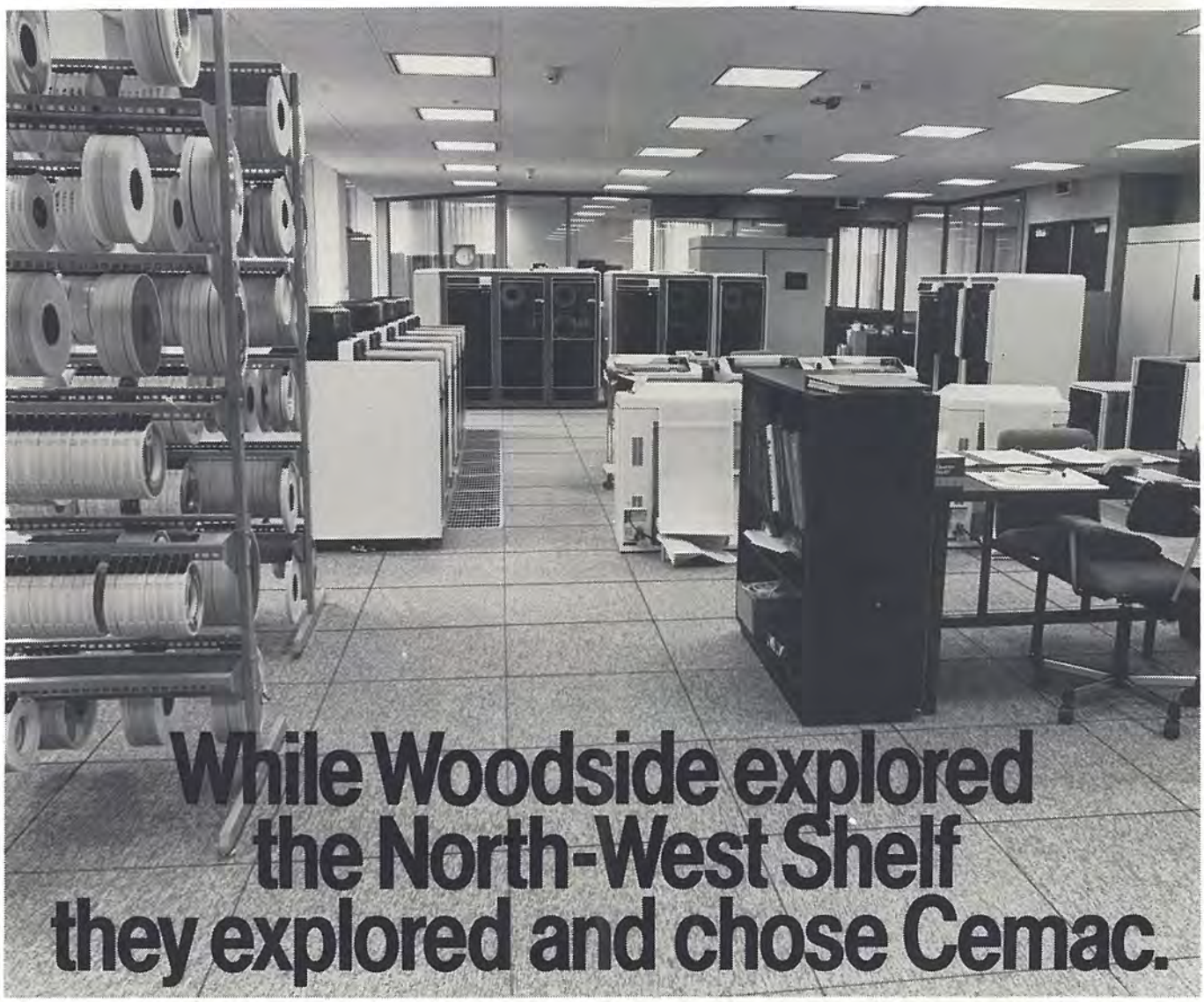
VOLUME TWELVE, NUMBER FOUR, NOVEMBER 1980. ISSN 004-8917

# THE Australian Computer Journal

PUBLISHED FOR THE AUSTRALIAN COMPUTER SOCIETY INCORPORATED
Registered for posting as a publication – Category B.



# While Woodside explored the North-West Shelf they explored and chose Cemac.

This is the 200m², Snaplock access floor installed in June 1980 at the Perth offices of Woodside Petroleum Limited. It supports computers used to process data for the North-West Shelf gas field development.

