.
(3) The middle level user who presents queries which require simple traversals of the data structures available.
(4) The high level user who can use all the features of LOFE which includes powerful logic programming.

In this spectrum, all users have access to "real" data structures and, depending on their status, may use them for more or less exotic tasks. The high level user has access to the powerful but potentially expensive tool of logic driven search, and to logic itself which may be used as a powerful programming language. Thus, the high level user may execute complex traversals of the data, and, for example, may construct procedures to perform complex computations, data manipulation and file searches. All this is achieved with a fair degree of homogeneity, and with acceptable efficiency.

The user spectrum referred to has one apparent discontinuity: only the high level user has access to logic driven search. We remove this apparent discontinuity by unifying the notions of "real" lists and "virtual" lists.

REAL AND VIRTUAL LISTS

We have shown (Debenham and McGrath, 1982a) how real lists, stored in some (traditional) data base system, may be accessed in a natural way by a system based on logic. For example, associated with the relation:

\[
\text{Item} (\text{item-no, item-desc})
\]

we might have the "inverse" relation:

\[
\text{Item-list} (\text{item-desc, item-no-list})
\]

which when given a particular item description will retrieve a list of item-no's of all items having that description. In our notation:

\[
\text{Item-list} (\text{item-desc, item-no-list})
\]

\[
\text{"pen"} \quad :X
\]

\[
:X \quad \text{would be a real list of item-no's of all pens. Unlike an ordinary logic relation such as Item, the Item-list inverse relation is in fact a function that will only operate when driven by its first argument.}
\]

If the "Item-list" inverse relation had not been available, then the "Item" relation could have been probed by a logic driven search:

\[
\text{Item} (\text{item-no, item-desc})
\]

\[
?: \quad \text{"pen"}
\]

where, as we will see, "?:" in this context means "find all by logic search", that is, by backtracking. In this case the set of answers found for ?: constitutes a virtual list. In general, the term "virtual list" refers to the (ordered) set of values found for a logical variable during some form of logic driven search, for example the backtracking procedure used in the "find all" search in PROLOG.

Real and virtual lists are indistinguishable to the high level user. For example, when probing the Item relation (see above) for the item-no's of all pens, if the Item-list relation had been available in the physical data structures then the backtracking mechanism would automatically use this list to find the successive values.

THE DEFINITION OF LOFE

We now define LOFE, a powerful data base query language based on logic. As we present it, LOFE is a language for the high level user with expertise in logic programming. It is not intended for day to day processing of high frequency queries. However we will show later on that LOFE may be restricted to operate on functions, rather than relations, that this restricted language, R-LOFE, provides a natural mechanism for phrasing queries which do not require logic driven search, and that it may be presented effectively to the untrained user. Thus LOFE (or restrictions of it) is usable by the whole user spectrum.

As we have said, the high level user may use logic as a powerful programming language to construct his queries. See Warren (1981) for examples. In addition, the user will have access to the powerful backtracking mechanism and to the metalogical features of LOFE.

LOFE consists of a notation and a procedural component which includes a set of metalogical features. We now describe these.

LOFE: THE NOTATION

We will only describe the notation adopted for query presentation. The definitions given below will be illustrated with reference to the following example of a query:

\[
\text{Item} (\text{item-no, item-desc}) \quad \text{Stock} (\text{item-no, qty-on-hand})
\]

\[
X: \quad \text{"pen"} \quad X: \quad ?:\quad \text{"pen"}
\]

which, as we will see, would find the quantity on hand of every "pen" in the data base.

Def. A query is a banner followed by a query form.
Def. A banner is a non-empty list of relations, each relation having one or more arguments, each argument being labelled with a domain name.
LOFE

In the above example, the first line is the banner and the second line is the query form. The Item relation has two arguments; the first is labelled item-no.

Def. A variable name is a single letter (subscripted if necessary), or the query symbol ?.
Def. A constant symbol is an alphanumeric string enclosed in quotation marks.
Def. A variable occurrence is a variable name followed by a (possibly empty) sequence of colons. The number of colons indicates the level of occurrence. Occurrences of a variable at its lowest level of occurrences are called base occurrences; other occurrences are called check occurrences.
Def. A query form is a non-empty list of variable occurrences and constant symbols, such that: (1) the list contains at least one variable symbol, (2) each element in the list is associated with a unique argument of a relation in the banner (usually written "beneath" it). (3) the query must be ordered so that the level of occurrence of any variable does not decrease at any point in a left to right reading.

In the above example, X is a variable name and X has two occurrences both at level 1; "pen" is a constant symbol and ? has one variable occurrence at level 2.

As we will see, condition (3) above is to prevent queries like:

\[ P(\text{X: \textit{\text{Y::}}}) \quad Q(\text{X: \text{Y}}) \]

which would probably cause meaningless thrashing between the relations P and Q.

As we shall see, the notation of level of occurrence of a variable is used to control the logical search, and enables the user to reduce the size of the search space considerably when compared with the "find all" solution space of PROLOG.

Def. The level of a variable is the level of its base occurrence(s).
Def. A query is said to have level n if the variable ? has level n.
Def. Variables with level no higher than the level of the query are called the template variables, other variables are called the check variables.

In the above example, X has level 1, ? has level 2, and hence the query has level 2. X is a template variable and there are no check variables. Note that a variable occurring without any colons has level zero.

As we shall see, the template variables determine how much of the space is searched by mechanising constructs such as "for all . . . find one ..". The check variables further reduce the search space by mechanising constructs such as ".. such that P if and only if Q".

AN ALTERNATIVE NOTATION

Our notation is very compact, we have found it convenient and, once mastered, meaningful. We stress that it is the declarative semantics of LOFE.

Hence the query:

\[ P(\text{X:::}) \quad Q(\text{\textit{\text{a::}}}, \text{X:::}) \quad R(\text{\textit{\text{a::}}, \text{X:::}}) \]

would be presented as:

\[ P(, \text{X:::}) \quad Q(\text{\textit{\text{a::}}, \text{X:::}}) \quad R(\text{\textit{\text{a::}}, \text{X:::}}) \]

Note that in this example, X is a check variable, at level 2, with two base occurrences and one check occurrence.

LOFE: THE DECLARATIVE SEMANTICS

A level n query expressed in LOFE may be read as follows: For the level zero variables identify one value, and for \( k = 1, \ldots, n, n+1, \ldots \) for the leftmost level \( k \) variable identify all values; for the other level \( k \) variables identify one value corresponding to each of these values such that the query is satisfied; subject to the conditions that for different values of the check variables, a literal containing a base occurrence of a particular check variable is satisfied if and only if any other such literal is satisfied; and, when these base occurrence literals are satisfied this implies that literals containing check occurrences of that check variable are also satisfied.

The answer to the query being the values identified with the variable name ?.

We now demonstrate the meaning of LOFE with some simple examples.

Example

\[ P(\text{X:::}) \quad Q(\text{\textit{\text{a::}}, \text{X:::}}) \]

would find a single ? and an X such that P and Q are satisfied. Recall that \( \text{\textit{\text{a::}}} \) is the constant symbol a.

Example

\[ P(\text{X:::}) \quad Q(\text{\textit{\text{a::}}, \text{X:::}}) \]

would find a single ? such that P is satisfied if and only if Q is satisfied. Note that ? occurs at level zero, and that X is a check variable.

Example

\[ P(\text{X:::}) \quad Q(\text{\textit{\text{a::}}, \text{X:::}}) \]

would find for each X satisfying P, a single ? satisfying Q.

Example

\[ P(\text{X:::}) \quad Q(\text{\textit{\text{a::}}, \text{X:::}}) \]

would find, for each X satisfying P all ?'s satisfying Q. In practice, the values found in answer to a query such as this could well contain a substantial amount of duplication. We will see that there is a metalogical device in LOFE for removing duplications. In fact this is achieved by enclosing the query variable in parentheses \{and\}, e.g. \{, ,\}.

We have found that queries frequently have the property that variables occurring in different relations are associated with the same domain name. Such queries are called homogeneous queries. For example:
Example

Contract ( #, item-no , supp-name )
X: "Parker"
Shelf ( #, dept-no , item-no )
?: X:

would find, for every item-number of those items supplied by Parker, all departments which sell that item. In this example, the only variable occurring more than once is X:, on both occasions it is associated with "item-no". Thus the query is called homogeneous.

Example

Contract ( #, item-no , supp-name )
X:: "Parker"
Shelf ( dept-no , item-no )
?: X::

would find, for every item-number of those items supplied by Parker, a department which sells that item. This example has the structure "for all . . . find one . . ." which would appear to be beyond the scope of QBE.

Example

Contract ( #, item-no , supp-name )
X::: "Parker"
Shelf ( #, dept-no , item-no )
?: X:::

would find the numbers of all departments which sell all the items supplied by supplier Parker, together with perhaps some other products.

Example

Contract ( #, item-no , supp-name)
X::: "Parker"
Shelf ( #, dept-no , item-no )
?: X:::

would find the numbers of all departments which sell only all the items supplied by supplier Parker.

Example

Shelf ( #, dept-no , item-no )
Contract ( #, item-no , supp-name )
X::: Z::
Stock-level ( item-no , amount )
X::: Y:::
Y::: >5

would find the numbers of all departments which stock only all the goods of any one supplier such that there are at least five items of all these goods in stock. In this example, the "greater than" predicate is provided just as in PROLOG.

If furthermore, each domain is complete, that is, if the range of values available in each of the various relations for the domain name is the same, then we may choose which relation to probe for a value.

**LOFE: THE IMPERATIVE SEMANTICS**

The procedural component of LOFE consists of: (1) the facility of logic programming, (2) our powerful backtracking mechanism, and (3) various meta-logical devices, for example, "to remove repetitions", "to count lists" and so on. We will not discuss logic programming explicitly: a good reference is Clocksin and Mellish (1981). We will now describe the backtracking mechanism and then the meta-logical features.

Our backtracking mechanism is an extended version of the left to right PROLOG (Clocksin and Mellish, 1981) strategy. We will shortly describe it, but before we do, we will describe the operation of "base" and "check" occurrences of a variable. Only relations in which variables occur at their base occurrence will be used by the backtracking mechanism to find new values for the variables. These values are checked (by resolution) in the relations where the check occurrences are. For example:

\[
P( X::, ) \quad Q(X::, )
\]

the P relation would be searched for values of X, each one found is only accepted if it satisfies the Q relation as well. Any base occurrence of a variable can be used to drive the search for values of the variable. As we will see, we attempt to choose the cheapest atom to probe. For example:

\[
P( , ) \quad Q(X::, ) \quad R(X::, )
\]

either the P or the Q relation could be searched for a value for X, each one found would only be accepted if it satisfies the other two relations as well. In effect we have specified that P and Q are to be searched concurrently. For an alternative, but less flexible approach to co-routining, see Clark and McCabe (1979). As we will see, with the exception of homogeneous queries which will be discussed, care must be taken with multiple base occurrences of a variable.

**Def.** The relation, containing a base occurrence of a variable, chosen to find a new value for that variable is called the base relation for that variable.

To describe the (operational) meaning of our notation we identify two types of query. First when there are no check variables, and second when check variables are present.

First, when there are no check variables, (i.e. all variables are template variables, and ? appears at the highest level in the query). Our backtracking algorithm is most easily described for the case when all the data is stored in the logic like a large PROLOG program. Answers are found to queries in the usual way; that is, resolution is used repeatedly on the query until values for the variables are found. This often occurs by resolution with an entry in a "real" relation. To backtrack at level k we identify the resolutions that fixed the previous set of level k variables and advance through the relations to find the next matching set: this is executed left to right as defined by the order of the relations in the query. In fact our data is not stored within the logic, and the backtracking operation just described is simulated. The complete search is achieved by using three procedures "initialise", "backtrack" and "saturate", defined by:

\[
\text{initialise level } k
\]
set the values for the level k variables to null

\[
\text{end}
\]
backtrack level k
  taking the variables of level less than k as found and fixed
  do
    backtrack to the next set of values for the level k variables using the level k base relations in a left to right order.
    if none then return "empty"
    else if there are level k+1 variables
      then saturate level k+1
  end
saturate level k
  initialise level k
repeat
  backtrack level k
until backtrack = "empty"
end

The whole process may now be described by:
initialise level 0
backtrack level 0
Note that variables at level zero are only instantiated once, and are then held fixed.

Second when check variables are present, (i.e. ? does not appear at the highest level in the query). The search operates as above, (i.e. hierarchically through the variable levels with resolution taking place on base occurrences before check occurrences of each variable), except that the values found for ? only contribute to the answer set if (1) the values found for the check variables are independent of the choice of base relation, and (2) the set of all values found in the base relations also satisfies the relations containing the check occurrences. E.g. in:

P(?, X::)
Q("a", X::)
R(Y::, X::)

X is a check variable and hence a value for ? contributes to the answer set only if (1) the set of all X's satisfying P is the same as the set of all X's satisfying Q and (2) if this whole set also satisfies the check occurrence in R.

We can now summarise the operation of our search strategy. For template variables we can choose which base occurrence to use, for check variables the choice of base occurrence must make no difference. For template variables if a check occurrence is not satisfied we backtrack to find a new value for the template variable; for check variables if a check occurrence is not satisfied we backtrack to find a new value for the query variable.

Our notation provides a powerful means of reducing the search space. In the hands of the trained user we believe it to be more effective than intelligent backtracking implemented by (Pereira and Porto, 1980) which was based on the work of Bruynooghe (1980). In the alternative notation, which was presented previously as a two dimensional schema, the operation of the search procedure can be seen as a top-down, left-right procedure directed by the schema.

Note that it is possible to execute a total search of the kind

(for all ... (for all ...( ... )))

for example consider

P(X::, Y::)
Q(Z::, ?::)

LOFE: THE METALOGICAL FEATURES

The metalogical features that we have incorporated in our experimental system comprise a device for removing repetitions from an answer set, denoted by placing { and } around the query symbol, and some principally arithmetic built-in procedures which operate in a way similar to the evaluable predicates available in PROLOG (see Clocksin and Mellish, 1981). We describe the operation of one of these.

The procedure Count(X,Y): the X argument must be occupied by a check occurrence of a variable within the query form. At the end of the computation, Y will be a numeric quantity equal to the number of references to Count.

It is easy to see how to specify similar procedures for summing (virtual) lists and so on.

IMPLEMENTATION

Our implementation of LOFE has been incorporated in the experimental system described by Debenham and McGrath (1981a). An important feature of that system is the availability, actually within the logic, of the inverse of certain relations identified by our transaction analysis (McGrath, 1981). These inverse relations are used when searching relations as described above, thus greatly improving efficiency.

Other efficiency measures include (a) not backtracking to non-key domains of normalised relations when the key is fixed, and (b) attempting to choose the cheapest domain to search in homogeneous queries. In the section on "query planning" in Warren (1981) a method is given for determining which relation to choose. This method cannot be conveniently included in our implementation because our physical data structures are buried in a traditional data base management system (CODASYL). Thus the costing exercise is more complex, but has been completed, although it does not take all aspects of the real data structures into account, for example, the paging mechanism. See the section "3.1 Query Optimisation" in Gallaire (1981) for a brief discussion of some other efficiency measures which could be included.

We have noted that LOFE may be implemented by refining the "find all" mechanism in logic programming. This has meant that LOFE was not difficult to incorporate in our system.

THE RESTRICTION OF LOFE

As we have stressed, the language LOFE described above has been designed for the trained logic programmer. We now describe a natural restriction of LOFE, R-LOFE, which is discussed and evaluated in Debenham and McGrath (1982b). There we show how the facilities may be rationed out to lower level users so as to prevent inefficient usage. Our approach to the low level user has been positive. We tell them what they can do rather than what they can not. This is achieved by exhibiting the logic data model.

The restriction, R-LOFE, does not have access to our backtracking mechanism, nor does it have the facility for entering the user's own logic programs. Thus the user of R-LOFE only has access to (possibly only a subset of) the relations in the logic data model. In general though, these will include some inverse relations. Furthermore, the relations that are available may only be "used functionally" as we will now describe.

All the relations in our data model have an identified
primary key. After all, the inverse relations have a "primary key", for example in

\[
\text{Meter-history ( account-number, meter-reading-list )}
\]

an account number functionally determines the list of meter readings. In a query form the construct associated with each (underlined) key domain may be a constant, or a variable whose value has been found by reference to a "previous" relation, where, by a previous relation we mean that the query will be processed in a strictly left to right order (see Debenham and McGrath [1982b] for examples). In addition, a query may drive a relation, for example

\[
\text{Contract ( #, item-no, "supp-name")}
\]

which would find the numbers of all contracts with supplier Parker. The quotation marks around the domain name supp-name mean that a secondary key back to the set of corresponding contract-no's is available in the data structures.

What of the queries that cannot be phrased in the restricted form described above? The short answer is that in our experience the queries from low level users naturally fall into this form as long as the database has been well designed, using transaction analysis, to support low level queries. However, if it transpires that the low level user will need to present queries which lie outside the restricted form, then this facility may simply be provided by a logic program which may, unknown to the user of R-LOFE, use backtracking.

Thus the user of R-LOFE will think naturally in real lists, whereas the user of LOFE thinks in terms of virtual lists. This led us to introduce analogous but different notation.

R-LOFE comprises a notation and a set of relations and operations. These are now discussed. Also a part of R-LOFE is its left to right operation described above.

**R-LOFE: THE NOTATION**

A list is either denoted by:

\[
\langle \text{variable name}\rangle
\]

which refers to the whole list, or by:

\[
\langle \text{variable name}\rangle, \langle \text{variable name}\rangle, \ldots, \langle \text{variable name}\rangle
\]

which denotes a list built up from the left in the usual way. For example, X, Y, Z will denote the list with first two elements X and Y and "tail" Z. Using this notation for real lists, the queries are presented beneath a banner as before, except that the primary key is underline. For example,

\[
\text{Meter-history ( account-number, meter-reading-list )}
\]

would retrieve the first and third values in the meter reading history of the account number 1234.

Note that the retrieval of particular elements of a list may be achieved in LOFE with a simple logic program.

**R-LOFE: THE RELATIONS AND OPERATIONS**

The elements of R-LOFE are thus elements and (real) lists. The following set theoretic relations are available:

- ordered equality, equal but for order, subset and proper subset.
- union, intersection, cartesian product, concatenation, removal of duplications from an answer list and an operation to extract a sublist having a specified property. The "usual" low level arithmetic facilities are provided.

**CONCLUSION**

Logic programming is assured of a prominent role in the Japanese Fifth Generation Computer Systems Project, (see Kowalski, 1982, and Moto-Oka et al., 1982). It is also clear that logic will have to be extended if it is to support fifth generation data bases (Kurokawa, 1982). In fact developments in Japan have already begun (Fuchi, 1982). The extensions to logic proposed here should be seen in this context.

The experiments with LOFE have demonstrated that a powerful query language may be implemented by refining the "find all" mechanism in logic programming. The language operates with acceptable efficiency and admits a restriction which may be used to prevent inappropriate usage across a full user spectrum. LOFE is based on positive logic; no attempt is made to implement "not-parallelism" (Kowalski, 1982), for example.

At the end of 1980 we embarked on a project to show that a large existing database could be effectively described, designed, implemented, maintained and operated using one language only, namely logic. The specification of our front end language completes that project. Future plans are to refine the whole procedure, and, in particular, to investigate the use of LOFE as a foundation for natural language query processing, see Dahl (1982) and Warren (1981).

**ACKNOWLEDGEMENT**

The permission of the Chief General Manager, Telecom Australia, to publish this article is acknowledged.

**REFERENCES**


GALLAIRE, H. (1981), Impacts of Logic on Data Bases, Proceed-
BIOGRAPHICAL NOTE

John Debenham is a Senior Lecturer in the School of Computing Science at the New South Wales Institute of Technology. He received his PhD from the University of Sydney in Computational Logic in 1971 before moving to the Institute. He has a long-standing interest in logic as a programming language and in logic as a data base language. Recent research interests include logic as a systems design language.

Mike McGrath is a Senior Project Manager in the Systems Development Section of Telecom's Information Systems Branch, New South Wales. He is currently supervising the development and implementation of two large national inventory/accounting data base systems. He joined Telecom in 1969 after graduating in Science from the University of Melbourne. He obtained his Master's degree in Computing from the New South Wales Institute of Technology in 1982. His research into the use of logic as a data base language was undertaken with the assistance of a Telecom postgraduate scholarship. Recent research interests include logic as a prototyping tool.

John Debenham is a Senior Lecturer in the School of Computing Science at the New South Wales Institute of Technology. He received his PhD from the University of Sydney in Computational Logic in 1971 before moving to the Institute. He has a long-standing interest in logic as a programming language and in logic as a data base language. Recent research interests include logic as a systems design language.

Mike McGrath is a Senior Project Manager in the Systems Development Section of Telecom's Information Systems Branch, New South Wales. He is currently supervising the development and implementation of two large national inventory/accounting data base systems. He joined Telecom in 1969 after graduating in Science from the University of Melbourne. He obtained his Master's degree in Computing from the New South Wales Institute of Technology in 1982. His research into the use of logic as a data base language was undertaken with the assistance of a Telecom postgraduate scholarship. Recent research interests include logic as a prototyping tool.
Communications Policy for Composite Processes

By C. J. Barter

The paper discusses group composition techniques, and the communications interfaces between processes when they are organised as a group. When a message is sent to a group of processes as a whole, one or more of the component processes may receive it. For each message type and group, a group policy expression determines the disposition of messages within the group, and a notation for group policy expressions is presented. The notation also provides for the encapsulation of one process by another without the use of shared variables. Policy expressions are also shown to provide a measure of control over message sequences, which is useful when specifying extended transactions in transaction processing systems.

