CONTENTS

RESEARCH PAPERS
1-8  A Query Interface for IMAGE Databases based on Relational Algebra
     C.S. NG and R. SACKS-DAVIS
9-17 Quokka: A Translator Generator Using Denotational Semantics
      T.N. VICKERS
18-25 A Clause Indexing System for PROLOG based
      on Superimposed Coding
      R.M. COLOMB and JAYASOORIAH
26-32 A Control and Data Flow Multiprocessor
      S.R. MAENG and J.W. CHO

SPECIAL FEATURES
17  Call for Papers for Special Issue on Information Systems
33  Letter to Editor and Reply
34-42 Book Reviews
43-44 Information for Authors

Published for Australian Computer Society Incorporated
Registered by Australia Post, Publication No. NBG 1124
TANDEM NONSTOP COMPUTER SYSTEMS PRIMARILY DELIVER ALL THE ELECTRONIC TRANSFER SERVICES IN AUSTRALIA AND NEW ZEALAND.

Tandem Computers control AUTOMATIC TELLER MACHINES, POINT OF SALE DEVICES and ELECTRONIC SWITCHING between our Financial Institutions.

TANDEM Computers have unique capabilities:

- **NonStop** – No single fault will cause the system to go down and the fault can be repaired while the computer system is still operational. Engineers can replace boards and modules plus the system can be upgraded whilst the system remains operational.

- **Growth** – A basic two processor Tandem System rated at 4.4 MIPS can be expanded in a selectable linear growth path to a 35.2 MIPS sixteen processor system without software changes or hardware replacements.

- **Networking** – Up to 255 systems can be interconnected to form a common, NonStop, data communications network, which is viewed as a single resource with the processing capacity of 4,080 central processors.

- **Relational Database** – A unique high performance relational database which permits views or relationships to be established at run time and which supports a truly distributed database across a network of Tandem systems.

- **Transaction Processing System (TPS)** – Which supports the efficient handling of a large volume of terminals in a multiple processor environment integrated with a unique dynamic load balancing to ensure maximum terminal response.

Companies base their business on computers today and cannot afford "not" to have the features available from “State of the Art” technology.

**TANDEM COMPUTERS**

- **MELBOURNE**
  - (03) 267 1577
- **ADELAIDE**
  - (08) 211 8655
- **SYDNEY**
  - (02) 957 5566
- **PERTH**
  - (09) 225 2242
- **BRISBANE**
  - (07) 229 3766
- **WELLINGTON (NZ)**
  - (04) 72 3292

**NCATC/43**
A Query Interface for IMAGE Databases based on Relational Algebra

C. S. Ng† and R. Sacks-Davis†

IMAGE is a widely used database management system developed by Hewlett-Packard for their HP-3000 series of computers. A query language for IMAGE databases based on relational algebra is proposed. The support provided by the IMAGE architecture for such a language is described. The issues involved in providing a full relational interface are discussed and the limitations of this approach are described. The new query language is compared to existing IMAGE query facilities.

Keywords and Phrases: IMAGE, query interface, relational algebra
CR Categories: H.2.3, H.2.4, H.3.3.

1. INTRODUCTION

IMAGE (Hewlett-Packard, 1983) is a database management system marketed by Hewlett-Packard and operates on the HP-3000 series of computers. Although IMAGE was first made available over ten years ago, it has remained quite popular, partly due to the introduction of new and powerful utilities such as application generators and report packages. Many business enterprises are heavily committed to IMAGE and the acquisition of new utilities to reduce programmer effort on IMAGE applications is a cost effective strategy for these enterprises.

In this paper we address the issue of providing a new query interface to IMAGE databases. Because IMAGE is a network database, one approach to this end is to provide a high level navigational query language. However, in Section 3, we show that the restricted network structure supported by IMAGE leads in many cases to the production of IMAGE schemas that do not reflect well the logical structure of the data being represented. As a consequence, the navigational paths through such databases that must be followed to answer a query become, in many cases, unnecessarily complicated.

On the other hand, we show that the IMAGE architecture is well suited to supporting a tabular view of data and we investigate the design of a relational front end for IMAGE databases. A set of rules for translating a relational schema into an equivalent IMAGE schema is presented in section 4. The limitations of such an approach are described.

Two of the most widely used query packages for IMAGE databases are QUERY 3000, a Hewlett-Packard product (Hewlett-Packard, 1981), and QUIZ, a package marketed by a third-party supplier Quasar Systems (1981). (Quasar Systems is now known as Cognos Inc.) In section 5, we describe the characteristics of these packages and propose criteria that a new query interface for IMAGE databases should meet.

A relational query language is proposed in section 6. This language can be used directly on IMAGE files. Compared to QUERY and QUIZ, the relational language has more expressive power. The query optimization techniques employed by the new language are compared to those used by QUERY and QUIZ. The support offered by the IMAGE architecture for the relational language is described.

Of course, we are not proposing that with this interface, we are providing a relational database. IMAGE does not allow a user to dynamically create new tables or indexes, an important component of a relational database implementation. Furthermore, IMAGE provides only limited support for relational integrity rules. Nevertheless, the architecture of IMAGE is such that IMAGE master and detail sets can be viewed as relational tables and the introduction of a query interface based on relational algebra offers a powerful tool for expressing IMAGE queries. We believe that this is a good direction for the design of new IMAGE utilities.

The paper does not assume any detailed knowledge of IMAGE features. The features of IMAGE that are
required for this paper are presented in the following section.

2. BACKGROUND ON IMAGE

IMAGE is a network database. It is similar in structure to TOTAL, a database management system marketed by CINCOM Systems. Both are one-level or simple network databases. Like network databases based on the CODASYL model, IMAGE supports owner-member record relationships. However, in IMAGE, a member record type may not in turn be an owner of other member record types. A description of IMAGE and TOTAL can be found in Kroenke (1983).

There are two types of file structures in IMAGE, a master set and a detail set. In the terminology of the CODASYL model, masters correspond to owner record types and details correspond to member record types.

3.2. Master Sets

Master sets are used to store single valued facts about uniquely identifiable entities. A master set contains one or more attributes, one of which is a key attribute. Rapid retrieval from a master set in the case when a key attribute is specified is supported by hashing. A master set can have only a single key attribute.

A master set is related to zero or more detail sets. Each such detail set has an attribute with the same name as the key attribute of the master set. In the detail set however, this attribute cannot, in general, be used to uniquely identify a record.

There are two types of master sets, manual masters and automatic masters. Automatic masters consist of only the key attribute. They merely serve as indexes to detail sets.

4.3. Detail Sets

Detail sets are used to store multivalued facts about an entity described in a Master set. They support, using a multilist structure, the efficient retrieval of these facts. All the records in a detail set relating to a particular record in a master set are chained in a linked list. In order to obtain the facts about an entity, the following access path is typically followed: the record in the master set is found by hashing using the record key as the identifier and the records with the same key value in the detail set are obtained by following the chain which starts from the master set. Throughout this paper, an index or indexed access to a detail set refers to the use of chains to obtain related records in the detail set. Detail sets can have one or more indexes, being related and physically linked to one or more master sets.

Consider the example shown in Figure 1. SUPPLIER and PARTS are both manual master sets. SUPPLIER has $S#$ as the key. Other attributes in this set represent non-repeating information for each supplier. PARTS has $P#$ as the key attribute while $SP$ has two indexes, $S#$ and $P#$, being related to SUPPLIER and PARTS respectively.

The physical implementation of the Supplier-Parts database is presented in Figure 2. Note that there is a good deal of redundancy in the way information is stored. As well as chaining in a linked list all the records in $SP$ with the same $S#$ values, the actual $S#$ values are stored in each record. In fact, both forward and backward pointers are stored in a chain as illustrated in Figure 3, so the storage overheads can be very large.

Because the pointer information is present, there is no need (aside from efficiency considerations in certain cases) to store in the detail records, the data values for the attributes common to both master and detail sets. However, for the purposes of providing a relational interface, the fact that the data values are stored in the records is an asset. Both manual master and detail sets can be viewed directly as relations in such an interface.

Automatic masters are linked to detail records in the same way as manual masters. They can be used to provide efficient access to detail records through a single attribute. Automatic masters do not contain data values that are not already present in the database and are automatically maintained by IMAGE. In the proposed relational implementation, automatic masters will serve as indexes (rather than relations).
5. LIMITATIONS OF IMAGE AS A NAVIGATIONAL DATABASE

There have been a number of high level languages proposed for network databases. These high level languages require the user to specify the attributes required for printing, the path through the database that must be navigated to obtain the appropriate records, and the restrictions or qualifications these records must satisfy before they are selected. Like relational languages, these high level navigational languages return a set of records as a result of a query: they are not "record-at-a-time". Examples of such languages are NUL (Deheneffe and Hennebert, 1976) and QRS, a language designed for the extended network database MDBS (Bonzek, Holsapple and Whinston, 1984).

The reason we did not follow this approach is that IMAGE, being a simple network implementation, does not, in many cases offer a user a view of the database that directly reflects the logical structure of the data being modelled. As a consequence, the specification of a navigational path through an IMAGE database is often unnecessarily complicated. In this section, we describe some of the limitations of using a one level network structure to represent logical relationships such as hierarchies, cycles and loops. In addition, we show that an IMAGE database design is also affected by performance requirements as well as by the logical relationships between entities. As a consequence of these limitations, we chose to support a relational view of the data to simplify query formulation.

To illustrate some of the limitations of the one level owner-member structure, consider the following examples:

1. Hierarchies

Consider the representation of a three level hierarchy (Figure 4) in which a faculty has many departments and each department in turn has many staff. The one level IMAGE representation does not allow a user to directly view the nature of this relationship. In IMAGE, DEPARTMENT and STAFF record types are represented as detail sets. They each have two indexes, which require the introduction of two automatic masters (Figure 5).

2. Cycles and Loops

Cycles and loops may not be represented directly in IMAGE. An example of a cycle is given in Figure 6. This example illustrates the representation of a genealogical tree. A relationship in which a manager is also a member of the staff can be represented using a loop (Figure 7).

3. Lack of Data Independence

A database design in IMAGE reflects both the physical design requirements as well as the logical relationship between entities. Its lack of data independence can be illustrated by this example:

Initially, SUPPLIER has one key S# and is represented as a manual master (Figure 8). Suppose that queries which specify Sname are shown...
to be very common and an index on Sname is required. A reorganisation of the database is forced. SUPPLIER becomes a detail with two automatic masters for its two indexes, S# and Sname (Figure 9).

6. IMAGE AS A TOOL FOR RELATIONAL MODELLING

As described in Section 1, in IMAGE databases, the data values of attributes common to both manual master and detail sets are stored within the IMAGE records, in addition to the pointer structures. Thus, it is possible to view manual master and detail sets as relational tables and only use the pointer information to achieve efficient access. Extending this idea, automatic masters can serve as indexes in the relational model.

In this section, we show how a relational schema can be translated into a corresponding IMAGE schema. For each relation in the relational schema, a manual master or detail set will be introduced in the IMAGE schema. The idea is that in order to provide a high level query interface, the user will be able to view the data as a relational database.

The rules used for schema translation are given below:

1. If a relation has one key, consisting of a single attribute, it becomes a master set.
2. If a relation has more than one key (here we include all candidate keys and foreign keys) or a key consisting of more than one attribute, it becomes a detail set. A key in a detail set is indexed by a manual master if it exists as a key to that manual master, otherwise, it is indexed by an automatic master.

As an example of a schema translation, consider the Stock-Order database (McDonell, 1983) below. Its relational schema is as follows:

CUSTOMER (Custno, Cname, City)
PRODUCT (Prodno, Description)
WHOUSE (Wno, City)
STOCK (Wno, Prodno, Qty)
ORDER (Ordno, Custno, Date)
ORDLINE (Ordno, Prodno, Qty)

Note that relations STOCK and ORDLINE both have keys consisting of more than one attribute. Relation CUSTOMER has a single key, Custno, but would have two keys if attribute Cname was designated a candidate key. Relation ORDER has more than one key, attribute Custno being a foreign key. A foreign key is not a defined key in the relation concerned but is a key to another relation. On applying the transformation rules, the equivalent IMAGE schema obtained is as shown in Figure 10.

Note that if Cname had been designated a candidate key in the relation CUSTOMER, the IMAGE schema would have been different. Note also that the translation rules would need to be modified in order to support indexes on arbitrary non-key attributes. As it stands, only indexes on key attributes are supported.

There are two major limitations to the described relational interface:

1. Relational database implementations generally allow tables and indexes to be dynamically created or dropped. This is not possible in our case due to the lack of data independence in the physical representation of the underlying IMAGE database. Changes to the structure of an IMAGE database require the database to be recreated.

2. Multiple attribute indexes (single indexes on concatenated attributes) are not catered for. Referring to the STOCK relation in the Stock-Order database, Wno and Prodno are both separate indexes. An index made up of Prodno concatenated to Wno is not provided.

In spite of these limitations, the relational interface does simplify the conceptual view of data and its relationships. An end-user need not know the difference between master and detail sets. Limitations in the
conceptual representation of certain logical relationships (as discussed in Section 3) are reduced.

To properly support the relational interface, it is desirable to support entity and referential integrity rules as they apply to relational databases. Entity integrity requires that key attributes may not contain null values and duplicate keys are not allowed. Referential integrity requires that foreign keys may not be null and the value of a foreign key must exist in the referenced relation where it is a primary key.

For relational schemas implemented using the translation rules described above, there is some automatic support for these integrity rules provided by the IMAGE software. We discuss entity integrity first and then referential integrity.

For manual master sets, IMAGE will automatically disallow duplicate key values or null values. For detail sets, however, duplicate values are possible and to support entity integrity in these cases, explicit checks must be written into the relational interface. However, because all key attributes are indexed by either automatic or manual masters, these checks can be performed efficiently.

Referential integrity is also implicitly supported by IMAGE when the referenced relation is an IMAGE manual master set. This is the case in the Stock-Order database where ORDER is a detail set and CUSTOMER is a manual master set. In IMAGE, by definition, for a detail record to exist, the corresponding master record must exist. If the referenced relation is a detail set, however, the check must be explicitly performed on update.

In summary then, IMAGE provides a useful tool for supporting a relational interface. It readily provides a tabular view of the data. Like all network databases in which the existence of a member record in a set occurrence requires the existence of the corresponding owner record, there is a degree of automatic support for integrity rules such as referential integrity.

We chose not to implement the relational interface in the above manner. The main reason for this is that many of the existing IMAGE utilities for inserting or updating data would have become unusable if the integrity of data was to have been supported automatically. Instead, we work with an existing IMAGE schema and provide a relational query language which operates directly on the IMAGE master and detail sets of that schema. These sets are viewed as tables of data. Note that, unlike CODASYL schemas, it is not possible, in general, to deduce a relational schema (containing the keys to each relation) from an IMAGE schema. However a relational query language does not require a knowledge of the keys of a relation. It can be used simply on tables of data.

7. IMAGE RETRIEVAL PACKAGES

A number of fourth generation packages with powerful query facilities have been built for IMAGE databases. A recent survey of these products can be found in Farquharson, Overteid, Marchand, Lloyd and Schneider (1983). In this paper we describe two widely used retrieval packages for IMAGE databases: QUERY (Hewlett-Packard, 1981) and QUIZ (Quasar Systems, 1981). QUERY was first released by Hewlett-Packard in 1976. While allowing for update operations, the retrieval facilities in the first release were rather limited. For example, queries were limited to paths that contained a single detail set. Subsequent releases included a JOIN (equijoin) command which could be used on any number of IMAGE sets. With this addition, the retrieval properties of the language were greatly enhanced. A serious disadvantage of QUERY is that, with one or two exceptions, queries can only be based on IMAGE files. Thus it is not possible, in general, to answer complex queries, by building up intermediate files in answer to subqueries, and then apply QUERY commands to the intermediate files.

QUIZ is possibly the most popular of the retrieval packages for IMAGE databases marketed by third party software companies. QUIZ is basically a report generator but it is very useful for ad hoc queries.

For retrieval, QUIZ supports the use of three commands: ACCESS, SELECT and CHOOSE, as well as a LINK TO clause. These are described below:

1. **ACCESS**

   The ACCESS statement specifies the files from which information is to be retrieved. Information may be retrieved from one or more files, linked together in the same way as a relational join (equijoin).

2. **SELECT**

   The SELECT applies a selection criterion on the records retrieved from the files specified in the ACCESS statement.

3. **CHOOSE**

   CHOOSE too applies a selection criterion but only to a specified key or indexed attribute of the first file in the ACCESS statement. Using a CHOOSE command rather than a SELECT, it is possible to avoid a serial scan of the first file. Hashed (on the key attribute of a master set) or chained (on an index of a detail set) access is used instead, based on the supplied key or indexed attribute values. A user must be aware of the difference between a SELECT and a CHOOSE, and use a CHOOSE rather than a SELECT whenever possible. The most efficient access method should really be transparent to the user. QUERY, on the other hand, automatically uses an index when it is available but has different commands for selection depending on whether the argument is an IMAGE file (in which case a FIND command is used) or a JOIN of one or more IMAGE files (in which case a MULTI-FIND command is used).

4. **LINK TO**

   A LINK TO clause serves the purpose of a join command and is used in the ACCESS statement.
when information is to be retrieved from more than one file. In

ACCESS SUPPLIER LINK TO SP,

SUPPLIER is known as the primary file and SP the secondary. There are two limitations in linking files in this manner:

a. A key or an indexed attribute in secondary files must be present as an attribute (not necessarily a key or index) in a preceding file. This allows QUIZ to use hashed or chained access to the secondary files when performing the join between files in the ACCESS statement. A link or join over non-key or non-indexed attributes is not allowed.

b. QUIZ does not optimize the order of which files appear in the ACCESS statement. Generally, placing a smaller file as a primary file will either have no effect or will yield better performance in query processing. However, QUIZ processes a query as it appears. Its efficiency is very much dependent on the skill of the user.

QUERY, on the other hand, imposes no restrictions on the files that can participate in the join. A join over non-indexed attributes is supported.

Both QUERY and QUIZ appear to be tuple or record orientated query processors. Queries which require the generation of intermediate files such as relational algebraic expressions involving MINUS or DIVIDE operators are either very difficult or impossible to express using either QUERY or QUIZ.

As well as the above commands, both QUERY and QUIZ contain many useful options for formatting reports. We do not address these issues here since these options are largely unaffected by the underlying access paths and they can be included in a package based on a relational query language in a similar way, to their incorporation into QUERY or QUIZ. Also, we do not discuss, in this paper, the support of arithmetic computation and statistical functions within the query language.

Some of the goals of the relational query interface are listed below:

1. Introduction of set based operators: PRODUCT, INTERSECTION, UNION, MINUS and DIVIDE.

   These are operators usually supported by query languages based on relational algebra. However a smaller subset can be supported without reducing the expressive power of the language. For example each of the operators JOIN, INTERSECTION and DIVIDE can be expressed in terms of the primitive operators SELECT, PROJECT, PRODUCT, MINUS and UNION (Date, 1986, p.271). Another requirement, not supported by QUERY, is that it should be possible to build intermediate files using these operators and then be able apply the operators to these intermediate files.

   Without such operators, there are queries that cannot be performed. Consider the following examples:
   a. List suppliers who supply all parts supplied by supplier S1.
      \[ X1 = \text{PROJECT SP ( S\# = 'S1') } \]
      \[ X2 = \text{SELECT SP ( S\# = 'S1') } \]
      \[ X3 = \text{PROJECT X2 ( P\# ) } \]
      \[ \text{RESULT} = X1 \text{DIVIDE BY} \ x3 \]
   b. List suppliers who do not supply part P1.
      \[ X1 = \text{PROJECT SUPPLIER ( S\# ) } \]
      \[ X2 = \text{SELECT SP ( P\# = 'P1') } \]
      \[ X3 = \text{PROJECT X2 ( S\# ) } \]
      \[ \text{RESULT} = X1 \text{MINUS X3} \]

   Neither can be directly formulated in QUERY or QUIZ.

2. Implementation of an unrestricted join.

   A join should be permitted over two relations irrespective of whether the joining attribute is a key to one or more of the relations. Without this property, there will be some queries that simply cannot be performed. Consider the following example (from the Supplier-Parts database described in Section 2):
   List suppliers located in the same city as supplier S1.
   This query can be formulated in two steps as:
   \[ X = \text{SELECT SUPPLIER ( S\# = 'SI') } \]
   \[ \text{RESULT} = \text{JOIN X SUPPLIER ( City )} \]
   This query may not be directly formulated in QUIZ.

3. Introduction of additional query optimization techniques.

   The requirement that a user should know whether to use a QUIZ SELECT or CHOOSE command and the requirement that a user order the sets participating in a JOIN in the optimal way obviously leads to inefficient query evaluation in many cases. IMAGE certainly provides information about which indexes exist so the first requirement is easily avoided and IMAGE also provides information that can be used to choose efficient evaluation paths for the JOIN operation.

   QUERY does not require the user to determine the optimal evaluation path. Very little information is presented in the QUERY manual about the way QUERY chooses access paths to answer queries, but the testing we have done indicates that efficient evaluation paths are chosen in most cases. There were one or two exceptions however (an example is given in the next section).

8. A RELATIONAL QUERY IMPLEMENTATION

   A relational query interface based on specifications described in the previous section is currently being implemented at RMIT. The purpose of this section is to briefly describe the implementation and to provide a description of the facilities available in IMAGE to support such an interface.

   The retrieval package implemented is based on rela-
Note that in order to implement other relational query languages such as SQL, a query optimizer must be written, one of the components of which will reduce the SQL query to a sequence of relational algebraic primitives (or something similar). Thus our implementation of relational algebraic primitives could serve as a basis for the implementation of a language like SQL.

The interface has been written so that the operands are IMAGE master and detail sets. The advantage of this approach is that the language can be used directly on existing IMAGE databases.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT SP ( S# = 'SI' )</td>
<td>JOIN X DEPARTMENT ( Dep# )</td>
<td>SELECT X ( Name = 'Smith' )</td>
</tr>
<tr>
<td>PROJECT STAFF ( Name, Age, Salary )</td>
<td>PROJECT Y ( Name )</td>
<td>JOIN STAFF DEPARTMENT ( Dep# )</td>
</tr>
</tbody>
</table>

This sequence of operations can be transformed to

\[ X = \text{SELECT SP ( S# = 'SI' )} \]
\[ Y = \text{JOIN X DEPARTMENT ( Dep# )} \]
\[ Z = \text{SELECT X ( Name = 'Smith' )} \]

The latter formulation is generally more efficient. Our implementation uses a fixed set of transformation rules, one of the objectives of which is to decrease the net area of relations that are passed to later operations. This is achieved by performing selections as early as possible.