The floor uses Series 800 metal panels on 300mm pedestals and has a Heritage II, monolithic, anti-static carpet finish.

Partitioning was also installed at the same time as the floor.

Cemac Tate's all-metal access floor system offers four panel sizes, with or without stringers and the widest range of finishes – high pressure laminate, wood parquet, cork and vinyl asbestos – as well as anti-static carpet.

Cemac also has a patented, adjustable pedestal base which ensures a level floor on a sloping slab. So, for access flooring, or complete office interior layout (floors, systems furniture, ceilings, partitions and task or ambient lighting), call Cemac Interiors.

cemac

**Brochures and details from Cemac Interiors:**

| | | Licensees: | |
|---|---|---|---|
| Sydney | 290 3788 | Adelaide | 45 3656 |
| Melbourne | 419 8233 | Hobart | 29 5444 |
| Brisbane | 221 5099 | Perth | 444 7888 |

CEIN0022

# News Briefs from the Computer World

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

## RESEARCH GUIDE AVAILABLE

The most comprehensive guide yet published on CSIRO's research activities throughout Australia became available in late October.

The guide, containing descriptions of all CSIRO's more than 700 research programs and sub-programs, is a valuable source of information to industry, government, research and educational institutions.

In clear, non-technical language, it outlines research problems being tackled by CSIRO and the implications of research findings, as well as providing details of where the research is being conducted, how many staff are involved, and how it is funded.

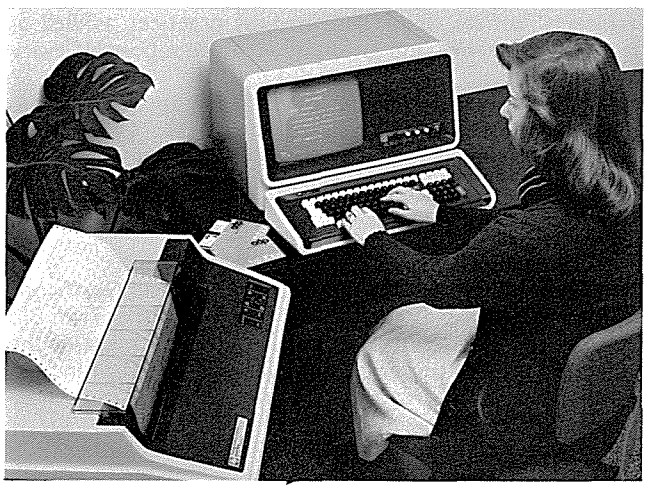
The CSIRO research guide was first published four years ago, and demand for the publication has indicated a strong requirement for information on the Organization's research.

A new-look edition published for the first time last year in line with a Government directive that CSIRO should provide a comprehensive research directory, went to a second printing.

More than 750 copies were sold, the majority to industry, while another 800 were sent to public libraries, college and university libraries, State and Commonwealth departments, parliamentary libraries and the Academy of Science.

Copies of the publication, titled CSIRO Research Programs 1980-81, are available for $12.50 (postage included) from the CSIRO Editorial and Publications Service, P.O. Box 89, East Melbourne, Victoria, 3002.

## THE ABC-24 MICRO-COMPUTER



G.M. O'Reilly & Associates, North Sydney, NSW are announcing the Australian release of a major new micro-computer – the ABC-24, a micro-computer designed to meet the Australian market's specific needs.

The ABC-24 is the product of the Japanese computer manufacturer, Ai Electronics and the design efforts and experience of G.M. O'Reilly & Associates.

The ABC-24 has been designed to operate in a stand-alone environment but is equally at ease as part of a communications network.

The ABC-24 can utilise virtually all computer langauges and protocols to link it with other computers. For instance, the ABC-24 can exchange information with others of its type, store the information and transmit it to large mainframe computers.

The ABC-24 can also handle a very wide range of standalone business applications. These applications include general ledger accounting, inventory management, order entry and invoicing, word processing, time recording and costing, job costing, maintenance of chemical formulae and many others. As such it has a major contribution to make to professional organisations such as accounting practices, architects' offices, legal and medical practices as well as commercial, service and government organisations of every type.

Much of the software for all these applications has been designed and developed by G.M. O'Reilly & Associates. Other software has been introduced from major suppliers in the US. All software is totally supported by the company.

## TRIM BIN DESIGN KIT

Tecnico Electronics, Lane Cove, NSW and Northcote, Victoria announces the arrival of the new Bourns Trim Bin in Australia. The Trim Bin contains an assortment of the most popular trimmers and "MFT" multi-function trimmer/resistors, all in a functional, convenient and attractive package.