Key words and phrases: concurrent programming, message passing, communications policy, process groups.


1. INTRODUCTION

The important aspects of concurrent language design are communications, synchronisation and composition of processes. The first two have been extensively studied, focusing on questions such as control, scheduling and non-determinism and problems such as deadlock, starvation and fairness. Less has been said about how complex processes may be composed from other processes, and ultimately from elementary sequential operations and communications primitives. This paper discusses group composition techniques, and the communications interfaces between processes when they are organised as a group.

When a message is sent to a group of processes as a whole, one or more of the component processes may receive it. We distinguish two aspects of group message reception in systems where messages are typed. Firstly, processes are typically provided with the means to select messages for reception, by scheduling arrangements such as system queues, or by user code involving local variables to choose between messages of different types. We define group input protocol to be the input behaviour of the group as a whole, for all message types. Secondly, we define for each message type, a group policy which determines the disposition of messages within the group.

We shall argue that policies have more to do with the transactions handled by groups than the reception of individual messages by processes, and are consequently better expressed at the group level. Because the control of group policy will be predicated on transaction attributes rather than process variables, and because the control issues seem simpler, a separate notation is proposed for group policy. The notation also provides for the encapsulation of one process by another without the use of shared variables.

2. PROCESSES AND MODULARITY

Language proposals for concurrent systems usually define a basic component, an asynchronous process with facilities for external communications and synchronisation. The process is basic in the sense that it is the building module of concurrent systems. The details of the proposals vary a great deal, and we shall mention some which have an influence on the way processes may be composed together.

One difference is whether communication is mainly by access to shared memory, or by message passing. In shared memory systems such as SimulatA (Dahl, 1970), Monitor (Hoare, 1979), Concurrent Pascal (Brinch Hansen, 1976), Modula (Wirth, 1980), processes communicate by writing and reading shared variables. Access to shared objects gives a tight coupling of processes and can result in inefficient implementations. Synchronisation can also be achieved by setting and testing shared variables, either by ordinary assignment or through special signalling facilities of the language.

Some early proposals which eschewed shared memory were PLITS (Feldman, 1979), Communicating Sequential Processes (CSP) (Hoare, 1978), Distributed Processes (DP) (Brinch Hansen, 1976), Actors (Hewitt, 1977). These promoted message passing in various forms on the grounds of simplicity, reliability and clarity of expression, at least with respect to communications and synchronisation. It is interesting to note that some of the most recent proposals such as Synchronising Resources (SR) (Andrews, 1981), E-CLA (Liskov and Scheifler, 1981), Modular Processes (MP) (Choi, 1981), allow shared variables (with the recommendation that they be used sparingly and with care). The sharing of variables occurs within an explicit grouping of processes (viz. the resource in SR, guardians in E-CLU and the node in MP).

Communications and synchronisation issues are often difficult to separate in particular language proposals, for it is frequently the case that both aspects are involved in the same language feature: for example, the input and output commands of CSP are the sole means of communication and synchronisation. These issues have been neatly separated by Choi (1981), where for each communication event there is a process which provides a service, and a process which is requesting a service (the sender of the message). Synchronisation is generally the concern of the sender of the message, and there are three possible arrangements, the
Communications Policy

no-wait send, the wait send and the remote call. With no-wait send, the sending process does not synchronise with the destination process, and continues execution after sending the message (e.g. PLITS). With wait send, the sending process synchronises with the receipt of the message by the destination process, then both processes continue independently (e.g. CSP). With remote call, the sending process synchronises with the completion of the service requested by the sender and invoked by the receipt of the message (e.g. DP).

From the point of view of the receiver, three kinds of service are identified, message service, procedure service and subprocess service. A message service simply receives the message, perhaps assigning values to local variables in the receiver, and the receiving process then continues normal execution. If the message requires a reply, it must be explicitly constructed and sent by the receiver as a new communication (e.g. CSP and PLITS). With procedure service, message reception invokes a procedure to handle the message, which may also construct the value of a reply (the "out" variables in DP and Ada [Ichbiah, 1979]). Lastly, a service may be provided by a process rather than a procedure, for greater concurrency. In MP, subprocesses are created dynamically to handle subprocess service requests, while in SR, all requests are handled by processes, but the processes are not dynamically created. The arrangements for sending and receiving described above are largely orthogonal, and all meaningful combinations have been proposed in the literature. The proposals for grouping processes advanced in this paper permit the construction of process groups which achieve all the arrangements for sending and receiving surveyed above.

Communication is mediated by arrangements such as sender-receiver pairing (as in CSP), ports (Balzer, 1971), message types (Milne and Milner, 1979), transactions (Feldman, 1979), constructions (Barter, 1978), pattern matching (Hewitt, 1979), and various notations such as Path Expressions (Campbell and Haberman, 1974), and Input Tools (van den Bos et al., 1981).

Major differences exist in the structure of the processes themselves, largely determined by the kinds of service provided. In CSP, the basic process is simply a list of sequential commands, using a non-deterministic guarded command notation to control input, output and ordinary sequential execution. The communications commands appear as in-line code. In contrast, DP provides a process with service procedures which may be called remotely, using a monitor-like discipline. A process may have a conventional process body as well, and the execution of the main body and the service procedures interleave in an unusual way (Welsh et al., 1980). Ada has both in-line message receivers (entries) and communications procedures, in an attempt to combine the advantages of CSP and DP. The proposals for grouping processes in this paper are independent of process structure; the example program at the end of the paper uses in-line code for services, but it is easy to see how the other kinds of service may be used.

2.2 The Composition of Systems of Processes

A simple way to compose processes is to form a loose grouping of processes within a common communications environment, with a global convention for process names and messages. Various refinements of this model have been proposed which provide ways of restricting the scope of these names. For example, Milne and Milner (1979) use an operator to restrict the visibility of port names. CSP uses textual nesting of process (parallel commands) and Algol-like scope rules for access to variables in different processes. Thus there are shared variables, but a "disjointness" property ensures that there is no shared write access.

Textual nesting has also been used to construct hierarchical groups of processes with scope rules on process names to hide the process structure of groups; from the point of view of the sender of a message, the destination is simply a process. The destination may in fact be a group of processes, and the primitive process within the group which receives the message is determined by the group composition and the type of the message (Barter, 1978). Structure hiding has been achieved in CSP by the use of a "hole-in-scope" rule whereby a process name is known in all of the enclosing processes but not in the named process itself; structure hiding is used in a stepwise refinement programming methodology, where each refinement step adds an additional process to a group in order to modify the group behaviour (Hoare and McKeag, 1979). Shapiro has developed this methodology through an extension of CSP which adds some flexibility to the naming conventions for processes and message constructors, and applied it to a large system design (Shapiro, 1980).

Some recent proposals (SR, E-CLU, MP), influenced by the additional considerations of distributed systems, have defined a middle-level structure involving a group of processes, and some shared objects (usually variables). This grouping may be regarded as the counterpart of a processor node in a network of processors. The authors of SR and MP regard these special groups as being different from processes, and do not allow arbitrary nesting of processes and groups.

The most interesting composition ideas have come from languages which were not primarily intended for concurrent programming, but had a strong object-oriented approach and with particular applications in mind Simula67, Smalltalk (Kay and Goldberg, 1977 and Ingalls, 1978), Thinglab (Borning, 1981) and Lisp Machine Flavors (Cannon, 1979, Weinreb and Moon, 1981). The reason is that without the complication of concurrency it is natural to exploit the advantages of shared memory, and this has been done in most imaginative ways. In these languages we see the composition of processes to mean the actual merging of state spaces, process bodies and service procedures, involving a much tighter composition than the loose coupling described earlier. Simula67 introduced the idea of class concatenation, where a class could inherit the attributes of another class. By this method, superclass hierarchies could be constructed. The original intention was to provide language support for program modularity, where the modules (classes) would correspond closely with the conceptual layers of a system design. Class concatenation also foreshadowed another important kind of group composition where one object encapsulates another (see later). The idea of class introduced by Simula67 has been extraordinarily influential, even though some of its details have been criticised (the details of concatenation, Algol scoping and remote accessing of class attributes).

2.3 Superclass Schemes and Process Composition

Languages such as Simula 67 and Smalltalk allowed class objects to inherit attributes (procedures, methods and even variables) from other classes by class concatenation. However, the structures which can be built this way are
strictly hierarchical, and may be classified as single superclass systems. Multiple superclass systems such as Thinglab and Flavors allow inheritance lattices. The inheritance mainly applies to the inheritance of methods (which may be viewed as message services), although there may be some state space sharing as well.

In the Flavor system, a flavor (a class-like specification) can be constructed from other flavors by a technique called "mixing". A mixed flavor may have components such that more than one component has a method with the same name; an important contribution of the Flavor system is that if an object of the mixed flavor is instantiated, and a method of that object is invoked, more than one method may be executed from the set of component methods. The programmer selects one of a set of method combinations to control which component methods are executed, and in which order. The default method combination is called daemon combination which allows methods to be classified as before, primary or after methods; all before methods are handled first, then the single primary method, and finally the after methods are handled. Within the before and after groups of methods, method order is determined by the order in which component flavors are mixed to form the composite flavor (in fact a tree walk order). In every case, the message handling policy is statically determined by the text of the flavors and methods. Our proposal differs in several ways. Firstly, the specification of group policy is separated from that of group composition; secondly, policy is expressed only at the group level, and not within methods, and finally, dynamic policies will be allowed (dynamic in the sense that method ordering can change depending on the execution environment).

3. COMMUNICATIONS POLICY

In this section we address a question which is fundamental to any proposal for forming processes into groups, namely how is a message received within a group when it is sent to the group as a whole? This question may be simplified by using message types and ensuring that there is always exactly one process in the group able to receive messages of that type. We define group policy to be a specification of how messages of a given type will be received within the group, and this will be the key concept upon which other ideas concerning transaction handling and encapsulation will be based. We shall now examine more flexible policies such as broadcasting to all processes able to receive the message, or the selection of some subset of those eligible. Of course policy may be implemented in an additional "policy manager" process (dispatcher) associated with the group, but we shall describe policies in a descriptive notation through policy expressions, examples of which now follow.

Consider a group of processes P, and a message type "msg". Let (P1, P2, ..., Pn) be those processes of P which accept messages of type msg. Three basic policies are now given by example.

- A policy of selection for P is written:
  
  \[
  \text{policy msg: (P1 \cap P2)}
  \]

  Only processes P1 and P2 are considered as possible destinations for messages of type msg. The choice between P1 and P2 is non-deterministic, all other things being equal. (An implementation could choose the first process ready to receive.) For example, consider a print request sent to a pool of print resources, and the request may be satisfied by any member of a

subset of printing resources (e.g. those three which are nearby). The policy for the group "printer-pool" may be expressed:

  \[
  \text{policy print-request: (print-resource(1) \cap print-resource(2) \cap print-resource(3))}
  \]

- A policy of broadcasting for P is written:
  
  \[
  \text{policy msg: (P1 \cup P2)}
  \]

  Both P1 and P2 receive the message, but the order is unspecified. For example, a request for some services may also be logged on an accounting file, and registered with a load monitor. The policy for such an encapsulated printer pool may be expressed:

  \[
  \text{policy print-request: (printer-pool \cup accounts \cup load-monitor)}
  \]

- A policy of serial broadcasting for P is written:
  
  \[
  \text{policy msg: (P1 \cup P2)}
  \]

  Both P1 and P2 receive the message, but process P1 must complete the processing of the message before P2 starts. Serial broadcasting is likely to be most useful in groups with shared memory; for example, it is the default policy for calling combined methods in the Lisp Machine Flavor system. Both forms of broadcasting require a convention when used with remote call, to determine which service sends the reply; see later for default policies.

An important degenerate case is policy msg: (P1), which simply directs all messages of type msg to P1.

A policy expression describes the disposition of every message received by the group, and therefore may be regarded as a repeating construct. (Additional notation will be introduced later to specify repetition of inner components.) A policy expression for a group cannot directly affect the reception of messages by that group; policy only determines the disposition of a message when it is received by the group.

Compound policy expressions may be formed in three obvious ways:

- By nesting groups as in:
  
  \[
  \text{policy for group P is policy msg: (P1 \cap P2)}
  \]

  \[
  \text{policy for group Q is policy msg: (Q1 \cap Q2)}
  \]

  where a message for group PQ is sent to P1 or P2 and also to Q1 or Q2.

- By expression nesting, e.g. policy for P:
  
  \[
  \text{policy msg: (P1 \cup P2) \cap (P3 \cup P4)}
  \]

  where a message for group P is sent to P1 or P2 and also to P3 or P4.

- As a sequence of policies:
  
  \[
  \text{policy msg: (P1 \cap P2) \Rightarrow (P3 \cup P4)}
  \]

  The initial policy is (P1 \cap P2), which directs one message to either P1 or P2. The policy then changes to P3 \cup P4, and after that the policy expression repeats. A sequence of policies achieves a similar effect to actor replacement (Hewett et al., 1979).

In a language using policy expressions, some convention for default policy would be useful, and perhaps some way of defining message type aliases (a reasonable default would be the selection of a single receiver, using a static criterion such as text order in the group description, or a dynamic selection over all eligible processes).

3.2 POLICY MODEL

The semantics of policies are now given as code for a virtual group message handler. The notation is CSP-like, where "P!msg" is the usual CSP wait send-of message
Communications Policy

"msg" to destination P. The notation is extended so that "P.msg" signifies a remote call to P: if a process Q executes a remote call "P.msg" which activates the guarded command "?msg -> command-list", then "P.msg" in Q does not terminate until "command-list" in P does. Also, the input command "?msg" differs from CSP in that it does not name a sender, but will receive messages of the appropriate type (Barter, 1978). The three basic policies are:

- **policy msg:(P1 [] P2) => [?msg -> [true -> P1.msg [] true -> P2.msg]]**
- **policy msg:(P1 [] P2) => [?msg -> [true -> P1.msg [] true -> P2.msg]]**
- **policy msg:(P1 [] P2) => [?msg -> [P1.msg [] P2.msg]]**

Note that all the virtual handlers have the same structure, [LHS -> RHS], where LHS is always the virtual input command for the group, and RHS is a simple transformation of the policy expression. Virtual handlers for nested policy expressions are similarly constructed by repeated transformation:

- **policy msg:((P1 [] P2) / / (P3 [] P4)) => [?msg -> [true -> P1.msg [] true -> P2.msg]]**

Sequences of policies result in sequential composition of virtual handlers; the operator ">>" takes precedence over the others in deriving the virtual handler:

- **policy msg:(P1 >> P2) => [?msg -> P1.msg]**
- **policy msg:(P1 >> P2) => [?msg -> true -> P1.msg [] true -> P2.msg]**
- **policy msg:(P1 >> P2) => [?msg -> true -> P1.msg [] true -> P2.msg]**

4. COMMUNICATIONS PROTOCOL

The meaning of a process may be given in terms of its input-output behaviour (Milne and Milner, 1979). Behaviour can also be expressed as the set of all possible communication sequences (Hoare, 1978). In the Actor model of concurrency, an actor receiving a message may change its local state, send messages to other actors and create new actors. The arrival of a message at an actor is called an event, and local time for an actor is the arrival of events. Message sending is not important in the event ordering as the model is asynchronous. However, an event can cause a message to be sent, and hence cause another arrival event; in which case the first event is said to activate the second event. Communications between actors is represented by such activation orderings. The meaning of a program is given by the combined ordering (Hewitt et al. 1979, Clinger, 1981).

In this paper we are interested in control over input messages, and input protocol will mean just the input behaviour of a process. We shall refer to input protocol as protocol for short (this is a narrower definition than used in the literature on networks).

The protocol of a process is determined by the mechanisms within the process for selecting the next message to receive from a set of pending messages. These mechanisms depend upon an ability to discriminate between messages by some global measure such as arrival ordering (Hewitt et al., 1979), or on the basis of message attributes such as type, sender and priority. Arrival ordering is sometimes used in combination with message attributes as a subsidiary selection criterion (PLITS, COSPOL [Roper and Barter, 1981]). Four basic mechanisms have been used:

1. Firstly, there are processes which have a process body which controls the selection of the next message to be received, using in-line receive commands (the input command of CSP and the entry of Ada). Local variables can be used to control message selection by normal flow of control and by guarding input commands.
2. Secondly, there are schemes which have service procedures or processes which are directly accessible to other processes, without the control of a "main body" (e.g. SR, E-CLU). Local variables can be used to guard access to services.
3. Thirdly, message reception can be entirely determined by arrival ordering; in Actor languages, messages can be typed (explicitly by pattern), and the pattern may be matched against a set of alternative actions, but the pattern matching determines the body which is to be executed, not the message to be selected. While languages such as CSP have "choice non-determinism" which affects message selection ordering, Actor languages only have "arrival non-determinism" due to asynchronous communication (Clinger, 1981).
4. Finally, there are languages which use a separate notation to control message selection, such as Path Expressions (Campbell and Haberman, 1974) and Input Tools (van den Bos et al., 1981). Path Expressions are based on regular expressions, using the names of the service procedures of a resource. As well as scheduling service requests, Path Expressions also control the amount of concurrency in the resource services.

The Input Tool Process model provides an event-driven model based on input events, controlled by input rules. An Input Tool has a name, an input rule, a tool body, and an initialisation section. Tools may be composed in parallel, or nested. An input rule is based on a regular expression notation, using the names of other tools. If an input rule is matched, the tool body is executed, and that tool name may cause further matching in an input rule at a higher level. Direct communications between processes involves a match between a send command in one process and a receive rule in another (a tool may specify a receive rule instead of an input rule). A parser uses the input rules to dynamically construct the currently "active" structure of input tools (a tree for each process, whose terminal nodes are basic tools with receive rules). Inputs which do not match the current structure are ignored. Thus input rules control input protocol and, as we shall show, some aspects of what we have called policy.

Input rules can be used to control both policy and protocol (indeed the authors do not draw the distinction). Because the Input Tool model has strong similarities with our policy proposals and strong differences with our treatment of input protocol, we shall discuss the model in some detail.

4.2 The Input Tool Model

The example of a printer server is given (van den Bos et al., 1981).

```python
tool printer = input (first-line: [more]: source -> line$) $ end
  bool more; process set source;
  tool first-line = input line end
  if more
    then source := sender
  fi
  tool line = receive string msg;
```

The input rule uses "::" for sequences of matches, "$" for repetition, and 
<boolean-expression>>: for guarding. The notation "source \(\rightarrow\)" restricts input messages to be from a particular sender, in this example it is the one bound by the assignment "source := sender ".

When the tool "printer" is activated, the parser activates the tool "first-line", and through it, the tool "line"; "line" is a basic tool which receives a message "msg", which matches its receive rule, and so the body of "line" is executed. This matches the input rule of "first-line", and so its body is executed, and the component "first-line" of the top-level is matched. The parser now moves to the next component of the top-level input rule: this will be "more: source \(\rightarrow\) lines" if the boolean guard "more" is true, but if "more" is false that component will be invisible to the parser, and so the next component will again be "first-line".

A second example shows input rules used to direct a message of the same type to alternative tools:

```plaintext
tool squash = input |go-on|: (star + nostar)$ end

... tool star = input character(c): | c = | end

... tool nostar = input character(c): | c <> | end

if c = EOF then go-on := false fi

end ...

init ... ; go-on := true end

end
```

The operator "+" specifies a choice between two tools, and the input rule "character(c): | c = |" uses a post-test on the value of the parameter "c", so that the post-test must succeed if the rule is to match.

An example of a bounded buffer is given to illustrate an input rule controlling a simple input protocol; the example is given here in abbreviated form:

```plaintext
tool buffer = input (kount <= size | put + kount >0 | get)$ end

... tool put = receive char c;

end ...

tool get = receive;

end ...

end
```

The parser does not activate the tool "put" if the buffer is full, and similarly does not activate the tool "get" if the buffer is empty. The boolean guards are computed within the bodies of put and get.

The example programs show three uses of input rules. The first example shows an input rule specifying a message policy: an input line is processed by either one or both tools. The purpose is to provide some encapsulation of tool "line" by tool "first-line". This particular encapsulation does not generalise well; encapsulation will be treated later.

Communications Policy

The second example also specifies policy for input characters, using the tool "star" or "nostar" depending on the data. In both examples, the input rules affect policy but not input protocol. In the third example, the input rule controls protocol in the sense that it directly determines the scheduling of requests. In all three examples the input rules exercise control through shared variables.

The use of program variables in these expressions allows arbitrary interactions between the expressions and the code of the processes controlled. But typical protocol and scheduling descriptions do involve variables which are local to (and sometimes shared between) the processes concerned; this is a strong reason not to place these descriptions in a separate expression, but to leave them in the code of the processes themselves. On the other hand, we shall show that policies have less to do with individual processes and their variables, and more to do with groups of processes and message sequences; for this reason we shall argue that group policy is better placed in a separate description associated with the group, and that a separate notation is useful for its description.

Because the method of process composition suggested in this paper does not involve message re-scheduling, the protocol of a group is simply the merge of individual protocols (i.e. all orderings which preserve the partial ordering of the component processes). Next we show how policy and protocol may interact without using shared variables in either the processes or the policy expressions.

5. POLICY-PROTOCOL INTERACTION

Consider a group of children and gifts arriving.