Rather than using a fixed set of transformation rules, another approach involves generating all possible access plans, estimating the cost of each of these plans, and then choosing the cheapest plan. An example of this approach is provided by Yao (1979). IMAGE provides the following parameters which are useful for the evaluation of the size of intermediate results. The number of records in each set is known. Also the length of each chain linking together records in a detail set with the same attribute value is also stored. As a consequence, the size of results formed from operations involving attributes that are keys in master sets can be estimated with some accuracy. On the other hand, for operations involving non-indexed attributes, estimation is much more difficult (see also Montgomery, D'Souza and Lee, 1983).

The other type of optimization that is heavily influenced by the underlying access paths is operator optimization. The problem here is to determine the most efficient way to carry out a primitive operation such as a SELECT or JOIN. Consider first the SELECT operator.

Hashed or indexed access as opposed to serial access is used to retrieve the records for evaluation when an index is supplied and the SELECT expression is of the form:

\[ X = \text{SELECT SP ( S# = 'SI' )} \]

An equality operator must be associated with the key or index because IMAGE only supports hashing as an access path to master sets. If B-trees indexing was also available, for example, other operators such as greater than (>) could also be supported efficiently. Similarly, for reports that require records to be sorted on the values of particular attributes, internal sort procedures must be used.

If the selection criterion is of the form:

\[ X = \text{SELECT SP ( S# = 'SI' AND P# = 'PI' )} \]

where S# and P# are both indexed in the underlying SP IMAGE detail set, one of two access paths may be traversed to answer this query. One path follows a chain starting from a SUPPLIER master record, the other chain starts from a PARTS master record. Since the length of each chain is stored in the associated master records, the query processor can always choose the shortest access path (the chain containing the least number of records) to answer a query. (This optimization technique was not used in our version of QUERY.)
Relational Query Interface

Note that with the multiset structure supported by IMAGE, queries such as the one above can be quite expensive to evaluate even when the shortest path is followed. The records retrieved by following this path could well be spread over different parts of the disk, and only some of these records will satisfy the query. Contrast this with the more common inverted file approach, where the lists of pointers corresponding to each of the given attribute values are first intersected, and only then are data records retrieved.

The cost of many relational algebraic operations can be reduced if advantage is taken of sort orders in the participating relations (Smith and Chang, 1975). As an example, consider the implementation of the merge-sort algorithm for computing the join of two relations. Obviously, if the participating relations are sorted over the join attribute, the efficiency of this relation is greatly facilitated. Unfortunately records in IMAGE master and detail sets cannot be accessed in a sorted order so it is not possible to take full advantage of these optimization techniques.

In order to implement the JOIN (equijoin) operator the following approach was adopted. Depending on the nature of the common join attribute, one of three methods for processing the join is chosen:

1. If the common join attribute is indexed in both of the relations involved in the join, the smaller relation is serially scanned. For each record obtained, the value of the key or index to the other relation is extracted for hashed or indexed access to this larger relation. If a record is found, it is concatenated to the record of the first relation as a resultant record.

2. If the common join attribute is indexed in only one relation, the relation for which it is not indexed is serially scanned. Hashed or indexed access is used on the other relation.

3. If the common join attribute is not indexed in either relation, both relations are sorted over the common join attribute and the merge-join technique is applied to the two sorted files.

In summary then the proposed query language overcomes the limitations of QUERY or QUIZ that were described in Section 3 and formulate queries in terms of the relational schema. The provision of a relational interface is limited by a number of factors however. IMAGE does not allow a user to dynamically create new tables or indexes and there is only limited support for relational integrity rules.

Acknowledgements
We would like to thank Jurgen Hinrichs, Norbert Riedl, Jo Shanks and Andrew Yeoh as well as the referees for their helpful comments.

References

Biographical Note.
Choon Seong Ng is a masters student in the Department of Computing at RMIT, where he obtained a B.App.Sci. (with distinction) in Computer Science in 1983. His research interests include database management systems and fourth generation languages.

Ron Sacks-Davis is currently a principal lecturer in the Department of Computing at RMIT. He obtained his Ph.D. in Computer Science from the University of Melbourne in 1977. His research interests include database management systems, information retrieval and data structures.
Quokka: A Translator Generator Using Denotational Semantics

T.N. Vickers†

Translator generation is the automatic production of a translator from a formal specification of a language. Quokka is a translator generator that accepts a denotational definition of a programming language's semantics. Using this definition Quokka generates an independent program capable of translating programs into lambda expressions. The lambda expressions can then be interpreted by an implementation of the lambda calculus. Alternatively the lambda expressions may be translated into code suitable for execution on a conventional run-time stack. Lexical and syntactic properties of a specified language are defined in a manner compatible with existing lexical analyser and parser generators. Quokka is intended as a research and learning tool for experimentation in the fields of denotational semantics and programming language theory.

Keywords and Phrases: translator generation, denotational semantics, formal specification.
CR Categories: D.3.1, D.3.4, F.3.2.

1. Introduction
Quokka is a program that automatically generates a translator from a denotational description of a language. A translator accepts a program in one language and produces a program in another. Translators generated by Quokka have the same target language, the lambda calculus, so only a specification of the source language is required.

Quokka is a tool for both students and researchers of semantics and programming language design and theory. For the researcher, Quokka can be easily used to explore ideas on language semantics and design. For the student, Quokka demonstrates the method and desirability of semantic specification, while giving first-hand experience of the implementation of text book theorems and concepts.

Denotational semantics (Stoy, 1977; Gordon, 1979) has been used in other translator generator systems (Mosses, 1975; Paulson, 1982). Other systems (Cordy, Holt and Wortman, 1979; Henson and Turner, 1982; Sethi, 1983) are based on various means of semantic specification. A general introduction to translator generation can be found in Aho (1980) and a collection of related articles appears in Jones (1980).

Because Quokka was designed partially as an aid to the teaching of denotational semantics, it processes denotational semantics in preference to other semantic forms. The other major design consideration was to include features of hand-crafted translators into automatically generated translators, especially those lacking from existing translator generation systems. To this end Quokka generates a self-contained program and has facilities for error recovery in the generated translator and in processing of the specification.

This paper provides an overview of Quokka. A complete description of Quokka will appear in the author's forthcoming doctoral thesis. The tutorial on Quokka (Vickers, 1985) is an informal guide to the use of Quokka.

The organisation of this paper is in three parts. The first is a short tutorial introduction to Quokka, covering aspects of language specification. Emphasis is placed on an explanation of semantic description.

The second part describes the strategies Quokka employs to generate a translator from a language specification, and how such a translator operates. Also included is an explanation of how a program may be executed after translation. Strategies of other translator generators using denotational semantics are presented.

Some comments on translator generation and areas requiring further research form the final part of the paper.

1.1 Example Language
The example language contains integer and boolean expressions and simple conditional and iterative constructs, as well as assignment. A program is a sequence of statements terminated by an arithmetic expression. The value of the final expression constitutes the result of the program.
Quokka: A Translator Generator

Specification of the language requires specification of its syntax and semantics.

2. Syntax
The syntax of the language to be described may be expressed in EBNF (Wirth, 1977) as shown in Figure 1.

```
prog = statlist aexpr.
statlist = stat { stat }.
stat = "skip" | ident "=" aexpr | "if" expr "then" statlist "else" statlist "fi" | "while" expr "do" statlist "od".
expr = aexpr (= "<" | "<=") aexpr.
aexpr = aterm { ("+" | "-"") aterm }.
aterm = ident | integer | ".(" aexpr ").".
ident = letter.
integer = digit { digit }.
letter = "a" | "b" ... "z" | "A" | "B" ... "Z".
digit = "0" | "1" ... "9".
```

Figure 1. Example Language

Quokka requires specification of a grammar in a manner compatible with Aardvark (White, 1981) and Llama (Dunn and Murphy, 1976). Aardvark processes the lexical definitions and generates a lexical analyser in Pascal code. Lexical tokens are defined using regular expressions. Llama performs an LALR(l) analysis (Aho and Ullman, 1977) on the grammar to generate a Pascal encoded parser. The grammar is given in a restricted form of EBNF. Figure 2 shows a possible Aardvark/Llama specification of the example language. Keywords in Aardvark and Llama begin with '&'. This convention has been adopted by Quokka.

Aardvark allows attributes to be returned with tokens (e.g. INT, IDENT). Llama makes provision in the generated parser for embedded code to be executed on reduction of a particular production. The syntactic definition of the language in the form suitable for consumption by Aardvark and Llama shows the framework in which the semantic specification is to be given. Further information on Aardvark and Llama can be found in the Aardvark (White and Vickers, 1983) and Llama (Hayes and Robinson, 1983) tutorials.

3. Semantics
Quokka requires the semantics of a language to be specified using denotational semantics. Gordon (1979) gives an introduction to denotational semantics, while the mathematical basis for this branch of semantic theory can be found in Stoy (1977).

Denotational semantics defines a language's meaning by a collection of semantic functions. These functions describe, or model, the execution of a program in the language. An important feature of the model is the program state. One can consider execution of a language construct as being a transition from one state to another. Such transitions are defined by the semantic functions.

For languages as simple as the example, the main component of the program state is the store, which holds the bindings of variables to values. Other components are required, such as input and output, if the language contains commands for their manipulation. Domains in a semantics form the type of the semantic functions; they contain the objects on which the functions operate. The following methods may be used to construct domains.

`\{*, *\}`
These enclose an enumeration of the components of a domain. The components are literals.

identifier
The identifier is the name of the domain, which may be defined elsewhere or be a basic domain.

D1 + D2
Forms the disjoint sum of the domains D1 and D2. Elements of the result domain are 'flagged' to indicate to which component domain they belong.
D1 × D2
Forms the Cartesian product of the domains D1 and D2. Elements of the product will consist of an element of each component domain.

D*
D* forms the domain of sequences of elements of D. An element of D* is a sequence of zero or more elements of D.

D1 − D2
The domain D1 − D2 is the domain of functions from D1 to D2. This symbol is right associative.

brackets
The bracket pairs ‘(’, ‘)’ and ‘{’, ‘}’ may be used for grouping.

A semantics capable of defining the example language may have the following domain definitions.

&domain
store = char → integer.
output = integer*.
The store is defined to be a mapping from elements of ‘char’ to elements of ‘integer’. An element of the domain ‘output’ is declared to be a sequence of integers.

The semantic functions must be declared to belong to a particular domain, and be defined for each language construct on which they act. The semantic functions for the example language are declared in the semantic function (&semfunc) section.

&semfunc
P : prog → output.
A : answer → store → output.
C : [stat + statlist] → store → store.
E : [aexpr + aterm] → store → integer.
B : expr → store → boolean.

A semantic function is a mapping from a syntactic object to the value it represents in the mathematical model of the language. A non-terminal symbol implicitly defines a syntactic domain, and is also taken to be an element of that domain.

Tokens will often have an associated value that is to be accessed during parsing. The tokens INT and IDENT will have attributes of integer and char. The functions that access token attributes must be declared in the interface (&interface) section.

&interface
N : INT → integer.
S : IDENT → char.

The notation used by denotational semantics, and used in Quokka specifications with only minor alterations, is the lambda notation (Church, 1941; Stoy, 1977). Before giving a description of lambda notation the semantic definition for the first syntactic construct of the example language is shown. It may be useful to refer to it while reading the description of the notation.

The semantics of particular constructs are given at the end of that construct in the grammar, and are enclosed by the parentheses ‘&(' and ‘&')’. For example, the semantics of the first construct is written as follows.

prog = statlist answer
& ( P[[]] = A[S2] (C[S1] null[store]) ) &.
The left hand side of the semantic definition, ‘P[[]]’, is short hand for ‘P[statlist answer]’. Notations of the form ‘$i’ refer to the i'th term in the right hand side of the production. The square brackets ‘[’ and ‘]’ are traditionally used to enclose textual objects. The expression ‘null[store]’ establishes an element of the domain store without any bindings. Writing the definition in full,

P[statlist answer] =
A[answer] (C[statlist] null[store])
looks more like text book denotational semantics, but this form is not understood by Quokka. Apart from these minor differences, the right hand side of the semantic definition is written in lambda notation.

The syntax of lambda expressions is given by the following.

λexpr = identifier
| "λ" identifier "." λexpr
| λexpr λexpr
| (" " λexpr ")."

Abstractions (expressions of the second form) extend to the right as far as possible. An abstraction, λi.e, has binding term i and body e. Applications (the third form) will be taken to be left associative.

The above syntax is that of the pure lambda calculus. Often it is extended to make expression of ideas easier. The form of lambda expressions used in Quokka have the syntax shown in figure 3.

λexpr = identifier
| "λ" identifier "." domain "." λexpr
| "μ" identifier "." domain "." λexpr
| λexpr λexpr
| unaryop λexpr
| λexpr binaryop λexpr
| cond 
| clist 
| basicvalue
| (" " λexpr ")." λexpr ")
| (" λexpr "/" λexpr ")"
| clist = cexpr
| cexpr = λexpr "→" λexpr.

Figure 3. Extended λ Expressions

The extended syntax has added simple boolean and integer arithmetic, and a conditional sequence. Basic (pre-defined) value domains are boolean and integer. The domain of the binding term is required to be declared (for type checking). The mu (‘μ’) expression is named an expression, allowing reference to its body.
(the final λexpr) by its name (the identifier). The name is known only in the body. Mu expressions are used when recursive references are needed. Notations of the form $s[v/i]$ are used to update associations in elements of mappings (such as store in the example). Here, v would become bound to i regardless of any previous bindings in s.

When writing any semantic definition, such as,

\[
P[\text{statlist answer}] = \\
A[\text{answer}] (C[\text{statlist}] \text{ null[store]})
\]

it is important that type compatibility is maintained. For example, the function C expects an object in the domain stat + statlist and another in the domain store. The first requirement is met by statlist ("[[S1]]") and the store in the example. The function null[store] returns an element of the domain store. Given these two arguments, C returns an element of store. The function A takes an answer ("[[S2]]") and produces an element of the domain output. The final check is that both sides of the definition are in the same domain.

The next construct of interest is the assignment.

\[
\text{stat} = \text{IDENT ASSIGN aexpr} \\
\quad & (C[[ ]] = \lambda s:store. s[E[[ ]] s / S[[ ]]])
\]

This statement is defined to produce an update of the existing store given the variable and the meaning of the expression.

The conditional statement might be given the following semantics.

\[
\text{stat} = \text{IF expr THEN statlist ELSE statlist FI} \\
\quad & (C[[ ]] = \lambda s:store. \text{cond}(B[p2] s - C[[4]] s, C[[6]] s))
\]

If the meaning of the guard is true (in the context of the current store), the construct denotes the first statement sequence, otherwise it denotes the second.

The semantics of the iterative statement may be defined by taking advantage of the self-referential property of the mu expression.

\[
\text{stat} = \text{WHILE expr DO statlist OD} \\
\quad & (C[[ ]] = \mu W:store \rightarrow \text{store. } \lambda s:store. \text{cond}(B[p2] s - W(C[[4]] s), s))
\]

If the guard is true in the current store, the store resulting from execution of the statement sequence is passed back to the expression for the while statement, where it becomes the current store. This process continues until the guard is found false in the current store, at which time the current store becomes the result of the expression.

Input and output are considered as domains of sequences. Normally one wishes a program to accept input and produce output. Quokka accommodates this wish by providing the means to process input and output as part of the usual transformation from one program state to the next. For example, the result of the function,

\[
\lambda o:output. o 1 <3>
\]

is the sequence, o, concatenated with the sequence containing only the integer 3. In addition, because the sequence was an element of output, the value 3 will be appended to the output stream. Similar processing of
input streams is maintained using the sequence operators, head and tail. Files are opened by the function file. This is considered to establish a sequence in the specified domain. These additional facilities are only available if input and output are defined (in the &domain section) to be sequences. Hence output is a sequence of integers, even though programs in the example language only ever produce a single integer. The definition of the semantic functions for the complete language are shown in figure 4.

The specification file for the example language would consist of the &domain, &semfunc, and &interface sections followed by the &token section of figure 2 and the &gram section of figure 4.

Other functions may be required to augment the semantic functions. These auxiliary functions are convenient for grouping often used expressions. They are declared in the &auxfunc section and are immediately followed by their definition. Their definition has the same form as the right hand side of a semantic function definition.

A hand written translator will almost certainly distinguish between compile-time and run-time actions. The compile-time actions may include constant expression evaluation, and type checking. Run-time actions include final checking and the mechanisms of procedure calls and variable access. Distinction between these classes of functions can be made in Quokka by declaring the functions to be static (compile-time) or dynamic (run-time). The default is dynamic. Static functions will be evaluated during the parse of a program, while a dynamic function will be evaluated only during execution of the program.

Occasionally one may feel that an implementation of a function or data structure is less efficient in the lambda calculus than a more usual programming language. For this reason, Quokka allows some auxiliary functions to be defined in a (well-defined) sequential language. They are defined in the &exfunc (external function) section, and may be referenced from the semantic function definitions. A strict definition of the interface of these functions to the rest of the semantics must be given. Care should be taken to maintain consistency of the semantic model, if external functions are in use, by updating the model to account for actions of the function. It must be stressed that these functions are not required for a workable semantic specification. They are an optimization only.

A translator cannot expect all input programs to be correct. A program might easily have 'missing semicolons', or have an integer variable in a context where a boolean value is required.

Simple syntax errors can be caught by the parser. For example, Llama allows error recovery at the statement level by the additional production, 

stat = lerror.

Of course, appropriate semantics must be specified for this construct.

More complex errors require more difficult measures. In the example the result of a look-up (s S[[IDENT]]) of an unbound identifier is undefined. Possible alternatives are to return a nominal value (perhaps with an error message), or to return an error condition (perhaps a literal 'unbound').

Whichever choice is made, a problem arises of how that choice is represented in the semantics. In the type of semantics shown (called direct semantics), an error condition would be propagated through the remaining semantics, with checks on the validity of arguments at each step. Neither this option or that of introducing nominal values is desirable.

The best solution involves the immediate termination of the program with an appropriate error message. To achieve this we can introduce into the model of program execution the notion of the 'rest of the program'. In most cases (those not producing errors), the rest of the program is just as it is for the direct semantics: simply what happens next. Where an error occurs an error function can be chosen in place of the normal 'rest of the program'. This form of semantics, known as continuation semantics, is also very useful for expressing the meaning of jumps (e.g. goto statements). Both direct and continuation semantics are forms of denotational semantics.

4. Strategies
This section is a description of how the language specification is transformed into a translator for that language, and how the generated translator operates.

4.1 Generation
Given the complete specification file, Quokka generates Pascal code to implement the language described by the semantics. The remaining lexical and syntactic sections are used by other programs (White, 1981; Dunn and Murphy, 1976) to generate appropriate Pascal routines to perform the functions of a lexical analyser and parser.

Implementation of the semantics is achieved in three sections, corresponding to the grammar (&gram), external function (&exfunc) and interface (&interface) sections of the specification. The remaining semantic sections are used by Quokka for type checking while processing the definition file.

Grammar
Each semantic definition of the grammar section is transformed into Pascal code which, when executed, reconstructs the lambda expression involved. The code is embedded with the corresponding production in the grammar given to Llama. The Llama-generated parser executes that code when the construct is recognised.

As a simple example the following construction code may be generated from the expression '2 + x'.

\[
t0 := \text{intnode}(2); \\
t1 := \text{derefnode}('x'); \\
t0 := \text{binarnode}(\text{plus}, t0, t1)
\]

where the above functions construct the appropriate
Quokka: A Translator Generator

lambda expression components.

External
The second part of the implementation is the generation of code to implement any external functions defined in the specification. These routines must integrate the function definition and the data structure of the lambda expressions. External functions, if they exist, are accessed during execution of a program, and so are implemented by Pascal routines.

Interface
The final section involves generation of components of the lambda expression data structure (those depending on arbitrary user-defined domains), as well as code to create expressions belonging to those domains. The latter code is used for results from external functions and for implementation of the interface functions. An interface function, for example, may take an integer value and produce an integer node of a lambda expression. Interface functions are employed during the transformation of a program to its equivalent lambda expression.

The generated file is a self-contained Pascal program consisting of a lexical analyser, LALR(1) parser, lambda expression constructor, necessary declarations, implementation of external functions, and the lambda expression interpreter. It is felt that generation of the translator as an independent program is important to the general goals of translator generation. If translator generation is to escape the realms of experimental and teaching uses, its generated translators must exhibit properties similar to hand-crafted translators. An independent program is capable of executing on machines on which the generator and language specification do not exist.

4.2 Parsing
This section describes the actions that take place when, the generated translator parses a given example program.

Llama-generated parsers have a value stack associated with the parse stack, which enables symbols to have attributes. The constructed lambda expression is taken as the attribute of the left hand side symbol of the production. When a non-terminal symbol appears on the right hand side of a production it will already have a lambda expression as its associated value. This attribute forms the ‘meaning’ of that non-terminal (e.g. E[[S]])) in the expression currently under construction. In this manner, a single lambda expression is constructed which represents the program’s meaning.