As in America the Trim Bin is offered at a price only about half that of the total cost of the components bought as separate lots.

The Bourns Trim Bin contains the following:

Fifteen of the most popular trimmers and "MFT" trimmer/resistor models with 50 varieties of resistance and pin styles, for a total of 127 units.

Design aids for Model 20, 3005, 3006, 3099, 3386, 3299, 3339 and 3359.

One each H-90 and H-91 screwdrivers.

These quality components include rectangular, square and round types; sealed and open-frame, wirewound and cermet, single and multi-turn.

Trim Bin should prove popular in engineering research and development labs where design-in and prototype models are produced.

## HONEYWELL TRACKS GOLFERS' SCORES AT MEMPHIS TOURNAMENT

A sophisticated golf scoring system using Honeywell hardware and software was unveiled recently at the Danny Thomas Memphis Golf Classic. While PGA personnel did the official scoring. Honeywell, through a Level 6 small computer and ten VIP 7700s, provided instantaneous, on-line scoring tabulations and summaries. A level 6 software support specialist working out of the firm's Gulf Coast Branch, worked long, 10-12 hour days perfecting the pilot program. During the tournament, fans could pick up a computer printout each morning detailing the play of the previous day on a hole-by-hole basis. Honeywell printed 2,000 of these reports each night. In keeping with the purpose of the tournament, all of Honeywell's hardware, software and maintenance services were donated.

## FREE 1981 ANNUAL TANDY ELECTRONICS CATALOGUE

Tandy Electronics is pleased to announce the arrival of their 148-page catalogue for 1981 featuring five additions to the TRS-80 family of micro-computers and accessories. It is free from any one of nearly 250 Tandy Stores and participating dealers across Australia.

The 1981 catalogue, with over 100 full colour pages, is a showcase for an exclusive range of more than 2,300 products under Tandy's brand names.

Easily one of the biggest outside commercial printing feats ever performed in Australia, the Catalogue consumed 135 tonnes of paper, 5,625 litres (2,250 gallons) of ink and required 158 hours of press time in its production. Thousands of catalogues are being given away.

## SOLDER REEL ATTACHMENT FOR ROYEL TOOL REST


Royston Electronics have now produced a solder reel atttachment for their popular Model TD150 solder tool rest.

The TD150, which accepts Adcola, Royel and other popular soldering tools, is a bench unit which provides a convenient receptacle for a soldering tool while idling. The tool is held firmly, at just the right angle, but inserts and extracts without effort. The unit also protects the tool itself, as well as the operator and the work bench.

Its heavy cast base gives the required stability without clamps or bolts, and a recess in the base holds the tip wiping sponge which is slotted to achieve the most effective, easy cleaning action.

The addition of the solder reel attachment now makes it the ultimate convenience accessory for all hand soldering operations.

The TD150 tool rest, and the new solder reel attachment, are available from electronics parts suppliers or from Royston Electronics, Notting Hill, Victoria or Punchbowl, NSW.

## ROBUST SOFTWARE FOR MICRO-COMPUTER USERS

The alliance "C. Abaci" has released robust software in several areas for scientific computer users. Exacting benchmarks on accuracy, efficiency, and testing which

*(Continued at back)*

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

Office Bearers. *President:* G.E. Wastie; *Vice-Presidents:* R.C.A. Paterson, A.W. Goldsworthy; *Immediate past president:* A.R. Benson; *National treasurer:* C.S.V. Pratt; *Chief executive officer:* R.W. Rutledge, PO Box N26, Grosvenor Street, Sydney, 2000, telephone (02) 267 5725.

Editorial Committee: *Editor:* C.K. Yuen, CSIRO Division of Computing Research, P.O. Box 1800, Canberra, A.C.T. 2601. *Associate Editors:* J.M. Bennett, T. Pearcey, P.C. Poole, A.Y. Montgomery, J. Lions.

SUBSCRIPTIONS: The annual subscription is $15.00. All subscriptions to the Journal are payable in advance and should be sent *(in Australian currency)* to the Australian Computer Society Inc., PO Box N26, Grosvenor Street, Sydney, 2000. A subscription form may be found at the end of the August issue.

PRICE TO NON-MEMBERS: There are now 4 issues per annum. The price of individual copies of back issues still available is $2.00. Some 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 640, Crows Nest, N.S.W., 2065. No trade discounts are given, and agents should recover their own handling charges. Special rates apply to members of other Computer Societies and applications should be made to the Society concerned.