The group is: (Sharon, Carol, Jenny, Michael)

The messages are: (gift, boy-gift, girl-gift)

Some example policies are:

- policy gift:(Sharon ▶ Carol ▶ Jenny ▶ Michael) - i.e. take turns.
- policy girl-gift:(Sharon [] Carol [] Jenny) — i.e. choice
- policy boy-gift:(Michael) - i.e single receiver

The three policies are independent, e.g. the policy for messages of type "gift" has no influence on the policy for messages of the type "girl-gift".

Consider the following policies:

- policy gift:(Sharon / Carol / Jenny / Michael)
- policy gift:(Sharon ; Carol ; Jenny ; Michael)

Incoming messages of type "gift" are received by the four group members, concurrently in the first policy and sequentially in the second.

The policy for "girl-gift" is not fully determined by the policy expression; an implementation may have some additional criteria for making the choice, such as choosing the process which has been waiting longest for a message of that type. (An alternative strategy is to choose without consideration of whether processes are ready or not, and to wait if the chosen process is not ready; this can lead to more deadlocks than the first strategy). Rather than regarding the previous as an implementation issue, the selection method could be part of the language definition and exploited to schedule message reception or synchronisation; but this encourages a dangerous interdependence between processes in a way which undermines modularity and clean interfacing. We now examine some policy-protocol interactions which depend only on more general aspects of communications:
A process may terminate, which is a most drastic change of protocol. The most desirable behaviour with respect to group policy if a component of the group terminates will depend on the composition method. If the terminated component is composed with the policy operators "/" or "\[("' then the process may be dropped (dynamically) from the policy provided that there is some live component to receive the message; if not, the group should abort.

A process may close a typed message service, which is similar to termination, but only with respect to that message type and the corresponding message policy.

The policy for a group is by definition a repeating construct, and as such associates with message sequences rather than a single message. The policies given earlier could have made this explicit with a repetition operator such as the Kleene star, e.g. policy msg((P1 | P2)*).

An explicit operator is necessary to express repetitions of policies within sequences of policies, e.g.

\[
\text{policy msg:}(\text{P1} \mid \text{P2}^* \Rightarrow (\text{P3} \mid \text{P4}^*))
\]

In this policy, a sequence of messages is dispatched under the policy (P1 \mid P2)*, before the policy changes to (P3 \mid P4)*. Some means of breaking the sequence is required, and we propose an explicit break-policy signal rather than a test on a program variable. A logically associated sequence of messages is usually called a transaction. It is useful to strengthen the attributes of a transaction by sender-receiver bindings, and two operations are proposed for this purpose: attach-sender and break-sender.

break-policy has the following effect: if the current policy is part of a sequence of policies and not the last policy in that sequence, the next policy becomes the current policy; otherwise the break is passed up to the next level, if any. When there is no "next level up" (the group is not a component of another group), the policy at that level does not signal a break, but restarts the entire policy expression at that level. (Repetition in policy expressions and the break-policy operation are similar to catch and throw in some versions of Lisp (Weinreb and Moon, 1981) attach-sender restricts all further messages received by the group to be from the sender of the last message, and this prevails until break-sender is executed within a process of the same group.

The three policy operations described above will be illustrated in an example after a discussion of encapsulation.

6. ENCAPSULATION

Simula67 supports a form of encapsulation through class concatenation; a special symbol inner is used to mark a point in the code of the body of a process, to identify where the code body of the encapsulated process may be regarded to notionally execute. A similar encapsulation facility with respect to method bodies is available in the Flavor system (wrappers).

Hewitt's serialisers/guardians may be used to encapsulate a resource process by intercepting and re-scheduling all communications with the resource. The guardian acts on behalf of the user of the resource. The purpose of the encapsulation is to enforce a stronger protocol than that of the resource itself; i.e. the resource may have been designed without considering the possibility of careless or malicious use, and the encapsulation is then designed to compensate for this.

Although shared variables are often exploited to provide the kinds of run-time environment encapsulation possible in the languages described above, we shall only discuss sharing through the communication environment, rather than through process state spaces. The most important communication attributes to be shared are those to do with transactions involving more than one message passing event. Examples of transaction attributes of interest such as policy-sequence bindings and sender-receiver bindings have already been mentioned.

We now introduce the construct inner to provide some encapsulation abilities in groups. The name inner is borrowed from Simula67, but because it is used without access to shared variables, its semantics is different from that in Simula. A process receiving a message will execute its command list (or service) up to the occurrence of the inner marker, and the process is then suspended. When the entire policy expression is complete, suspended processes are re-activated in the reverse order to that in which they were suspended. Thus processes may be regarded as nested or encapsulated with respect to message passing.

Thus an encapsulating process encapsulates the transactions or message sequences of the encapsulated process, rather than its execution environment; but this is often what is required anyway. A common use of encapsulation is resource locking, where only requests of the current transaction are allowed to access the resource, and all other requests are locked out for the duration of the transaction. To achieve this effect, the encapsulating process could contain the following code:

```plaintext
... lock; inner; unlock; ...
```

In the following example, inner is used to illustrate head and tail encapsulation in the printer problem. The example is an extended version of the earlier printer example, with the added requirements that each file be printed with a header and a trailer, both containing information extracted from the first line of the file, and that empty files should not cause a page-skip. The programming language used is the same as that used for the description of virtual group policy handlers (Barter, 1978) for the sake of example, and it is not intended that the group policy model associate with any specific language.

```plaintext
[Printer::
  group-members : (Newfile, Printlines, ...)
  policy line : (Newfile* >>> Printlines*)
     ... policies for other message types
]
```

```plaintext
[Newfile ::
  [*?line ->
    [line.eof -> skip
     not line.eof -* attach-sender;
     print-header(line);
     break-policy;
     inner;
     print-trailer(line);
     page-skip;
     break-sender
    ]
  ]
]
```

```plaintext
[Printlines ::
  [*?line ->
    [line.eof -> break-policy
     not line.eof -* print(line)
    ]
  ]
]
```


Communications Policy

The modularity achieved is typically that which is to be expected from careful encapsulation. The process Newfile only performs operations at the file level, either empty ones which are skipped, or non-empty ones which have headers, bodies (for which inner is a surrogate), and trailers. The process Printlines just handles sequences of lines under some prevailing policy, breaking at end-of-file.

Note that the sender-receiver binding is now handled in the same process, and that the header and trailer procedures use the same value of line as their parameters.

CONCLUSIONS

Message policy has been defined to be the description of the disposition of messages of the same type, when received by a group of processes. Group policy applies to all the processes of a group, but to a single message type. It is proposed that group policy be specified in an expression which is separate from the code of the processes of the group, and in a separate notation. Separate specification seems natural, for policies are associated with transactions and message sequences rather than the details of processes; for this reason it is possible to write policy expressions which are independent of process state variables, and as well use a simpler control notation based on regular expressions.

Input protocol, on the other hand, applies to single processes (or a group as a whole) for all message types. When policy aspects are separated from input protocol, scheduling is what usually remains, and scheduling often has strong associations with process state variables; for this reason it is often difficult to specify protocol expressions without using control constructs which access process state variables. Accordingly, we leave control over protocol in the code of the processes themselves.

Encapsulation of processes is presented with an unusual emphasis on the transactions and resources which associate with an encapsulated process rather than the state space of the process environment. This is due to the notion of encapsulation without shared variables, and to the association between group policies, message sequences and transactions.

We have tried to avoid commitment to any particular language with the general message-passing group surveyed, though there are important interactions which will affect group composition and policy expression, as well as implementation (e.g. the presence of remote call in a language which is separate from the code of the processes of the group, and in a separate notation. Separate specification seems natural, for policies are associated with transactions and message sequences rather than the details of processes; for this reason it is possible to write policy expressions which are independent of process state variables, and as well use a simpler control notation based on regular expressions.

ACKNOWLEDGEMENTS

Marc Shapiro was an invaluable colleague in the early stages of this work, and contributed the name “group policy” to the notion of controlling the disposition of messages, within a group. Mike Brady read many drafts, and tried to steer me down the path of rigour with a mixture of criticism and encouragement. Thanks to Howard Cannon for Flavors and the LISP Machine windows which vindicate them, and to Richard Stallman for discussions which suggest that the ideas presented here may be most interesting in a LISP environment. Thanks are also due to William Kornfeld, Maurice Herlihy and Pierre Lescanne.

REFERENCES


The papers cover a wide range of research goals and applications of image modelling. The need to develop models for images generally relate to the design, refinement and evaluation of algorithms for image processing. Applications for such models therefore arise in all aspects of image processing. In the book, ten papers are concerned with segmentation techniques based on edge and texture representations, while other papers address such topics as symbolic description of image and graphic data, image restoration, image coding and image classification.

Various kinds of models are presented in the papers. Some are based on mathematical and statistical treatments of image representations and processes; for example, mathematical functions for intensity edges, co-occurrence matrices for texture. Other models are derived from such theories as random sets and fields, geometric probability, and integral geometry, and stereology.

The book provides an excellent overview of the state-of-the-art in image modelling. It shows the diversity of research interests, and the generally poor performance of current modelling techniques. Part of the reason for this performance (indicated by Tenenbaum, Fischler and Barrow) is that the models fail to account for the scene conditions underlying an image. It is difficult for example to characterise the textures produced by shadows on rough surfaces without modelling the scene generation and viewing conditions. The papers are therefore disappointing to the extent that they do not discuss the purpose of their approaches to image modelling and do not provide adequate evaluations of their models.

The book is of interest to those who are developing image processing techniques and those who are looking for a fertile area to apply mathematical and statistical theories.

But why buy the book? The papers constitute Vol. 12 of Computer Graphics and Image Processing, which should be neatly bound in your library.

J.F. O’Callaghan,
CSIRO Division of Computing Research

Communications Policy

The book possesses an excellent index and while at $36.20 it appears a little expensive, it may yield more lasting value than a good lunch.

M.J. Lawrence,
University of New South Wales

Book Reviews


The papers cover a wide range of research goals and applications of image modelling. The need to develop models for images generally relate to the design, refinement and evaluation of algorithms for image processing. Applications for such models therefore arise in all aspects of image processing. In the book, ten papers are concerned with segmentation techniques based on edge and texture representations, while other papers address such topics as symbolic description of image and graphic data, image restoration, image coding and image classification.

Various kinds of models are presented in the papers. Some are based on mathematical and statistical treatments of image representations and processes; for example, mathematical functions for intensity edges, co-occurrence matrices for texture. Other models are derived from such theories as random sets and fields, geometric probability, and integral geometry, and stereology.

The book provides an excellent overview of the state-of-the-art in image modelling. It shows the diversity of research interests, and the generally poor performance of current modelling techniques. Part of the reason for this performance (indicated by Tenenbaum, Fischler and Barrow) is that the models fail to account for the scene conditions underlying an image. It is difficult for example to characterise the textures produced by shadows on rough surfaces without modelling the scene generation and viewing conditions. The papers are therefore disappointing to the extent that they do not discuss the purpose of their approaches to image modelling and do not provide adequate evaluations of their models.

The book is of interest to those who are developing image processing techniques and those who are looking for a fertile area to apply mathematical and statistical theories.

But why buy the book? The papers constitute Vol. 12 of Computer Graphics and Image Processing, which should be neatly bound in your library.

J.F. O’Callaghan,
CSIRO Division of Computing Research


The sub-title of the book gives an accurate guide to its contents and objective. These have been clearly oriented to the needs of the executive who finds himself as a manager or user in close proximity to the computer for the first time.

The author deals lucidly with the topics of cost benefit analysis associated with purchasing a computer or commissioning a new systems project, and distinguishes carefully between efficiency and effectiveness of computer applications. There is also a short chapter, of perhaps less general interest, on the thorny issue of pricing computer services.

Unfortunately the chapter on systems development is inadequate to provide executives with a full appreciation of the need for a sound development methodology. The lifecycle concepts presented in this chapter do not stress the successive reviews and approvals that take place as the development project progresses from stage-to-stage. In addition, a grab-bag of techniques including structured analysis, HIPPO, software engineering, chief programmer teams, decision tables and simulation are all briefly touched on in this chapter but little appreciation of the tools is communicated. It is likely that the main advantage will be to arm the executive with a few new buzz words of jargon to impress his colleagues.

The chapters on planning present some good material but suffer a little from lack of an overall framework, possibly reflecting the fact that a number of these chapters were adapted from journal articles previously published by the author.

Two oversights are the lack of treatment of project management and control, and no discussion of the heavy drain on resources from the inevitable continuing maintenance of applications systems. Both these topics can significantly impact computer productivity.

The book possesses an excellent index and while at $36.20 it appears a little expensive, it may yield more lasting value than a good lunch.

M.J. Lawrence,
University of New South Wales
1983 National Computer Conference

U.S.A

May 16 – 19 1983
(Monday to Thursday)

Held at:
Anaheim and Disneyland Convention Centers,
Anaheim, California.

Sponsored by:
American Federation of Information Processing Societies, Inc., Association for Computing Machinery;
Data Processing Management Association; IEEE Computer Society; Society for Computer Simulation.

Australian arrangements:
Sponsored by: Australian Computer Society.

Travel & Accommodation:
San Michele Travel Pty. Ltd.— Sydney Perth Adelaide
Melbourne & Brisbane.

Official air carrier: Qantas.

Reservations & Data
Sydney 02/221 2222
THEME — "The Emerging Information Age; Computers, Communication and People"

- one of the most extensive exhibitions of computer hardware and software to be seen in the world
- Conference has 80 sessions in 11 different tracks ranging from academic and highly professional papers to office automation and microcomputers
- Professional Development Seminars (a conference within a conference) offers a further 22 sessions ranging from new technology to data security and computer graphics
- this is the one worldwide conference available yearly which offers equal scope to the professional or to the businessman who is planning extension or installation of computer equipment.

San Michele Travel Pty. Ltd.

(Official Australian organisers to the Conference) will also be providing the following services: *

- Two senior executives will be attending the Conference —
  (1) Mr. J. Forsyth — with 14 years overseas travel experience — to assist with onward travel arrangements, telex contact, urgent business messages and providing liaison on a daily basis.
  (2) Mr. J. Pozdena (System Services) — to assist with contacts to computer companies and sales links throughout the U.S.A.
- Professor Phillip Grouse (University of N.S.W.), will be attending the Conference assisting the interests of the Australian Computer Society and San Michele Travel. He will be available daily for discussion on technical points relating to hardware & software and its compatibility to systems already on the Australian market. In addition, (on each night of the Conference), a discussion group will be held for interested delegates to compare notes relating to exhibits, lectures, etc.
- Special rates for Avis self-drive cars have been established. Reservations can be made either before departure or whilst at the Conference.
- San Michele Travel will complete your Conference registration in advance (waiting in long registration lines is an American habit which you should happily avoid!)
- All U.S.A. visa work for Australian passports will be completed free of charge.

* Services based on min. number of delegates booking N.C.C Tour.
Tour Cost: $1395 *

INCLUDING:

- Air Travel Sydney/Los Angeles/Sydney with Qantas Airways.
- 5 nights share twin accommodation in 1st class hotels in Anaheim, Los Angeles, Buena Vista or Newport. (Children's rates on application — see under)
- Round trip transfers to and from Los Angeles Airport.
- Free Disneyland Entrance Pass.
- Free daily transfers from convention hotels to Convention Centre.
  (Departure from other Australian States have varied fares applicable — please inquire.)
- Additional stopovers at Honolulu either outbound or return (at no extra airfare cost)
- 5 Outbound Departure Dates from May 1, 1983.
- 5 Return Travel Dates from May 20, 1983 (Extended return travel available to June 30, 1983).

BOOKING: $100 per person deposit plus Convention Fee.

Children: Tour and accommodation prices on request. Many of the hotels don't charge for children under 12 sharing the same room. We need the age and number of children together with rooming requirements in order to give the applicable price.

Hotels: Specific hotels can be requested. However, application has been made to A.F.I.P.S. for allocation from their pre-booked list — particularly in relation to Anaheim accommodation for exhibitors.

Optional tours: See over for a range of "See America" fares and holiday ideas.

Taxation: We are advised that companies or persons with genuine interest in computer installation or associated data may use the San Michele Conference Tour invoice for taxation purposes.

“See America” Air Fares: We recommend the following 4 airlines for travel inside the U.S.A.

<table>
<thead>
<tr>
<th>Airline</th>
<th>Price</th>
<th>San Michele Conference Clients Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>American Airlines</td>
<td>A$250.00</td>
<td>$250.00 Aust.</td>
</tr>
<tr>
<td>T.W.A.</td>
<td>A$250.00</td>
<td></td>
</tr>
<tr>
<td>United Airlines</td>
<td>US$250.00</td>
<td></td>
</tr>
<tr>
<td>Republic Airlines</td>
<td>US$250.00</td>
<td></td>
</tr>
</tbody>
</table>

* Based on min. number of conference participants & selected 1st Class hotels.
MICRO COMPUTER TOUR
U.S.A. — MAY 1983
conducted by *

LES BELL and ROD WHITWORTH
Editor "Your Computer" and Technical Manager "Automation Statham"
A Tour in conjunction with the NATIONAL COMPUTER CONFERENCE Anaheim May 16-19, 1983.

Tour Price: $1100 (land & domestic air USA)
Incorporates: travel through U.S.A. by air, coach and car
12 nights accommodation (share twin basis)
Tours, lectures & conferences as detailed.

(Tour Price is in conjunction with the NCC Tour Price of $1395 ex Australia)

TOUR DESCRIPTION

Friday May 13 — Sydney/Honolulu — Qantas flight QF11 leaves Sydney at 8 p.m. and arrives at Oahu Island (Honolulu) at 9.55 the same morning. Before travelling onto "the Big Island", Hawaii, (for a study of hotel computer systems and informal pre-conference discussions) we are taking you by air-conditioned coach or car via Waikiki Beach to Diamond Head and round the east coast of Oahu to have lunch overlooking the ocean. Then through the mountains and major tunnel network returning to the domestic airport and flight to the Kona Coast on "the Big Island" (Hawaii) Landing at Kona is unusual as the airport is cut out of sheer volcanic basalt (looking as though it has flowed recently although actually several hundred years ago.) Transfer is arranged to the Sheraton Hotel.

Saturday May 14 — Provided in the tour cost is a Roberts rental car (self-drive) for each minimum 2 persons. After morning discussions, why not drive into the township of Kona to see something of Hawaiian lifestyles or — better still — into the nearby hills where the famed Parker ranch is established. Close by is the utterly magnificent Mona Kea Hotel — established by Rockefeller for pure understated luxury living. Everything is expensive (they don't accept credit cards!) but well worth a lunch or dinner during your stay on Hawaii.

Sunday May 15 — The purpose of the initial stay in Hawaii is twofold. Firstly, to meet with other tour delegates and discuss common interests. More importantly, the relatively short travel time onto Los Angeles via San Francisco means that you'll be arriving at the Conference rested and ready for one of the most expansive computer conferences held in the world. We fly from Kona at 6.57 a.m. arriving back at Honolulu at 7.30 a.m. and then travel with Qantas at 10.30 a.m., arriving at San Francisco at 6.20 p.m. and then with United Airlines to Los Angeles. Transfer is arranged to your Convention Hotel.

Monday May 16 — Thursday May 19 — NATIONAL COMPUTER CONFERENCE (See elsewhere in brochure)
NEW YORK
Friday May 20 — The United Airlines flight 4 leaves Los Angeles at 9.10 a.m. and arrives at Newark at 5.50 p.m. Hotels throughout the tour are of 1st class standard and where possible the group will be kept together in the same hotels. Transfers will be a blend of taxi (organised by the Tour Conductor) or use of coach and car where applicable. In addition to conferences and visits to computer centres, tours of areas of interest will be arranged and details available through San Michele Travel Pty. Ltd. On many evenings, special “get-togethers” have been arranged with computer people in various cities and your tour cost includes a contribution so that the “Australian contingent” can return some of the hospitality offered.

Saturday & Sunday May 21-22
During this time, local tours of interest will be arranged and if you would like to attend a Broadway show or similar, please advise in advance so that bookings can be made.

TRENTON
Monday May 23 — Early today by car or coach to Trenton where a visit will be made to the Sig/M Micro Centre where one of the largest libraries of software in the world is kept. Special arrangements can be made here to purchase (at very low cost) software applicable to your needs. Tonight, a social get-together with the micro people of Trenton.

ATLANTIC CITY
Tuesday May 24 — Onto Atlantic City by car/coach where we will be visiting Plum Hall, famous for Unix and C language. Again, considerable contact with the local micro people and discussion groups has been arranged.

SAN FRANCISCO
Wednesday May 25 — By car/ coach to Philadelphia where the United Airlines flight 33 leaves for San Francisco at 5.45 p.m. arriving at 8.30 p.m.

Thursday & Friday May 26-27 — A very interesting schedule of visits has been arranged, particularly at Morrow establishment to see the factory and for a key subject address. Next day, onto Apple Computer Co. with a discussion group during Friday afternoon and evening.

Saturday May 28 — Whilst many computer and business people know Los Angeles and the East Coast, they aren’t too familiar with one of America’s most interesting cities, San Francisco. We can arrange a self-drive car or Greyhound tour to nearby areas such as Yosemite National Park.

Sunday May 29 — Qantas flight QF4 leaves at 9 p.m., arriving Sydney at 8.40 a.m. Tuesday May 31st — just in time for you to get back into the work syndrome!