As an example, consider the following program in the previously specified language.

\[
\begin{align*}
  x & := 0 \\
  x + 1
\end{align*}
\]

Recognition steps (figure 5) in the parse of the program together with associated lambda expression demonstrate the way in which a single expression is constructed which represents the program. Reference to the complete collection of semantic definitions (figure 3) may aid understanding of the lambda expressions constructed.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Lambda Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT</td>
<td>‘x’</td>
</tr>
<tr>
<td>INT</td>
<td>0</td>
</tr>
<tr>
<td>aterm</td>
<td>λs.0</td>
</tr>
<tr>
<td>aexpr</td>
<td>λs.0</td>
</tr>
<tr>
<td>assignstat</td>
<td>λs.(λs.0 s/’x’)</td>
</tr>
<tr>
<td>stat</td>
<td>λs.(λs.0 s/’x’)</td>
</tr>
<tr>
<td>statlist</td>
<td>λs.(λs.0 s/’x’)</td>
</tr>
<tr>
<td>IDENT</td>
<td>‘x’</td>
</tr>
<tr>
<td>aterm</td>
<td>λs.’x’</td>
</tr>
<tr>
<td>aexpr</td>
<td>λs.’x’</td>
</tr>
<tr>
<td>INT</td>
<td>1</td>
</tr>
<tr>
<td>aterm</td>
<td>λs.1</td>
</tr>
<tr>
<td>aexpr</td>
<td>λs.(λs.s ’x’) s + ((λs.1) s)</td>
</tr>
<tr>
<td>answer</td>
<td>λs.file(output) 1</td>
</tr>
<tr>
<td>prog</td>
<td>(λs.file(output) 1</td>
</tr>
<tr>
<td></td>
<td>&lt;(λs.(λs.s ’x’) s + ((λs.1) s) s &gt;)</td>
</tr>
<tr>
<td></td>
<td>((λs.s(λs.0 s/’x’)) null(store))</td>
</tr>
</tbody>
</table>

Figure 5. Lambda Expression Construction

Static functions are evaluated during parsing. Once a compile-time function is encountered, the interpreter is called to reduce the expression. The resultant expression takes the place of the original in all future expression construction.

4.3 Evaluation
At this point, Quokka could progress in two directions. The first is the straightforward interpretation of the final lambda expression. The second involves generation of applicative code from the lambda expression, and subsequent execution of that code.

Interpretation
The final lambda expression is handed over to the interpreter on successful completion of the parse for evaluation of dynamic functions. The possibility of preserving the lambda expression for future evaluation is not practical since the interpreter is required as part of the generated translator to evaluate static functions.

The lambda expression interpreter is based on Landin’s SECD machine (Landin, 1964), an implementation of the lambda calculus (Church, 1941). Figure 6 shows an evaluation trace of the expression corresponding to the given example program. The final expression, <1>, is the irreducible meaning of the program. As a side effect of catenation the value 1 has been appended to the output stream.

Code Generation
Before applicative code can be generated from the lambda expression, that expression must undergo a transformation. The lambda calculus treats functions as first-class objects – an unfamiliar concept to most
conventional languages. The transformation removes function-valued functions while maintaining the original meaning, and is an extension of the algorithm proposed by Georgeff (1984).

The resultant expression may itself be evaluated using a standard run-time stack, without the usual encumberances of environment manipulation evident in lambda calculus interpreters. Alternatively, since special environment support is not required, code resembling conventional programs may be generated directly from the expression and executed in the appropriate fashion.

A possible code generation scheme might produce the following program from the lambda expression,

\[(\lambda p. \lambda q. p + q) \, 3 \, 5\]

program X(output);
var X : integer;
  function P(p,q : integer) : integer;
  begin P := p + q end;
begin X := P(3,5) end.

To demonstrate the failure of this strategy on expressions in which functions return functions (in applicative order of execution), consider generating code from the following.

\[(\lambda u. u \, 3) \, (\lambda p. \lambda q. p + q) \, 5\]

The following might be the program generated.

program X(output);
var X : integer;
  function P(p,q : integer) : integer;
  begin P := p + q end;
begin X := P(3,5) end.

The function call \(P(5)\) will form a binding of 5 to \(p\). That binding will be lost on return from \(P\), before any addition involving \(p\) is carried out. The problem exists independent of the remaining implementation and type compatibility details of the above program.

After transformation, however, the original expression becomes,

\[(\lambda u. u \, 3) \, (\lambda k. (\lambda p. \lambda q. p + q) \, 5 \, k)\]

which might be converted to the following program.

5. Other Translator Generators

Peter Mosses (1975, 1976, 1979) developed a translator generator called SIS (for Semantics Implementation System). The processes involved in SIS are as follows. Initially, two files – one containing the lexical and syntactic specification, and the other containing the semantics – are passed to SIS. From the first, SIS produces an SLR(1) parser for the language. To this parser is fed the program of interest, producing an abstract syntax tree of the program. SIS then uses the semantics file to generate an 'encoder'. This encoder, given the previously produced abstract syntax tree, will produce a (pseudo) lambda expression representing the initial program. Finally, that lambda expression is applied to a further lambda expression representing the inputs to the program. The application is evaluated using a lambda calculus interpreter, and the lambda expression representing the result is produced.

The path from specification to result is long and complex. Even having specified the semantics of a language, translation of a new program requires retracing of many earlier steps. Also, no independent translator is produced. The actions of translation occur as a result of SIS interpreting a generated data file.

The shortcomings and restrictions of SIS, it was, and remains a very important first step toward true translator generation.

The next step was a translator generator sometimes called PSP for Paulson's Semantic Processor (Paul-
son, 1981; Paulson, 1982). PSP differs from SIS by using an attributed grammar for syntactic specification. Although attributed grammars are complex, PSP is conceptually simpler than SIS.

PSP accepts a single specification file which contains semantic definitions embedded in the attribute grammar for a language.

Paulson's generator consists of three, separate phases. The first, the 'grammar analyser', takes the specification file and produces a language description file containing the semantic rules and a generated LALR(1) parser. A 'universal translator' constitutes the second phase. It transforms the previously generated file, along with a sample program, into a form of lambda expression representing that program: a complex transformation. The final phase is interpretation of the lambda expression.

PSP, unlike SIS, performs type checking on semantic function definitions and their use. It also distinguishes between run-time and compile-time actions, allowing the user to specify semantics for both phases of translation. PSP does not generate an independent translator.

6. Comments
This section presents some informal views on translator generation: its glitches and pitfalls.

An important property of a generated translator is that it conforms to the language specification. This is clearly so since it is an implementation of that specification. A translator written by hand often faces the problem of being inconsistent with the language definition.

Being based on a formal notation, semantics of varying languages can be compared. Development of a new language using a translator generator would be more consistent than current methods due to the need to formally define every intended meaning. Inspection of an almost complete specification can reveal glaring errors that might go unnoticed in conventional language development. Not only can some problems be more easily detected from a formal specification, but properties of a language are more easily abstracted. The language specified earlier, for example, is free of side effects in expression evaluation. Since expression evaluation is effected with the functions E and B, which return only integers and booleans, it can never modify the store used by the rest of the program. Abstractions of this nature are difficult to verify for hand written translators.

One of the problems of generated translators is inefficiency. Some inefficiency must be expected simply because of the generality of the system, as anyone who has used automatic generating tools will appreciate. It is unreasonable to expect large-scale tailoring of the generated translator to take advantage of properties of a specific language. The remaining inefficiency, and perhaps the greater part of it, can possibly be attributed to the youth of translator generation as a field of endeavour. Successive generators have improved in efficiency. Future generators will continue this trend.

Another problem shared by translator generators involves the size of the program to be processed. Any generated translator that chooses to represent a large program as its equivalent lambda expression may easily find itself short of memory. (For SIS and PSP this problem occurs in the system itself.) Hand written translators cope with this problem by treating in relative isolation small sections of a program (say a procedure at a time), rather than attempting to process the complete program in one action. It may be possible to incorporate these strategies in translator generators, but as yet this idea needs further development.

Acknowledgement
The author wishes to express his gratitude to Ken Robinson and Ian Hayes and to the referees, for their thoughts and constructive criticisms of an earlier draft of this paper.

References
Quokka: A Translator Generator


Biographical Note.
Trevor Vickers graduated from the University of New South Wales in 1982 with the degree of Bachelor of Science (Honours) in Computer Science. He has continued at the university as a candidate for the degree of Doctor of Philosophy, with the thesis topic 'Translator Generation using Denotational Semantics'. For the duration of his doctoral work, Trevor has been in receipt of an Australian Commonwealth Postgraduate Research Award.

CALL FOR PAPERS
SPECIAL ISSUE ON
INFORMATION SYSTEMS

The Australian Computer Journal will publish a special issue on "Information Systems" in November, 1986. This will be under the joint guest editorship of Michael Lawrence and Ross Jeffery, both of the University of New South Wales. Research papers, tutorial articles and industry case studies on all aspects of the subject will be welcome. Both full papers and short communications will be considered.

Prospective authors should write as soon as possible to:
Associate Professor M.J. Lawrence
Department of Information Systems
University of New South Wales
Kensington, NSW 2033

to inform the editors of their intention to submit material for the issue and to provide a brief summary of their proposed contribution.

In order to allow adequate time for refereeing and editorial review, complete manuscripts will be required by 31 May, 1986 at the latest.

Manuscripts should be prepared in accordance with the guidelines published in the May 1984 issue of the Journal. Authors are requested to pay particular regard to the Journal's preferred style for references.
A Clause Indexing System for PROLOG Based on Superimposed Coding

R.M. Colomb† and Jayasooriah†

Superimposed coding, a general method of partial match data retrieval, is adapted to give a general solution to the Prolog clause indexing problem. Two implementations are described, one using a conventional computer and the other using a special purpose attached processor for performing the calculations needed. The conventional implementation is superior to other proposed methods, while with the attached processor, the indexing scheme becomes effectively an associative memory.

Keywords and Phrases: Superimposed coding, information retrieval, partial match, hashing, Prolog clause indexing, associative memory.

CR Categories: H.3.1, I.2.3

Introduction
Prolog is a programming language based on logic. A good introduction may be found in Sammut and Sammut (1983b). It is well suited to symbolic computation, expert systems and other artificial intelligence problems, and especially to deductive data base problems (Lloyd, 1983).

This paper first describes the problem of indexing Prolog clauses and considers other solutions presented in the literature. It then describes the technique of superimposed coding and shows how it may be applied to the clause indexing problem, giving details of the implementation and results of performance tests. A specialized hardware device is then described which implements superimposed coding fast enough to be considered pseudo-associative memory. The impact of this device on the clause indexing problem is sketched, and some of the potential of this type of technology considered.

A simple expert system problem will illustrate enough of the essential features of the language that the problem addressed in this work will make sense. The system is to assist in finding a new residence by screening out the clearly unsuitable, leaving a short list of possibilities for further investigation. A set of descriptions will be held in a knowledge base, some describing residences which will be considered further, and others describing residences which will explicitly not be considered further. No description will be considered that is not found in the knowledge base. In operation, a real estate agent will present to the system a description of a residence, and the program will respond either yes or no.

A description is represented in Prolog as a predicate expressed as a clause, as follows:

A residence can be either a house or a unit:

\[
\text{residence(house(Bedrooms, Storeys), Price, View).}
\]

\[
\text{residence(unit(Bedrooms, Balcony), Price, View).}
\]

where:

a. \( \text{Bedrooms} \) is a number from 1 to 5.
b. \( \text{Storeys} \) is either one, split, or two.
c. \( \text{Price} \) is either low, medium or high.
d. \( \text{View} \) is either none, harbour or ocean.
e. \( \text{Balcony} \) is either yes or no.

A sample predicate in the knowledge base is

\[
\text{residence(house(3, _), low, harbour).}
\]

This statement is interpreted as: 'If the residence presented by the real estate agent is a three bedroom house with a harbour view and a low price, answer yes regardless of the number of storeys'.

The Prolog representation of the 'residence' predicate looks very much like relational data base notation. This similarity can be misleading. Relations are in fact represented in logic as axioms or facts, which are included in the notion of predicate, but a predicate is a more general concept. The form 'house(three, _)' is technically a function mapping the number of bedrooms and the number of storeys into a set of classes of house, specifically the class...
A Clause Indexing System for PROLOG

characterized by the indicated number of bedrooms and storeys. More detail may be found in the references above.

A few additional predicates will be added to the knowledge base:

A five bedroom house will be considered if it is two storey. Price and view are then not important:

residence(house(5, 2), _, _).

A unit will be considered only if it has three bedrooms, a balcony and a harbour view. Price is then not important:

residence(unit(3, yes), _, harbour).

The program in this simple system consists only of the knowledge base of patterns. The Prolog interpreter will accept a description of a residence from a real estate agent in the form of a query:

residence(unit(4, no), high, none)?

It will attempt to match this query against each predicate in its knowledge base in sequence. The interpreter will answer yes if one of predicates is true for the pattern of attributes given in the query, no if none are true. The process of pattern matching as performed by the Prolog interpreter is called "unification". (See Sammut and Sammut, 1983a, for a description of a typical Prolog interpreter.)

It is important that the Prolog interpreter searches its knowledge base of predicates in sequence. For example, the clause below will express that a high priced residence with no view will be rejected:

residence(_, high, none) :- !, fail.

The ':-!' signifies to the interpreter that if the predicate residence(_, high, none) is true, the actions following will be performed. In this case, the '!' signifies that no other alternatives will be considered, i.e. stop the sequential search. The command 'fail' signifies that the candidate will be rejected, i.e. 'fail' directs the interpreter to answer no.

To be effective, this clause must be placed before any other clause which might match the candidate and allow it to be accepted.

Some terminology is useful:

1. The main name of the clause is the principal functor (e.g. "residence").
2. The predicate containing it is the clause head.
3. Simple words of lower case letters are constants (e.g. "high", "none").
4. An item with brackets is a function, its name is a functor, and the items within the brackets are arguments. (For example, "house(four, split)" is a function, "house" is its functor, and "4" and "split" are arguments.
5. The don't care condition, represented as "_", is an example of a variable.
6. Either a function, constant or variable is a term.
7. A term which is a function is said to be structured.
8. The collection of all predicates with the same name is called a procedure.

Much of the computational effort in a Prolog program is consumed in the process of unification. In problems involving large data/knowledge bases, unification becomes a much greater proportion of the computation as it must include the effort of retrieval of data residing on slower secondary storage. Since unification conceptually requires a linear search of the data, techniques are needed which index the set of clauses. They will either permit sections of the file to be bypassed or will allow a pre-unification which eliminates incorrect patterns cheaply. It is important that no correct pattern be missed by the indexing procedure.

Indexing for unification differs from conventional database indexing in that in general a Prolog clause can have an arbitrarily structured set of arguments, and in particular can contain variables which match all candidates in a given argument position. In the example, structured arguments are the descriptions of house and unit with their different meanings, while variables are the 'don't care' conditions.

Most Prolog implementations follow Warren (1977) and Tick and Warren (1984) by indexing on one or more of the principal functors of the clause, number of arguments of the principal functor, whether the first term is a constant, variable, list or structure, or the first few arguments of the principal functor. These ad hoc indexing methods can be useful in problems of moderate size, but in many cases a procedure with very many clauses can be uniform with respect to the indexing method, so effectively reverting to a linear search.

MU-Prolog features an integrated relational database manager as described in Naish and Thom (1983) using a hashing technique for database access as described in Ramamoharan, Lloyd and Thom (1983). Only clauses without structure or variables may be indexed. Chomicki and Grudzinski (1983) describe a related implementation extended to terms with variables.

Wise and Powers (1984) describe a technique called field encoding based on \( m \) in \( n \) encoding of constant terms organized to handle both structured terms and variables. This technique and the previous one are described in more detail in the next section.

Motivation

The present work is motivated by two factors. First, the technique of superimposed coding, on which it is based, is a highly regular and simple indexing scheme which can be implemented in hardware, as described below. Second, the published methods for clause indexing are not satisfactory. The most important of them are analysed in the following.

Hashing is the technique employed in the MU-Prolog deductive database system. The clauses to be indexed are stored in a number of buckets on disk. When a clause is to be stored, the address of its bucket is calculated as a function of its constant terms. Most hashing schemes are oriented to retrieval on a single key attribute, so that the bucket address is
A Clause Indexing System for PROLOG

a function of that attribute. Here the address is a function of many or all of the attributes. The hashing function is organized so that different attributes contribute specific different bits to the final address.

When the data base is queried, the address of the buckets to search is calculated from the query attributes in the same way. Some of the attributes are typically missing from the query, so some of the bits of the bucket address remain undetermined. The retrieval program will search all buckets whose address matches the partial address calculated from the query.

One problem with this hashing method is that it does not lend itself well to matching variables or structured terms. Its main problem, however, is that unless the data base has quite particular statistical properties, it will tend to return a large number of clauses as potential matches which will fail in subsequent unification. The problem seems to be fundamental to this class of method, and is analysed more fully in Colomb (1985a).

Field encoding is based on an \( m \) in \( n \) coding of the clauses. A code word is built which is a given number of bits wide. Portions of the code word are allocated to each term in a way that mirrors the structure of the clause head. A fixed proportion of bits in the code word is allocated to each first level argument, then a fixed proportion of the bits for each first level argument is allocated to each element in a second level structure, and so on.

For example, if 256 bits are allowed for encoding a clause head, and a maximum of 8 arguments are allowed, 32 bits will be allocated to each first level argument. A second level structure will be assigned to the 32 bits allocated to its position in the argument list, and therefore if eight elements are allowed for this structure, only four bits will be allocated to each element.

The main problem with the technique is that unless the structure of the clause heads is known fairly precisely, and also all the terms have the same structure, a large proportion of the bits in the code word will be allocated to a structure which may not be present. For example, in the residence example above, suppose the structure of the ‘house’ term and ‘unit’ term were very different. If the index were tuned to the clauses with the ‘house’ term, it would not perform well on the ‘unit’ terms, and vice versa.

Basic Superimposed Coding Method

Superimposed Coding is a technique based on linear search of a highly compressed version of a file which has been proposed as a method of implementing file access for partial match retrieval, most forcefully in Roberts (1979). Further development of the concept may be found in Samanek (1982) and Sacks-Davis and Ramamohanarao (1983). A history of the technique, which first appeared with the introduction of edge-notched cards for information retrieval, and other references can be found in Knuth (1973). It is particularly interesting as special purpose hardware can be used to increase performance, e.g. Ahuja and Roberts (1980). The attached index machine hardware referred to in this work is reported more fully by Allen, Jayasooriah and Colomb (1986).

The technique of superimposed coding is first described here in terms of records as it was originally conceived, and will later be adapted to represent clauses. The scheme represents each record in the file with an encoding of all the attributes useful in partial match retrieval equality constraints. The record is encoded one attribute at a time. The attribute is encoded by setting exactly \( k \) bits to one in a word of \( b \) bits width. The encoding for the record is obtained by or-ing together the encodings for the individual attributes (hence superimposed). The encoding for the record functions as a compressed index for the file. Roberts shows a compression of at least five to one on the original data record.

The parameters \( k \) and \( b \) depend on the number of records in the file, the number of attributes encoded per record, and on the amount of collision allowed between encodings for different attributes in the record. (\( b \) is typically 10 times \( k \), so the encoding for one attribute is sparse). Formulas for calculating \( b \) and \( k \) in particular situations are given in the appendix.

Retrieval involves encoding the query attributes to be matched, then finding all records in the superimposed code index which have a one in at least all bit positions in which the resulting query superimposed code mask has the value 1.

Roberts (1979) notes that the volume of index data processed in a query can be reduced substantially by recognizing that the bit positions in the query containing the value 0 correspond to positions in the index which do not need to be examined at all. If the index is stored with bit one of all the records gathered into one string, followed by bit two of all the records, and so on (i.e. in bit slice fashion), only those bit positions containing one in the query need be processed. In addition, if as the bits are processed a count of the number of potential matches is kept, the pre-unification can switch to full unification when the number of potential matches gets low enough.

Application of Superimposed Coding to Prolog

The indexing system examined in this work takes advantage of the storage of a superimposed code index in bit slice fashion. It is fairly simple to process the query term by term. Each step will result in a bit map of the knowledge base indicating which clauses are potential matches at that step. The system allows both variables and structure, incorporating position information.

Calculation of the superimposed code word index for a clause head requires first reducing each item to be indexed to a 32 bit word (the word size of the machine on which it is implemented), then calculating exactly \( k \) bit positions in a field of width \( b \). Those \( k \) bit positions set to one will identify the item in the code word.
Variables are represented by their position, while constants are represented by the text of their name combined with position. Integers are encoded as themselves, combined with position in the same way as for constants. Position is represented by the tree address of the item in the clause head. For example, the variable \( X \) in the term \( f(a, b(c, d(X, Y))) \) has position \((2, 2, 1)\) since it occurs as the first argument of \( d(X, Y) \), which is the second argument of \( b(c, d(X, Y)) \), which is the second argument of the term. The method of combination is to exclusive or the two 32 bit words together.

In more detail, a 32 bit word is used to encode position, divided into four 8 bit fields, holding the first four numbers in the tree address. (A term has a maximum of 255 arguments.) Zero indicates that the address is completed. Tree addresses of depth up to three are represented exactly. If the fourth position is non-zero, it indicates a position of depth at least four, but full unification will be needed to resolve the exact position.

The principal functor can be indexed either explicitly or implicitly by having a separate index for each principal functor. If explicitly indexed, no position code is used, so that the encoding for the principal functor is distinct from the encoding of any of its arguments at any depth.

Terms will be indexed until depth of structure exceeds a specified maximum. Should any term of depth greater than three be included in the index, its position will be ambiguous.

Structure must be indexed at a level at a time because the query may contain a variable at any level, and so a system based on a fixed number of terms will not necessarily know how many terms in the query to index.

A query against a procedure guaranteed to contain no variables is processed very simply. The query is encoded in the same way as the clause heads, then the encoded query processed against the clause heads as described in the section on superimposed coding, above.

When the procedure may contain variables, the query processing is more complex. The goal term must be processed one term at a time. A particular clause in the procedure matches the goal if the constant matches, or if a variable in any position higher in the structure matches.

Consider the structure \( a(a_1, a_2, \ldots, a_n) \). If, for a particular clause, we represent:

1. by \( A \) the proposition ‘\( a \) matches a functor in the clause head’
2. by \( A_i \) the proposition ‘\( a_i \) matches a constant in the \( i \)-th argument position in the clause head’
3. by \( V \) the proposition ‘the functor \( a \) matches a variable in the clause head’
4. and by \( V_i \) the proposition ‘there is a variable in the \( i \)-th argument position of the clause head’

then the proposition ‘\( a(a_1, \ldots, a_n) \) matches the clause head’ is calculated as the boolean algebra expression

\[
V + A \& (V_1 + A_1) \& \ldots \& (V_n + A_n)
\]

where + denotes ‘inclusive or’ and \& denotes ‘and’, which involves each symbol only once for an expression of any depth.