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.

REPRINTS: 50 copies of reprints will be provided to authors. Additional reprints can be obtained, according to the scale of charges supplied by the publishers with proofs. Reprints of individual papers may be purchased for 50 cents each from the Printers (Publicity Press).

PAPERS: Papers should be submitted to the Editor, authors should consult the notes published in Volume 12, pp. 71-75 (or request a copy from the National Secretariat).

MEMBERSHIP: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries. Canberra: P.O. Box 446, Canberra City, A.C.T., 2601. NSW: Science House, 35-43 Clarence St, Sydney, N.S.W., 2000. Qld: Box 1484, G.P.O., Brisbane, Qld, 4001. S.A.: Box 2423, G.P.O., Adelaide, S.A., 5001. W.A: Box F320, G.P.O. Perth, W.A., 6001. Vic: P.O. Box 98, East Melbourne, Vic, 3002. Tas: P.O. Box 216, Sandy Bay, Tas, 7005.



**Published by:** Associated Business Publications, 28 Chippen Street, Chippendale, N.S.W., 2008. Tel: 699-5601, 699-1154.
*All advertising enquiries should be referred to the above address.*

**Printed by:** Publicity Press Ltd., 29-31 Meagher Street, Chippendale, N.S.W., 2008.

ISSN 004-8917

# THE Australian Computer Journal

*VOLUME 12, NUMBER 4, NOVEMBER 1980*

## CONTENTS



This Journal is Abstracted or Reviewed by the following services: :

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

# Editorial

Journal bashing is something of a favourite sport among ACS members. Like most sports, this interesting activity follows a set of standard rituals. A typical exchange consists of a practitioner complaining "the Journal's too academic" and an academic answering back "because practitioners don't write papers for it". The numerous discussions that have taken place in the past seldom went beyond variations on this basic theme, never giving results other than a simmering dissatisfaction on both sides.

This unhappy state of affairs really reflects a fundamental controversy about the function and purpose of the Journal. If one regards the Journal simply as a vehicle of service to ACS members, then one is bound to reach the conclusion that the Journal is not giving most members the material they want to see. Now one can always argue that the fault is with the reader rather than the material, because members "don't know what's good for them", "didn't tell us what they need" or "should write some industry-oriented papers instead of just complaining about lack of them". Although this may be good for winning arguments, it does not win the Journal many friends. The question is too important to be quickly dismissed in this way.

What one has to get across to the practitioner is that, the Journal is more than just a publication for supplying technical reading material to ACS members. It is meant to serve a range of functions. One of these is to provide a medium for publishing new findings of computing research and new experiences gained in computing practice, to bring the work described to the attention of an international audience, and provide a permanent record of the authors' contributions to computing knowledge. By publishing such a journal, ACS establishes in a concrete manner its claim to be a society of learning, a society with the advancement of computing knowledge as part of its charter. In this indirect and rather elusive way the Journal does provide a service to ACS members. What is unfortunate is the lack of balance, in that we do not receive enough material of other types that serve members at large in a more direct fashion.

It is often thought that the Journal represents a heavy financial burden to the members, and indeed publishing-cost discussions on the Journal virtually constitute a standing item on the Council agenda. When, however, one works out the net cost (after subtracting subscription incomes), it comes to about $2 per head, hardly a major component of the annual membership fee. Whatever shortcomings the Journal might have, extravagance is not one.

Another charge frequently levelled at the Journal is editorial bias towards academia. Indeed the Editorial Committee, authors, referees and book reviewers of the Journal are largely drawn from university and college computing departments and government research organizations. But the main reason for this is, simply, that these are where we can find people prepared to take on, without pay, the time-consuming efforts involved. In the past numerous attempts were made to obtain more contributions from industry, and it was our lack of success on that front, for whatever reason, that made the Journal what it is.

Nevertheless, we have not yet entirely given up the hope of making the Journal please everyone, at least a little. While continuing to publish research type papers, we are hopeful that some of our traditional authors would also be willing to spare a little time to write tutorial papers oriented at a broader audience. The special issues we intend to publish (Computer Networks, May '81; Database Management, November '81) represent another effort with the same objective. If you would like to see the Journal do more for ACS members, perhaps you could start by sending in some material for the special issues?