* Tour based on min. number of clients booking Micro USA Tour.
National Computer Conference
May 1983
Extend your Travel either for business or pleasure using:
• 14 day Visit U.S.A. Air Pass for $250 Aust.
• Greyhound Coach Tours (Ask for special Coach Tour Brochure)
<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Keynote Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONDAY, MAY 16</td>
<td>8.30 a.m.</td>
<td>(A) Writing Less Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B) Strategic Systems Planning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(C) Fifth Generation Computers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(D) Computer Teleconferencing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(E) Decision Support Systems: Design Methodologies (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(F) High Performance Computing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(G) Voice Processing</td>
</tr>
<tr>
<td>TUESDAY, MAY 17</td>
<td>8.30 a.m.</td>
<td>(A) Software Management</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B) Increasing Productivity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(C) Database Management</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(D) Decision Support Systems: Design Methodologies (11)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(E) Microcomputer Design Topics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(F) Human Voice Communication with Computers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(G) Business Planning on Micros</td>
</tr>
<tr>
<td>WEDNESDAY, MAY 18</td>
<td>8.30 a.m.</td>
<td>(A) Software Engineering: Now and the Future</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B) Systems Maintenance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(C) Managing Computer Changes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(D) Planning the EIS/DSS/EUC Environment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(E) Super Micros: Commodity Computer &amp; Software</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(F) Special Applications of Technology</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(G) Software Applications of Technology</td>
</tr>
<tr>
<td>THURSDAY, MAY 19</td>
<td>8.30 a.m.</td>
<td>(A) Software Technology for Adaptable Reliable Systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B) Software Maintenance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(C) Measuring Impact of Information Techniques</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(D) Corporate DSS on Micros</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(E) Videotex Systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(F) Network Services Planning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(G) The Next 5 Years in Microcomputers</td>
</tr>
<tr>
<td></td>
<td>10.30 a.m.</td>
<td>(A) Effective Software: On-Line Documentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B) Computer Literacy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(C) Computer Aids to the Handicapped</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(D) Personal Computers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(E) Management Through Enhanced DSS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(F) Multiprocessor Computer Designs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(G) CAD/CAM</td>
</tr>
</tbody>
</table>
**N.C.C. PROGRAM 1983**

<table>
<thead>
<tr>
<th>1.30 p.m.</th>
<th>3.30 p.m.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) Communicating with Databases</td>
<td>(A) Artificial Intelligence</td>
</tr>
<tr>
<td>(B) Maintenance &amp; Motivation of Personnel</td>
<td>(B) Career Development</td>
</tr>
<tr>
<td>(C) Current Database Machines</td>
<td>(C) Research Database Machines</td>
</tr>
<tr>
<td>(D) Office Automation</td>
<td>(D) Organisations &amp; Systems</td>
</tr>
<tr>
<td>(E) High Performance Winchester Trends</td>
<td>(E) Measuring Office Productivity</td>
</tr>
<tr>
<td>(F) Protocols for Computer Communications</td>
<td>(F) Advances in Communications Networks</td>
</tr>
<tr>
<td>(G) Microcomputer Software Protection</td>
<td>(G) Personal Computer Industry: Experts Forecast Future</td>
</tr>
</tbody>
</table>

| (A) Reducing Program Risks with Reusable Code      | (A) Software Engineering Techniques & Approaches |
| (B) Practices in Staffing Software Projects        | (B) Audit of Complex Computer Communication Systems |
| (C) Modelling Enterprise Data                      | (C) Micro Database Management Systems           |
| (D) Management and Professional Workstations       | (D) Electronic Mail                            |
| (E) Non-Conventional Approaches to DSS            | (E) Dynamic & Reconfigurable Software Problems |
| (F) Fault-Tolerant Systems                         | (F) Dynamic & Reconfigurable Hardware Structures |
| (G) Distributed Processing                         | (G) Network Security                           |

| (A) Experience in ADA Applications                 | (A) Future Visions: ADA 1990's                  |
| (B) Control of the Maintenance Function            | (B) Computer & Video Disks in Learning          |
| (C) Living with Computers: Social Impact           | (C) Applications in Database Management         |
| (D) Organisational & Implementation Issues         | (D) Computerized Society: Resilient or Vulnerable |
| (E) Computers and the Law                          | (E) Network Management                          |
|                                                   | (F) New Applications for Briefcase Computers    |
STOPOVER HAWAII!!
(without extra international air fare!)

NCC delegates travelling Qantas are offered a special range of stopover holidays throughout the Hawaiian Islands. We have excellent rates with hotels and self-catering apartments, Hawaiian Air offer discount air fares to visit several islands during the stopover holiday.

Qantas flights leave Los Angeles for Honolulu on Thursdays and Saturdays at 9 p.m. arriving 11.30 p.m. the same night. Departure from Honolulu to Sydney with Qantas is daily.

HAWAII’S ISLANDS

OAHU — the Capital island, home of Waikiki Beach, Diamond Head, Pearl Harbour and the Iolani Palace — the interior offering beautiful valleys and dramatic mountain ranges.

MAUI — remote communities of great beauty, where old-time customs are still revered. At Kaanapali, there is a magnificent range of hotels on excellent beaches with a sweep to golf courses, sugar palm plantations and a volcanic mountain range.

KAUI — This is probably Hawaii’s most beautiful island with deep valleys and canyons and vegetation. Coco Palms is a hotel with a unique flavour — huge clam shells for bathroom basins — a blend of old and new with excellent and inexpensive dining.

HAWAII — the Big Island — the most dramatic and interesting. Land at Kona Airport which seems to be carved out of acres of black basalt. This area appears as though volcanic activity happened yesterday but dates back to last century. Magnificent hotels abound including the Sheraton and Rockefeller’s Mona Kea. Drive into the hills to one of the world’s most famous cattle studs — the Parker Ranch.

Explore 2 or more islands — it’s easy to arrange an itinerary visiting several islands, particularly using the combination of Hawaiian Air and self-drive cars awaiting you at each airport. (Don’t forget your International Drivers’ Licence!)

U.S.A. — GREYHOUND

COACH TOURS

The largest coach company in America has an unsurpassed range of escorted or independent tours throughout the U.S.A.

- a 10 day tour from Los Angeles visiting Las Vegas, Cedar City, the Grand Canyon and Salt Lake City
- a 3 day tour taking in the magnificent scenery of Yosemite National Park
- a 4 day tour visiting San Diego and Tijuana
- Cross country tours by the north or south route taking 11 days
- a combination of air and Greyhound visiting Niagara Falls on a 3 day tour from New York

Greyhound are famous for their National Park tours — the Black Hills National Forest in South Dakota, the Mount Rushmore Memorial, the Grand Canyon, Yellowstone National Park and of course Yosemite.

Special interest tours are also included including ranch round-ups, river-running raft trips and scenic jeep safaris.

Read the Greyhound “Coach Travel” Booklet.
San Michele Travel Pty. Ltd.

CONDITIONS OF TRAVEL

DEPOSIT $100 per person. Balance payment is due 45 days prior to travel. If less than 45 days — payment in full is required. Hotel reservations, etc. are commenced after receipt of deposit. Where reservations are necessary under 35 days, telex confirmations are chargeable to the client.

TOUR ALTERATIONS Minor alterations to itinerary cost $15 per file. Major alterations have a charge of $45 per file plus any necessary telex fees.

IF YOU CANCEL YOUR HOLIDAY Cancellation (in writing) after deposit has been paid but prior to 35 days before travel, incurs a fee of $50 per person plus any applicable airline cancellation fees or outgoings such as visas, telex, etc. Under 35 days — after deduction of applicable airline cancellation fees, outgoings and non-refundable prepayments — the following apply:

<table>
<thead>
<tr>
<th>Less Land Content</th>
<th>Air Ticket Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 days up to 50%</td>
<td>$50</td>
</tr>
<tr>
<td>7 days up to 100%</td>
<td>$50</td>
</tr>
</tbody>
</table>

DOCUMENTATION (IMPORTANT) It is the responsibility of the passenger to completely check all tickets and vouchers before departure. A “Check List” assists clients and consultants to agree on documentation. Brochure conditions cannot be altered or waived except by the Company General Manager in writing.

AIRPORT TAX Payable by the client.

REFUNDS Tours, accommodation and other vouchers are issued on a non-refundable basis. If a service or accommodation is not provided as pre-arranged, refund of the actual sector or voucher cost will be effected where the responsibility lies with the tour operator.

TRAVEL AGENT’S LICENCE San Michele Travel Pty. Ltd. New South Wales Government Licence No. is B879.

VISA REQUIREMENTS Any visa required for San Michele itinerary will be processed at a specified fee by ourselves or your travel agent. However, it is the clear responsibility of the client to ensure (and double-check) that passports, visas, re-entry permits, etc. are in order for intended travel.

OUR BROCHURE (E/or Itinerary) Every effort has been made to ensure that descriptions and details are correct at the time of printing/preparation. The client recognises and accepts that subsequent changes can occur (particularly relating to Asian travel) and that

1. such changes are not grounds for claim (other than refund of differences in actual fares/land costs which can be reasonably estimated),
2. San Michele Travel Pty. Ltd. and/or his servant/agent may change an itinerary/schedule to achieve a reasonable parallel to the originally intended tour.

DEVALUATION & INCREASES It is a fact of life that currency rates, overseas travel, accommodation costs and air fares alter rapidly. In accepting our travel arrangements. The client must be aware that such surcharges may be levied at any time.

COMPANY RESPONSIBILITY In accepting our tour arrangements, the client agrees that the operator is not liable or responsible at law for illness, death or injury of any passenger or other customer or for any damage or loss of any luggage or personal possession. Any client accompanied by a minor or otherwise legally disabled person is deemed to have agreed to indemnify the operator from any claim or legal action. The passage contracts used by the transportation companies or firms concerned constitute the sole contract between these transportation companies and the client and such tickets, coupons, etc. are furnished subject in all respects to those terms and conditions.
Please complete the booking form below and return to your travel agent or San Michele Travel Pty. Ltd. with the deposit of $100 per person.

IMPORTANT: In completing this form please READ CAREFULLY the tour itinerary, and "Travel Notes & Conditions" (overleaf) — as all bookings are subject to that data.

**BOOKING FORM**

<table>
<thead>
<tr>
<th>MR/MRS</th>
<th>FAMILY NAME</th>
<th>INITIALS</th>
<th>BIRTH DATE</th>
<th>PASSPORT N' LITY</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Phone</td>
</tr>
</tbody>
</table>

**TOUR NAME**

<table>
<thead>
<tr>
<th>Tour Duration (Days/Nights)</th>
<th>Departure Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tour Quote (Air & Land)**

Per Person
(Not incl. visaes/insurance etc.)

<table>
<thead>
<tr>
<th>Adult</th>
<th>Child</th>
<th>Infant</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>

Note: Some tours have a "selected plan" all hotels, others allow your choice.

**CLIENT SIGNATURE**

Enclosed is deposit of $100 per person for the tour/travel as noted above. The brochure, Travel Notes & Conditions have been read and understood — particularly in regard to group fares & departures/cancellation fees/airline confirmations. You are authorised to disperse my funds to airlines/overseas and local operators as necessary to complete my travel arrangements. I am aware of the possibility of changes/variations to my itinerary/schedule with or without notice, and also aware of the need to maintain contact with my travel agent.

Client Signature: __________________________ Date: __________________________

---

San Michele Travel Pty. Ltd.

**IN CONJUNCTION WITH:**

**INDONESIAN TRAVEL CENTRE**

**SALES OFFICES:**

SYDNEY: Imperial Arcade, 168 Pitt Street, 2000 Phone 231 5444
PERTH: National Bank Arcade (off London Court) 6000 Phone 325 2500
ADELAIDE: Cnr. Tour World 144 Nth Terrace 223 7777
MELBOURNE: Embank Arcade, 319 Collins Street, 3000 Phone 52 4961
BRISBANE: Piccadilly Arcade (off 340 Queen St.) 4000 Phone 229 3677

**ADMINISTRATION:** American Express Tower, King & George Street, 2000 Phone 221 2222
and Agency Sales

Sydney Telex ITCSM 27334
A Comparison of the Concurrency Constructs and Module Facilities of CHILL and Ada

By C. J. Fidget and R. S. V. Pascoet

The concurrency and modularisation features of two new programming languages, both the product of large design efforts, and both with similar aims and application areas, are summarised and compared. CHILL and Ada, despite global similarities, present quite different approaches to these two problem areas and it is shown that the Ada mechanism is slightly superior in both cases.

Key words and phrases: CHILL, Ada, programming languages, modularisation, concurrency, abstract data types, history of programming languages.
CR categories: D.1.3, D.3.3.

1. INTRODUCTION

This paper describes and compares the concurrency constructs and modularisation facilities available in the programming languages CHILL and Ada. Both languages are designed to provide support for "embedded systems", including provision for real-time, concurrent and systems programming while still being suitable for general purpose, procedural applications. Since CHILL and Ada are backed by the CCITT and the US Department of Defence respectively, it is anticipated that they will be used extensively in the near future, and, as noted by Nissen (1980) "it is an unfortunate historical accident that a single language is not being adopted" since either language could suit the requirements of both organisations.

More general comparisons of the two languages can be found in Boute and Jackson (1981), Nissen (1980) and Rahko and Hemdal (1981), however this paper will concentrate on the modularisation and concurrency aspects of the languages because these are the two areas where CHILL and Ada differ from virtually all widely used, traditional programming languages.

For the purposes of this paper a "module" is defined to be a language facility which explicitly controls the visibility of program objects and which may contain a special code segment that can be used to initialise data objects local to the module. While a module controls visibility, it does not influence the lifetime of its local objects. It may or may not be a separately compilable entity. "Concurrency constructs" are those which enable two or more program units to be executed at the same time, at least from a logical viewpoint. Of particular importance is the ability of concurrently executing program units to synchronise and/or communicate with each other.

The Ada Reference Manual (DoD, 1981) and the CHILL Language Definition (CCITT, 1980a) are the definitive references for their respective languages although several other works were found to be very helpful during the preparation of this paper. Both language designs are now frozen; however, full compilers are not available for either CHILL or Ada at the time of writing.

2. WHAT IS CHILL?

CHILL (the CCITT High Level Language) is a language developed by a CCITT study group specifically for programming SPC (Stored Program Control) telephone exchanges. However, it is general enough for use in most real-time environments (e.g. message switching, packet switching, modelling, etc.) as well as for general systems programming and conventional procedural applications. It is based on Pascal, PL/I and Algol 68 and the final version of the language was given in CCITT Recommendation Z.200 (CCITT, 1980a), although development of a programming environment is still under consideration (for example, the language definition does not reference problems such as separate compilation, input/output or debugging tools).

As described in the language definition, the primary aims of the language design were to:

- enhance system reliability by allowing for extensive compile time checking,
- permit the generation of highly efficient object code,
- be flexible and powerful in order to cover the required range of applications and to exploit various kinds of hardware,
- encourage modular and structured program development,
- be easy to learn and use.

Whether or not CHILL has managed to live up to these aims is largely a matter of opinion; however, in a thorough analysis of the concurrency features offered by CHILL, Kedy (1981) uncovers some inadequacies in the definition of "regions" which suggests that the requirement for reliability may already be endangered. Also, due to CHILL's origins as a committee product, several alternative constructs were provided whenever a unanimous agreement could not be reached. This has resulted in a strangely mixed, non-uniform syntax, and in consequence the language does not appear to completely satisfy the requirement for being easy to learn. Finally, Rahko and Hemdal...
further design refinement, the "Green" language was selected, i.e. all names defined in an outer block are automatically visible in inner blocks, but names defined in inner blocks are not visible in outer blocks (by definition, the scope rules for modules and regions are totally different, of course). Finally, a complete CHILL program is defined to consist of a sequence of regions and modules.

3. WHAT IS ADA?
Ada (named after Augusta Ada Byron, Countess of Lovelace and considered to be the first computer programmer) is a language designed by a Paris-based team under the leadership of Jean D. Ichbiah for the US Department of Defence (DoD). Sixteen designs were originally presented in 1977 by various industrial and educational organisations, conforming to a set of requirements specified by the DoD and, after a process of elimination and further design refinement, the "Green" language was selected in 1979 and renamed Ada. The final version of the language is described in the Ada Reference Manual for July 1980 (DoD, 1981). Ada is primarily based on Pascal and, while containing many constructs that Pascal does not provide, attempts to show the same syntactic elegance and uniformity that has led to the widespread use and popularity of Pascal in recent years. Pascal derivatives including Euclid, Lis, Mesa, Modula and Sue also provided some inspiration as well as Algol-68, Simula, Alphard and Clu. Although intended mainly for embedded computer applications, Ada is also suitable for general systems programming, real-time industrial work, general applications programming, numeric computation and even for "teaching good programming practices" (Brenden and Nassi, 1981).

According to the Ada Reference Manual the principal design goals were to:
- promote reliability and simplify maintenance by emphasising program readability over ease of writing, using strong typing checks, allowing separate compilation, etc.
- cater for the needs of the human programmer by keeping the language as small as possible, providing language constructs which map onto a user's intuitive view of the underlying concept, and avoiding excessive invocation of constructs and allowing for the increasingly decentralised and distributed nature of programming.
- allow for the generation of efficient object code by rejecting any proposed language construct whose implementation was unclear or required excessive machine resources in the light of present implementation techniques.

In general, it appears that Ada lives up to its design goals somewhat better than CHILL does (and it is interesting to note the similarities between the two sets of aims). Nevertheless, Nissen (1980) suggests that the scoping rules in Ada are too complex and Boute and Jackson (1981) comment that the inability to use structural equivalence rather than name equivalence is a disadvantage with typing in Ada.

An Ada program consists of one or more (separately compilable) program units, which may be packages, tasks or subprograms. A package is described as the basic unit for defining a collection of logically related entities. It controls data object visibility and fills the aforementioned requirements for the definition of a module. As such it will be discussed in section 5. A task is syntactically similar to a package and is the basic unit of concurrency in Ada. It is described in detail in section 8. A subprogram (which may be a procedure or function) again conforms to the familiar idea of a call sequence of action statements. Each of the three types of program units normally consist of two parts: a specification containing information visible to other units, and a body containing implementation details which are not visible. Ada, like CHILL, also has blocks which are optionally named sequences of statements delineated by BEGIN and END with optional local declarations. Since they implicitly control object lifetime they cannot be considered as modules. Scope rules for Ada subprograms are again similar to Pascal; however Ada allows access to global objects that have been redefined in a nested routine by the use of a qualifying prefix. The visible part of packages can be made accessible to other units via the use clause but it is also noted that this visibility is non-transitive (see section 5). Finally, Ada allows "overloading" of identifiers. The value of this technique with respect to compiler complexity and readability by the human programmer is debatable and it may be considered to be a violation of the strong typing principle (Boute and Jackson, 1981), but is arguably justified in the context of large scale program development (DoD, 1981).

4. MODULARISATION IN CHILL
The basic unit of modularity in CHILL is the module. It is a means of restricting the visibility of names and is one of the three major constructs for program structuring. It does not influence the lifetime of locally created locations (variables).

Textually a module consists of a sequence of declaration and definition statements (which include location declarations as well as mode [type], process and procedure definitions), visibility statements, regions and a sequence of action statements (which may further be composed of modules and/or begin-end blocks). Any or all of these items are optional and they may be arranged or interleaved in any order desired, the only restriction being that the action statement list (if any) must be the last item enclosed in the module. Thus a module has the following syntactic appear-
Comparison of Concurrency Constructs

An entire module is considered to be a single (compound) action statement and can therefore be "nested" within any program construct which may contain action statements. "Visibility statements" (GRANT and SEIZE) are the method used by a module to explicitly control name visibility and are described later.

4.1 Lifetime of locations

As mentioned earlier, modules have no effect over the lifetime of locally declared locations. The CHILL constructs that control lifetime are blocks (the main members of which are processes, procedures and begin-end blocks). As a result, the lifetime of an object declared in a module is the same as if it were declared within the first surrounding block.

Since a complete CHILL program was defined in section 2 as a sequence of modules and regions, a natural question at this point is what constrains the lifetime of locations declared in these outermost modules. As described in section 7.7, this problem is solved by the fact that the language definition states that a CHILL program is treated as if it were surrounded by an imaginary "outer block" (specifically a process) and it is therefore the lifetime of this all-embracing block that also defines the lifetime of objects declared in the outermost nesting level.

As a final point on the lifetime of objects it is noted that locations declared as STATIC have a lifetime covering the span of the entire program (as if they were declared in the imaginary outer process) no matter how deeply they are nested.

4.2 Visibility rules

The visibility rules for modules are extremely simple: a name declared in a module is local to that module only. It is not automatically visible to "outer" or "inner" nested modules or other modules at the same nesting level (of course, the name will be visible to inner nested blocks which follow different scope rules).

A local name can be made visible "outside" of a module by including it in a grant statement. This has the effect of making the named object visible in the directly surrounding nesting level, i.e.

\[
\text{GRANT granted_element_list;}
\]

A syntactic shorthand is also provided for granting all of the names visible in a module ("GRANT ALL"). Also, when granting a newmode structure (i.e. record type), selected fields of the structure can be made inaccessible outside the module by naming them in a forbid clause attached to the grant statement (again a shorthand notation is available that forbids all the fields of a structure being granted). Finally, the reserved word PERVASIVE following a grant statement causes the granted item to become automatically visible in "inner" nested modules (without the need to explicitly "seize" the item).

For a module to make use of an item that has been granted by another module it must seize it (except for the special pervasive case). The seize statement has the effect of making the seized name(s) directly visible within the seizing module, i.e.

\[
\text{SEIZE seized_element_list;}
\]

The shorthand "SEIZE ALL" enables a module to seize all names that are not presently visible to the module but are visible in the surrounding nesting level. A second shorthand seize statement ("SEIZE module_name ALL") has the effect of seizing all names granted by the named module. A pervasive name that is seized will have the pervasive property within the seizing module.