Evaluation of one of the propositions in the underlying indexing scheme is based on bitwise ‘and’ operations and produces a bit map of clauses matching that term. Evaluation of a query against a procedure containing variables mirrors exactly the boolean expression above, using bitwise ‘and’ and ‘or’ operations on the bit maps. Although somewhat more complex than the constant case, it requires examining only twice as many bit positions.

This representation also shows that if the probability of a variable in the functor position is comparable to the probability of any specific constant functor, there is not too much to be gained by indexing below the first level. This is so because the bit map generated by the \( V \) proposition will dominate the final bit map.

Design Problems
A number of design problems have been solved to bring to preliminary implementation the system described. The solutions are in general capable of implementation in hardware. Two are described in this report:

1. Parameters of the indexing system.
2. Special treatment of common items.

Colomb (1985b) contains a more extensive description of the method.

Parameters of the Indexing System
The parameters of the code word calculation are the width of the code word \( (b) \) and the number of bits set to 1 per constant encoded \( (k) \). The formulas given by Roberts (1979) and reproduced in the appendix involve the number of constants to be encoded per clause, the number of clauses to be encoded, the average number of constant terms in a query and the permissible number of false drops.

Of these, the number of constants per clause to encode is not easily found because the clauses have structure. Consider the clause head \( p(f(a, b), g(c)) \) and the query \( p(X, g(c)) \). If we are encoding a fixed number of three constants, we would build the code word from \( f, g \) and \( a \). We would not encode \( b \) or \( c \). When we come to encode the query, we do not know whether to encode \( c \), because the presence of the variable \( X \) masks the structure in that argument. The solution is to encode all constants in a fixed number of levels. In the example, we would encode either only \( f \) and \( g \) or all of \( f, g, a, b \) and \( c \).

As the algorithm for indexing clauses where variables are allowed is more complex than that for indexing clauses containing constants only, the programmer can specify as a parameter whether or not variables
are allowed. The parameters chosen for the examples are described in the section on results, below.

**Frequently Appearing Items**

Items which appear very commonly in the coded index present a special problem. If a rarely appearing item collides with a frequently appearing one, there will be a large number of false drops when a query is made on the rarely appearing item. Note that the converse is not a problem, as a query on the commonly appearing item will generate a large number of true drops in any case.

Roberts (1979) shows that if the commonly occurring items are encoded with fewer bits than the rest, it is impossible for such a collision to arise. In addition, the discriminating power of commonly occurring items is weak, so that fewer bits in the code are needed to distinguish them. Unnecessary calculation is thereby avoided.

In this implementation, the following commonly occurring items are treated specially:
- The list functor '.' in position (1, 0, 0, 0) to (8, 8, 0, 0).
- Integers 0 to 9 in position (1, 0, 0, 0) to (8, 0, 0, 0).
No facility is provided for the programmer to add to this list dynamically, although not all the available codes are used and therefore the list could be extended.

These commonly occurring items are encoded with 2 bits in the $b$ bit code word, sequentially from (1, 1, 0, ..., 0), (1, 0, 1, 0, 0, ..., 0) to (0, 0, ..., 0, 1, 1). The list functor is allocated the first 72 codes, and small integers the following 80.

The smallest value of $b$ used is 32, so that the worst case is a two in 32 encoding, which has 496 different values. The first 152 of these are used in this scheme.

Note that variables, no matter how common, may not be encoded with a small number of bits. A constant in a query will match a variable in its position as well as the constant. The query evaluation shown in expression (1) will attempt to match both a variable and a constant for each position in the query containing a constant. Many false drops will occur if a constant in the clause head has the same encoding as a variable. The more bits in the encoding (the larger $k$), the less likely a collision (so long as $k$ is much less than $b$). A variable must therefore be encoded with as many bits as possible. In the test cases, constants and variables were encoded with the same number of bits.

**Performance Measurement**

The implementation described in this report has the index and all clauses held entirely in memory.

Warren's Population Density program (Warren, 1977) generates a large number of unifications, with a very large number of failures. Individual unifications are very simple, involving only one constant term. A four in 32 encoding was used. Details of performance measurements are shown in Colomb (1985b). The following points are noteworthy:

1. The number of false drops is very low, a maximum of 0.2 per successful unification where variables are not allowed.
2. A moderate number of false drops are generated if the indexing allows the possibility of variables. This ranges from two false drops for one successful unification in 25 clauses to four false drops for one successful unification in 100 clauses.
3. A significant cost saving is found, even where extra false drops are generated when variables are allowed in the clause heads. With a data base of 100 clauses, and taking the time consumed by a full sequential search as 100%, superimposed coding without allowing variables takes 25% and allowing variables 31%.
4. The crossover point where superimposed code indexing without variables is equal to sequential search comes at about 7 clauses with one successful unification.

Further tests were made using two data bases from Wise and Powers (1984), described in detail in Colomb (1985b).

The simpler of the two consists of 216 clauses with five terms, having neither variables nor structured arguments. Results are similar to the Warren example, except that since unifications are more expensive and the data base larger, the savings are greater: more than 10 to one when one unification is successful and no variables allowed, seven to one when variables allowed. A factor of more than 13 savings was found when no clauses responded to the goal. Even where 36 of 216 clauses responded to the goal, a saving of two to one was achieved. Numbers of false drops are still acceptable, the worst case being 10% of clauses in the procedure, and in most cases, none at all.

The more complex example consisted of 206 clauses with up to 5 levels of structure. Where no variables were allowed in the data, saving of superimposed coding over sequential search was from 3 to 1 to 5 to 1 for a query in which a failed unification is cheap, and from 10 to 1 to 50 to 1 for a query in which a failed unification is expensive. False drops ranged from zero to 17% of clauses. Two levels were encoded, with an average of nine terms.

Performance improvement where variables are allowed ranged from 28 to 1 to 4 to 1 where unifications were expensive. Where unifications were cheap, improvement was much less, ranging from 5 to 2 to 3 to 2. False drops ranged from 0 to 19% of clauses.

A 3-in-32 encoding was generally as good as a 4-in-32 encoding, although where variables are allowed, the 3-in-32 encoding generated more false drops. This emphasizes that when encoding variables more bits are needed to reduce false drops. The trade-off is between sufficient bits to distinguish the variables and saturation of the code word by having too many bits set to 1. (According to Roberts, 1979, the practical limit would have 50% of the bits in the final superimposed code word set to 1.)

The index for the simple data base took 14% of the space needed for storing clauses, while that for the complex data base took only 6%. There is therefore scope to increase the width $b$ and therefore to increase $k$. 

---

*The Australian Computer Journal, Vol. 18, No. 1, February 1986*
A query against a simple data base containing variables generated false drops ranging from zero to 16% of clauses.

Hardware Implementation: the AIM
As the calculations needed for superimposed code indexing are simple and regular, they lend themselves to implementation in hardware. A special purpose processor called the Attached Index Machine (AIM) has been developed by one of the authors (Jayasooriah) to perform these calculations, among others. It is described in detail in Allen et. al. (1986).

The AIM is based on the Texas Instruments 4161 256 x 256 x 1 bit dual ported Video RAM, which has the ability to extract 256 bits from, and restore to, an integrated high speed shift register in one memory cycle (300 nanoseconds). The AIM has an arithmetic unit which allows it to perform logical operations and simple comparisons on the bit streams in the shift registers. It is intended to function as a tightly coupled attached processor performing specialized functions, somewhat in the manner of a floating point unit.

In its prototype implementation, the AIM is integrated with a test system based on a National Semiconductor 32016 central processor. The video RAM chips are organized into two banks of 64k 16 bit words (128k bytes each) one port of which looks to the processor's bus and therefore functions as normal memory to the CPU. There is another 256k bytes of memory on the bus, so that the processor has 512k of RAM in all.

The AIM itself is a logic processor with its own registers also attached to the main bus. It is controlled by 16 bytes of instruction and status registers. It has its own memory bus (AIM bus) which is attached to the second port of the AIM memory. The AIM and the CPU can therefore access the AIM memory independently and simultaneously, with very little interference.

The AIM looks at the AIM memory as divided into pages of 256 sixteen bit words (512 bytes). A selected page can be transferred to the 16 256 bit shift registers built into the video RAM in one cycle. The AIM processor can then perform its operations on the 4096 bits, change pages, then restore the shift registers to a different page of RAM in another 300 nanosecond cycle.

Its instruction set includes the standard boolean operations on three addresses. The initial implementation is able to perform a logical operation on two 4096 bit pages and deposit a result in another 4096 bit page in 50 microseconds, including time to set up the next page.

Application to Superimposed Coding
Superimposed code word retrieval is shown in Colomb (1985b) to take about 500 microseconds per bit for a 256 bit slice on the VAX 780. If we multiply by 16, we get an estimate of 8 milliseconds per bit for a 4096 bit slice. This is an operation that the AIM can perform in less than 50 microseconds. The AIM therefore improves performance of this function by a factor of more than 100.

From the same source the overhead per query excluding any bit slice calculations is shown to be about 5 milliseconds. This overhead is independent of the number of bits in a slice, so that if the bit slices had been 4096 bits instead of 256 bits, we can estimate that the time needed for a query requiring examination of six bits would have been five milliseconds overhead plus 8 milliseconds per bit, or 53 milliseconds. Using the AIM, the bit slice calculations would take less than 0.3 milliseconds, so the whole operation would take an estimated 5.3 milliseconds. This shows a gain of a factor of 10 in the whole operation, including overhead.

We will develop some estimates for a realistic problem. Let us assume we have an expert system with a knowledge base organized as a single procedure with 4096 rules. This is the order of the number of rules in the expert system R1 (McDermott, 1982), which is fairly large as expert systems go. Using the formulas in the appendix, if we allocate a 128 bit code word and encode 5 bits per constant, we will get an average of one false drop if we encode 16 terms per clause and have an average of three constant terms in a query. The index therefore takes 4096 x 128 bits, or 64k bytes. This can easily permanently reside in the AIM memory.

If we allow variables, the three constant terms in a query each encoded with 5 bits evaluated by expression (1) above will require about 35 boolean calculations, and will result in a 4096 bit bitmap of the clauses, with a one in the bitmap for every clause passing pre-unification. If each of the constant terms has 16 equally probable possible values, then the specification of the three constant terms could identify a single clause in the 4096 clauses. It is therefore reasonable to assume that the bitmap has 3 bits set to one, therefore three clauses responding. Of these three, two clauses will be assumed to pass unification, and one will be assumed to be a false drop.

The AIM performs the boolean calculations in 50 microseconds per bit, so that the 35 boolean operations will take 1.8 milliseconds. Adding the five millisecond overhead for the query, the query will take 6.8 milliseconds to process and produce the bitmap.

The next step is to retrieve the three clauses from disk. It is clear that this process will take far longer than it took to produce the bitmap. Access time depends greatly on details of disk organization and the operating system, but the rotational latency alone is usually more than eight milliseconds. Combined time to read the three clauses would surely be in the order of 100 milliseconds.

Because the AIM is capable of performing a linear search of the file in a time small compared with retrieval of the required records, a superimposed code index implemented with the AIM is effectively an associative or content addressed memory in problems of this size.

Potential of the AIM Technology
It should be clear from the last section that the AIM is so much faster than current disk technology that its real power is only apparent if the superimposed code word index is permanently available in its memory. Fortunately, the superimposed code word index is highly
compressed, so that a small amount of RAM can index a large amount of data on the disk. We will extend the previous estimates to explore possibilities. In these estimates we will assume a data base without variables.

Suppose we had a megabyte of AIM memory. Many present-day workstations have this much memory for a single user. Applying the formulas in the appendix, we find that, as before, a 128 bit code word will index 16 constants in 64k of clauses with five bits set per constant. False drops allowed is still one, and the average query is assumed to have three constants. One megabyte would therefore suffice to index 64,000 clauses.

Assuming 3 constants in the query and no variables allowed in the data, an index calculation will take 15 boolean operations, each taking 50 microseconds per operation, for a total of 12 milliseconds. Adding the five millisecond overhead gives a total for the query of 17 milliseconds. This is comparable to a single disk access.

Suppose we had a data base of 1 million clauses. Using the same performance parameters as before, the code word would be 168 bits (21 bytes) and each constant would be encoded with seven bits. The index would therefore occupy 21 megabytes. A query with 3 constants would need 21 boolean operations, for a total of 269 milliseconds of calculation using the AIM.

Note that the superimposed code word index calculations are independent from clause to clause. There is therefore no reason not to perform them in parallel. Since the AIM can access its own memory independently of the central processor and independently of any other AIM accessing different blocks of memory, there is no technical reason not to put eight AIMs on the processor. Working in parallel (single instruction multiple data mode), these eight AIMs could perform the index calculations for one million clauses in 34 milliseconds—still comparable to a single disk access.

If the index must reside on disk, it is possible using the AIM to process the index at disk rotation speed. The dual ported AIM memory could have its shift registers gated directly onto the disk controller, so that 4k bits could be transferred to memory in one cycle without using either the processor bus or any processor cycles. (This facility is not implemented in the present version.)

One million bits can be processed in a boolean operation in about the time it takes the disk to make one rotation. A query on one million records is shown above to require retrieval of 21 bit slices. If the disk held one million bits = 128k bytes per track and the index were held in adjacent cylinders, it would take less than 400 milliseconds to retrieve the 21 tracks and produce a bitmap from a query on one million records.

### Additional Features of the AIM

The AIM has a number of other capabilities besides boolean calculations on bit strings, some of which are useful in superimposed code index processing. It can count the bits in a string in approximately one operation time. It can return the offsets of one bits, assisting in converting the bit map to clause addresses. It can shift the bit string left or right by one bit.

Shifting the bit string is important in updating the superimposed code index when a new clause is added. If the code word is 128 bits wide, adding a new clause requires inserting a bit into each of 128 bit strings. It takes four AIM operations to insert one bit. At 50 microseconds per operation, it would take only 26 milliseconds to update the index held in memory. A similar time would delete a record from the index.

### Conclusions

The prototype implementation reported in this work shows that the superimposed code scheme for pre-unification of Prolog procedures is workable, and that even when implemented on a conventional computer, it represents a significant and sometimes dramatic improvement over sequential search. Storage requirement for the index is a small fraction of that needed for the knowledge base.

When implemented on the AIM, the superimposed code index for problems of moderate size is so fast it is effectively an associative memory. The AIM technology can be extended to very large problems with little conceptual difficulty.

Colomb (1985b) shows that the method of this work when implemented conventionally is roughly comparable in performance to the field encoded word scheme of Wise and Powers (1984) described above. Its advantages are:

1. Superimposed coding is simpler and more regular, and therefore more adaptable to hardware implementation.
2. Superimposed coding is more stable. The field encoded word experiments reported were the result of carefully tuning the encoding scheme to the pattern of arguments in the clauses used. As noted above, if this tuning is not done, the efficiency of bit use drops a great deal.
3. Superimposed coding will give greatly superior performance in applications where the index is too large to reside in memory. Its use of bit slice representation allows a large reduction in the amount of the index actually processed in a given query.

### Acknowledgements

The authors wish to thank Professor Murray Allen for the inspiration of his long-time interest in superimposed coding, Dr. Claude Sammut for his advice and encouragement, and David Hume, John Hiller and the referees for many helpful comments.
A Clause Indexing System for PROLOG

References


COLOMB, R.M. (1985b): A Clause Indexing System for Prolog Based on Superimposed Coding, DCS Report 8506, School of Electrical Engineering and Computer Science, University of New South Wales, Australia.


NAISH, L. and THOM, J. (1983): The MU Prolog Deductive Database, TR 83/10, Department of Computer Science, University of Melbourne, Victoria, Australia.


Appendix: Design Parameters for Superimposed Code Words

The design parameters for a system of superimposed codes are given in Roberts (1979) as:

- \( N \) number of records to index
- \( r \) number of attributes per record to be indexed
- \( q \) average number of constant attributes in a query
- \( f \) average number of false drops allowed per query

If the width of the code word \( b \) is much greater than the number of bits set \( k \), and \( r \geq 5 \), then we have:

\[
q = \frac{0.96}{N/f} \times \log_2 (N/f) \quad \text{and} \quad b = 1.5kr.
\]

Biographical Note.

Robert M. Colomb received an S.B. in Mathematics from the Massachusetts Institute of Technology in 1964, and has since held a number of positions in industry in the USA and Australia. He has recently been appointed Principal Research Scientist with the CSIRO Division of Information Technology, responsible for advanced software engineering. Since 1984 he has been engaged in part-time study at the University of New South Wales towards his Ph.D. degree. His current research is in the application of superimposed coding to the implementation of Prolog.

Jayasooriah received his B.Eng. degree from the National University of Singapore in 1978. After working in industry, he joined the National University as a tutor, and obtained his M.Eng. degree in 1982. Since then he has been on leave at the University of New South Wales completing his Ph.D. degree. He returns to Singapore in March, 1986.
A Control and Data Flow Multiprocessor

S. R. Maeng† and J. W. Cho†

This paper presents the design of a highly parallel multiprocessor, a control and data flow heterogeneous multiprocessor (CDFM), which combines data flow, parallel forms of control flow, and decoupled access/execute architecture into a hybrid system. CDFM is designed to exploit the potential concurrency at the level of individual operations. A high level program is translated into a set of acyclic directed data flow program graphs and a control program graph in order to exploit the potential concurrency. Each node of the control program graph initiates and controls an data flow program graph. A brief analysis shows that the CDFM architecture and its program organization can exploit concurrency as much as conventional data flow approaches, but reduce greatly the overhead in implementing the loop constructs and array handling operations.

Keywords and Phrases: data flow architectures, control flow architectures, decoupled access/execute architecture
CR Categories: C.1.2, C.1.3

INTRODUCTION
We are doing research on and development of a control and data flow heterogeneous multiprocessor (CDFM) for fast parallel execution in computation bound applications. CDFM is designed to exploit the potential concurrency at the level of individual operations. This work has been motivated by the need to reduce the system overhead caused by implementing control dependency (e.g. conditional branch operations and loops) and to handle nonfunctional operations (e.g., I/O operations and data structure modifications) in data flow computing.

Data flow computers represent a radical departure from von Neumann architecture (see Arvind, Gos telow, and Plouffe, 1978b; Dennis and Misunas, 1975; Dennis, 1980; Treleaven, Brownbridge, and Hopkins, 1982b; Watson and Gurd, 1979). It is well understood that data flow machines are well suited for scalar processing. In data flow approaches, control flow structures such as conditional statements, loops, etc. are implemented using special purpose data flow operators such as SWITCH and MERGE (see Arvind et al., 1978b), but these require significant overhead for implementation and some segments of the program may be executed unnecessarily. Other problems for data flow approaches arise from trying to handle nonfunctional operations. I/O operations and data structure modifications do not fit into the functional view of data flow computing. Scalars must be, and can be, copied each time they are used or modified. However modification of a single element in a structure means copying the entire structure to avoid any side effects. This entails prohibitive communication and computation costs for large structures. The nonfunctional operations described above can be supported efficiently by the control flow machines. But the conventional von Neumann type control flow architectures are not suited for exploiting potential concurrency.

By architecturally decoupling data access from execution (see Smith, 1982), implementing the execution part using data flow computing, and constructing the data access part using ‘multi-thread’ control flow computing (see Farrell, Noordin, and Treleaven, 1979), it is possible to improve performance without any significant overhead in implementing the loop constructs and array handling operations.

Therefore, CDFM combines ideas from data flow machines, decoupled access/execute computer architectures (Cohler and Storer, 1981; Smith, 1982), and ‘multi-thread’ control flow machines (Farrell et al., 1979) and integrates them in a well structured manner.

Several authors have proposed new hybrid computer systems that combine the concepts of data flow and control flow computing. These include the Expression Processor (VanAken and Zick, 1981), PDF (Requa and McGraw, 1983), and the ‘Combined’ model (Treleaven, Hopkins, and Rautenbach, 1982a). These models are related to ours but have been independently conceived.
PROGRAM ORGANIZATION AND REPRESENTATION

A number of models for parallel computation and analytical techniques for exploiting parallelism have been studied. These include compiling suitable code from given programs written in conventional or functional programming languages (Ackerman, 1979; Padua, Kuck, and Lawrie, 1980; Peterson and Bredt, 1974; Treleaven et al., 1982a).

This paper attempts to show how the parallelism in a program written in a conventional language may be exploited. However, we are also convinced that the program organization in this paper is well suited for a functional programming language.

1. Program Organization

Programs for CDFM are defined at two levels: at the lower level, data flow programs perform pure functional, side-effect free operations. At the higher level, control programs initiate and control data flow programs and data accesses. Each program can be represented as a directed graph to represent the parallelism easily. Figure 1 shows a relationship between the programs.

Given a program, consider a block of assignment statements which contains no branching into or out of the block except at the beginning and/or ending. Then the block of assignment statements is said to be a DF-block. A DF-block can be translated easily into a data flow program graph and a node of the control program graph (see Maeng, 1984).

The presence of subscripted variables poses particular problems in the determination of a DF-block. For example, assume two assignment statements $A_1$ and $A_2$ are such that a subscripted variable $S$ appears on the left-hand side of $A_1$ and on the right-hand side of $A_2$. To determine whether $A_1$ and $A_2$ can be grouped into a DF-block, we need to determine whether the element of $S$ modified by $A_1$ is the same as the element of $S$ fetched or modified by $A_2$. If we can determine this at compile time, these two statements $A_1$ and $A_2$ must be separated into two DF-blocks executed sequentially (see Maeng, 1984). The transfer of control among the DF-blocks caused by the conditional, iterative, branch statements, or the data accesses are implemented by the control program graph.

2. Program Representation

An example of program representation is given in Figure 2. The computational part, $y \times (r \times z + t \times z)$, which can be thought of as the template for the computation, is carried out by the data flow program graph shown in Figure 2(c). The data flow program graph is made of actors connected by arcs. Data flow program graph representations and their interpretations are well described by Arvind and Gostelow (1978a), Arvind et al. (1978b), and Treleaven et al. (1982b). However, in our approach, the special purpose data flow operators such as SWITCH, MERGE, L, and D (Arvind et al., 1978b), are eliminated and only pure functional operators are considered, because a data flow program is translated from a DF-block.