The following example (based on one from the Introduction to CHILL [CCITT, 1980b]) shows the effect of some of these visibility rules:

\[
\begin{align*}
m1: & \quad \text{MODULE} \\
& \quad \text{DCL} \ d \ \text{INT}; /* \text{declaration of integer } \text{"d")(*)} \\
& \quad \text{m2:} \\
& \quad \text{DCL} \ c, e, f \ \text{INT;} \\
& \quad \text{NEWMODE} \ h = \text{STRUCT} \ ((x \ \text{INT}, y \ \text{BOOL}); \\
& \quad /* \text{define a new type } \text{"h" with two fields } \text{"x" and } \text{"y")(*} \\
& \quad \text{SEIZE} \ d; \\
& \quad \text{GRANT} \ c, e; \\
& \quad \text{GRANT} \ h \ \text{FORBID} \ ALL; \\
& \quad /* \text{c, d, e, f, h, x and y are all visible here (*)} \\
& \quad \text{END m2;} \\
& \quad /* \text{c, d, e and h are visible here (but not x and y)(*)} \\
& \quad \text{m3:} \\
& \quad \text{MODULE} \\
& \quad \text{DCL} \ g \ \text{BOOL;} \\
& \quad \text{SEIZE} \ e; \\
& \quad /* \text{e and g are visible here (*)} \\
& \quad \text{END m3;} \\
& \quad \text{m4:} \\
& \quad \text{MODULE} \\
& \quad \text{SEIZE} \ m2 \ \text{ALL;} \\
& \quad /* \text{c, e and h are visible here (*)} \\
& \quad \text{END m4;} \\
& \quad \text{b1:} \\
& \quad \text{BEGIN} \\
& \quad \text{DCL} \ d \ \text{BOOL;} /* \text{local version of "d" — the integer d} \\
& \quad /* \text{cannot be accessed from within b1 (*)} \\
& \quad /* \text{c, e and h are also visible here (automatically in­} \\
& \quad \text{herited by b1 since it is a block)(*)} \\
& \quad \text{END b1;} \\
& \quad \text{END m1;} \\
\end{align*}
\]

Since a module is treated like an action statement it is entered and left rather than being called. For instance, in the example above, if the three inner modules and the begin-end block each contained some (non-branching) action code it would be executed in the following order:

\[
m2, m3, m4, b1
\]
Comparison of Concurrency Constructs

In fact action statements could be scattered between these inner items. In all cases they are simply executed sequentially, top to bottom, totally ignoring the module boundaries (this system of entering and leaving a program unit does not apply to procedures and processes, of course, which must be called or started respectively).

This concludes our summary of the modularisation facilities in CHILL. The visibility rules for modules also apply to regions which will be discussed in section 7.6. The reader is directed to the CHILL Language Definition (CCITT, 1980a) for a complete, definitive description of these facilities.

5. MODULARISATION IN ADA

The basic unit of modularity in Ada is the package; used to define a separately-compilable collection of logically related entities. As mentioned earlier, they are one of the three basic program structuring units in Ada, along with tasks and subprograms. The Ada Reference Manual identifies three intended uses for packages: providing a named collection of logically related data declarations, grouping of related subroutines and abstract data type support.

An Ada package consists of two parts: a specification and an optional body. The specification and body may be separately compiled (a package body can be replaced with an equivalent implementation without the need to recompile any other program units, including the specification). Syntactically a package specification and body have the following form (Wegner, 1980):

```ada
PACKAGE name IS
  declarations of visible resources
[PRIVATE
dclarations of private data types]
END [name];

PACKAGE BODY name IS
  [declarations of local resources]
  [subprogram declarations to implement visible subprograms]
  [BEGIN
  initialisation part]
END [name];
```

5.1 Package specifications

A package specification consists of a list of declarative items that can be accessed by other program units (the visible part) and an optional list of declarative items that can not be accessed outside the package (the private part). A package specification without an associated body is usually used to provide a set of common constants, variables and/or types (i.e. it provides other program units with access to information). For example, the following package may be used as part of a game playing program:

```ada
PACKAGE checkers_data IS
  - a common pool of data and types
  - for a game playing program
  board_size: CONSTANT INTEGER := 8;
  TYPE square IS (empty, white_man, black_man,
   white_king, black_king);
  TYPE board IS ARRAY (1..board_size, 1..board_size) OF square;
  checkers_board: board;
END checkers_data;
```

A package which contains only object (i.e. variable) declarations is similar to a named common block in FORTRAN. However, as noted in the Ada rationale (Ichbiah, Heliard, Roubine, Barnes, Krieg-Brueckner and Wichmann, 1979), the package concept has three important advantages: a package can be declared in a nested block and is thus not global to the entire program, storage reservation can be performed at scope entry time, and the contents of the package need only be described in full once whereas the specification of a common block in FORTRAN must be replicated in its entirety wherever it is used (thus allowing accidental incorrect declarations). Variables declared in a package specification (or body) retain their values throughout the lifetime of the package ("own" variables). The method used to access items declared in the visible part of a package is described in section 5.3.

A package specification may also contain declarations following the reserved word PRIVATE. Particularly useful when using a package to create an abstract data type, this mechanism allows declaration of a type or constant in the visible part with the description of its detailed representation deferred until the private part. In this way, even though other program units may use the declared object they cannot access its actual representation. Private object representations need to be described in the package specification to allow a compiler to allocate variable space without the need to refer to the body.

For example, the following specification for a key managing package from the Ada Reference Manual has a private type and constant value:

```ada
PACKAGE key_manager IS
  TYPE key IS PRIVATE;--detailed representation is deferred
  null_key: CONSTANT key := 0;
  PROCEDURE get_key (k: OUT key);
  FUNCTION "<" (x,y: key) RETURN boolean;
  PRIVATE
  TYPE key IS NEW Integer RANGE 0 .. integer'last;
  null_key: CONSTANT key := 0;
END;
```

Any other program unit using this package can in no way use the fact that a key is actually represented by an integer subrange or that the constant "null_key" is represented by the value 0. Also the above specification contains the declaration of two subprograms "get_key" and "<".

As described in the next section, the body for these two routines must be found in the package body corresponding to use the "key_manager" specification.

Notice that objects of a private type may still be used outside the package in assignment statements or tested for equality and inequality. However, even these operations can be forbidden to users of the package by declaring a type as "LIMITED PRIVATE" in which case the only possible operations on objects of this type are those defined by procedures in the package specification.

5.2 Package body

Package bodies provide the means to supply other program units with a related group of subprograms, or the operations that can be performed on an abstract data type. Whenever the visible part of a package specification contains the declaration of a (nested) subprogram, task or other package then the behaviour of this package must be described in a corresponding package body. Package bodies may also contain their own declarations and program units which are local to the body only and are never visible anywhere outside the package.

Referring back to the "key_manager" example in section 5.1 the two subprograms referred to would be found in a corresponding package body (DoD, 1981), i.e.
5.4 Separate compilation

Packages, like other Ada program units, are separately compilable entities (and as noted earlier the specification and body of each package can also be separately compiled, body following specification). A compilation data base allows separate compilation without giving up the ability to provide program-wide type checking.

When using the separate compilation facilities in Ada the with clause defines the dependences between compilation units. Any compilation unit about to be compiled that accesses a previously compiled package must be preceded by

```
WITH package_name;
```

This results in the compiler recognising that the unit presently being compiled will make use of this package and causes the mentioned library unit to be directly visible to the one now being compiled.

The final important point about Ada packages is that they may be generic, i.e. a "skeleton" package definition with parameters may be produced to provide a template from which a number of different packages can be instantiated. This concludes our brief examination of modularisation in Ada.

6. COMPARISON OF THE MODULE FACILITIES OF CHILL AND ADA

Having summarised the modularisation facilities provided by the two languages we are now in a position to be able to compare the two techniques offered to prospective programmers.

6.1 General

The most striking difference between modules in CHILL and packages in Ada is the greater syntactic complexity of the Ada mechanism. Rather than being a disadvantage, however, this is a point in favour of packages. The tightly defined structure of a package helps ensure that the programmer adopts a good programming style whereas the "looseness" of the CHILL construct, while being flexible, nevertheless allows an ad hoc programming style to be used unless care is taken, thus placing greater onus on the programmer to ensure a well designed, manageable program.

The grant and seize mechanism in CHILL has a problem that is not found with the dot-notation and use clause accessing methods of Ada. Since a seize statement cannot say which module an identifier to be seized belongs to there is a possibility of unresolvable ambiguities when the same identifier is granted by two other modules (the "SEIZE module_name ALL" statement offers a partial solution but is not satisfactory for all situations).

In its simplest form a module acts as a repository of information. As mentioned in section 5.1, the Ada rationale identifies three advantages of the package over traditional FORTRAN common blocks, all of which may equally be applied to modules in CHILL.

In Ada the "local data initialising statements" that were originally mentioned as one of the pre-requisites for the definition of a module, are a specific, well-defined language feature. Although the CHILL programmer can duplicate this facility with action statements inside of a module, the "loose" syntax of a module again places the burden of ensuring that this scheme is not misused on the programmer rather than the language. There is no restriction to stop a program from jumping into (or around) the use clause:

```
USE package_name
```

and the second is via the use clause which makes all the information in the visible part of a package directly visible to the program unit which included the clause, i.e.

```
USE package_name
```

The board from our checkers example in section 5.1 could be accessed with, e.g.

```
checkers_data.checkers_board(4, 3)
```

or the package could be made visible to a program unit with the use clause:

```
USE checkers_data;
```

The visibility provided by the use statement is non-transitive. If a package d contained the clause "USE m" then a program unit accessing package d with "USE d" will not automatically have access to package m. The Ada rationale suggests the use of explicit renaming if this sort of transitive package visibility is required (the Ada renames clause allows the programmer to provide local synonyms in package d for objects in package m, in effect making these objects automatically visible to any program unit using d).

5.3 Accessing a package

There are two ways to access the identifiers made visible by a package. The first is with dot-notation, i.e.

```
package_name.declared_identifier
```

and the second is via the use clause which makes all the information in the visible part of a package directly visible to the program unit which included the clause, i.e.

```
USE package_name
```

The board from our checkers example in section 5.1 could be accessed with, e.g.

```
checkers_data.checkers_board(4, 3)
```

or the package could be made visible to a program unit with the use clause:

```
USE checkers_data;
```

The visibility provided by the use statement is non-transitive. If a package d contained the clause "USE m" then a program unit accessing package d with "USE d" will not automatically have access to package m. The Ada rationale suggests the use of explicit renaming if this sort of transitive package visibility is required (the Ada renames clause allows the programmer to provide local synonyms in package d for objects in package m, in effect making these objects automatically visible to any program unit using d).

5.4 Separate compilation

Packages, like other Ada program units, are separately...
a module and causing statements originally intended for
initialising common locations being executed again and
again. In view of this, the CHILL programmer may prefer
to make use of the facilities available for including an
initial value for a location in its declaration whenever action
statements are not required to calculate initial values.
CHILL provides two alternatives: lifetime bound initiali-
sation in which a value is assigned to the location when it
is first created, and scope bound initialisation where the
given value is assigned to the location whenever its parent
module is entered.

Of the three intended uses of an Ada package both
CHILL and Ada provide sufficient mechanisms to support
the first two, a named collection of related data declara-
tions and grouping of related subroutines, although, for
the reasons stated above Ada is slightly superior and easier
to use. The third possible use, support of abstract data
types, is more complicated and will be studied in the next
section.

6.2 Abstract data type support

The emphasis placed on the use of data abstractions
in programming has grown enormously in recent years
and this area is by far the most important application for modu-
larisation facilities. Very briefly, an abstract data type
defines a class of abstract objects which are completely
characterised by the operations available on them. The
implementation of these objects and operations is comple-
tely hidden from (and hopefully inaccessible to) the user.
Modularity has obviously been given much more
thought in Ada than in CHILL and the precise separation
of specification from implementation in Ada is a powerful
abstraction tool that CHILL lacks.

In a thorough analysis of the abilities of CHILL and
Ada to support abstract data types, Boute and Jackson
(1981) make the distinction between two different
methods of providing abstract types. The "type module"
approach involves operations being visibly performed on
objects of an abstract type. A type module, therefore
presents other program units with an abstract type which
may contain exception handlers. A process definition state-
mence to be executed sequentially. A process definition
statement which includes the forbidden clause in CHILL provides a similar
mechanism it suffers from the disadvantage of only being
applicable to structure (record) types. Also CHILL does not
appear to have a facility equivalent to the "limited private"
declaration.

The second style of definition is the "encapsulated
data type" which considers the operations as attributes on
particular instances of that type. A module here presents its
users only with a group of operations. The objects that
these operations are performed on are enclosed within the
module. This means that to provide different objects of the
abstract type we need to create different instances of the
module. Both Ada and CHILL can handle a single instance
of an encapsulated type, but CHILL fails to provide a suit-
able method for creating multiple instances. In Ada it is
possible by using the generic package facilities to create

6.3 Summary

Although the comparison criteria used here have cer-
tainly not been all-embracing, it is apparent that the pack-
age in Ada is a superior modularisation facility than the
module in CHILL in many respects.

It may be argued that it is "unfair" to compare pack-
ages and modules as we have done since their intended pur-
poses are different, i.e. modules are primarily meant to re-
strict visibility while packages are intended to provide other
program units with a service. Nevertheless, we feel that this
comparison is justified since CHILL contains no other
language constructs with capabilities similar to the package
in Ada.

Before leaving this subject it is noted that while Ada
has a clearly defined method for separate compilation, the
CHILL language definition does not cover this topic
(although the CCITT is currently considering the problem).
However, it would seem reasonable to assume that CHILL
implementations will allow modules to be separately com-
piled. This will possibly require minor extensions to the
syntax of a module to allow the compiler to recognise
those units destined for separate compilation (similar to the
with clause in Ada) (Rudmik and Moore, 1981; British Ad-
ministration and Philips' Telecommunicatie Industrie,
1982).

7. CONCURRENCY IN CHILL

CHILL processes are entities which may operate in
parallel. They consist of the sequential execution of a series
of statements and may contain special features to synchro-
nise and/or communicate. The behaviour of a process is de-
scribed by a process definition statement which includes the
objects local to the process and the series of action state-
ments to be executed sequentially. A process definition
may be parameterised (but it cannot return values) and
may contain exception handlers. A process definition state-
ment is considered to be a template from which individual
processes may be created.

Syntactically a process definition statement (which is
a CHILL block) is similar to a procedure definition. It
may contain a formal parameter list, declaration and
definition statements (i.e. location declarations, mode and
procedure definitions) and action statements (including
nested modules and begin-end blocks):

name:
PROCESS ([parameter list]);
 [declaration statements]
 [definition statements]
 [action statements]
END [name];

A process definition statement may not be nested
within a region or block (except the imaginary outer
process). A process is, at the logical program level, always
in one of two states: active or delayed.
Comparison of Concurrency Constructs

7.1 Creation and termination of processes
A process is created (and thus made active) by the evaluation of a start expression:

```
START name ([actual parameter list])
```

Any number of processes can be created from a single process definition. Parameter passing is analogous to procedure parameter passing, however, the language definition states that "additional actual parameters may be given [in the start expression] with an implementation defined meaning". No guidelines for the use or intended purpose of this facility are given. The creation of a process also results in the creation of its locally declared procedures, values and locations (except in the case of locations declared as static for which a single location exists for all instances of the process; a rather dangerous capability considering the mutual exclusion problem that would arise).

A process can be terminated by executing a stop statement, reaching the end of its body or reaching the end of one of its exception handlers. An entire CHILL program can only terminate when all of its subsidiary processes have terminated.

7.2 Identifying processes
To enable the identification of any particular process a special mode "INSTANCE" is provided. Every process created has a unique instance value attached (the special value NULL is used to identify no process).

The instance value for a particular process may be obtained in two ways. Firstly, when the process is created, the start expression will return the unique instance value identifying the process, which can be assigned to a location declared to be of instance type. It is now possible to reach the process via the value held in this location. Also, a process can learn its own identity by using the this operator which returns the instance value of the process executing it.

The following code segment demonstrates the effect of using instance values (CCITT, 1980b),

```
DCL call1, call2 INSTANCE := NULL;
call_handler:
PROCESS (line_no INT);
  /* local declarations */
call2 := THIS;
  /* further action code */
END call_handler;

call1 := START call_handler(2, 200, 5);
/* call1 now identifies the process just created. When the process executes its use of the THIS operator, call1 and call2 will both hold the same instance value. */
```

7.3 Events
CHILL provides several mechanisms for process coordination. The first of these, event mode locations and their associated operations are used to enable explicit synchronisation of processes.

An location declared to be of event mode can be acted upon by continue, delay and delay case statements. The declaration of an event object may optionally include an event length that specifies the maximum number of processes which can be concurrently delayed on this event (Keedy, 1981).

A process executing a delay statement, i.e.

```
DELAY event_location [priority]
```

will be delayed (and placed in a queue of processes delayed on this event) until the event "occurs" (i.e. another process executes a continue statement on this event). The delay statement may contain an optional priority to be used when deciding which delayed process to awaken.

A process executing a delay case statement is delayed awaiting activation by any of the named events. For each named event a different sequence of action statements to be performed upon reactivation may be specified. As with the delay statement a priority may be given, however, the delay case also offers the ability to identify which process caused re-activation of the present process (by setting a locally declared instance location).

Execution of a continue statement,

```
CONTINUE event_location
```

causes re-activation of a process delayed on the named event with highest priority (if any). If no processes are delayed on this event the continue statement has no effect (i.e. a continue action is not "remembered").

7.4 Signals
Signals provide a means of communication and synchronisation between processes. They are used in send and receive case statements. Unlike events, CHILL signals are persistent, i.e. if no process is waiting to receive a signal it is saved ("becomes pending") until a process needs it.

A signal is defined in a signal definition statement. It may have an optional message part consisting of a set of modes and may also specify which process type may receive this signal:

```
SIGNAL name [= (mode_list)] [TO process_name];
```

A signal send statement will send the named signal with a list of values (if it has a message part), an optional priority and an optional instance value identifying the intended receiver (which must not conflict with the name given in the signal definition statement, if any):

```
SEND name [[message_value_list]] [TO instance_value]
[priority]
```

A process can receive a signal by executing a receive case statement which specifies a list of signals which may be received (each with its own action code segment). If none of the named signals is pending the process is delayed (unless an ELSE clause has been included in the receive case statement). The receive case also offers the ability to identify the sending process (by setting an instance location),

```
RECEIVE CASE [SET instance_location;
  (signal_name [IN message_receiving_names]):
    action statements
  ]
[ELSE action statements]
```

ESAC

If more than one appropriate signal is pending the signal with highest priority is chosen; if several signals share the highest priority then the choice is implementation dependent.
7.5 Buffers

Buffer mode objects and their operations are another method of synchronisation and communication between processes. When an object is declared to be of type BUFFER, the mode of the elements in the buffer must also be supplied and optionally the number of elements that the buffer can hold, e.g.

DCL mail_box BUFFER(10) message;

declares a buffer “mail_box” containing up to 10 entries of type “message”. An object of buffer mode can be acted upon by send statements, receive expressions and receive case statements.

A buffer send action,

SEND buffer_name (value) [priority]

causes the specified value to be stored in the buffer location. If there is no room in the buffer the process executing the send statement is delayed (with optional priority) until a space becomes available or the value being sent is consumed.

There are two ways of receiving a value from a buffer. The receive expression,

RECEIVE buffer_name

delivers a value out of the specified buffer or from any delayed sending process. If there are no values in the buffer or delayed sending processes, the process executing the receive expression is delayed until a value is sent to the buffer.

The buffer receive case statement allows a process to receive a value from one of a number of named buffers with separate action code segments for each and an optional else clause (syntactically it is almost identical to the signal receive case). The process executing the receive case is delayed if no values are present in any of the buffers, no sending process is delayed on any of the named buffers and no else part is specified. The identity of the sending process may also be obtained.

A significant point about the buffer receive case statement and the receive expression is that their execution may result in the re-activation of a delayed sending process.

CHILL buffers differ from signals primarily in that they enable the user to explicitly control the allocation of buffers, whereas allocation for signals is performed automatically (CCITT, 1980b).

7.6 Mutual exclusion

The CHILL region is a means of providing processes with mutually exclusive access to locations. Regions may not be nested and must not be surrounded by a block (apart from the imaginary outer process). A region may consist of location declarations, definition statements (including mode and procedure definitions) and visibility statements (GRANT and SEIZE). It may also contain exception handlers.

    [name:]
    REGION [declaration statements] [definition statements] [visibility statements]
    END [name];

Processes wishing to access the locations declared in a region may only do so by calling procedures defined within and granted by the region (the common objects for which processes are competing may not be granted outside the region).

Procedures granted outside of a region are known as critical and may not be recursive.

Only one process can be active within a region at any given time. A region is locked if it is entered (by the outer process), one of its critical procedures is called or a process delayed in the region is re-activated. Any process attempting to access a region while it is locked will be delayed. A region is released (unlocked) if it is left, the critical procedure returns or the critical procedure becomes delayed. If more than one process is suspended awaiting access to a region and the region is released a process will be selected according to an implementation defined algorithm.

7.7 The outer process

As mentioned earlier, a CHILL program is considered to consist of a language defined “outer process” which is started when the host system begins execution of the CHILL program. However, despite its description as a process, Keedy (1981) points out that this program unit does not obey the same rules as other (user defined) processes, and that the language definition leaves some questions about it unanswered. For example, it is not clear whether it has an instance value or can synchronise with other processes using the normal mechanisms, it is not started or stopped in the usual way and, unlike other processes, has the ability to enter regions (presumably to initialise region data). He suggests that the CHILL programmer should not attempt to use the outer process as a concurrency tool and should confine its use to initialisation of data and the creation of other processes.

8. CONCURRENCY IN ADA

Ada tasks are entities which may operate in parallel. They are one of the three forms of Ada program units (with subprograms and packages) and may communicate and synchronise via entries.

Textually a task is similar to a package, with a separate specification and body. The specification may contain entry declarations (see section 8.3) and a representation specification which may specify how the entries or the task itself map onto the underlying hardware. The task body may contain local declarations, action code and exception handlers.

    TASK [TYPE] name IS
    [entry declarations]
    [representation specification]
    END [name];

    TASK BODY name IS
    [local declarations]
    BEGIN
    action statements
    [EXCEPTION exception handlers]
    END [name];

Notice that two different forms of task declaration are available. In the more general case, a task type specification defines a type which can be used to declare objects (instances) of this task (in the same way as standard variable declarations). Alternatively, a single task can be
defined in the same way as a task type except that the reserved word TYPE is omitted from the specification. This is considered to be equivalent to declaring an anonymous task type followed immediately by the declaration of an object of that type. Here, as in the reference manual, the task type will be considered, thus covering both cases.

Finally, task objects can be components of records or arrays or the objects designated by values of an access (pointer) type. A task object behaves as a constant and has the properties of a limited private type (see section 5.1).

8.1 Activation and termination of tasks

All tasks in Ada are enclosed in the declarative part of another program unit. Any objects declared to be of a task type will automatically begin execution before the first statement following the declarative part of the enclosing program unit. For example, assuming that a task type "resource" has been defined (DoD, 1981):

```
PROCEDURE p IS
  a, b: resource;  -- elaborate a and b
  . . c: resource;  -- elaborate c
BEGIN
  - a, b and c are activated in any order
END;
```

If the programmer needs the declared tasks to remain inactive until required, this could be achieved by including an accept (see section 8.5) as the first executable statement in the task so that it will be immediately suspended following activation until explicitly re-awakened.

Also, if it is not possible to statically predict the required number of task instances, tasks can be dynamically created by declaring a linked list structure in which each node contains a task object. The dynamic addition of a new node to the list, thus also results in the creation and activation of a new task.

An active task is normally terminated by reaching the end of its body. Any program unit containing a task can not be left until all of its subsidiary tasks have terminated. A task may also terminate by executing the terminate alternative of a selective wait statement (see section 8.6). Abnormal termination of one or more tasks is possible with the abort statement,

```
ABORT task_names;
```

which will unconditionally terminate the named tasks and any (nested) tasks "dependent" on the named tasks.

8.2 The delay statement

The delay statement simply suspends the task executing it for at least the specified time interval. Apart from this simple use it also has a special purpose in the selective wait statement (see section 8.6).

8.3 Entries

The primary means of communication and synchronisation between tasks are the entry call and accept statements. Task specifications may contain entry declarations (textually similar to subprogram declarations),

```
ENTRY name [(discrete_range)] [(formal parameter list)];
```

thus specifying the entries that other tasks may call. The actions to be performed when a declared entry is called are specified by a corresponding accept statement. The optional "discrete range" is used to declare a family of entries with the same name and parameters. In this case the individual entries are accessed via indices.

8.4 Calling entries

A task can call an entry in another task by specifying the entry name and an actual parameter list in the same way as a subprogram call (parameter passing is also analogous to subprogram calls). The calling task is suspended if the task owning the called entry has not reached a corresponding accept statement. Calls to the same entry are placed in a FIFO queue. An exception is raised if an attempt is made to call an entry in a terminated task or if the value in an entry family index is out of range.

8.5 The accept statement

A task body may contain one or more accept statements for each of its entries:

```
ACCEPT name [(formal parameter list)] [DO
  action statements
END [name] ];
```

A task executing an accept statement prior to any call of the named entry is suspended until such a call occurs. This means that the use of entry calls and accepts always results in a rendezvous. The calling task is suspended until the called task completes the action statements specified in the accept statement (if any). Following the rendezvous the tasks continue their (independent) parallel execution.

8.6 The select statement

Since it is often impossible to predict the order in which entry calls will occur, Ada provides the select statement which can be used in conjunction with the accept statement to enable a task to be suspended awaiting a call on any one of a number of named entries. Three different forms of select statement are available.

The selective wait statement allows two or more alternatives to be named, each with an optional condition which must be satisfied before the associated alternative may be selected. An else part may also be included.

```
SELECT
  [WHEN condition => ]
  alternative
  OR [WHEN condition => ]
  alternative
  . .
  [ELSE
  action statements]
END SELECT;
```

An alternative may consist of an accept statement followed by action code, a delay statement followed by action code or the reserved word TERMINATE. The selective wait may contain at most one terminate alternative and may not contain both a terminate and a delay alternative. Also, the use of either a terminate or delay alternative excludes the inclusion of an else part.

When executed a selective wait statement behaves as follows. Each alternative is examined to see which are open
(an alternative is considered to be open if its when clause evaluates to "true" or if there is no when clause). If there is an open alternative with an accept statement for which a rendezvous is possible (i.e. the named entry has been called by another task) it can be immediately executed, otherwise the task is suspended until a rendezvous is possible for an open alternative. An open alternative starting with a delay statement will be selected if no other alternative is selected before the specified time has elapsed. An open terminate alternative (which causes natural termination of the task) will only be selected if the "parent" program unit "owning" the task is itself ready to terminate or be left and is waiting only for the termination of its dependent tasks (including the task executing the selective wait). Finally, where included, an else part will be executed if no alternative can be immediately selected. Where several accept alternatives may be selected the choice is implementation dependent.

The second type of select statement is the conditional entry call which issues an entry call only if a rendezvous is immediately possible, otherwise the else part is executed:

```
SELECT
  entry_call [action statements]
ELSE
  action statements
END SELECT;
```

The third select statement, the timed entry call issues an entry call if the call can be accepted within a given delay,

```
SELECT
  entry_call [action statements]
OR
delay_statement [action statements]
END SELECT;
```

If a rendezvous does not occur within the specified interval the delay alternative is executed.

8.7 Some other properties of tasks

A task may optionally have a priority attached by including a priority pragma in its specification. The given value is used in the scheduling algorithm but unfortunately there is no provision for dynamic modification of priorities. Ada has certain pre-defined attributes for task objects that can be accessed from within the program. These include the priority of the task, number of storage units allocated for the task and whether or not the task has been terminated. The number of calls presently queued on an entry may also be determined.

Hardware interrupts can be handled by interpreting them as external entry calls and using a representation specification to link them with the physical object, e.g. an interrupt vector address (Ledgard, 1981). For example,

```
ENTRY stop_button;
FOR stop_button USE 8#7060#;
```

Finally, Ada assumes that it is the programmer’s responsibility to ensure mutual exclusion between tasks using common global variables. However, to enable the program to make sure that a variable is always updated with its latest value (since it may be maintained in a local register), the generic library procedure “shared_variable_update” is provided which can be called whenever the variable is to be accessed.

9. COMPARISON OF CONCURRENCY IN CHILL AND ADA

On evaluating both designs, one is struck by the more compact, possibly more elegant concurrency constructs in Ada. The Ada “rendezvous” concept contains most of the concurrency facilities required for task communication and mutually exclusive access to data. In CHILL, these facilities are provided by buffers, events, signals and regions. This large number of constructs may needlessly clutter the language. Keedy (1981) suggests that either signals and modules or buffers and processes are sufficient to provide all of the concurrency requirements of the CHILL programmer. Even the CHILL introduction warns that “care should be taken not to mix the various methods within one subsystem” (CCITT, 1980b).

Furthermore, whereas CHILL introduces a special mode INSTANCE to reference specific processes which have been started, Ada achieves greater orthogonality by allowing tasks to be types, and hence allowing the declaration of arrays and linked lists of tasks. The CHILL instance mode would also seem to be a threat to programming security, since it allows an instance to refer to any defined process (and hence there is no “strong typing” for processes). Against this, Ada has been heavily criticised since it lacks a facility equivalent to the THIS operator in CHILL. There is no obvious way for a task to identify itself in Ada.

Also, the CHILL approach to concurrency may map more closely onto the application domain for which it is intended, and the provision of regions is very close to the well accepted and disciplined concurrency construct of monitors suggested by Brinch-Hansen (1977) and Hoare (1974).

However, Ada allows the creation of all the apparently missing concurrency facilities, by providing enough primitive constructs to allow generic library facilities for tasking to be standardised. This keeps the impact of tasking on the syntactic core of the language to very few constructs.

Calls to the same entry in Ada are queued in FIFO order but the CHILL language definition appears to leave the queueing mechanism for signals undefined. The authors’ experiences with CHILL indicate that signals are by far the most widely used concurrency tool among its intended user community, and this “grey area” will make it very difficult to write portable CHILL programs. In fact programming with signals becomes excessively complicated if, for example, it is not possible to assume that the digits dialed at a subscriber’s telephone will arrive in their correct sequence. Nevertheless, the examples given to illustrate the use of signals often rely on FIFO queueing so an implicit assumption regarding this matter exists, however a clear statement on signal queueing in the language definition (which represents the definitive reference for compiler writers) would be greatly appreciated.

Finally, the authors believe that the rendezvous concept of Ada is a new and elegant contribution to controlling concurrency. Rather than being restrictive, the discipline imposed by using the rendezvous concept will free the programmer to create and manage the complexity of large real-time programs. Although not a new concept, the same comments apply to monitors (CHILL regions).

10. CONCLUSION

The modularisation and concurrency features of
CHILL and Ada have been compared by two authors currently engaged in a research project to automate program code generation in the telecommunications area. Both CHILL and Ada are large modern languages which accept restrictions and limitations to their power of expression in the interests of program reliability and efficiency of implementation. Both languages are the product of large design exercises and both languages would seem to achieve their major aims.

In summary, it was found that the package in Ada provided a superior mechanism for supporting the use of "abstract data types" than the module in CHILL. Also, it was felt that the multitude of concurrency features in CHILL presented the programmer with a rather confusing setup, although powerful, set of concurrency tools compared to the simpler approach offered by Ada.

If this paper suggests that Ada is a slightly better product than CHILL, then perhaps this reflects a number of political concessions made during the CHILL design. The second author was horrified when, attending the Melbourne CHILL design conference approximately three years ago, he noted language facilities included essentially because a politically powerful group had "guessed the standard" and had committed their compiler development to the inclusion of these features!

It may also be fair to conclude on the point that the cost of the Ada development effort is at least an order of magnitude higher than that of CHILL, and - in some respects at least - this shows.

REFERENCES


BIographies

Colin Fidge is currently a full-time postgraduate student in the Department of Computing, Royal Melbourne Institute of Technology, working towards a Masters Degree in Computer Science. His research interests include techniques for the automatic generation of high-level language code from Telecommunications Program Specifications, and in particular the Data Structures required to support this process. He received the BAppSc Degree (with distinction) from RMIT in 1981.

Bob Pascoe is a Lecturer in Computer Science at the Royal Melbourne Institute of Technology, where his research interests are high level languages, programming methodologies and high level language compilers. He represents Australia on the CCITT (Telecommunications Affiliate of the United Nations) where he sits on Study Group XI. Although he is principally concerned with the design of the Specification and Description Language, SDL, one of Study Group XI's roles is the design of the CHILL Language discussed in this paper.

Both authors would like to acknowledge the assistance of Telecom Australia, this paper being a sideline of a research program conducted at RMIT and funded by the Telecom Research Laboratories.
Commercial Programming Productivity — An Empirical Look at Intuition

By M. J. Lawrence* and D. R. Jeffery*

Two empirical studies have been carried out to examine a number of hypotheses related to the effects of organisation and technical factors on programming productivity. In the first study a comparison of three organisations was carried out, while in the second study, data from twenty-two organisations was analysed. The first revealed similar programming time equations for each organisation, with lines of code having the highest correlation coefficient with programming time. In both studies a number of counter-intuitive results were observed. For example, the data indicated that programmer experience, programming methodology or on-line testing had very little impact on productivity.

Keywords and phrases: programming productivity, structured programming, programming tools.


1. INTRODUCTION

The control of software development costs is a primary concern of data processing managers. The proliferation and marketing of software development aids and methodologies promising productivity gains and improved software maintenance has further forced the manager to consider this issue. However, many data processing managers, because of the lack of a suitable performance metric and public statistics, are unsure of how to assess their own department's performance and of how this compares to other organisations and industry averages. From our experience there is lack of agreement in industry as to what constitutes good and bad performance, even in programming, and furthermore most organisations do not collect statistics on their own software development. This denies them the essential tool for management control of a standard to compare actual performance.

The purpose of this paper is to discuss the programming productivity research being carried out at the Information Systems Department of the University of New South Wales and place it into the context of published material in this field.

2. PRIOR RESEARCH

The published material in programming productivity spans some twenty years indicating an enduring interest in the topic, but the lack of consensus as to the definition and determinants of programming productivity are witness to the difficulty of the field.

Most of the published studies in the area of systems development productivity focus on programming and are based on regression analyses of relatively small samples of empirical data collected from one organisation. A number of problems prevent results from one study being easily related to another study. These problems include: the use of subjective assessment variables, differences in data collection methods, lack of publication of the primary data and differences in the productivity metric used. Another drawback in the published material is the absence of empirical studies comparing the value of software development aids and discovering principles that can be used to guide their selection.

A satisfactory metric is a pre-requisite for measuring programming productivity. Although lines of code per unit time is the most widely used measure it does have a number of drawbacks. Some of these are:

(a) the influence of language level and re-used code on the line count,
(b) the influence of tools and methodology used on the line count,
(c) the inability to account for quality, and degree of difficulty of the job, and
(d) the measurement variation possible in counting lines (all lines, executed lines, statements, procedure lines) and in measuring time.

Some of these problem areas have been addressed by authors such as Gilb (1976), Boehm (1973), Boehm, Brown and Lipow (1976), Halstead (1977), Freburger and Basili (1979), Andersen and Schneiderman (1977), Brooks (1975) and Jones (1978). Despite the limitations of the lines of code per unit time measure Jones concludes it is a useful measure and capable of further development. Jeffery and Lawrence (1981) considered the problems of interpreting the current literature on programming productivity from the practitioner's point of view and advanced some possible explanations for the apparent inconsistencies.

One of the earliest studies, which still ranks as one of the largest, was carried out by the System Development Corporation and reported by Nelson (1967). A sample of 169 'programming projects' was analysed to identify the factors causing variability in programming time. Elapsed time and computer run time. The sample was drawn from both defense and non-defense organisations and that constitutes one of the very few multi-organisation studies. Unfortunately the data was regarded as being drawn from the one population and no analysis of variability between organisations was carried out. Perhaps this assumption contributed to the poor regression fit obtained ($R^2 = 0.58$)
(Nie 1975). Jeffery and Lawrence (1979) discuss the usefulness of the equation given for estimating total effort and show that under a plausible and consistent set of assumptions the effort equation gives the value of minus 39 man months of effort! Clearly one of the basic rules for a model, that it does not yield silly results, is violated in this case.

Perhaps the most quoted research in this field was carried out by Sackman, Erikson and Grant (1968). These authors performed a laboratory designed experiment to compare the influence of on-line and off-line computer access on total programming time. Two groups in a latin square design, coded and debugged a maze problem and an algebraic problem. The most discussed result of the study was the 20:1 ratio between best and worst individual times for both the coding and debug stages.

In 1977 Walston and Felix (1977) reported the results of the analysis of sixty projects developed by the IBM Federal Systems Division. They defined productivity as the ratio of delivered source lines to total effort (man months). Their analysis objectives were to:
(a) Provide data for the evaluation of improved programming technologies.
(b) Provide programming data to management.
(c) Gather and preserve software development historical records.
(d) Foster a common programming terminology.

Using a simple correlation Johnson (1977) analysed records. In a later paper Chrysler (1978a) the results of attempts to relate program development time to program and programmer variables is presented. The sample for the regression analysis consisted of 36 programs from one organisation. He found the most important variables influencing programming time were:
- number of input files,
- number of control breaks and totals,
- number of input edits,
- months of programmer experience at this facility, and
- number of input fields.

In a later paper Chrysler (1978b) explored the variables which contribute to the variability in program size. He found that programmer identity and experience did not appear to impact program size.

3. THE THREE ORGANISATION STUDY

The studies reported in the previous section each use different independent variables for their models and employ various approaches to data collection. Thus it is not possible to determine whether for each of these organisations a model of similar structure, but perhaps with different parameters, could be used to calculate programming effort successfully. Furthermore it would be useful to know, if a commonly structured model was used, how much less accurate it was than the ideal model for each organisation. These questions are motivated by a desire to understand the nature of the organisation specific aspects of program development, and to know the extent to which this impacts the structure of the programming effort model.

Research to examine these questions has been undertaken by the Information Systems Department at the University of New South Wales. Jeffery and Lawrence (1979) collected data on 93 programs developed by three organisations with each organisation contributing roughly a third of the programs. The organisations selected represented quite different programming styles and environments. Two organisations used structured code and one of these used walkthroughs on all programs while the third organisation developed quite unstructured code and exhibited fairly traditional attitudes to program design.

The same data definitions and data collection approach were used with each organisation. Seventeen data elements were collected for each program:
- programmer identification,
- company name,
- programming experience,
- program type,
- intended program use,
- program to be operated batch or on-line,
- data base interface,
- programming methodology (e.g. structured programming, top-down, walkthrough),
- testing facilities (e.g. batch, on-line),
- average turn-around,
- number of parallel tasks assigned to the programmer during work on this program,
- lines of code,
- procedure division (or equivalent) lines of code,
- program development time (man hours from receipt of program specification to end of unit testing),
- number of files interfaced to program,
- number of datatypes in program,
- number of procedural elements.

Since one of the aims of the research was to study the between-organisation effects the data was regarded as drawn from three distinct populations, and not merged together.

For all three organisations the variable most highly correlated with program development time (T) was number of lines of procedure code (PL). A linear equation gave the best model for each organisation. The equations are:

\[
\text{Org. 1 } T = 0.09 \text{ PL} + 4 : R^2 = 0.79 \\
\text{Org. 2 } T = 0.14 \text{ PL} + 9.5 : R^2 = 0.81 \\
\text{Org. 3 } T = 0.08 \text{ PL} + 43 : R^2 = 0.52
\]

Adding additional variables to the regression made some slight difference in organisation 3 (R^2 increased to around 0.60) but almost none of the other two organisations' models. Variables which are normally held to substantially impact program development time such as programmer experience and testing turn-around time did not contribute to the model, even when greatly enlarged.

While the fit obtained for regressions on program length (R^2 range from 0.36 to 0.71) were not nearly as good as for program development time, the same model structure appeared appropriate for each organisation with not greatly dissimilar coefficients. Thus where D is the number of data types, the equations are:

\[
\text{Org. 1 } \text{PL} = 13.6D - 8 : R^2 = 0.71 \\
\text{Org. 2 } \text{PL} = 20.5D - 22 : R^2 = 0.38 \\
\text{Org. 3 } \text{PL} = 18.9D + 393 : R^2 = 0.36
\]

Programmer experience did not significantly impact program length.

It would appear that the two organisations using more modern programming practices, organisations 1 and 2, are fairly similar in their models. Organisation 3 seems to
stand out as an exception, but it still conforms to the same model structure and with similar model coefficients.

The lack of impact of programmer experience and the failure, using the programmer identification, to isolate individuals with greatly above average programming productivity was quite unexpected. Some comments on this are advanced later.

Thus in summary it was concluded that:

- The model for program development time is similar for each organisation, although the goodness of fit varies.
- The model for program length is similar for each organisation but with a generally lower goodness of fit (R²) than for development time.
- The model coefficients are roughly the same but the constant terms are strikingly dissimilar.

4. THE MULTI-ORGANISATION STUDY

The productivity research to date has not focussed directly on productivity differences between organisations and investigated its magnitude and to what extent it can be explained by technology factors (e.g. quality of computer interface for testing), methodology factors (e.g. use of structured programming, use of walkthroughs) programmer characteristics and organisation industry. This topic has formed the research objective of the study reported by Lawrence (1981).

Data on 248 commercial type programs was obtained from 22 organisations to enable a number of hypotheses to be investigated. The principal hypotheses were:

- Significant variations in productivity exist between organisations,
- Industries like banks and insurance companies where EDP is a vital success factor, have above average productivity,
- Certain programmer characteristics, tools, methodologies and technologies substantially improve productivity.

Examples of the programmer characteristics, tools, methodologies and technologies mentioned in the above hypothesis are:

- Programmer experience,
- Structured programming,
- Program walkthroughs,
- On-line testing facilities, and
- Database interface.

The data was obtained using the same questionnaire and data forms as used in the 3 organisation study. The productivity metric chosen was lines of procedure division code per hour where the time was measured in man hours from the receipt of program specifications to the completion of unit testing. This metric has been shown to be a reliable productivity measure when used for organisations utilising the same program development standards. Since not all the 22 organisations in this study do observe the same development standards it is possible that this metric exhibits a bias for some organisations. However it is believed that this bias will be significantly smaller in magnitude than the reported size of the productivity effect due to the use of some of the programming productivity techniques. For example Yourdon (1975) quotes programming productivity improvement claims of “five or more” following the introduction of these techniques, while Jeffery and Lawrence (1979) estimated the number of lines of code used by one organisation writing particularly unstructured code at 50% greater than that used by another organisation writing structured code.