The FOR loop control and the array data accesses are carried out by the control program graph. The control program graph is also a directed graph consisting of a finite set of nodes and a finite set of arcs as shown in Figure 2(b). A directed arc called a control arc represents the order, in which the nodes are to be executed. A control signal is carried by the control token which travels along the control arc. The passing of data between two nodes in the control program is accomplished by reference to a shared memory location.

Each node contains all the information and instructions necessary for controlling data flow programs translated from a DF-block. The information contained in the node is as follows:

1. The header part of the node contains the control information such as the number of input control tokens required for enabling the control node, the number of control tokens arrived currently to this node, the size of the node, the number of
address(es) of the successor node(s), and the status of the node.

2. The body section contains instructions which control the flow of data and synchronize the usage of the shared resources. These instructions perform necessary operations for fetching data, passing data to the data flow machine, storing result data, synchronizing executions of data flow programs and control programs, and selecting successor node (s).

3. The tail section contains address(es) of the successor node(s).

\[
\text{FOR } k = 1, 400 \text{ DO}
\]
\[x(k) = y(k) \times (r \times z(k+10) + t \times z(k+11))\]
\[
\text{ENDFOR}
\]

(a) Loop construct in a high-level language

\[
\begin{array}{|c|c|c|c|}
\hline
\text{# of the CTS} & \text{# of the CTS} & \text{node size} & \# of the next \text{addresses} & \text{Status} \\
\text{required} & \text{currently arrived} & & & \\
\hline
1 \text{ LOAD } R7, -400 & \text{ LOAD } R2, 0 & \text{ LOAD } R3, 1 & \text{ SEND } & \\
2 \text{ LOAD } R2, 0 & \text{ index increment} & & \text{ipt1, z+10,R2} & \\
3 \text{ SEND } & & & \text{send z(k+10) to DFPG} & \\
4 \text{ SEND } & & & \text{ipt2, 1} & \text{ with the color } (\text{node } \text{addr}, \text{ITER}) \\
5 \text{ SEND } & & & \text{ipt3, z+11,R2} & \text{send z(k+11) to DFPG} \\
6 \text{ SEND } & & & \text{ipt4, y,R2} & \text{store y(k) to DFPG} \\
7 \text{ STORE } & & & \text{opt0, x,R2} & \text{store result from DFPG into x(k)} \\
8 \text{ INC } & & & \text{iter} & \text{increment iteration number} \\
9 \text{ INC } & & & \text{r7} & \text{increment loop counter} \\
10 \text{ JUMP } & & & \text{if neg. 4} & \text{next address selection} \\
11 \text{ Select } 1 & \text{next address selection} & & & \\
\hline
\end{array}
\]

(b) control program

\[\text{(c) data flow program opt0}\]

Figure 2. Example of a Program Representation

3. Interpretation of a Program

The interpretation of the control program graph is accomplished by the enabling and execution rules for the node in the graph. A node is enabled when all of its necessary input control signals are available on its input control arc(s). The number of control signals required to enable a control node is contained in the header section of each node.

An enabled node is executed according to the following rules in a finite time after it has been enabled:

1. Enabled nodes are executed in a random order.
2. When the execution of the node commences, no other control signals enter this node until the execution of this node is completed.
3. While a node being executed, it performs several operations:
   a. absorbs control signals from the input control arcs,
   b. sets the iteration number register (ITER) to zero,
   c. executes instructions in the body section sequentially,
   d. and puts the control signal(s) on the output arc(s) selected in step (c).

In Figure 2, the computational part \(y \times (r \times z + t \times z)\) can be thought of as the template for the computation, which is executed 400 times. This template can be executed in parallel regardless of other executions of the template if input data are available.

To support such concurrent execution of each successive iteration and multiple instances of a data flow function, the ‘colouring scheme’ is used (see Arvind et al., 1978b; Watson and Gurd 1979). A node of a control program graph issues memory read or write operations as soon as it calculates memory addresses for the reads or writes, and distinguishes an input data set for one invocation from others by its colour. Therefore, before a data flow code is invoked, a new ‘colour’ in which its instructions can be safely executed, is created for it and attached to the input data for it by the control processor. A colour consists of a control node address and an iteration number. The iteration number is incremented by one whenever the iteration terminates. Thus, each iteration of a loop will have a different iteration number. The control processor has an iteration number register (ITER) to support this mechanism.

By using this colouring scheme, the control program can execute without waiting for the results from the data flow program graphs. Therefore, the statements 9-13 in Figure 2(b) can be executed before the results from the data flow program graphs arrive. Thus, the loop can be unfolded and executed in parallel in CDFM.

Now, consider the following loop construct:

\[
\text{for } k = 1, N \text{ do}
\]
\[S = S + f(k)\]
\[
\text{endfor}
\]

In this case, the output of the current iteration, variable S, is used to the input of the next iteration. As mentioned earlier, memory addresses for reads and writes may be computed well in advance of when the data is available from the data flow program graph. This raises a problem known as the ‘read-after-write’ problem in asynchronous parallel processing (Ramamoorthy, 1977). To resolve the 'read-after-
write' problem and avoid unnecessary read and write accesses to the same memory locations, the CONNECT instruction is included. These instructions colour the outputs of the current iteration from the data flow program with a new iteration number, sending them to the inputs of the following iteration without storing the current results into memory locations. This reduces the traffic between processors and memory and eliminates the problem of synchronization to resolve the 'read-after-write' phenomenon. These mechanisms are described in detail in the next section.

In multiple loop constructs such as:

```
for i = 1,N do
  for j = 1,M do
    endfor
  endfor
```

each iteration of the loop body can be numbered sequentially from 0 to NM-1. Therefore, CONNECT instructions can be used also, if necessary. In general loop constructs, the loop decomposition technique is used to construct the multiple loops as described above for the general multiple loops (see Maeng et al., 1984).

THE ARCHITECTURE OF CDFM

The CDFM architecture is shown in Figure 3. It consists of two types of processors, data memories, and interconnection networks: a number of control processors (CP's), a number of data flow processors (DFP's), data memories (Data MEM's), and two interconnection networks. The control processor and the data flow processor contain the local control program memory and the local data flow program memory, respectively. Program execution on CDFM is supported by two levels of activities. At the higher level, the control processors perform all the necessary operations for transferring data to and from data memory, initiating the data flow programs, global coordination and program control. At the lower level, each data flow processor performs pure functional, side-effect free arithmetic operations on its own execution stream. The data memory holds persistent data while the program memory is associated with each processor. Similar concepts are proposed by several authors (Requa and McGraw, 1983; VanAken and Zick, 1981), but in CDFM concepts of data flow and control flow computing are combined hierarchically.

The control processors and the data flow processors in CDFM correspond to the access processor and the execute processor in the conventional decoupled access/execute computer architectures (Cohler and Storer, 1981; Smith, 1982), respectively.

1. Data Flow Processor

Data flow processors in CDFM, depicted in Figure 4, are similar to the ones proposed by Arvind, Kathail, and Pingali (1980) and Watson and Gurd (1979). The data flow processors communicate with each other or with the control processors through the interconnection network.

When a data token arrives from the network, it is associatively checked in the waiting and matching queue (WMQ) for tokens destined for the same instance of an instruction. When a match is found, the matched tokens are sent to the instruction fetch unit. After the instruction has been fetched, the instruction and the necessary operands are sent to the functional unit for processing. After processing, a local result is sent to WMQ, while a non-local result is routed through the interconnection network to the desired data flow processor or the desired control processor.

A node in a data flow program graph is executable only if all necessary input tokens have the same colour. The colouring method permits the use of pure static code and enables the maximum use of any parallelism that exists in the problem specification. In pure data flow approaches, the lack of a centralized control makes it difficult to generate unique colours (Arvind et al., 1980). In CDFM, token colouring is easily carried out in the control processor in contrast to conventional data flow machines.
2. Control Processor
Each control processor contains three functional units: an enabled node selection unit (ENSU), a node execution unit (NEU), and a queue management unit (QMU) as shown in Figure 5. The execution of a node in a control program graph consists of three asynchronous tasks executed on three functional units, respectively.

The ENSU manages two queues: the control token queue (CTQ) and the enabled node address queue (ENAQ). The control token generated by the predecessor node enters CTQ. The enabled node selection unit takes one control token from the front of CTQ, and fetches the information contained in the header part of the node designated by the control token. It then tests the status of the node. If the subject node is enabled currently, the control token is inserted back into the rear of CTQ. Otherwise, the number of control tokens currently waiting at the subject node is incremented and tested to see whether the node can be enabled. If it can be enabled, the address of the enabled node is put into ENAQ and the number of control tokens currently waiting is cleared and the status is set to 'enabled'. When the node cannot be enabled due to the absence of some of the control tokens, the number of control tokens currently waiting is merely updated.

The NEU takes the address of the enabled node from ENAQ, and then begins execution. When the execution of the enabled node terminates, the ‘enabled’ state is reset and the control token(s) containing the address(es) of the successor node(s) is put into CTQ or routed to CTQ in the desired control processor through the interconnection network.

Once a node of a control program begins to execute in the NEU, other nodes cannot preempt the NEU until the node completes its execution. Therefore, no context switching problem arises and no state information needs to be retained by the NEU. The iteration number register (ITER) in the NEU is used to distinguish among multiple instances of a data flow program. Before an instance of a data flow program is invoked, ITER is set to a different iteration number.

The data paths according to the instructions executed by the NEU are also depicted in Figure 5. The SEND instructions send data with the colour to the data flow processors through the interconnection network. Thus, the data flow programs are initiated. Each data flow processor receives data, performs the operations according to the instructions within its local program memory, then returns the results via the result data queue (RDQ).

As mentioned earlier, to exploit more concurrency, the NEU issues STORE operations or CONNECT operations as soon as it computes the store address or the destination address of data which is returned back to a data flow processor, respectively. It does not wait until the results from the data flow machine are received in RDQ. The store or the destination addresses awaiting results are held temporarily in the write address queue (WAQ) or the destination address queue (DAQ), respectively. As the data arrives at RDQ, it is paired with the write address in WAQ or the destination address in DAQ and is sent to data memory or the data flow machine, respectively, as shown in Figure 5.

Conditional branches and loops are implemented in the control processor, but condition expressions are calculated in data flow processors. The results of condition expressions are returned via branch status queue (BSQ). Therefore, condition test instructions may access data in BSQ, if the condition expression is executed in the data flow machine.

To reduce the task of the NEU, the management of the queues described above is achieved independently by the QMU.

Figure 5. Functional Block Diagram of CP
ANALYSIS OF THE CDFM ARCHITECTURE

In this section, several comparisons of CDFM with the data flow machines are made. The U-interpreter (Arvind and Gostelow, 1978a) is selected for comparison because it can exploit the maximum parallelism in a given data flow program graph. Figure 6 shows the translation of the program in Figure 2(a) for the U-interpreter.

First, in order to analyze only the relative efficiencies of the architectures and to eliminate effects of specific implementations, three measures (S, M, E) are selected. These measures serve as reasonable first-order approximations. Similar measures to analyze architecture performance are considered by Dietz and Szwerenko (1979).

1. \( S \): the size of a program, i.e. the total amount of memory required to represent the program, including instructions, operand addresses, control information, temporary work area, place holders for operand values, and result destination addresses.
2. \( M \): the total number of words transferred between memory and the processors during the program execution.
3. \( E \): ratio of \( M \) to the number of executions of instructions which perform computational functions. This value relates to the system overhead.

To simplify the analysis, several assumptions are made. It is assumed that 0.5 memory word is required for an operation code (opcode), 0.25 for the register address specification, 1 for a memory address, place holder for an operand value, or an operand, and five for the header in Figure 2(b). CDFM contains a file of eight registers in the control processor and WMQ in the data flow processor as the temporary working storage while the U-interpreter contains the waiting and matching queue for the temporary working storage. Under these assumptions, CDFM requires 52 memory words for the representation of the test program in Figure 2(a), i.e. 7 for opcodes, 8 for the temporary working registers, 27 for the addresses and the constants, and 10 for the data flow program while the U-interpreter requires 115 memory words, i.e. 14 for opcodes, 51 for the temporary working storage and the constants, and 50 for the destination addresses. In CDFM, the traffic among CP's, DFP's, and data flow program memory is also included in \( M \) because it affects the system performance. The corresponding value for \( M \) of the U-interpreter is the traffic among the processor, the waiting and matching queue, and the data flow program memory.

A comparison of three measurements for the test program in Figure 2(a) on CDFM with those on the U-interpreter is shown in Table 1. For \( S \) and \( M \), CDFM is smaller by the factor of about 2. It also appears that \( E \) of the CDFM is much smaller. Therefore, by use of CDFM, execution time overhead can be reduced greatly. These improvements result from the elimination of the complex mechanisms imple-

### Table 1. Performance Comparison

<table>
<thead>
<tr>
<th></th>
<th>( S )</th>
<th>( M )</th>
<th>( E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>U-interpreter</td>
<td>115</td>
<td>( 239N + 54 )</td>
<td>( \frac{239N + 43}{4N} = 59.7 )</td>
</tr>
<tr>
<td>X-machine</td>
<td>52</td>
<td>( 102N + 7 )</td>
<td>( \frac{102N + 7}{4N} = 25.5 )</td>
</tr>
</tbody>
</table>

\( N = \) number of iterations

Concluding remarks and structure operations in the U-interpreter (e.g. control operators such as \( D \), \( D^{-1} \), \( SWITCH \), \( L \), \( L^{-1} \), \( SELECT \), \( APPEND \), etc.) and the introduction of the concept of the shared memory. The U-interpreter does not have the concept of storage. Therefore, implementation of the loops or the array operations incurs the significant overhead.

Second, to estimate the execution times, the degree of parallelism that can be exploited in program execution needs to be considered. It can be shown that the U-interpreter finds more parallelism than the usual way of interpreting a data flow graph (Arvind and Gostelow, 1978a). As mentioned earlier, since the control program carries on executing without waiting for the results from the data flow machine, the statements 9-13 in Figure 2(b) can be executed before the results from the data flow machine arrive. When this is the case, the next iteration can be started with a different colour. Therefore, CDFM can exploit parallelism much the same as the U-interpreter. It is true that, if the U-interpreter employs I-structures (Arvind and Thomas, 1981), it can exploit more parallelism than CDFM. However, an attempt to read an element not yet defined is deferred until the element is defined. A linked list called the deferred read request list is maintained for each I-structure element. There are no figures that show how far this list can inhibit the gained parallelism. Also, in some cases, a backlog of instructions that will never be executed may be created, cluttering the system.

If each instance of the data flow program requires a great deal of computation, then more unfolding takes place. In the U-interpreter, this would pose a significant practical problem of managing limited processors and memory resources because no control mechanism is provided. In CDFM, unfolding can be managed easily by the control processor.

CONCLUSIONS

This paper has presented a highly parallel multiprocessor, CDFM, which combines data flow architecture, decoupled access/execute architecture, and ‘multi-thread’ control flow architecture into a hybrid system. Its program organization and an example of the program representation are also presented.

It has been shown that the program organization consisting of two levels can exploit parallelism as much as conventional data flow programs, but reduce the overhead greatly in implementing the loop con-
structs and array handling operations. It is contended that the modular and asynchronous nature of the architecture makes the system design simple.

But also, since CDFM is a new hybrid system and under development its weaknesses are not fully understood and much work on the CDFM architecture design remains to be done. Multiple queues, interconnection networks, memory system, the interface between CP and DFP, and load balancing among CP’s and DFP’s deserve a substantial amount of further study.

One of the studies currently in progress is the design of interconnection networks. Communication among the control processors and the data flow processors may be achieved by using a ring network, a dedicated set of channels, cross-bar switches, time-shared buses, multi-stage networks, or a combination of these modules. A ring network is a logical choice for CDFM (Farrell et al., 1979), since it offers simplicity, flexibility, and modularity.

The control processors are also connected with the data memory by an asynchronous interconnection network. Asynchronous interconnection networks between processors and memory modules have been studied by several researchers and several proposals (Barnes and Lundstorm, 1981; Dias and Jump, 1981; Padua et al., 1980) are being considered for CDFM.

Also, much work on more efficient program representations remains to be done. Load balancing among CP’s and DFP’s requires further research. Detailed simulation and performance evaluation studies are being made by the authors.

References


Biographical Note.

S.R. Maeng is currently an assistant professor of Computer Science at the Korea Advanced Institute of Science and Technology (KAIST). He received the M.S. and Ph.D. degrees in Computer Science from the KAIST, Seoul, Korea, in 1979 and 1984, respectively. His main research interests include dataflow computer architectures, parallel computer architectures, and Prolog machines.

J.W. Cho is a professor of Computer Science at the KAIST. He received the M.S. degree from the University of Wyoming in 1968 and the Ph.D. degree from the Northwestern University in 1973. His research interests include distributed and parallel computer architectures and fault-tolerant computing.

The two authors have been involved in a national ‘Next Generation Computer Project’ since 1984.
Letter to Editor and Reply

COMMENTS ON KNOWLEDGE REPRESENTATION ARTICLE

I was surprised to read an assertion by Garner and Tsui regarding the limitations of symbolic logic ("Knowledge Representation for An Audit Office", ACJ August 1985, page 108). They say that while symbolic logic is "highly general and flexible", it cannot avoid certain fallacious inferences, and give the following examples:

- Clyde is an elephant
- Elephant is a species

Hence, Clyde is a species.

Now this is certainly a very elementary fallacy. It is hard to imagine anybody being seriously confused by it. So what usefulness could there be in a logic which could not avoid it, no matter how "general and flexible" it might be otherwise?

In fact, it was the inability of traditional ("syllogistic") logic to deal with fallacies considerably more subtle than the above which spurred the development of symbolic logic, and led to the placing of mathematics on a logical foundation (see Beth (1959) and Russell (1908)).

The fallacy above is one of logical level or "type", and it is true that a simple first-order logic (such as Prolog implements) is not directly capable of expressing the second premise. But this does not mean that the logic "cannot avoid" this fallacy, because the erroneous conclusion cannot be derived either.

The correct way to deal with this situation is to embed the premises in a system of set theory, itself expressed in first order logic. So one could have

```
... axioms for "member_of_set" ...

member_of_set( clyde, elephants ).
member_of_set( elephants, species ).
```

From this the conclusion "member_of_set( clyde, species )" only follows if "member_of_set" is transitive, which it should not be. If, however, one wrote a Prolog program containing the clause

```
species( X ) :- elephant( X )
```

one would merely be saying that all (individual) elephants are species, and the language cannot be blamed for the nonsense that would ensue. Such an error seems to be the explanation for Garner and Tsui's further assertion that a "direct mapping" of the fallacy into Prolog yields the same result.

REFERENCES


AUTHORS' REPLY

We appreciate the interest shown by Brian Currie in our comments on the limitations of PROLOG for mapping directly from natural language into symbolic logic. The use of ingenious contrivances to avoid fallacies of the type referenced is not disputed. What is in dispute is the need to use such devices for trivial examples and the virtual impossibility of coping with the real world in the construction of complex, dynamic knowledge bases using only PROLOG without the benefit of a specific notation such as the theory of conceptual graphs provides.

The avoidance of fallacious inference by using conceptual graphs is well documented elsewhere (Sowa, 1984) and by keeping the concept type label and its referent as distinct entities provides the best practical approach that I am aware of for the average professional knowing little of the mysteries of PROLOG to implement their own knowledge bases and inference engines.

The work by Doug Skuce (1984) and the need he discovered to invent an intermediate language (LESK-Language for the Exact Statement of Knowledge) should also be noted in any discussion of Knowledge Source Systems. The contribution of Helm, Marriott and Lassez (1985) on the Evaluation of PROLOG for Expert Systems in Government is also worthy of detailed study by the interested reader.

Without wishing to detract in any way from the power and flexibility of PROLOG for mapping symbolic logic, we should be careful not to encourage the development of a race of specialists who become bottlenecks in the application of knowledge engineering techniques to real world problems. The need for notations with a good underlying theory, such as Conceptual Graphs, is paramount if users are finally to transcend the many barriers created by computing specialists. In our view, J. Sowa's contribution will be recognised as a landmark in the development of such user-friendly notations with the potential to provide unambiguous resolution of the many semantic problems encountered in capturing real world knowledge. The recent review of his book by W.J. Clancey should be compulsory reading for all Knowledge Engineers in our view.

REFERENCES


Brian J. Currie
Department of Main Roads, Oxford Street, Sydney

Deakin University
Book Reviews


Why do we teach programming in schools? By and large our students will never become programmers, so why waste their time trying to teach them this difficult science?

I recommend that all teachers of programming read this book. It is a collection of the proceedings of the IFIP TC3 Working Conference on Teaching Programming held in Paris in May 1984, and is divided into nine main sections:

- Teaching programming to everybody
- Are there cultural reasons for teaching programming
- The context of the profession
- The role of new languages
- Programming curriculum
- Importance of theory
- Tough nuts in teaching programming
- Techniques and methods
- Final discussion.

About 25 fairly brief papers are included in the book (along with any discussion which arose from the presentation of the papers). Although some suffer from language limitations (17 countries were represented at the conference), there are sure to be some of interest to anybody involved in programming and education. The papers represent some quite divergent views on the hows, whys, and should-we's of teaching programming, and the problems involved.

Among the papers I found particularly interesting was one by J. Hebenstreit from L'Ecole Superieure d'Electriite discussing the traditional argument that programming should be taught to everybody including children. He suggests that what the typical user will eventually need to know (and almost everybody will be a typical user in a few years) is how to use a computer and its applications programs — not how to write Mickey Mouse programs in BASIC.

An article by Ronald Ragsdale and Betsy McKevery from Ontario states that the current emphasis on programming teaching in Canada is because of expectations of Canada becoming an exporter of software. The authors wonder if the demand for programming skills will grow as anticipated.

Many of the papers tackle the problem of whether there are cognitive benefits from learning computer programming, or even benefits to the development of logical thought. The suggestion is made that there may be better ways of teaching the skills of abstraction and analysis.

The participants express concern about the shortage of trained and experienced teachers. A number of different types of languages are discussed (Pascal, PROLOG, Smalltalk-80). Teachers from a number of different countries discuss the methods they have used to introduce students to programming.