The majority language in the sample (216 programs) was COBOL with the remainder largely PL/I and BASIC. Assembly programs were excluded from the sample due to the problems they raise with the productivity metric.

Table 1 lists the average productivity for each organisation in the sample and the average for the four industry groupings. There are no significant industry effects on productivity except perhaps that the manufacturing and mining industry group exhibits far greater between-organisation variance. Two-thirds of the organisations exhibit productivity that could have been sampled from the one population (using a test based on sampling theory). Thus there is limited support for significant organisational differences in productivity and no support for the hypothesis of industry effects.

It is widely reported that structured programming and walkthroughs impact productivity by reducing testing time. The sample data was analysed two ways to investigate this claim. Firstly the proportion of time spent on the various activities of programming were calculated. Time was recorded by the following breakdown, preparation and design, coding, debug to obtain clean compile, debug to unit testing, other (e.g. documentation). Table 2 lists the proportion of time for

- All COBOL programs
- COBOL using structured programming (SP)
- COBOL not using SP
- COBOL using walkthroughs.

This table is prepared using only COBOL programs to remove one additional source of variation. It can be seen that the percentage of time devoted to debugging is about...
Commercial Programming Productivity

Table 2. Allocation of Programming Time — COBOL

<table>
<thead>
<tr>
<th></th>
<th>All COBOL</th>
<th>COBOL using SP</th>
<th>COBOL not using SP</th>
<th>COBOL Walkthrough</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cases</td>
<td>155</td>
<td>129</td>
<td>26</td>
<td>77</td>
</tr>
<tr>
<td>Mean</td>
<td>16%</td>
<td>15%</td>
<td>21%</td>
<td>15%</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>11.2</td>
<td>11</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>Prep. and Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td>37%</td>
<td>38%</td>
<td>32%</td>
<td>38%</td>
</tr>
<tr>
<td>Total debug</td>
<td>40%</td>
<td>40%</td>
<td>39%</td>
<td>40%</td>
</tr>
<tr>
<td>debug compile</td>
<td>12%</td>
<td>11%</td>
<td>11%</td>
<td>13%</td>
</tr>
<tr>
<td>debug test</td>
<td>28%</td>
<td>29%</td>
<td>28%</td>
<td>27%</td>
</tr>
<tr>
<td>Other</td>
<td>7%</td>
<td>7%</td>
<td>8%</td>
<td>7%</td>
</tr>
<tr>
<td>PL/T</td>
<td>8.1%</td>
<td>7.9%</td>
<td>8.6%</td>
<td>7.4%</td>
</tr>
</tbody>
</table>

the same for all four groups, indicating that the technique does not impact time allocation. Also in Table 2 the productivity (PL/T) for each group is given. This exhibits no significant differences between the four groups indicating that productivity is roughly constant independent of programming technique used.

Table 3 gives the effect of programmer experience on productivity, with the sample divided by program type to remove this as a possible source of bias. Only in the edit programs is there sufficient depth to make a comparison. Here it appears that productivity peaks after one year with no difference in productivity between the intermediate (2-3 years experience) and the experienced (greater than 3 years). The total sample reveals the same picture, thus giving no support to the hypothesis of experience greatly impacting productivity.

The effect of on-line testing was investigated. Using regression analysis the coefficient for the on-line testing variable was found to be significantly less than zero. Thus in this sample on-line testing was associated with lower productivity, once again not supporting the hypothesis.

In summary no empirical evidence could be found supporting the hypothesis that 'better' tools and more experience aid productivity. Additionally only scant evidence could be found supporting widespread organisational differences in programming productivity and no industry effect was present.

5. CONCLUSIONS

This paper has discussed some of the software engineering research that has been carried out at the University of New South Wales recently, focussing on two studies which were concerned with programming productivity. The first of these, the three organisation study, was devised to overcome some of the weaknesses of prior research by specifically investigating inter-company differences with respect to a basic model for programming effort and program length. It was found that for each organisation the model was very similar in structure and the same variables are significant in their influence on programming effort and program length in all three organisations.

During this study certain counter-intuitive results, such as the role of programmer experience and development environment on programming productivity, led to the multi-organisation study. In this study it was found that there were no industry effects on productivity and only limited support for organisational differences. Furthermore, the programming methodology used did not impact the proportion of time spent in different activities within the programming task, nor did it significantly impact programming productivity. Additional tests did not support on-line testing of years of programming experience as contributors to increased productivity.

These results do not support the claims made over the years for some of the 'intuitive' productivity advantages in different approaches to software development. Some of the possible reasons for these results, such as the influence of staff allocated to tasks, have been investigated (Jeffery and Lawrence, 1979), but it is also quite possible that any productivity improvements resulting from the variables investigated are small and not detectable using the simple measurements made to date. Better productivity metrics which capture more of the 'essence' of software may be needed before the 'intuition' can be supported by empirical evidence.

While it is not yet clear what the explanation is for these counter-intuitive findings, other research in this area has yielded similar results. Shiel (1981) surveys a wide body of empirical research in programming and concludes a lack of evidence supporting much of today's conventional wisdom.

Table 3. Average Productivity by Experience and Program Type.

<table>
<thead>
<tr>
<th>Program – Type</th>
<th>Experience</th>
<th>Trainee 0-1 year</th>
<th>Intermediate 2-3 years</th>
<th>Experienced &gt; 3 years</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cases</td>
<td>Mean</td>
<td>Cases</td>
<td>Mean</td>
</tr>
<tr>
<td>Utility</td>
<td>1</td>
<td>6.4</td>
<td>1</td>
<td>8.4</td>
</tr>
<tr>
<td>DB Interface</td>
<td>6</td>
<td>5.4</td>
<td>23</td>
<td>7.6</td>
</tr>
<tr>
<td>Edit</td>
<td>29</td>
<td>7.1</td>
<td>35</td>
<td>8.3</td>
</tr>
<tr>
<td>Update</td>
<td>1</td>
<td>4.2</td>
<td>1</td>
<td>19.3</td>
</tr>
<tr>
<td>Processing</td>
<td>6</td>
<td>4.8</td>
<td>7</td>
<td>7.9</td>
</tr>
<tr>
<td>File Extract</td>
<td>5</td>
<td>9.8</td>
<td>11</td>
<td>10.0</td>
</tr>
<tr>
<td>Report</td>
<td>7</td>
<td>8.7</td>
<td>9</td>
<td>11.2</td>
</tr>
<tr>
<td>Others</td>
<td>1</td>
<td>11.2</td>
<td>1</td>
<td>6.5</td>
</tr>
<tr>
<td>All programs</td>
<td>56</td>
<td>7.1</td>
<td>87</td>
<td>8.7</td>
</tr>
</tbody>
</table>
Commercial Programming Productivity

Further work is in progress at the University of New South Wales by the authors, focusing on behavioural factors influencing productivity.

REFERENCES

ANDERSON, N. and SHNEIDERMAN, B., Use of peer ratings in evaluating computer program quality, IFSM TR No. 20, Department of Information Systems, University of Maryland (June 1977).


CHRYSLER, E., Some basic determinants of computer programming productivity, Communications of the ACM, 21, 6 (June 1978a), pp. 472-483.


NELSON, E.A., Management handbook for the estimation of computer programming costs, System Development Corp., Santa Monica, California (March 1967).


BIOGRAPHICAL NOTE

Ross Jeffrey is a senior lecturer in Information Systems at the University of New South Wales. He holds a B.Com (Hons) in Accountancy from the University of Queensland and an M.Com (Hons) in Information Systems from the University of New South Wales. He has also held positions at the University of Queensland, Bendigo College of Advanced Education, and the University of Minnesota. His publications include papers in the areas of organisational objectives, programming productivity, and EDP management, and three books on information systems.

Michael Lawrence is senior lecturer in Information Systems at the University of New South Wales. He holds a BSc, BE from the University of Sydney and MS and PhD from the University of California, Berkeley, majoring in operations research. His current research is in the areas of software models and metrics and forecasting decision support systems. He is a member of TIMS, ACM, IEEE Computer Society and ACS.
Student Selection for Computing Courses

By D. Woodhouse*

This article refers to the Diploma in Computer Science at La Trobe University. This course is open to graduates of other disciplines, and so it was difficult to determine appropriate criteria for selecting from the large number of applicants. Over a four year period, a number of tests were administered to students selected for the course, and the results correlated with performance in the course. Significant positive correlations with the results of the Computer Programmer Aptitude Battery have been obtained. This test is now being used as a major selection mechanism for the course.

Keywords and phrases: Aptitude tests, diploma course, selection criteria.

1. INTRODUCTION

In 1975, the 'Barry/Barry report' (Smith and Ferranti 1975) recommended an increase in the number of diploma courses in computing: that is conversion courses, taking graduates of other disciplines and training them in computing.† A course of this type was then developed at La Trobe University, and has been offered each year from 1977 onwards. It was quickly found that the number of (graduate) applicants regularly exceeded the number of places by a factor of about four. This necessitated a stringent selection process, and raises the question 'What are the appropriate selection criteria?' A number of possibilities spring to mind.

1.1 Performance in computing courses

This can be quite useful. However, the whole point of the diploma course is to take people with no computing background. It is therefore anomalous to make prior experience a prerequisite.

1.2 Achievements in mathematics

This attempts to overcome the objections to 1.1. Firstly, mathematics is seen to be cognate with computer science, and secondly, most people have prior experience in mathematics. However, mathematical achievement has not proved to be a good predictor of achievement in computing (see, for example, Leeper and Silver, 1982). Furthermore, many current applicants for diploma courses have been repelled by mathematics for a variety of non-academic reasons, and not simply a lack of mathematical ability.

1.3 A science degree

This has the effect of excluding a large number of applicants, and telling them that they cannot simply retrain in computing but must go even further back in educational process. It also defines computer professionals as scientists. This is unfortunate, since those in computing must communicate with people from a wide spectrum of backgrounds and abilities (cf. Snow, 1964).

1.4 A commerce degree

This has the same defects, mutatis mutandis, as 1.3.

1.5 High academic performance

This is what we have used, but it is a very crude criterion. Over the years, people with doctorates have dropped out or performed badly; people with minimal qualifications have succeeded admirably; and all results between these extremes have also occurred. Gearing (1959) says 'choose plodders who will attend to detail, rather than those who may be very bright, but be inclined to spend too long searching for short cuts'.

1.6 Motivation and/or personality

There have been a number of attempts to investigate the relationship between personality and success in the computing profession (Crawley and Morris, 1970; Morris and Martin, 1977; Avery and Whitaker, 1976). However, the correlations are not high. Motivation is difficult to determine, especially since it may have so many different sources. (See Woodhouse, D., "Course Design and Implementation: Theory and Practice", to appear, in which the nature of this particular diploma course is discussed in more detail.) We have considered interviewing applicants, but should prefer to avoid conducting hundreds of interviews.

1.7 Aptitude and/or intelligence

This is really what we want: not a retrospective look at an applicant's performance so far, but some sort of crystal ball to predict his performance in future. A prospective student needs either certain basic capabilities or the ability to acquire them (Woodhouse, op. cit.). It may be possible to ascertain the former from the results of his first degree and/or other completed courses. It may be possible to ascertain the latter by applying certain tests. This possibility has been investigated since the course began, and it is the purpose of this paper to report on the results to date.

* Department of Computer Science, La Trobe University, Bundoora, Victoria 3083. Manuscript received June 1982, revised October 1982.

Copyright © 1983, Australian Computer Society Inc.

NOTE: In this article, the term 'computing' includes the areas commonly called 'computer science' and 'electronic data processing'.

Department of Computer Science, La Trobe University, Bundoora, Victoria 3083. Manuscript received June 1982, revised October 1982.

Student Selection for Computing Courses

2. APTITUDE TESTS

Many computer manufacturers use aptitude tests as part of their selection process for employees. These tests have the form of a typical intelligence test, possibly augmented with some specifically computing terms. Two such tests were investigated by Avery and Whitaker (1976). Performance on a course unit in elementary computing as part of the Bachelor of Business degree at the Tasmanian College of Advanced Education was correlated with:

(a) the two manufacturers' aptitude tests,
(b) the factors of neuroticism and introversion of the Eysenck Personality Inventory, and
(c) academic performance in all units previously studied at TCAE,

with the following results.

Factors compared                      Correlation
Performance on computer unit/aptitude test A  0.25
Performance on computer unit/aptitude test B  0.30
Performance on computer unit/EPI neuroticism -0.36
Performance on computer unit/previous academic performance  0.58
Aptitude test A/aptitude test B            0.78

It is clear that the two manufacturers' aptitude tests are measuring similar factors, but these seem to be independent of whatever makes for success in a computing course. Conversely, performance in other, non-computing courses at the college correlates fairly well with performance in the computing course. This suggests that previous academic performance should be a useful guide.

Since the inception of our course, therefore, students have been selected principally on the basis of their academic record. As noted under point 1.5, however, the results of this procedure have not been as good as we should like. Therefore, all students selected have been given a variety of tests before starting the course. At the end of the course, the test results and the students' achievement on the various sections of the course have been examined for significant patterns, correlations, etc. Rowan (1957) calls this the follow-up method, and says that it is 'perhaps the best way of establishing a selection programme'. Over four years, some test results have, like those reported above, borne little relation to the course results. Other tests, however, namely a certain set of computer-oriented aptitude tests, have shown much closer correlation with course results. There seems, therefore, to be some hope of improving the selection process by using these tests.

3. THE TEST PROGRAMME

A priori, it seems that a student's ability to acquire the necessary capabilities (1.7) may depend on:

(a) intelligence, and/or
(b) character, and/or
(c) aptitude.

By 'intelligence' we mean what is measured by intelligence tests. (This apparent tautology is now widely accepted as the only possible definition.) By 'character' is meant some innate or developed aspects of a person's nature, such as perseverance, sensitivity, self-assurance. By 'aptitude' is meant the possession of certain knowledge, skills or methods of working, which, prima facie, seem to dispose to the acquiring of the capabilities in question. A number of tests were selected to assess these three features, namely Raven's Advanced Progressive Matrices (RAPM) and the Watson-Glaser Critical Thinking Appraisal (WG) for intelligence; Cattell's Sixteen Personality Factor Questionnaire (16PF) for character; and the Computer Programmer Aptitude Battery (CPAB) for aptitude. The choice of these tests was influenced by their appropriateness, by their availability, and by other work in the field. Crawley and Morris (1970) and Morris and Martin (1977) have found some correlations between performance on some of these tests and professional competence. Bell (1976) found some positive and some inverse correlation in a student environment. Anderson and Gorgone (1975) and Wolfe (1977) have also studied the use of aptitude tests for predicting performance in study and employment respectively. Bauer (1968) reports attempts to predict performance in study, while Hanson (1975) and Stalnaker (1970) consider problems of selecting for systems analysis work in the UK and USA respectively.

Test 1: Raven's Advanced Progressive Matrices (Raven, 1965) (RAPM)

This is a non-verbal test, which thereby avoids some 'culture-specific' problems. All the questions require a pattern to be completed by the insertion of the appropriate one of eight pieces. Figure 1 shows the form of the questions. The test was designed:

(a) to determine quickly whether a person is in the dull 10%, average 80%, or bright 10% of the population;
(b) to assess a person's total capacity for observation, coherent perception and orderly thinking; and
(c) to test intellectual efficiency, that is, ability to perform well under stress.

![Figure 1. A frame typical of Raven's Advanced Progressive Matrices (See section 3).](image-url)
Test 2: Watson-Glaser Critical Thinking Appraisal
(Watson and Glaser, 1964) (WG)

Critical thinking has been defined as involving the following abilities. The ability to:
(a) define a problem;
(b) select pertinent information for the solution of a problem;
(c) recognize stated and unstated assumptions;
(d) formulate and select relevant and promising hypotheses; and
(e) draw conclusions validly, and judge the validity of inferences.

These abilities are assessed by the WG through the medium of five tests, designed to test inference, recognition of assumptions, deduction, interpretation and evaluation of arguments, respectively.

An example of an inference exercise is as follows:

Two hundred eighth grade students voluntarily attend a recent week-end student forum conference in a mid-western city. At this conference, the topics of race relations and means of achieving lasting world peace were discussed, since these were the problems the students selected as being most vital in today's world.

On the basis of this statement classify the following statements as true, false, probably true, probably false or indeterminate (due to insufficient data):
1. As a group, the students who attended this conference showed a keener interest in humanitarian or broad social problems than have most eighth-grade students.
2. The majority of these students were between the ages of 17 and 18.
3. The students came from all sections of the country.
4. The students discussed only labour relations problems.
5. Some eighth-grade students felt that discussion of race relations and means of achieving world peace might be worthwhile.

In the deduction section, one is asked to say whether certain conclusions do or do not follow. For example:

Some holidays are rainy. All rainy days are boring.
Therefore:
1. No clear days are boring.
2. Some holidays are boring.
3. Some holidays are not boring.

The WG contains quite different material from typical IQ tests, and correlations between WG and intelligence tests usually produce values in the range 0.55 to 0.75. This suggests that intelligence may be necessary but not sufficient for critical thinking.

In this study, only the exercises on inference and deduction were used.

Test 3: Computer Programmer Aptitude Battery
(Palermo, 1974) (CPAB)

The CPAB comprises five separately timed tests, measuring skills and aptitudes as follows:
1. Verbal Meaning — Each item presents a word followed by five alternatives, from which is to be chosen the one nearest in meaning to the first word. The vocabulary represented is that commonly used in mathematical, business, and systems-engineering literature.
2. Reasoning — A test of the ability to translate ideas and operations from word problems into mathematical notations. For example, 'What is the total cost of twelve chairs which cost c dollars each, three tables at t dollars each, and a bar costing b dollars?'
3. Letter Series — Each item is a series of letters, in which a pattern is to be discerned. This is seen as a test of abstract reasoning ability.
4. Number Ability — A test of the ability quickly to estimate reasonable (but not necessarily exact) answers to computations.
5. Diagramming — A test of the ability to analyse a problem and order the steps for solution in a logical sequence, by completion of flow charts.

Before the development of the CPAB, the few research studies on predicting achievement in the computer programming field suggested that standardised tests of aptitudes and mental abilities would be no more than moderately successful. One defect is that the typical general aptitude test is designed to yield measurements across the total population, whereas the people usually considered for programming employment are a more restricted group. Thus, only a few of the test items operate effectively with this group. It was hypothesised that the predictive efficiency of the tests could be increased by making the range of difficulty of the items more appropriate for the typical prospective programmer.

The CPAB was developed by having supervisors rank their programming and systems analysis personnel, and then analysing the performance of these people on the tentative CPAB test items. The CPAB was then used in a variety of situations, and appears to have given moderate correlations.

Test 4: Cattell's Sixteen Personality Factor Questionnaire
(Cattell, 1970) (16PF)

This is designed to give as complete a coverage of personality as possible in a short time. It comprises 105 questions, each designed to probe one of 16 independent dimensions of personality. (There are 6 or 8 questions for each of the 16 factors, and 7 questions for checking purposes.) The 16 factors are:
A: reserved/outgoing
B: dull/bright
C: affected by feelings/emotionally stable
D: humble/assertive
E: sober/happy-go-lucky
F: expedient/conscientious
G: shy/venturesome
H: tough-minded/tender-minded
I: trusting/suspicious
J: practical/imaginative
K: forfhright/astute
L: self-assured/apprehensive
M: conservative/experimenting
N: self-sufficient/group dependent
O: self-controlled/self-conflict/controlled
P: relaxed/tense

4. RESULTS
The Diploma in Computer Science course at La Trobe University is divided into four sections, thus:
1. Programming (CP) — including Pascal, data structures, assembly language, software engineering.
Student Selection for Computing Courses

Table 1. 1977 Correlations (see section 4).

<table>
<thead>
<tr>
<th></th>
<th>CP</th>
<th>CO</th>
<th>CI</th>
<th>CM</th>
<th>DCS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>RAPM</td>
<td>26</td>
<td>0.05</td>
<td>22</td>
<td>-0.02</td>
<td>19</td>
</tr>
<tr>
<td>WG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.54*</td>
</tr>
<tr>
<td>INFER</td>
<td>15</td>
<td>0.39</td>
<td>13</td>
<td>0.40</td>
<td>13</td>
</tr>
<tr>
<td>DEDUC</td>
<td>15</td>
<td>0.58*</td>
<td>13</td>
<td>0.73**</td>
<td>13</td>
</tr>
<tr>
<td>TOTAL</td>
<td>15</td>
<td>0.53*</td>
<td>13</td>
<td>0.62*</td>
<td>13</td>
</tr>
<tr>
<td>CPAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERBAL</td>
<td>28</td>
<td>0.23</td>
<td>23</td>
<td>0.20</td>
<td>24</td>
</tr>
<tr>
<td>REASON</td>
<td>28</td>
<td>0.17</td>
<td>23</td>
<td>0.53**</td>
<td>24</td>
</tr>
<tr>
<td>LETTER</td>
<td>28</td>
<td>0.63**</td>
<td>23</td>
<td>0.55**</td>
<td>24</td>
</tr>
<tr>
<td>NUMBER</td>
<td>28</td>
<td>0.35</td>
<td>23</td>
<td>0.33</td>
<td>24</td>
</tr>
<tr>
<td>DIAGRM</td>
<td>15</td>
<td>0.55*</td>
<td>13</td>
<td>0.62*</td>
<td>13</td>
</tr>
<tr>
<td>TOTAL</td>
<td>28</td>
<td>0.46*</td>
<td>23</td>
<td>0.56**</td>
<td>24</td>
</tr>
</tbody>
</table>

** denotes 'significant at or beyond 0.01 level'.
* denotes 'significant between 0.05 and 0.01 level'.
N is the number of students involved in the respective correlations.