The book concludes with a final discussion and summary. There was general agreement amongst the participants that there is no consensus on what programming is, or how it is changing. It is too early to make any resolutions about the teaching of programming except to say that the position should be reviewed at another conference in a few years' time.

Although the conclusions seem unsatisfactory, a reading of the book will show clearly why these conclusions were reached.

A valuable book for the shelves of any teacher or potential teacher of programming.

Sue Trahair
Franklin College of TAFE.


This book contains seven chapters which develop an understanding of the two verbs, SOUND and ENVELOPE, allowing the reader to fully exploit the B.B.C. microcomputer's sound capabilities. The book is written by a trained teacher and in this regard, it carefully leads the reader through the maze of parameters associated with the two verbs, by giving well explained examples and hints on technique and experimentation.

The author has taken the trouble to explain the terms pitch, amplitude, timbre and then has described the generation of sound within the computer itself. Chapter 3 examines in detail the SOUND keyword and gives a full description of the four parameters associated with it. Several programs complete with easy to follow documentation, are provided to allow a novice to become familiar with the full capabilities of the SOUND verb. The channel parameter and noise channels are explored at the end of this chapter, complete with some ideas of sound effects production.

Chapter 4 looks at the ENVELOPE statement, with its 14 parameters and again describes, with examples, the use of these parameters. Some envelope theory is given in this chapter together with a program listing to design envelopes. The fifth chapter of the book was a disappointing culmination of the previous two chapters, in that too much time was spent looking at the graphics of musical notation, and not enough on the combination of the SOUND and ENVELOPE statements to produce certain 'known' sound types. The author has rightly indicated that many sounds are not possible to synthesise but has largely left the work up to the reader. Perhaps a few 'instrument-types', tried by the author, could have been added.

The use of the speech synthesiser is briefly explored in Chapter 6, with examples of tones, pauses, compound words etc. being looked at in depth. Again programs are provided to allow guided use of the speech synthesiser.

The final chapter has been written for people with some expertise in ASSEMBLER programming and effectively shows how some of the earlier constructs can be written in ASSEMBLER. The layout and style of the book make it a very useful acquisition for the owner of a B.B.C. micro, the programs provided, together with the extensive appendices will suit both the novice and the more experienced programmer. At $14.95, it is definitely good value.

R. MacGregor
University of Wollongong


This book describes in detail the use of Sprite Graphics using BASIC on the Commodore 64 micro-computer. The book is probably only relevant to neophyte programmers who find the description of Sprite Graphics in the Commodore Reference Manual daunting and who are not familiar with 'bits and bytes' computer terminology.

With this audience in mind the book provides a gentle introduction to Sprite Graphics and provides yet another screen editor which is used in the book for developing the sprites. However no description of the operation of this editor appears in the book.

All of the features of sprites available on the Commodore 64 are covered including multi-colour sprites, priorities, collision detection and movement. A method for saving sprite information on tape and retrieving it for later use in a program is described.

The book dismisses the slowness of sprite graphics when using BASIC by saying "Movement doesn't have to be lightning fast to make a good game". No mention is made in the book of the use of machine code routines to speed sprite movement. It does mention animation by switching various sprites into view.

There are many useful tips for beginning programmers, and most of the techniques developed in the book come with a good explanation.

Glyn W. Poady
Australian Atomic Energy Commission


The authors state that the book may be approached on three different levels. Firstly, as a collection of programs which may be used to

The Australian Computer Journal, Vol. 18, No. 1, February 1986
produce complex drawings on a computer screen. Secondly, and similar to the first, as a package to produce and label "data diagrams" for Business and Laboratory applications, ie, pie-charts, histograms and graphs. Thirdly, and the main objective of the book, as an introductory text on computer graphics.

The chapters are arranged in a logical order which progressively introduces the reader to new concepts and then proceeds to practical application, with worked examples and complete program listings. Diagrams and figures clearly show theory and typical results that can be expected. Chapters include: Introduction, Graphics Operations of the Acorn Electron, From Real Coordinates to Pixels, Two-Dimensional Coordinate Geometry, Matrix Representation of Transformations on Two-Dimensional Space, Character Graphics on the Acorn Electron, Diagrams and Data Graphs, Three-Dimensional Coordinate Geometry, Matrix Representation of Transformations on Three-Dimensional Space, Orthographic Projections, Simple Hidden Line and Hidden Surface Algorithms, Perspective and Stereoscopic Projections, A General-Purpose Hidden Line and Hidden Surface Algorithm, Assembly Code Manipulation of Pictures, Advanced Programming Techniques, A Worked Example of a Video Game and Projects. In all there are 16 chapters in the book.

The main points throughout the book appear under separate headings, as do the examples and exercises. References are given for further reading and a good index is provided to help locate a subject/item quickly.

The authors emphasize the top-down modular approach in the introduction of the book and this is borne out by the method of analysis and program listings which clearly demonstrate its use. This approach, although obvious to the experienced person, is of fundamental importance in any problem solving exercise, and especially so in any introduction to graphics programming.

Although the authors are dealing with specific hardware, the capabilities discussed are present in many machines available these days and in general the characteristics may be mapped out and adapted to the reader's situation, ie, selection of palette and colours, physical hardware characteristics, programming environment/language. Real coordinate to pixel selection and character generation are also discussed and treated in a manner which should allow easy adaptation.

The authors' treatment of Two and Three-Dimensional Coordinate Geometry and the use of Matrix representation to efficiently manipulate and store scenes is presented clearly and reinforced by the worked examples and program listings. Orthographic, Perspective and Stereoscopic Projections are addressed in turn and finally the problems of Hidden Line and Surface handling are discussed and a general solution given.

In conclusion, although the programs are written for the Acorn Electron, BASIC is the principal language used and therefore all the programs should be adaptable to the reader's situation. The Assembly code routines given would require more work, but these are relatively short and probably within the means of any reader seriously studying this book. Within the main objective of the book, the reader is always encouraged to adapt the routines and so to learn by doing. Despite the title, this book should be considered a good introduction to computer graphics, whatever machine/hardware/software used.

J. Del Floramo
Road Construction Authority, Victoria

Computer Graphics is defined in terms of Modelling, Storage, Manipulation and Viewing. The emphasis being on whether the generated image is to be used for scientific purposes or, for example, for a video game. A number of different methods are described and illustrated, including explanations of how interactive graphics has moved from the light pen to the digitising tablet, joysticks, tracker balls and finally to mice.

Six authors are acknowledged, and each chapter has a good written description of the subject matter of that chapter. The history and techniques are fully described, and lots of suitable, interesting illustrations are provided. Although this is a seriously written book, it can be enjoyed purely at the picture book level. Among the hundreds of fascinating pictures there are some unusual ones, such as the portrait of John F. Kennedy in wire frame boxes (from the Computer Techniques Group in Japan), a digital reconstruction from a photograph of an unreadable car number plate; and stills from a T.V. commercial showing distorted images of a can of Coca Cola. Examples of simulation are provided, including a fascinating sequence of images simulating a journey past the planet Saturn (the work of Dr. James Blinn for NASA).

Irritatingly there is no index, nor a list of the illustrations. Thus, for example, a reference to a still life by Turner Whitted which is likened to Magritte's pipe, is very difficult to find. The book is a rather awkward shape to flip through quickly, but lends itself to a gentle perusal of the book. The whole book at leisure. Although primarily it is a beautifully produced picture book, the text is generally well written and relates well to the illustrations. For the lover of beautiful books or for the computer graphics freak, a really good book, and very reasonably priced too.

Sue Trahair
Frankston College of TAFE


Eurographics Conferences are now held annually by the European Association for Computer Graphics; Eurographics 84 marked the fifth anniversary of the Association.

The 36 papers in the Proceedings cover a wide range of topics: data capture, applications, geometric modelling, art and animation, visualization, tools, algorithms, education, user interface, business graphics and presentation. In addition, three papers from Japanese authors address computer graphic developments and applications in Japan. It is unfortunate that the opening address by van Dam on the Conference theme "Computer Graphics: A Tool for Innovation" was not included in the Proceedings.

Most of the papers describe research and development activities at a variety of European Centres; twelve papers have authors from North America and Japan. The papers are evenly divided between solutions to application problems and computer graphics tools and techniques.

Two of the invited papers are of general interest. Badler and Carlbom outline the computer graphics scene in the U.S., commenting on graphics systems (PCs, minis, workstations, supercomputers) and software (standards, interfaces, 3D modelling techniques, animation).

Marrchal and Mattys express concern that the European graphics community is not collaborating sufficiently to transfer their research to commercial products. They cite the story of GKS, which started in Europe, but which has been commercialized largely by North American and Japanese companies. They see Eurographics, Standards Organizations and European funding agencies (e.g. ESPRIT) as mechanisms for technology transfer and for maintaining European opportunities to use existing European systems and software.

Eurographics Conferences have now reached the status where their Proceedings contain useful descriptions of computer graphics activities of interest to people in the field. However, the delays in production and the costs of production in book form are likely to restrict Eurographics Proceedings to computing libraries.

Eurographics '85 was held in September, Nice, France.

J.F. O'Callaghan
CSIRO/NET, Canberra

BOOK REVIEWS


This lovely coffee-table-type book will be of interest to, and give pleasure to, everyone. The book is divided into nine sections with headings such as Modelling Techniques, Design and Industry, Art, Television and Feature Films. Each section is profusely illustrated in colour, and each illustration has a brief, but useful, summary of how and where it was produced.

Among other things, the book is concerned with the ability of computer graphics to enhance the creativity of people who work in industry, engineering, art and science. It is also about the people who work in computer graphics, with their machines and amazing techniques, enabling us to see impossible things.

The Australian Computer Journal, Vol. 18, No. 1, February 1986

I.F. O'Callaghan
CSIRO/NET, Canberra

35

Despite a previous edition of this book having been recently reviewed (see Burn, 1985) and that there are no obvious changes, I feel some further comments on it are justified.

It is still "very good value" being well thought out, easy to read and (oh joy!) nearly completely error-free. And so I have no hesitation in insisting that all Fortran programmers read it.

But what about all the non-Fortran programmers in our world? Serious optimization problems occur with all languages and much of the available computing resources at our disposal are wasted by poor code in languages other than Fortran. It seems a shame to try and educate Fortran devotees and leave the rest to muddle along.

And so I would go further. Any serious programmer who knows enough Fortran should read this book irrespective of which language he uses in practice. Although many points in the text are Fortran specific, we can all learn much of value by adopting the principles which are espoused.

That is very unacademic! Many of Metcalf's suggestions run counter to the divine edicts of structured programming; requiring us to write peculiar code to help compilers for example. But what is the point of insisting on style if it causes unacceptable system performance and user frustration? I am sure that many Pascal and Ada programmers would benefit from this work (at least until gigaflop machines with intelligent compilers are available at "Woolies").

Reference


This is a teach-yourself BASIC book aimed at the large groups of students in our high schools with access to BBC micros. Each one of the eighteen lessons is divided into four sections: a statement of the aim of the lesson, followed by the lesson proper and a summary. A housekeeping section introduces commands or techniques which are essential but which would not fit into the lesson material. Exercises end each lesson and most answers to the exercises are provided in Appendix I.

After an explanation of the keyboard in lesson #1 the author introduces the graphics commands in lesson #2. The heavy emphasis on graphics in this text book is a deliberate teaching strategy to provide the student with immediate feedback on his or her progress.

In order to appeal to the wider audience who find the traditional syntax-oriented approach to the teaching of computer languages too daunting and mathematical formulae intimidating, the syntax is introduced incidentally and not many formulae appear in the text.

The text abounds with sample programs, some of which lack comments. Good programming style is emphasized through proper indentation in FOR-loops, REPEAT-UNTIL loops and IF-THEN-ELSE-statements. Variables are consistently printed in lower case and keywords, in upper case.

The author shows a distinct preference for the REPEAT-UNTIL loop with its attendant problems of not being able to cope with boundary conditions, such as empty files. (Either this, or BBC BASIC simply does not support the DO-WHILE loop).

A minor irritation in the sample programs is the inconsistent initialization of all numeric variables to zero and string variables to null regardless of whether the variables are used later for summing (or concatenation) or not. There does not seem to be a syntactic requirement to initialize all variables, but if there is, then all complete programs have to show the initialization.

A more significant criticism of this introductory text is the lack of emphasis on the design phase and testing. Not one design methodology is presented, suggested or used. Most searching and sorting algorithms are not well explained; perhaps students are expected to copy the algorithms from the book rather than try to understand the logic behind the algorithms.

The underline character, which is a valid character in identifiers, is set as a long hyphen, while the minus sign is printed sometimes as a short hyphen or a long one. More attention could also have been paid to proof-reading.

BBC BASIC is a Pascal-like superset of the "normal" BASIC available on other micros as the REPEAT-UNTIL loop is supported along with procedures communicating via parameter lists, and local variables can be declared in procedures.

Overall, at S14.95, this is a good text for the training of future computer users. Teachers using it as a text book for the teaching of introductory computer programming will have to supply their own material for the topics of design, desk checking, run-time testing and internal documentation.

Dominic Wild Perth Technical College


This book is aimed at the popular market. No time is wasted on explaining what a computer is or how and why it works. After being instructed to plug in and turn on, the action is on! The author sees "no reason why you should give yourself a headache trying to fathom the mysteries of binary code and silicon chips".

All steps to starting programming for the complete computer novice are given (including plugging the machine in and turning it on). One wonders if you are unable to do these things for yourself how you are going to manage to write programs? The student is encouraged to sit down at the keyboard and start typing straight away. The first program to be written prints I AM A COMPUTER I WILL OBEY YOU which gives some idea of what sort of a book this is!

The book is very wordy – there is a lot of information presented, but there is no logical separation of topics and few paragraph headings. For instance, there is a heading LISTing your Program, but not one on RUNNING your Program: TAB and the semi-colon in a PRINT statement are mentioned, but the comma is ignored. BASIC language words are all printed in upper case typeface, which makes them easy to find. However, in general, the setting out of instructions and the order in which words are introduced, is very disorganised and hard to follow. The contents page lists the topics of each chapter but gives no page numbers! There is no index of contents. Any beginning student of programming could very easily get lost.

As new $BASIC words are introduced, various programs are printed in full, supposedly to give examples of previous topics. In general I found these programs confusing and often pointless or trivial. Many of the programs are silly games (eg. "A Word Guessing Game" or "A Fake Astrological Program" etc.).

On page 45 the first flowchart appears, along with the statement "It helps to draw a flowchart before you start devising a program." However the author doesn't seem to draw them before HIS programs, so what is the reader to believe?

There is no attempt to introduce style or structured programming design. There is no mention of logic. In fact this is a book written just as a beginner would have written it!

The title of this book is obviously an attempt to capture the novice who may associate programming with maths. In fact the book refers to maths no less nor any more than many other similar but better books.

The title is not referred to in the main body of the book.

One advantage of this book over some others with similar aims, is that it is written so that it can be read by users of a number of different micros (Apple II, BBC, TRS-80, IBM PC etc.). However, since the book is so poorly presented the beginner will find other, more general texts, much easier to understand.

Definitely not recommended. Go out and buy a good book like LEARNING BASIC FAST by De Rossi.

Sue Trahair
Frankston College of T.A.F.E.

The Australian Computer Journal, Vol. 18, No. 1, February 1986

No adequate discussion of programming methodology can be successfully decoupled from linguistic concepts: the lack of specification of various pseudo-codes and “algorithm design” notations and their consequent lack of utility in an environment which demands increasing levels of mathematical precision testifies to this. Correspondingly, the presentation of programming language material should always proceed with reference to some programming methodology, for of what use is a programming language other than as the vehicle for expression of a methodology? The lack of adherence to this pedagogy in Traister’s book leads us to raise immediate and profound reservations regarding its net usefulness to humanity.

Further examination leads to the impression that the book is aimed at persons who know only BASIC and who wish, with naive sincerity, to better themselves. It would seem that such an audience is most desperately in need of some elementary education in programming methodology. This dose of Traister would conceivably have the effect of transforming bad BASIC programmers into bad C programmers. Instead of wreaking havoc with TRS-80’s and Apple II’s, they would be able to have their way with VAX’s, Pyramid’s and the like.

In terms of the competence with which the book achieves its goals, the job appears to be well done. However, this appraisal is given rather in the spirit into which one would be forced in applauding the engineering of a jumbo jet out of brown paper, glue and wire. The book may have some value in exposing BASIC troglodytes to a wider world, but we can all imagine better world views than from an exclusive BASIC-C perspective.

This reviewer tends to regard the book as computer science pornography. In the hands of the mature student of programming languages, there are some harmless hints as to the relationships between the two languages. However, the immature reader, if not already intellectually crippled by BASIC, will likely be hopelessly perverted.

Paul A. Bailes  
Griffith University


This is a good concise book summarizing the main properties of the FORTH computer language. It would make a good text on the language.

FORTH is growing in popularity. It’s very much a programmer’s language. For a long time few universities presented courses in FORTH. Books useful as texts therefore are long overdue.

This is a translation by M.J. Stewart of the book of the same published in French by Editions Eyrolles in 1983. The authors are systems engineers. It covers all the features of the language (FORTH shares many of its features with threaded interpretive languages in general), beginning with a short preface, leading on from there to a discussion of the main stack manipulation words and programming structures, the workings of the FORTH semi-interpreter, semi-compiler (a brilliant structural innovation FORTH shares with other threaded languages), compiling and definition words, and vocabularies.

By now there are many books on FORTH. Among them are Leo Brodie’s *STARTING FORTH* and Kevin McCabe’s *FORTH FUNDAMENTALS I, II*. Unlike these, the book by Salman, Tisserand and Toulot contains an extensive set of exercises (with answers for some of them). It is also much more concise, yet goes more deeply into the language on many points. It contains one of the best treatments of FORTH vocabularies I have seen.

On the other hand, its conciseness makes it read less easily than (for instance) the book by Brodie. The latter, however, has a cuteness some may find cloying, so these matters are questions of taste.

On one question, that of recursion, I wish that the authors had discussed the use of EXECUTE to implement recursion with the address of a routine on the stack, rather than the popular use of SMUDGE. The difficulty with SMUDGE is that it only allows a word to call itself, rather than allowing more complex sequences such as WORD1 which calls WORD2 which calls WORD1. Constructions using EXECUTE are much more general.

The price is good, $A19.00. It’s certainly worth the money and I’d be willing to buy it for myself.

Thomas Donaldson  
Guiltech Research Corporation, Sunnyvale, California


FORTH is a programming language which is easy to implement, has compact code and a fast execution speed. I wish that I had discovered this book a few months ago, when I was contemplating implementing a FORTH-like language on my microcomputer. Indeed, I can agree with the author that by the time readers of this book have read to the end, they should be in a position to implement FORTH for themselves.

Emery introduces FORTH in the best way that I have yet seen in a FORTH text-book, from the perspective of postfix arithmetic, or reverse Polish notation. This explains the title of his first chapter, *HSILOP*. The second chapter gives a thorough overview of the basic concepts of FORTH, variables, constants, words, operators, and the control elements. Examples from real FORTH systems (POLYFORTH and FORTH-83 specifically) are used to good effect in Chapter Three where the (rather crude) FORTH file system is explained.

The first substantial hint of the underlying details of FORTH are revealed in the FORTH (or was that fourth?) chapter, as the dictionary structure is elucidated. A concise explanation of threaded code, the threaded code interpreter, and the vocabulary mechanism, is given.

The compilation and execution of FORTH words is among the topics addressed in the final chapter, and it is here that the bulk of the implementation information resides. It is unfortunate that FORTH is not a recursive language, despite its strong stack orientation. An ingenious solution is offered in this chapter which partially overcomes this restriction.

Exercises are scattered throughout the book. These would be helpful to beginners in FORTH, but those with a computer science background could skip them without serious consequences. The answers (given in the back of the book) to the few exercises that I attempted seemed to be correct.

Whether you are interested in learning how to program in FORTH, or desire to implement your own FORTH system, this is a suitable book as far as price, conciseness and breadth of coverage go. It is inexpensive, has an excellent glossary, is up-to-date, and is strong on implementation issues.

P.C. Brebner  
University of New South Wales


The intent of this book is to teach students “how to design non-trivial programs that are correct, reliable, robust, and reasonably efficient”. The book starts out by providing a summary of the syntax of the PASCAL language. The summary is only brief and not sufficient on its own to allow students to gain a mastery of the language.

After the introduction to PASCAL, attention is switched to the primary focus of the book – that of program design. The program development strategy advocated is built around the design of algorithms, the design of data structures and the design of input/output requirements. To convey some of the more important practical aspects of program design a well-chosen set of examples are considered. In the discussion of examples good use of data structures is made. Most students who work through this material in the way advocated could be expected to derive valuable practical programming experience.

Some other more specific comments about the book are:

- The use of procedures to give structure to a solution is sometimes undone.
- The “state” approach to exception handling as advocated by Atkinson (1984) is used fairly consistently throughout the text. One exception is the clumsy handling of the solution to the problem of deleting trailing blanks from a line of text (see pp.107-8).
With regard to the claim about the design of correct programs it is usually expected that some consideration of formal correctness arguments is needed for this to be possible. No attention has been paid to these requirements.

In summary, the book provides what it says, a good practical introduction to the design of non-trivial programs. The question of whether training in program design that ignores the use of formal specifications and constructive proof techniques is adequate, remains open.

References


Consider some of the ways of evaluating the properties of the programming language ADA: one way is to consult the primary documents (which we might call semantics by specification); a second way is to execute programs with a validated compiler (which we might call semantics by experiment); a third way is to consult secondary textbook sources (which we might call semantics by anecdote). We are all aware that obtaining a language understanding (let alone a language appraisal) by these means is a slow and painful process. This book is the result of a study group set up during 1980 under the auspices of Actel (Association Francaise pour la Cybernetique Economique et Technique). They evaluated the language by working from the draft Reference Manual and by consultation with the implementation team led by Jean Ichbiah (your sources can’t be any more primary than that). Their study was completed by the end of 1981, and was influential to some extent in changes to the language incorporated in the final standard. This English translation has the original study systematically updated to take account of these changes.

The structure of the book is that of a number of chapters which address different aspects of the language (declarations and types, numerical names and expressions, sequential control structures, modularity, scope and visibility, tasks, exceptions, generic units, separate compilation, adaptation of programs, input/output, and syntactic, lexical and textual elements). Each chapter has a different set of authors. No great differences in style were noted by the reviewer, perhaps due in part because of the translation from French to English by a single translator.

Each chapter has much the same structure. The author(s) assume that the reader is already familiar with ADA and has a reasonably sophisticated grasp of programming language concepts (the book is written by and for computer scientists). For each chapter/topic the basics are carefully analysed and the consequences of the design of the language topic are established by a series of elegant examples. The chapter generally ends with an appraisal, both complimentary and critical, by the author(s). The discussion and analysis of the topic has, of course, given the reader a detailed understanding of the topic and the reader is then in a position to appraise ADA from the point of his/her application. There is a great deal in the book to be learned by those of us whose knowledge of ADA is essentially restricted to secondary sources. For instance, more things can go wrong with the use statement than I had ever realised, and the (unpleasant) interactions between the task concept and the exception concept are spelled out.

In the end the authors are enthusiastic about the language. The reader gains the none-too-subtle feeling that they see into their study expecting the worst, and in the end gave credit where credit was due. I will merely quote one comment: “Because of the programming style and the discipline it imposes, ADA is seen to be a language for professionals, whose skills it both respects and enhances.”

The authors appear to have a healthy disdain of amateurs in computer science.

This book is a very valuable addition to the literature of ADA in particular and to programming languages in general. It provides a model for the analysis and appraisal of a large language, and makes the reader wish for equivalent studies of other languages. It is a must for all computer science departmental libraries, and is an important source book for comparative programming language courses.

References


Consider some of the ways of evaluating the properties of the programming language ADA: one way is to consult the primary documents (which we might call semantics by specification); a second way is to execute programs with a validated compiler (which we might call semantics by experiment); a third way is to consult secondary textbook sources (which we might call semantics by anecdote). We are all aware that obtaining a language understanding (let alone a language appraisal) by these means is a slow and painful process. This book is the result of a study group set up during 1980 under the auspices of Actel (Association Francaise pour la Cybernetique Economique et Technique). They evaluated the language by working from the draft Reference Manual and by consultation with the implementation team led by Jean Ichbiah (your sources can’t be any more primary than that). Their study was completed by the end of 1981, and was influential to some extent in changes to the language incorporated in the final standard. This English translation has the original study systematically updated to take account of these changes.

The structure of the book is that of a number of chapters which address different aspects of the language (declarations and types, numerical names and expressions, sequential control structures, modularity, scope and visibility, tasks, exceptions, generic units, separate compilation, adaptation of programs, input/output, and syntactic, lexical and textual elements). Each chapter has a different set of authors. No great differences in style were noted by the reviewer, perhaps due in part because of the translation from French to English by a single translator.

Each chapter has much the same structure. The author(s) assume that the reader is already familiar with ADA and has a reasonably sophisticated grasp of programming language concepts (the book is written by and for computer scientists). For each chapter/topic the basics are carefully analysed and the consequences of the design of the language topic are established by a series of elegant examples. The chapter generally ends with an appraisal, both complimentary and critical, by the author(s). The discussion and analysis of the topic has, of course, given the reader a detailed understanding of the topic and the reader is then in a position to appraise ADA from the point of his/her application. There is a great deal in the book to be learned by those of us whose knowledge of ADA is essentially restricted to secondary sources. For instance, more things can go wrong with the use statement than I had ever realised, and the (unpleasant) interactions between the task concept and the exception concept are spelled out.

In the end the authors are enthusiastic about the language. The reader gains the none-too-subtle feeling that they set out on their study expecting the worst, and in the end gave credit where credit was due. I will merely quote one comment: “Because of the programming style and the discipline it imposes, ADA is seen to be a language for professionals, whose skills it both respects and enhances.”

The authors appear to have a healthy disdain of amateurs in computer science.

This book is a very valuable addition to the literature of ADA in particular and to programming languages in general. It provides a model for the analysis and appraisal of a large language, and makes the reader wish for equivalent studies of other languages. It is a must for all computer science departmental libraries, and is an important source book for comparative programming language courses.

References


This volume results from a “Symposium on Software Validation” held in West Germany, September 1983. The theme of the symposium was inspection, testing, verification and alternatives. These all refer to various methods for increasing confidence in software reliability or correctness. Coverage of this broad area is achieved by various papers from management of software construction through to proposals for techniques for software specification, and formal verification. The latter area is covered descriptively, there being little heavy mathematically based notation or methodologies present.

Under the heading “Validation by inspection” are three papers on the benefits of small peer group review of software specification and coding. Emphasis is placed on the formal nature of this inspection process with example forms and printouts which are used and
produced. R.D. Buck and J.N. Dobbins also reveal their "natural numbers of programming"; these being 8 to 12 defects per 1000 lines of code and 3 to 5 man hours expended on a major defect. The assumption underlying testing is succinctly described as reducing software construction costs by finding defects early in the process.

In the section on testing an interesting paper entitled "Introduction to the Formal Treatment of Testing" by J.S. Gourlay presents a definition of a testing method, as a relation between sets of specifications and sets of programs. This relation has a natural ordering corresponding to more powerful methods. Two other papers describe toolkits which enhance or enable methodical testing strategies. A claim made in the paper by S.H. Salt is that use of one of these toolkits saves 20 person years on a large project.

Rather than testing a program on example data, symbolic evaluation can be used to "run" the program on symbolic data. Such a method is feasible in the presence of an automatic algebraic simplifier and theorem prover to discover whether various paths in a program are executable. The paper by L.A. Clark and D.J. Richardson is a good introduction to symbolic evaluation.

The section headed "Validation by formal verification" contains papers dominated by formal specification issues. In the case of the Aura specification language for Ada, described by D.C. Luckham, specifications may be best described as formalised comments. These can provide information on relationships that should hold within the program, and an expressed goal is to build a theorem prover which will check that these relationships hold. Another paper by M. Moriconi and A.I. Lanksy treat a graphical representation of specifications, which still has a formal interpretation.

The issue of automatic program construction from specification is not ignored, even though this is a research topic in its infancy so far as practical systems are concerned. A. Goldberg and G. Kotik write about a wide spectrum language called V. This enables specifications at a high level involving set-theoretic data type constructors, for example, to be transformed to a more efficient algorithm. The two major aspects of this treated in the paper are data type refinement and contrast structure refinement. If such refinement can be shown not to introduce errors such a technique would be invaluable.

Clearly the articles in this volume are diverse, but they seem to cover examples of many important ideas in the area of software validation. Rather too few pages in the book relate to the various philosophies involved; even so they contribute to making it a volume well worth browsing through.


This is the proceedings of the IFIP TC2 Working Conference on PDE Software: Modules, Interfaces and Systems held at Soderkoping, Sweden in August, 1983. It was the third conference organized by the IFIP Working Group 2.5 (Numerical Software). The previous conferences involved; even so they contribute to making it a volume well worth browsing through.

Rather too few pages in the book relate to the various philosophies involved; even so they contribute to making it a volume well worth browsing through.

The section headed "Validation by formal verification" contains papers dominated by formal specification issues. In the case of the Aura specification language for Ada, described by D.C. Luckham, specifications may be best described as formalised comments. These can provide information on relationships that should hold within the program, and an expressed goal is to build a theorem prover which will check that these relationships hold. Another paper by M. Moriconi and A.I. Lanksy treat a graphical representation of specifications, which still has a formal interpretation.

The issue of automatic program construction from specification is not ignored, even though this is a research topic in its infancy so far as practical systems are concerned. A. Goldberg and G. Kotik write about a wide spectrum language called V. This enables specifications at a high level involving set-theoretic data type constructors, for example, to be transformed to a more efficient algorithm. The two major aspects of this treated in the paper are data type refinement and contrast structure refinement. If such refinement can be shown not to introduce errors such a technique would be invaluable.

Clearly the articles in this volume are diverse, but they seem to cover examples of many important ideas in the area of software validation. Rather too few pages in the book relate to the various philosophies involved; even so they contribute to making it a volume well worth browsing through.


This is the proceedings of the IFIP TC2 Working Conference on PDE Software: Modules, Interfaces and Systems held at Soderkoping, Sweden in August, 1983. It was the third conference organized by the IFIP Working Group 2.5 (Numerical Software). The previous conferences have dealt with performance evaluation of numerical software (1979) and the relationship between numerical computation and programming languages (1981).

The papers presented in the previous conferences have been of high quality and these proceedings are no exception. Also the papers in these proceedings would be of interest not only to researchers in the area of numerical solution of partial differential equations (PDEs) but also to researchers interested in other areas of mathematical software since most papers deal with issues that are relevant to solving PDEs as well as solving other mathematical problems.

Most of the papers included in the book deal with the following four topics related to solving PDEs:

1. PDE Software Design — Designing software for solving PDEs is perhaps the most challenging task in scientific computing. Several papers in this book discuss issues involved in designing large mathematical software for solving PDEs that is to be used at different sites on different machines. For example, Chen et al. discuss design of MARS simulator, a 130,000 line code for simulating petroleum reservoirs and discuss how the software was developed and how it is maintained for the IBM and CRAY computers. Biorstat discusses

2. Solution of Linear Equations — A substantial proportion of CPU time used by most PDE software is spent in solving linear equations. Reid discusses a FORTRAN package for solving large sets of linear equations while Young and Kincaid discuss the well known ITPACK package for solving linear equations using iterative methods.

3. Language Issues — Since most mathematical software continues to be written in FORTRAN, new features planned for the language are of interest to the designers of such software. Papers by J.K. Reid and B. Smith discuss features planned for inclusion in FORTRAN 8X. These include array expressions and assignments and data abstraction mechanisms.

4. Vectorization of PDE Software for use on Supercomputers — Solution of PDEs that model real life phenomena often require a very large amount of computer time and it is therefore not surprising to note that much PDE software is run on supercomputers. Papers by Hemker et al., Schonauer et al., Bossavit, Houstis et al. and Duff discuss some of the difficulties involved in vectorizing PDE software.

In summary, most of the papers in these proceedings present aspects of modern approach to building PDE software. The modern approach involves building modular packages with interfaces carefully defined so that maintenance and modification of the software is simplified. Any researcher interested in building PDE software should acquire a copy of this book and anyone interested in numerical software would benefit by reading it.

Dr. C.K. Gupta
Monash University


The solution of elliptic boundary value problems has often played a key role in the modelling of physical systems of interest to the scientific and engineering communities. Numerical schemes to solve such problems almost invariably require the full power that large-scale scientific computing offers, and have frequently prompted advances in theoretical procedures and computer architectures.

This proceedings volume contains the proceedings of the Elliptic Problem Solvers Conference held at the United States Naval Postgraduate School in Monterey in 1983. Conference sessions focused on many aspects of the solution of linear and nonlinear elliptic problems, including software packages, vector and parallel processing, iterative solution of equations, finite element and multigrid methods, and advances in modelling and physical applications. Papers in each of these sections have been contributed by authors foremost in their field, and provide a comprehensive summary of recent, developments together with a detailed account of several new innovations in both computer software and architecture.

Contributions to the section concerned with the use of computational techniques in modelling physical phenomena explore a variety of discretization procedures used to solve the nonlinear systems which arise in modelling semiconductor devices, the Rayleigh-Benard convection cell in fluid flow and combustion devices. Readers involved in the field of numerical modelling will find the case studies in this volume of considerable interest.

In recent years, many attempts have been made to develop finite element and multi grid software which offers adaptive local grid refinement. The section of this book concerned with these issues provides a very useful account of progress being made in some specialized fields of application, and highlights some of the problems which remain to be resolved. Acceleration of the iterative solution of the large sparse systems of algebraic equations which result from finite element and finite difference procedures are essential to the efficient

The Australian Computer Journal, Vol. 18, No. 1, February 1986

SESM '80, a finite element program that involved 130 man-years of work to build and is even larger than MARS. Rice discusses how very high level elliptic problem solvers could be built from a collection of software parts. Machura discusses issues that need to be considered when designing general purpose software for PDEs. The authors emphasize the need for PDE software to be modular and designed so that new techniques when available may be easily incorporated in the software.

3. Language Issues — Since most mathematical software continues to be written in FORTRAN, new features planned for the language are of interest to the designers of such software. Papers by J.K. Reid and B. Smith discuss features planned for inclusion in FORTRAN 8X. These include array expressions and assignments and data abstraction mechanisms.

4. Vectorization of PDE Software for use on Supercomputers — Solution of PDEs that model real life phenomena often require a very large amount of computer time and it is therefore not surprising to note that much PDE software is run on supercomputers. Papers by Hemker et al., Schonauer et al., Bossavit, Houstis et al. and Duff discuss some of the difficulties involved in vectorizing PDE software.

In summary, most of the papers in these proceedings present aspects of modern approach to building PDE software. The modern approach involves building modular packages with interfaces carefully defined so that maintenance and modification of the software is simplified. Any researcher interested in building PDE software should acquire a copy of this book and anyone interested in numerical software would benefit by reading it.

Dr. C.K. Gupta
Monash University


The solution of elliptic boundary value problems has often played a key role in the modelling of physical systems of interest to the scientific and engineering communities. Numerical schemes to solve such problems almost invariably require the full power that large-scale scientific computing offers, and have frequently prompted advances in theoretical procedures and computer architectures.

This proceedings volume contains the proceedings of the Elliptic Problem Solvers Conference held at the United States Naval Postgraduate School in Monterey in 1983. Conference sessions focused on many aspects of the solution of linear and nonlinear elliptic problems, including software packages, vector and parallel processing, iterative solution of equations, finite element and multigrid methods, and advances in modelling and physical applications. Papers in each of these sections have been contributed by authors foremost in their field, and provide a comprehensive summary of recent, developments together with a detailed account of several new innovations in both computer software and architecture.

Contributions to the section concerned with the use of computational techniques in modelling physical phenomena explore a variety of discretization procedures used to solve the nonlinear systems which arise in modelling semiconductor devices, the Rayleigh-Benard convection cell in fluid flow and combustion devices. Readers involved in the field of numerical modelling will find the case studies in this volume of considerable interest.

In recent years, many attempts have been made to develop finite element and multi grid software which offers adaptive local grid refinement. The section of this book concerned with these issues provides a very useful account of progress being made in some specialized fields of application, and highlights some of the problems which remain to be resolved. Acceleration of the iterative solution of the large sparse systems of algebraic equations which result from finite element and finite difference procedures are essential to the efficient

The Australian Computer Journal, Vol. 18, No. 1, February 1986
solution of elliptic systems by these methods. This volume contains a
survey of recent results in this area together with suggestions on new
approaches which are likely to prove computationally efficient as array
and parallel processors continue development.

The discussion of available software for the solution of elliptic
boundary value problems (including the widely available packages
ELLPACK, MODULEF and ITPACK) will prove especially valuable to
readers engaged in solving problems of this nature as the summary of
the capabilities of each package is accompanied by a statement of
design aims and the areas of future development of the software. The
versatility of modern software for the solution of elliptic problems
may surprise many readers.

This volume would be a valuable acquisition for any professional in
areas involving computational techniques and their application in a
scientific or engineering setting, and is available at very reasonable cost.
The articles included have been contributed by authors who have made
a significant and sustained contribution in their field, and cover many
aspects of the solution of elliptic systems. Much information provided in
this book will be of immediate use to workers in this field, while some
sections serve to give an insight into the nature of recent developments
and likely directions of progress in the near future.

H. Connell
Royal Melbourne Institute of Technology


This book studies functional equations, in particular ordinary differential
equations and integral equations, with the aim of producing algorithms
to compute high accuracy lower and upper bounds for the solution.
The algorithms are generally of the form: find an appropriate solution
by any technique you care to use and then use iterative residual correc­
tion to converge to an accurate solution. Thus, the main theoretical
tools used are a variety of nonstandard fixed-point theorems of
functional analysis.

However, because the aim is to produce algorithms that guarantee
existence, uniqueness, and appropriate computable bounds, the imple­
mentation of these ideas relies heavily on a theory of rounding (approxi­
mation) in function spaces which can be implemented. Thus, a large
portion of the book discusses aspects of Taylor series, Chebyshev
polynomials, splines, etc., which, when implemented, require a tie-in
with the ideas of interval arithmetic.

Consider the 2-point boundary value problem

\[ 4y^{(4)} - y = 0.5 + 0.5x \]
\[ y(-1) = 0 \]
\[ y(1) = 1 \]

on the interval \([-1, 1]\). In this book, a solution of this problem is defined
_to be a pair of computable functions, say \(l(x)\) and \(u(x)\), such that the
true solution \(y(x)\) exists, is unique, and satisfies \(l(x) \leq y(x) \leq u(x)\),
for all \(x\) in \([-1, 1]\). The key point in this book is that the
computation itself should validate the existence and uniqueness of the
solution for each particular example.

For this example, a simple approach might be to choose \(l(x)\) and \(u(x)\)
from the class of 12th degree polynomials in \(x\), with coefficients of
type real, with the proviso that any evaluation of \(l(x)\) and \(u(x)\) must use
appropriate rounding operations. With this choice, the authors compute a solution using a floating point arithmetic with a 12 decimal
digit mantissa. For this solution, \(l(x)\) and \(u(x)\) differ, for all \(x\) by at most
4 in the last decimal digit. For example, \(l(0.5)\) and \(u(0.5)\) differ by
2.18243676220 \times 10^{-1}\) respectively, while \(l(-0.4)\) and \(u(-0.4)\) take the
values 1.10852360497 \times 10^{-1}\) respectively. The
arithmetich used is an interval arithmetic with precise inner product.

Thus, in this example, the user is guaranteed that a unique solution
exists, is bounded below by \(l(x)\) for each \(x\), and is bounded above by
\(u(x)\) for each \(x\). Moreover, the difference \(u(x) - l(x)\) is a
precise measure of the computational uncertainty in the solution. Thus, the user can
accept the results with a certain knowledge of their meaning, without
having to resort to re-runs, side calculations and the like.

For the approach used in the above example, you will observe that

the approximation space, viz. truncated Taylor series, is chosen a priori
without any knowledge of the solution. Thus, it is clearly possible that
such a space may be (numerically) inappropriate for the problem at hand.
One of the more interesting aspects of this book is the discussion of
algorithms in which the approximation space is generated implicitly
as the algorithm proceeds.

Overall, this is an interesting book but not one that I would
recommend to the casual reader. It is a research monograph in a
technically (mathematically) nontrivial area.

D.J. Clements
University of New South Wales

CHRISTIE, L.G. and CHRISTIE, J. (1985): The Encyclopaedia of Micro­
computer Terminology, George Allen & Unwin Australia Pty. Ltd.,
North Sydney, 352 pp., $12.95 (paperback).

This is the sort of reference book one would like to have on hand when
trying to decipher a typical microcomputer manual: More than 4000
microcomputer technical jargon terms are defined in dictionary style,
and there is very thorough cross-referencing.

Despite an obvious attempt to cater for users with a minimal
background in electronics, many of the definitions in this area assume
some knowledge. An example is the electronic definition of ‘field’,
which includes the sentence: ‘Electrical fields can induce current in
other conductors such as a transformer and can transmit interference.’
I can see some people scuttling for definitions of ‘induce’, ‘current’,
‘conductor’, ‘transformer’, ‘transmit’ and ‘interference’ (all present!) and
even then needing further explanation in order to understand the
concept of a field.

There is actually a preponderance of electronic terms defined, but
this is not surprising, considering John Christie’s 20 years of experience
in electronics. Also well-covered are software, operations, hardware,
etc., and current technology such as videodisc and robotics.

At the end of the book are a number of appendices, dealing with
specialized topics such as logic functions and symbols, music synthesis
and number systems. There is even a list of BASIC keyword definitions,
although I cannot see the point, as the owner’s BASIC manual should
obviously be preferred. A word-processing glossary also seems out of
place in a book of this kind.

The book is most suited (and well-priced) for the microcomputer
hobbyist, rather than the student, business person, or teacher it also
claims to cater for. Its biggest problem is that it will lose credibility in
future years unless kept up to date (although some of its terms are
already out of date – see paper tape!).

S. Lichtenstein
Royal Melbourne Institute of Technology

Publications 173 pp., $24.95 (paperback).

This is a disappointing book. The aim of the author is “to provide the
reader with a concise text describing microprocessor architecture,
microcomputer design, interfacing and assembly language
programming”. The book is certainly concise, but at the expense of
failing to address many important topics — for instance subroutines are
mentioned only in passing, and certainly no mention is made of
subroutine linkage. The use of the stack is mentioned very briefly in
Chapter 1, at which stage the reader has not been confronted with
instructions or even registers!

The author is to be commended for selecting the Motorola 6809 as a
sample machine for teaching computer architecture, as it has most of
the features found on larger machines. However the machine is
described by a barrage of facts, rather than a coherent discourse on
computer architecture. The author is also to be commended for
discussing "structured programming" (albeit without the use of
_procedures), using Pascal. Having done so, he then fails to comment his
sample programs to take advantage of the Pascal program_description:
the Pascal should appear, properly indented, in the comment field to
highlight the structure of the program. The author also fails to show
how data structures, as declared and used in Pascal, can be
implemented in Assembler.

The Australian Computer Journal, Vol. 18, No. 1, February 1986
Save for a couple of good chapters on interfacing and analogue-to-digital conversion, the book reads more like a collection of data sheets. It certainly does not teach good programming practice, and one wonders if just as much information could be gleaned by poring over data sheets.

P.G. McCrea
University of New South Wales


This book is intended to constitute the material of an elementary course introducing students in computer engineering to the structures and mechanisms of operating systems. The emphasis is on the organisation and practical aspects of operating systems, rather than theoretical considerations. There are literally thousands of introductory operating system textbooks and in the opinion of the reviewer this one does not stand out in the crowd.

The author does indicate in the preface that the book is not intended as a stand-alone course, but rather as lecture notes offering detailed information to help students. In fact the general terse style makes the book difficult reading and this, combined with the poorly typeset diagrams, makes the book quite unsuitable as an introductory text.

Despite the above comments the book would be quite useful as a reference text, both for lecturers responsible for an operating system course and for students. The material presented is quite up-to-date and includes an excellent bibliography at the end of the text. Each chapter concludes with suggested further reading classified by sub-topic as well as a list of references.