2. Organisation (CO) — including computer logic and design, microprocessors, compilers, operating systems, networks.
3. Information Systems (CI) — including systems theory, analysis and design, file structures and access, COBOL, DBMS.
4. Miscellany (CM) — Including managerial economics, accounting, operations research, psychology of groups and individuals.

For the years 1977 to 1979, the students' results on CP, CO, CI, CM and the total DCS mark were correlated against RAPM, WG (inference section, deduction section and total mark) and CPAB (individual sections and total mark), with the results indicated in Tables 1, 2 and 3. In 1980, only the CPAB was used (Table 4).

The 16PF test was used in 1977 to 1979, but the results were treated rather differently. For each of CP, CI and DCS, the mean score was obtained on each factor, for two groups of students, namely those with above average scores on that part of the course, and those with below average scores on that part of the course. This was done in order to distinguish the personalities of the two groups, if possible. The results are set out in Figures 2 to 10, in which the performance of above average students is indicated by the solid line, and the below average ones by the broken line. The central band denotes the level obtained by 40% of the population.

5. DISCUSSION

5.1 Some of the RAPM and WG results showed significance at the 1% level for some of the years studied, but none showed even 5% significance for all the years (see Tables 1 to 4).

5.2 The CPAB, however, performed much better than the RAPM and WG tests. The results are summarised in Figure 11, in which the entries are to be interpreted as follows: m,n means that, out of 4 years of observation (1977-80), this correlation was significant at between 5% and 1% on m occasions, and at ≥ 1% on n occasions.

** denotes 'significant at or beyond 0.01 level'.
* denotes 'significant between 0.05 and 0.01 level'.
N is the number of students involved in the respective correlations.

Table 2. 1978 Correlations (see section 4).

<table>
<thead>
<tr>
<th></th>
<th>CP</th>
<th>CO</th>
<th>CI</th>
<th>CM</th>
<th>DCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAPM</td>
<td>0.35</td>
<td>0.13</td>
<td>0.39</td>
<td>0.07</td>
<td>0.30</td>
</tr>
<tr>
<td>WG</td>
<td>-0.06</td>
<td>0.31</td>
<td>0.44**</td>
<td>0.07</td>
<td>0.25</td>
</tr>
<tr>
<td>INFER</td>
<td>0.28</td>
<td>0.23</td>
<td>0.19</td>
<td>-0.13</td>
<td>0.15</td>
</tr>
<tr>
<td>DEDUC</td>
<td>0.17</td>
<td>0.32</td>
<td>0.36</td>
<td>-0.05</td>
<td>0.23</td>
</tr>
<tr>
<td>TOTAL</td>
<td>0.16</td>
<td>0.34</td>
<td>0.61**</td>
<td>0.01</td>
<td>0.29</td>
</tr>
<tr>
<td>CPAB</td>
<td>0.25</td>
<td>0.62**</td>
<td>0.48**</td>
<td>0.21</td>
<td>0.49**</td>
</tr>
<tr>
<td>VERBAL</td>
<td>-0.01</td>
<td>0.13</td>
<td>0.20</td>
<td>0.25</td>
<td>0.15</td>
</tr>
<tr>
<td>REASON</td>
<td>0.34</td>
<td>0.60**</td>
<td>0.55**</td>
<td>0.29</td>
<td>0.56**</td>
</tr>
<tr>
<td>LETTER</td>
<td>0.39*</td>
<td>0.52**</td>
<td>0.39</td>
<td>0.32</td>
<td>0.52**</td>
</tr>
<tr>
<td>NUMBER</td>
<td>0.11</td>
<td>0.52**</td>
<td>0.62**</td>
<td>0.28</td>
<td>0.56**</td>
</tr>
</tbody>
</table>

** denotes 'significant at or beyond 0.01 level'.
* denotes 'significant between 0.05 and 0.01 level'.
N is the number of students involved in the respective correlations.

Table 3. 1979 Correlations (see section 4).

<table>
<thead>
<tr>
<th></th>
<th>CP</th>
<th>CO</th>
<th>CI</th>
<th>CM</th>
<th>DCS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(N=33)</td>
<td>(N=34)</td>
<td>(N=34)</td>
<td>(N=34)</td>
<td>(N=34)</td>
</tr>
<tr>
<td>RAPM</td>
<td>0.29</td>
<td>-0.32*</td>
<td>-</td>
<td>0.46**</td>
<td></td>
</tr>
<tr>
<td>WG</td>
<td>0.43*</td>
<td>0.59**</td>
<td>0.44**</td>
<td>0.30</td>
<td>0.59**</td>
</tr>
<tr>
<td>INFER</td>
<td>0.25</td>
<td>0.27</td>
<td>0.22</td>
<td>0.02</td>
<td>0.30</td>
</tr>
<tr>
<td>DEDUC</td>
<td>0.37*</td>
<td>0.47**</td>
<td>0.36**</td>
<td>0.16</td>
<td>0.49**</td>
</tr>
<tr>
<td>TOTAL</td>
<td>0.12</td>
<td>0.50**</td>
<td>0.42*</td>
<td>0.23</td>
<td>0.50**</td>
</tr>
<tr>
<td>CPAB</td>
<td>0.38</td>
<td>0.66**</td>
<td>0.43*</td>
<td>0.31</td>
<td>0.57**</td>
</tr>
<tr>
<td>VERBAL</td>
<td>0.34</td>
<td>0.51**</td>
<td>0.20</td>
<td>0.32</td>
<td>0.52**</td>
</tr>
<tr>
<td>REASON</td>
<td>0.43**</td>
<td>0.48**</td>
<td>0.36**</td>
<td>0.21</td>
<td>0.51**</td>
</tr>
<tr>
<td>LETTER</td>
<td>0.40**</td>
<td>0.69**</td>
<td>0.48**</td>
<td>0.39</td>
<td>0.62**</td>
</tr>
<tr>
<td>NUMBER</td>
<td>0.37**</td>
<td>0.67**</td>
<td>0.46**</td>
<td>0.35</td>
<td>0.62**</td>
</tr>
</tbody>
</table>

** denotes 'significant at or beyond 0.01 level'.
* denotes 'significant between 0.05 and 0.01 level'.
N is the number of students involved in the respective correlations.

Table 4. 1980 Correlations (Computer Programmer Aptitude Battery only; see section 4).

<table>
<thead>
<tr>
<th></th>
<th>CP</th>
<th>CO</th>
<th>CI</th>
<th>CM</th>
<th>DCS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(N=43)</td>
<td>(N=37)</td>
<td>(N=36)</td>
<td>(N=37)</td>
<td>(N=36)</td>
</tr>
<tr>
<td>RAPM</td>
<td>0.53**</td>
<td>0.37*</td>
<td>0.17</td>
<td>-0.07</td>
<td>0.29</td>
</tr>
<tr>
<td>WG</td>
<td>0.67**</td>
<td>0.66**</td>
<td>0.20</td>
<td>0.18</td>
<td>0.58**</td>
</tr>
<tr>
<td>INFER</td>
<td>0.54**</td>
<td>0.50**</td>
<td>0.16</td>
<td>0.27</td>
<td>0.50**</td>
</tr>
<tr>
<td>DEDUC</td>
<td>0.51**</td>
<td>0.37*</td>
<td>0.08</td>
<td>0.31</td>
<td>0.39**</td>
</tr>
<tr>
<td>TOTAL</td>
<td>0.66**</td>
<td>0.54**</td>
<td>0.11</td>
<td>0.21</td>
<td>0.52**</td>
</tr>
</tbody>
</table>

** denotes 'significant at or beyond 0.01 level'.
* denotes 'significant between 0.05 and 0.01 level'.
N is the number of students involved in the respective correlations.

Table 5. Frequency of Occurrence of Non-Average Personality Factors, Shown by Students with Non-Average Course Marks (see section 5).

<table>
<thead>
<tr>
<th>Factor</th>
<th>Quality</th>
<th>Above-average students</th>
<th>Below-average students</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>reserved</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>bright</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>F</td>
<td>sober</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>H</td>
<td>venturesome</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>tough-minded</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>trusting</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>imaginative</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>O</td>
<td>self-assured</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Q1</td>
<td>experimenting</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>Q2</td>
<td>self-sufficient</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Q3</td>
<td>controlled</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>Q4</td>
<td>relaxed</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Obviously, the most interesting entries are those where \( m+n=4 \), and particularly those where \( m=0, n=4 \). It is encouraging to observe, therefore, that \( m=0 \) and \( n=4 \) for the overall CPAB/DCS correlation. This suggests the use of the CPAB for course selection, a process upon which we are now embarking.

The generally lower correlations with the mark on the 'miscellaneous' section of the Diploma course (CM) is probably because this section contains least directly computer-related material.

Another useful observation is the following. The designers and users of the CPAB have reported high correlations between the battery and programming competence. We have found moderately high correlations between the battery and performance in the Diploma course. We hope, therefore, that those who do well in the Diploma course will develop further when they are employed. Naturally, however, there will be some who succeed in the academic environment who fail to produce the same performance in practice. Conversely, some people who do not perform well in formal assessments have a tenacity which makes them effective programmers.

5.3 The 16PF showed the following pattern. Inspection of Figures 2 to 10 show that the above average and below average 'graphs' go outside the 'median strip' on a number of personality factors, on a number of occasions. The occurrence of these 'extreme' values is summarised in Table 5. Results of significance, are that success in the La Trobe Diploma Course seems to go to the person who is reserved yet experimenting, with trust, control and intelligence also...
being of value. However, these scores do not appear to depart from the mean sufficiently to justify the use of the 16PF in student selection.

The aptitude tests devised by individual companies often mix typical IQ test questions and personality questions with more computing-oriented questions. The results summarised in this section suggest that the effect of the latter types is diluted by the effect of the former types.

6. CONCLUSION
Among graduate students selected for a Diploma in Computer Science course, performance in the course correlates well with performance on the Computer Programmer Aptitude Battery of tests. This device will now be used (in conjunction with academic record) to select students for the course.

Copies of the CPAB may be obtained from the Australian Council for Educational Research, 9 Frederick Street, Hawthorn, Victoria, 3122, but only by persons and organisations competent and registered to administer such tests.

7. REFERENCES

Student Selection for Computing Courses

Figure 8. Personality Profiles, 1979, Programming Section.

Figure 9. Personality Profiles, 1979, Information Systems Section.

Figure 10. Personality Profiles, 1979, Total Diploma Result.

CPAB     CP     CO     CI     CA     DCS

verbatim
reason    0,3          3,0          1,3

letter
number   1,2          0,3          0,3

diagram  3,1          1,3          0,3

total    2,1          1,3          1,2

Figure 11. Correlation of the Computer Programme Aptitude Battery with the Diploma in Computer Science (see section 5). m,n means that, out of 4 observations, the indicated correlation was significant at between 0.05 and 0.01 on m occasions, and at or beyond 0.01 on n occasions. Correlations with m=0,n=4 are doubly hatched; other correlations with m+n=4 are singly hatched.

BIOGRAPHICAL NOTE

David Woodhouse obtained his MA from Oxford in Mathematics, MDPhil, also from Oxford, in abstract algebra. He taught in the University of East Africa for four years, and took an MSc in computer science. He joined La Trobe in 1969 as a lecturer in pure mathematics and introduced and developed computer science courses through the 1970s. He has been chairman of the Department of Computer Science since its establishment in 1978 and treasurer of the Victorian Branch of the ACS for five years. He was a founder member of the Computer Education Group of Victoria, and is chairman of its conference organising committee.


SMITH, B.W. and DE FERRANTI, B.Z. (1975): Computer Education Needs and Resources at Tertiary Level, Australian Commission on Advanced Education.


Book Reviews


The 1965 publication of J.A. Robinson's "A machine-orientated logical inference mechanism" marked the beginning of a new era in automated inference research. Logic programming (based on the premise that computation is controlled inference) grew from this work, and came of age in 1972 with the efficient implementation of PROLOG. The third edition of this book has been organised, as the title suggests, into various computer graphics features, selective backtracking and multiprocessor architecture. Logic Programming is concerned with the use of computers to simulate the processes of logic.

The basis of the volume, which consists of twenty-three papers, was in the first International Workshop on Logic Programming held in Debrecen, Hungary in 1980 where thirteen of the papers were prepared. Others were new papers or revised versions of papers previously only privately circulated, while one is an English translation of a manuscript previously published in French. The papers are grouped into ten sections under the following topics: introduction to logic programming, applications of logic programming, natural language understanding (including data base interfaces and story understanding), implementation issues (variable bindings etc. for PROLOG-like languages), specification and transformation (logic programs can often be seen as specifications of what they compute, and can be transformed using formal inference techniques), metalevel inference (discusses features that enable PROLOG programs to manipulate and run other PROLOG programs), control issues (parallelism, compilation, backtracking), logic programming languages (infinite data structure PROLOG; IC-PROLOG), logic in LISP (LOGLISP; PROLOG in LISP) and Horn clause computability.

The papers emphasise how successful the view of computation as controlled inference has been (an initiative taken by the Japanese in the design of their fifth generation machines) and serve as an introduction to, and review of, research in the field to date. Of special note is work reported in the areas of metalinguage features, selective backtracking and multiprocessor architecture.

Logic Programming would prove an invaluable addition to the reference library of the research worker in logic or programming. This value would extend to the student or professional programmer with an interest in the area, but they may find the cost prohibitive — especially for a book with such ordinary typesetting. A subject index and extensive bibliography are included.

Bradley W. White, University of New South Wales


It is difficult to place this book into a particular category. It is not a text, although it would make an excellent accompanying volume to one of the standard computer graphics texts. Its outstanding feature is the large number (40 pages) of high quality, full page size, colour plates of computer generated pictures, which are organised, as the title suggests, into various computer graphics applications: computer aided design, animation of animated films, synthetic landscapes, models of molecules, computer-aided design, remote sensing, and astronomy. The plates selected are quite different to those available at the annual Siggraph conferences.

Apart from this selection of computer generated pictures, there are four chapters surveying various aspects of computer graphics, each written by a different author. The largest of these, by Donald Greenberg, is an overview of computer graphics hardware and software as such is most commonly encountered in the standard computer graphics texts. The summary is particularly clear, non-mathematical and amply illustrated.

The second chapter follows the colour plates, and is an overview of colour, covering such topics as the components of colour, its production, and its interaction, but not the psychology of colour perception.

The third chapter, entitled "Computer-Aided Business Graphics", is the only real applications-oriented chapter. It describes some graphics systems which are currently commercially available for areas such as marketing, financial statements, financial trends, and investment analysis, and ends up with a basic businessman's guide to the hardware and software required.

The fourth and shortest chapter entitled "Polaroid Instant Photography in the Computer Graphics Camera" appears to be somewhat of an anomaly, until you turn to be confronted with the fact that the authors are connected with Polaroid, and that the book was "planned, prepared and produced by the Publications Department of Polaroid Corporation"! This chapter is technical in nature, providing the background and instructions on how to use Polaroid cameras to take pictures of computer generated images.

The main advantage of this book is its excellent colour plates, which would prove an invaluable addition to both the instructor and student of computer graphics alike. A lot of material in the section on colour is not in the standard texts, while the summary chapter would be invaluable for providing an overall perspective on the student who has become entrenched in the mathematics of some of the more involved parts of graphics. Overall, a good buy at the price.

P.G. McCrea, University of New South Wales


This is one of four volumes for 1982 in a new serial publication which provides information on research and development in Japan in the general area indicated by the title. The other three volumes for 1982 are in the areas of Semiconductor Technologies, Amorphous Semiconductor Technologies and Devices, and Optical Devices and Fibres. The latter volume was reviewed in this journal in November 1982. Computer aided design and network systems are among the areas to be treated in forthcoming volumes.

The volume under review consists of 22 papers, together with an introduction and an appendix which consists of an essay on policy making in Japan. The papers are grouped into five chapters under the following headings:

1. Pattern Information Processing System (PIPS).
2. Natural Language Processing.
4. Data Base and Knowledge Base.
5. Knowledge Engineering.

Each chapter commences with an introductory article which attempts to summarise the current state of the art in Japan, and is followed by some selected papers to represent the Japanese research scene.

Such a volume presents problems to a reviewer. One approach would be to separately review the individual papers, and this method has been followed by Computer Abstracts, which dealt with the book in May 1982. Several different reviewers would be needed to do justice to this task. For the present purpose a few general comments will be made.

The first chapter describes the PIPS project which ran from 1971 to 1981 and dealt with pattern processing and related topics. It demonstrates the ability of the Japanese research community to concentrate a substantial effort into an area of perceived importance.

The second chapter, dealing with natural language processing, is strongly influenced by the special problems of dealing with the Japanese language, including its translation and from English.

As the chapter headings would indicate, there is an AI flavour about much of the work, although I did not notice that term used anywhere. This emphasis is perhaps a little surprising, although the Japanese interest in robots is well known.

As would be expected in a book written in English by Japanese workers, the incidence of spelling errors and infelicities is higher than is normal in books written by native English speakers, but the meaning is always clear.

Because of the deliberate restriction to Japanese work, the book could not be recommended as an introduction to the topics with which it deals. However, it meets its aim of providing international readers with information on the design and development of artificial intelligence systems in the field of computer science. The book is handsomely bound and very expensive.

G.B. McMahon, University of New South Wales

"News Briefs from the Computer World" is a regular feature which covers local and overseas developments in the computer industry including new products, interesting techniques, newsworthy projects and other topical events of interest.

DATA PROCESSING VOCABULARY

The Standards Association of Australia has published new editions of two parts of its standard containing a data processing vocabulary, AS 1189 — Part 19 — Analog computing, and Part 24 — Numerical control of machines, are part of a series of revisions to supersede the existing standard publishing in 1972. The revisions are identical with the corresponding parts of ISO 2382 — Computers and information processing.

AS 1189 presents data processing concepts and associated terms which should be used in communicating with specialists in this field. This vocabulary deals with the main areas of data processing as well as particular applications.

Copies of AS 1189 can be purchased from any SAA office at a cost of: Part 1 — $5.80 plus $1.25 postage and handling charge; and Part 2 — $8.00 plus $1.50 postage and handling charge.

TEKTRONIX ANNOUNCES HIGH-QUALITY COLOUR GRAPHICS COPIER

Tektronix has added the Model 4691 colour graphics copier to its line of hard copy units. The unit uses an ink-jet technology to produce eight-colour copies from computer graphics systems. The 4691 is intended to address applications including simulation, design previewing, stress analysis, ultrasonic scanning, and surface modelling which generate complex plots requiring colour to aid discernibility.

The 4691, produces B-size (279 x 432 mm) and A-size (216 x 279 mm) output. Images can be produced in a horizontal or vertical format for 'A' or 'B' sizes, and multiple copies may be produced under program control.

The high quality images produced by the 4691 are the result of a combination of elements. According to Noel Winter, Information Display Division Manager, "Users are demanding higher copy quality from colour hard copy devices than from monochrome units. Factors like high resolution and excellent colour saturation, which provide optimum area fill capabilities, allow the 4691 to produce copies that meet the high expectations of colour graphics users."

Ink is supplied from separate ink cartridges containing yellow, cyan and magenta colour which mix to print red, green and blue. Black comes from a fourth ink cartridge, rather than from a mixture of primaries, resulting in a dense, solid black. The ability to copy fine patterns — a result of the 4691's high resolution and accurate dot placements — allows the 4691 to produce additional colours through a shading technique called dithering. For example, the 4113 Local Easy Graphing software will include routines which produce 125 callable shades for use on the 4691 colour copier.

For further information on the 4691 Colour Graphics Copier, please contact your local Tektronix office.

SPECIAL ISSUE ON SOCIAL CONSEQUENCES OF COMPUTING TECHNOLOGY

The Australian Computer Journal will publish a special issue on "Social Consequences of Computing Technology" in November 1983. Research papers, tutorial articles and industry case studies on all aspects of the subject will be welcome, and both full papers and short communications will be considered.

Prospective authors should write as soon as possible to:

Ashley W. Goldsworthy,
P.O. Box 554, Fortitude Valley, Qld. 4006

to notify him of their intention to submit material for the issue and provide a brief summary of their intended contribution.

In order to allow adequate time for refereeing and editorial review, complete manuscripts will be required no later than 15 June 1983.

Papers should be prepared in accordance with the guidelines published in the November 1982 issue of the Journal. Authors are requested to pay particular regard to the Journal's preferred style for references.
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 November 1982 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:

- Publisher: Service
- ACM: Bibliography and Subject Index of Current Computing Literature
- ACM: Computing Reviews
- AMS: Mathematical Reviews
- CSA: Computer and Information Systems Abstracts
- Data Processing Digest
- ENGINEERING INDEX INC.: Engineering Index
- INSPEC: Computer and Control Abstracts
- INSPEC: Electrical and Electronic Abstracts
- ISI: Current Contents/Compumath
- ISI: Compumath Citation Index
- SPRINGER-VERLAG: Zentralblatt fur Mathematik und ihre Grenzgebiete

Copyright © 1983. Australian Computer Society Inc.

Production Management: Associated Business Publications, Room 104, 3 Small 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: Publicity Press (NSW) Pty. Ltd. 66 O’Riordan Street, Alexandria, NSW 2015.