A good feature of the book is the inclusion of two topics often only briefly mentioned in operating system texts. These are the relationship between naming and addresses (i.e. linking), and protection and sharing. The chapter on linking covers all of the essential issues from static link editors through to dynamic linking. It also introduces some of the additional problems in distributed systems. The chapter on protection includes a motivational section on why protection is necessary. Such motivation is essential in introductory texts. Topics covered include encryption techniques and capability-based addressing.

An excellent list of references is provided.

The book would be enhanced by the provision of one or more complete case studies of real operating systems. Although reference is made throughout the text to a number of well-known systems a complete case study could bring together all of the concepts and give the reader a clearer understanding of the fundamental problems of operating system design.

In general this book would make a useful addition to a reference library on operating system concepts but is not really suitable by itself as an introductory text.

J. Rosenberg
Monash University

This book is a collection of papers presented at the Seminar on Distributed Data Sharing Systems, held in Parma in 1984. The Seminar was the third in the series. Papers are presented from six streams:

1. Multidatabase Systems and Architecture (5 papers)
2. Distributed Query Processing (3 papers)
3. Distributed Data Sharing Security (2 papers)
4. Performance Evaluation (1 paper)
5. Distributed Services and Applications (2 papers)
6. Special Systems (1 paper)

Unlike the previous seminars where it was assumed that the distributed data constituted logically one database, there was a lot of interest in the Parma Seminar in multidatabase systems. Over one third of the papers were presented on this topic. There was also increased emphasis on new technologies such as high speed satellite links, local networks or videotelex networks as well as new types of host machines such as microcomputers, workstations or database machines.

The book will be useful in particular for researchers working in the distributed database area and is recommended for library acquisition.

R. Sacks-Davis
Royal Melbourne Institute of Technology

The book is edited from the proceedings of the IFIP WG 10.1 Working Conference on Methodologies for Computer System Design, North Holland, 343 pp., SUS55.50.

The book is edited from the proceedings of the IFIP WG 10.1 Working Conference on Methodologies for Computer System Design held in September, 1983. It is organised in seven separate sections of invited papers on specific topics, together with five discussion group sessions.

The main interest of the Conference was to address the difficulties inherent in designing computer systems in a coherent manner, from applications through to chip levels. A number of papers tackled this problem directly, notably Marwedel’s paper on the Minola design system and Gueth and Kriz’s paper on a pseudo language approach based on Modula 2. These and others such as Chen and Mead’s paper on VLSI clearly illustrate the mismatch between a rigorous hierarchical approach to design levels and the needed multi-level designs actually achieved.

It is the insights gained into the design processes that make this book a worthwhile addition to any system designer’s library. The discussion sessions in the book are unfortunately trite, as brilliantly summarised by G. David in the wind up presentation, so that the value as a true workshop is questionable.

F. O’Brien
N.S.W. Institute of Technology


The branch of Artificial Intelligence dedicated to understanding natural language is a complex one, which has dominated the interest of many researchers over recent decades. There have been many attempts at this task, with that used by Mellish being significant among them.

This book is a description of his approach and a brief discussion of how it fits within the traditional framework. It describes the work around 1980 undertaken by Mellish at the University of Edinburgh and forms part of the ‘Ellis Horwood series in Artificial Intelligence’.

The task of understanding a sentence has traditionally been broken into two smaller tasks: one concerned with the analysis of its syntax, i.e. its structure, and the other with its semantics or meaning. Early researchers built systems which were dominated by the treatment of these as separate phases of a single process. Many people still maintain that the entire structure of a sentence must be known before any meaning can be extracted.

A controversy has reigned for a long time within the Artificial Intelligence community as to the degree to which this syntactic analysis has to be completed prior to extracting meaning. Many researchers claim that we understand a sentence as it progresses rather than in a single burst at the end.

Two approaches to the early evaluation of a sentence’s semantics are apparent. The first is to treat each individual noun phrase as a single unit describing a set of objects and analyse each as it is encountered and to retry again later if we get more information. The other, advocated by Mellish amongst others, is to extract only partial meaning from each noun phrase and to wait until the sentence structure helps complete any doubtful features.

Some assumptions are made as to prior knowledge by the author but any specific terms are defined clearly. While the index is adequate and the bibliography is excellent, later chapters of the book rely heavily upon material in earlier chapters. It does not lend itself to browsing unless the reader already has some familiarity with incremental semantic analysis.

The book must be read as a whole if the reader is to gain a firm grasp of
of the author's approach to the understanding task.
A set of appendices are included and contain sections of PROLOG code showing how the author achieved particular ends. The discussion is presented in a manner that does not presume a knowledge of PROLOG and presents the examples clearly.

My recommendation is that this book will appeal to researchers endeavouring to keep up-to-date in this changing field, and to graduate students interested in natural language understanding. However, some time is required to read it thoroughly, rather than just to skim it.

W. Kneipp
University of New South Wales


Prolog is a language which has 'taken off' in recent years. There is considerable interest in it around the world, mainly for its use in Artificial Intelligence projects, although it does have wider applications, especially in the database area. Its style, declarative rather than imperative, is appealing, and it has an impeccable ancestry in mathematical logic.

From the point of view of a Prolog teacher, first impressions of this book on the language from Warsaw University are disappointing. Inexperienced programmers or students will find the going rather hard. In the first section are discussions of associativity, functions mapping components into objects, and cryptic allusions to LISP which might be disconcerting. The first program does not arrive for some time, and it is a procedure for splitting lists with an inexplicable name (except to LISPers) and in a non-standard notation. On page 95, the authors shift to a more common notation for lists. Even the publisher's blurb is uncharacteristically defensive, and commences: 'This book is not a primer'.

In spite of these criticisms, the book has a number of strengths. Among its virtues I would count:

- programming hints which should be useful even to readers with some experience in the language;
- a Prolog bibliography extending to 1984;
- a great deal of information on Prolog implementation, including complete listings of a Prolog interpreter for small computers written in Pascal and Prolog;
- two case studies, including complete listings. One of these implements a database query language;
- discussion of some Prolog dialects.

The writers use fluent idiomatic English, and are sufficiently modern to write 'the majority of' rather than 'most'. Perhaps a better title might have been 'Prolog for Computer Scientists' or 'Prolog for Implementers'. These audiences would obtain value for the price, which is at a normal level for these days of the devalued Australian dollar.

G.B. McMahon
University of New South Wales


Section 7(1) of Australia's Telecommunications (Interception) Act, 1977, states: "A person shall not -
(a) intercept;
(b) authorise, suffer or permit another person to intercept; or
(c) do any act or thing that will enable him or another person to intercept,
a communication passing over a telecommunications system."

Against this background, we have in Australia the report of Special Prosecutor, Ian Temby, Q.C., in relation to the so-called "Age Materials" consisting of recording tapes that allegedly contained voice material that may have arisen from "... illegal interception of telephone conversations ...".

With this background, the time is right to obtain a copy of Henry Beker and Fred Piper's new book on voice communications security or, put more basically, "How to thwart phone taps!" The authors set out their objectives as: "The aim of this book is to discuss both the techniques and principles underlying current scrambler design, and "Our aim is to provide sufficient background information on the subject of scramblers that the reader will feel he knows how to begin his selection of a piece of equipment or even how to begin to actually design a system ... we will try to evaluate and compare the various techniques which are currently in use."

The authors proceed through seven chapters to achieve their goal admirably. The chapters cover the areas of User Perspective; Speech Communication: Principles of Cryptography; Frequency/Time Domain and Two-Dimensional Scrambling and finally the emerging technology of Digital Scrambling. These chapters, over some 267 pages, provide the reader with the necessary technology to understand the basic construction of secure speech communication systems and the basic cryptographic "boxes", or "scramblers" that are used.

The incorporation of speech scramblers into a communications network can often give rise to considerations that are not just technological in nature. As the authors state:

"In any telecommunications system, whether it employs scramblers or not, the listener's perception characteristics are important. When scramblers are included in the system then it is necessary to test the intelligibility of both the received speech and the transmitted speech. Obviously we want the received speech to be understandable while the transmitted speech should have little or no intelligibility to any listener."

The authors clearly show that achieving this technological goal is not at all easy. Essentially the nature of speech has to be understood as does the transmission characteristics of the medium over which speech is to be transferred. These are then combined with cryptographic techniques to provide the scrambling system. The book covers analogue, digital and "hybrid" or combined analogue/digital techniques for scrambling. It does not, however, aim to give a complete treatment of cryptography and the reader is immediately referred by the authors to their previous book "Cipher Systems: The Protection of Communications".

This book gives an excellent introduction to the topic of voice security and is to be thoroughly recommended. The reader will have to be prepared to work through the necessary mathematics and electronic engineering, which assumes a basic knowledge of these topics. In this sense the engineering sections of this book are not "for management". However, the early chapters set the theme of security for management decisions.

The emerging world of digital systems is left to the last chapter of this book. The point is clearly made by the authors that it is in this area that the most advances are being made today with truly practical systems emerging to market. Perhaps this section should or will form the basis of a further book on digital systems particularly as the integrated digital communications system (I.S.D.N.) emerges. The book belongs on the shelf of all professionals involved in the assessment of the security of today's information systems for any enterprise, large or small.

Bill Caelli
Nerang, Queensland


Books like this should never be written. Having been written, they should never be sold or given away, only burnt. Consider the following quotation from this convoluted, twisted, irrelevant outpouring:

"Anne had been a typist for many years. When computer-based word processors were introduced into the office it seemed natural enough to transfer to be [sic] new equipment. ... But the first time she tried to use a word processor, she vomited – quite unaccountably and in circumstances of rising panic."

The author has also written books on seolology, robots, witchcraft, censorship and women's rights. Just remember the name: it's bne to avoid.

J. Lions
University of New South Wales

The Australian Computer Journal, Vol. 18, No. 1, February 1986
Information for Authors

The Australian Computer Journal, which is published quarterly by the Australian Computer Society, invites original contributions relating to the design, understanding and application of digital computers. These include research papers, tutorial and review papers, industry case studies, short communications and letters to the editor.

Each paper should comply with accepted practices for the style and organisation for scientific papers; must be in English; and must be certified by the author that it is his or her own original work, that it has not been copyrighted, published or submitted for publication elsewhere. If the work described has been sponsored, the paper must be cleared for publication by the sponsoring organisation. The author is asked to agree to assign copyright in the paper to the Australian Computer Society Incorporated. Where there is more than one author, each should sign the letter of submission.

Each paper submitted is reviewed by the editor and, if the form of submission is satisfactory, it is then evaluated for accuracy, originality and relevance by at least two referees. Revisions may be requested. Every effort is made to notify the authors promptly of the referee's comments. Revisions may be requested, e.g. for concern (e.g. a particular journal article). The writer's full title and address must be given (not necessarily for publication).

Full papers will normally be equivalent to four to six printed pages (3500 to 5500 words). Longer papers may be accepted if space permits.

Short communications (one or two printed pages) describing novel applications, new ideas, work in progress or summarising post-graduate research theses accepted for degrees are also invited. These should aim to inform readers of current work, to provide abstracts of current Australian research for overseas readers, and to provide for prompt reporting of new ideas in Computer Science.

Letters may be sent to the editor for publication. These should be typed and clearly indicate the matter for concern (e.g. a particular journal article). The writer's full title and address must be given (not necessarily for publication).

Book reviews are normally prepared at the invitation of the editor only.

Manuscript Preparation.
The guidelines given below are intended to facilitate editorial assessments. Where manuscripts are inadequately prepared, the assessment process may be prolonged unnecessarily.

Papers should be typed double spaced with 30 mm wide margins, on one side only of A4 size paper (210 mm x 297 mm). Three identical copies on good quality paper are required; the first page should contain the title, author information, an abstract and cross-referencing information. The title of the paper should characterise the contents of the paper in as short a span as possible. Author names should be given as initials and last name only—no titles or degrees should be shown. If there is more than one author, the principal author should be listed first. For each author, a sufficient postal address (possibly including the name of an employer) should be supplied. The abstract should be informative and summarise the paper as concisely as possible in about 100 words. Since an abstract may be republished separately, it should contain no uncommon acronyms, footnotes or reference citations. Cross-referencing information should consist of a list of Key Words and Phrases, and a classification of the paper within the categories published in ACM's Computing Reviews (see the January issue for 1983 or later years).

The main text should be clearly written and follow the English conventions for spelling and punctuation. Text prepared using a word processor should not be hyphenated or right-justified. Subheadings and paragraphs should be clearly shown. Major and minor sections may be numbered at the author's discretion. Footnotes should not be used. (Reference citations should conform to the Harvard style—see below.)

Unusual mathematical symbols, subscripts and mathematical formulae can be difficult to set in type. Choose symbols carefully, avoid excessive use of subscripts, plan formulae to fit within the standard column width and use forms that are convenient to set in type, e.g. \( \exp(-x) \) rather than \( e^{-x} \). Avoid using the letters 'o' and 'l' where they may be confused with the digits '0' and '1'. Extensive mathematical treatments or supporting data are often best included in one or more appendices at the end of the paper. These should be numbered if there is more than one, and have suitable headings.

Tables should be constructed to fit neatly into a single column (83 mm x 229 mm) or exceptionally, across two columns (width 178 mm), and should be numbered consecutively. Each table should have a brief explanatory heading, and column headings should be brief and clear. Camera-ready copies for each figure, diagram or flowchart should be supplied. Drawings should be about twice the expected finished size (single column width is 83 mm, and double, 178 mm.). Legends and labels should be clearly drawn, positioned suitably and large enough to remain legible after reduction. Programs that illustrate the text or that convey important algorithms may be published. Clear, camera-ready listings of such programs are requested, e.g. printed on white paper using a letter-quality printer with a single-strike carbon ribbon. The proper location for each table and figure should be noted in the margin of the text.

Careful adherence to the Journal's style for citation and quotation of references is required. This style is the so-called Harvard style and has several advantages: the citations are not intrusive, they are often sufficient for the reader to recognise the work without
further effort, and they may be added or deleted easily during drafting without disturbing any pre-arranged numbering scheme.

Citations in the main text should be made as in the following examples: 'It was shown by Curtis and Osborne (1966) that . . .' or 'It has been shown elsewhere (Paine, 1966) that . . .' or 'It may be shown (e.g. see Knuth, 1973a) that . . .'. Note that the citation is composed of the author's last name and the year of publication. Where this is ambiguous, different works by the same author(s) in the same year are distinguished by adding a single lower case letter to the year. Only one level of parenthesis is used. The author's name appears outside the parentheses if the reference is direct, or inside if the reference is indirect. In the latter case, the name and year are separated by a comma. If there are two authors, both names are used. If there are three or more authors, then names for all authors should appear in the initial citation but subsequent citations may be abbreviated by replacing the second and later author names by the phrase 'et al.' Thus, for example, a second citation of the book 'Newey, Stanton and Wolfendale (1978)' may be made as 'Newey et al. (1978)'.

References to unpublished works or private communications should be avoided. If these are desired, they should appear within the main text as e.g. 'Lone and Ryder (to appear)' or 'F. G. Smith (private communication)'. No accompanying entry should appear in the final reference list unless publication has already been arranged.

Full details for all references cited in the text (and no others) should be given in a Reference section at the end of the paper. Each entry must be sufficient to allow an ordinary reader to locate the reference without undue difficulty. Entries should be arranged so that the textual citations can be located readily, i.e. ordered alphabetically by author name and then year. The following illustrates the style of a reference list:

References

Authors are responsible for ensuring the accuracy of all details of all references quoted. Author names are capitalised (for prominence) with initials following the principal name; the year of publication is enclosed in parentheses and followed by a colon. For books and monographs, the colon is followed by the title (underlined or italicised), a page reference (if appropriate), the name of the publisher, and the place of publication. For journal articles, the colon is followed by the full title of the article (not underlined and without quotation marks), the name, or a standard abbreviation for the name, of the journal (underlined or italicised), the volume number, the issue number within the volume, and the page number range (introduced by the symbol 'pp.').

Journal titles appearing in citations should be abbreviated in accordance with accepted standards. (See ISO 4, Documentation--International code for the abbreviation of titles and periodicals and ISO 833, Documentation--International list of periodical title word abbreviations. A relevant summary of these can be found in Computing Reviews, 22. Sample abbreviations are Aust. Comput. J., Datamation and IEEE Trans. Softw. Eng. Titles consisting of a single word are never abbreviated, and a leading The is usually omitted. For longer titles, the following abbreviations should be used: ACM (Association for Computing Machinery), Abstr. (Abstracts), Appl. (Applied Applications), Aust. (Australian), Bull. (Bulletin), Commun. (Communication(s)), Comput. (Computer(s), Computing), Des. (Design), Eng. (Engineering), J. (Journal), Math. (Mathematics), Proc. (Proceedings), Program. (Programs, Programming), Rev. (Review), Sci. (Science(s), Scientist), Softw. (Software), Surv. (Surveys), Syst. (Systems), Trans. (Transactions).

Final Manuscripts. Final versions for manuscripts should be prepared as described above, and should be accompanied by the original versions of figures, diagrams, etc. Each author is asked to supply a short narrative biographical note (about 60-80 words) for publication.

Where manuscripts have been prepared using a word-processing system, the submission of a copy of the final version in computer readable form (magnetic tape or diskette) will be of considerable assistance. Papers that are accepted for publication will be typeset and, shortly before publication, a proof copy will be sent for correction to the (first) author, or to his nominee (overseas authors are requested to nominate a representative in Australia for this task). Symbols for correcting proofs may be found in the Style Manual published by the Commonwealth Government Printer, Canberra, pp. 76-79. Authors should note that proof reading is intended to correct errors introduced by the editing and typesetting processes. Extensive changes or additions to an article introduced during proof-reading may delay publication and be charged to the author. Corrected proofs must be returned promptly or publication cannot be guaranteed.

Fifty reprints of articles will be supplied free of charge.

Papers and correspondence should be addressed to: Associate Professor J. Lions, Editor, Australian Computer Journal, Department of Computer Science, University of New South Wales, Kensington NSW 2033, Australia.

The Australian Computer Journal, Vol. 18, No. 1, February 1986
Avoid the conventional.

As the heart of the Orient and the world’s third most important financial centre, Hong Kong’s efficiency, communications and facilities are first rate. Its hotels and meeting venues are the world’s finest.

But there is another side to Hong Kong; the age-old Chinese customs and traditions that still carry on amid the glittering skyscrapers. Photogenic sailing junks still ply the harbour.

Fascinating festivals, classical Chinese opera, ancient temples and street-side food-stalls still add colour to everyday life in Hong Kong.

The Hong Kong Tourist Association can show you how to make the most of this amazing city. We’ll help you with the planning, preparation and promotion of your meeting, to make it your most successful ever. In Hong Kong.

To find out how, just clip the coupon.

Hong Kong

To: Hong Kong Tourist Association
20/F., National Australia Bank House
270 George Street
Sydney, N.S.W. 2000

Please tell me more on how I can meet with success in Hong Kong.

Name ___________________________
Organisation _______________________
Address ___________________________
Tel ____________________________
MEMBERSHIP: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries.

MEMBERS: The current issue of the Journal is supplied to personal members and to Corresponding Institutions. A member joining partway through a calendar year is entitled to receive one copy of each issue of the Journal published earlier in that calendar year. Back numbers are available from the National Secretariat, P.O. Box 319, Darlinghurst, NSW 2010. No trade discounts are given, and agents should recover their own handling charges.

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

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

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

M A N U S C R I P T S : Manuscripts should be typewritten in double spacing on A4 size paper, with 2.5cm margins on all four sides. The original, plus two clear hard-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:

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Bibliography and Subject Index of Current Computing Literature.</td>
</tr>
<tr>
<td>ACM</td>
<td>Computing Reviews</td>
</tr>
<tr>
<td>AMS</td>
<td>Mathematical Reviews</td>
</tr>
<tr>
<td>CSA</td>
<td>Computer and Information Systems Abstracts.</td>
</tr>
<tr>
<td>CSA</td>
<td>Data Processing Digest.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Indexer</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENGINERING INDEX INC.</td>
<td>Engineering Index.</td>
</tr>
<tr>
<td>INSPEC</td>
<td>Computer and Control Abstracts.</td>
</tr>
<tr>
<td>INSPEC</td>
<td>Electrical and Electronic Abstracts.</td>
</tr>
<tr>
<td>ISI</td>
<td>Current Contents/Computer Science.</td>
</tr>
<tr>
<td>ISI</td>
<td>CompuMath Citation Index.</td>
</tr>
<tr>
<td>SPRINGER-VERLAG</td>
<td>Zentralblatt fur Mathematik und ihre Grenzgebiete.</td>
</tr>
</tbody>
</table>

Copyright © 1986. 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: Ambassador Press Pty Ltd, Parramatta Road and Good Street, Granville, NSW 2142.

The Australian Computer Journal is an official publication of the Australian Computer Society Incorporated.

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


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

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

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

M A N U S C R I P T S : Manuscripts should be typewritten in double spacing on A4 size paper, with 2.5cm margins on all four sides. The original, plus two clear hard-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:

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Bibliography and Subject Index of Current Computing Literature.</td>
</tr>
<tr>
<td>ACM</td>
<td>Computing Reviews</td>
</tr>
<tr>
<td>AMS</td>
<td>Mathematical Reviews</td>
</tr>
<tr>
<td>CSA</td>
<td>Computer and Information Systems Abstracts.</td>
</tr>
<tr>
<td>CSA</td>
<td>Data Processing Digest.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Indexer</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENGINERING INDEX INC.</td>
<td>Engineering Index.</td>
</tr>
<tr>
<td>INSPEC</td>
<td>Computer and Control Abstracts.</td>
</tr>
<tr>
<td>INSPEC</td>
<td>Electrical and Electronic Abstracts.</td>
</tr>
<tr>
<td>ISI</td>
<td>Current Contents/Computer Science.</td>
</tr>
<tr>
<td>ISI</td>
<td>CompuMath Citation Index.</td>
</tr>
<tr>
<td>SPRINGER-VERLAG</td>
<td>Zentralblatt fur Mathematik und ihre Grenzgebiete.</td>
</tr>
</tbody>
</table>

Copyright © 1986. 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: Ambassador Press Pty Ltd, Parramatta Road and Good Street, Granville, NSW 2142.