*C.K. Yuen*

# FACETS: A Language Feature for Security and Flexibility

Warren Burton* and Brian Lings*

Current designs in programming languages stress the need for data abstraction facilities. This paper views a data abstraction as one facet of the behaviour of its underlying data type. This facet hides the implementation details but presents a full view of allowable operations on objects associated with it. It is argued that, particularly in an environment where security is important, it is necessary to provide a facility for precisely defining other, more limited facets of the behaviour of a type. These can then be used for fully controlling access to sensitive data.

Language facilities for defining and using facets of both simple types and type combinations are developed in a Pascal environment. Various examples of 'safe' generic procedures and their use with sensitive data are presented.

Keywords and Phrases: Abstract data types, access control, generic procedures, iteration statements, programming languages, programming methodology, security, type checking.

CR Categories: 4.20, 4.22, 4.33

## 1. INTRODUCTION

The motivation behind much work in programming language design has been the realization that the reliability and understandability of programs can be significantly improved through the use of abstraction.

Abstraction is used not only for program segments, through the use of procedures, but also for data objects, through the use of abstract data types (Dahl, Myhrhaug and Nygaard, 1970; Guttag, Horowitz and Musser, 1978; Liskov, Snyder, Atkinson and Schaffert, 1977; Wulf, London and Shaw, 1976). Both concepts break up the implementation problem into manageable pieces, and delay many implementation decisions until the shell, or program abstraction, has been fully designed.

For each abstract data type a set of primitive operations are defined. The representation of instances of the type and the implementation of the primitive operations are completely internal to the abstract data type definition. The set of operations must be complete, in the sense that every operation required for a given object during the execution of a program must be derivable from the set of operations provided for the type with which the object is associated. Strong type checking forces conformity. One further advantage follows: the contexts in which a data object may appear are defined by the legal operations for that object. Hence code can be written not only without knowing the underlying representation of an object (data independence) but also without knowing more about its type than that certain operations are defined for it. Such *generic* procedures allow the expression of the logic of a routine without over-defining the objects on which it can work. The saving on programmer effort is obvious: if several abstract data types have certain operations in common, then it is possible to write generic procedures in terms of these common operations (Gries and Gehani, 1977; Liskov *et al.*, 1977; Wulf *et al.*, 1976).



Even in a language such as Pascal, which does not support data abstractions as such, there exist operations which are defined for many different types. These include *succ* , *pred*, '>' , '<' etc. for ordered types, ':=' for all types, and '+','−','*' etc. for numeric types. Any routines expressed in terms of such operators could, in theory, be generic. There is a strong parallel here with variant record types. With a variant record, routines can be written which will accept any of the possible variants of a record but access only the fixed part of such records. Just as such a routine can only 'use' those fields which all records of the variant have in common[1], a generic routine can only use those operations which all objects appearing in that context have in common. The necessary set of operations in such a context for an object we call a *requirement*. Consider the requirement *sortable*. A list is *sortable* if

(i) an ordering relation is defined on the components of the list, and
(ii) any two elements in the list can be interchanged.

Thus the primitive operations required before an object can appear in a sortable context are

(i) an *inorder* operation, which acts on a given pair of list components to yield a boolean result; and
(ii) a *swap* operation, which interchanges two components of a list.

Although the first of these operations is likely to be a primitive for a data type (if it applies at all), the *swap* operation is not. In general the basic operations for a procedure, in this case a *sort* routine, may be at a higher level than is likely for the primitive operations of a data type. The latter is likely to support a general *copy* operation (or *assignment*) instead. If, rather than the swap operation given above, the assignment operator were to be specified in requirement *sortable*, we can see that we would be demanding more power than is absolutely necessary. When considering the general operation of generic procedures this fact is not particularly worrying: the

1. (Short of testing the tag field to find out more about the object).

*Warren Burton is with the School of Computing Studies, University of East Anglia, Norwich, England. Brian Lings is with the Department of Computer Science, University of Queensland, St. Lucia, Australia, 4067. *Manuscript received 16 May 1980.*

swap procedure can be included in line, or written as a generic procedure itself. However, the implications for generic procedures in a security-conscious environment are somewhat greater.

It has been suggested that security can be provided in programming languages by allowing a user to restrict the set of operations which a procedure may apply to a parameter (Jones and Liskov, 1978). For example, a user may pass a file as a parameter to a procedure, but restrict usage by the procedure to the *append* operation. The procedure would be prohibited from accessing or overwriting confidential information.

It is the thesis of this paper that, if security of data is required, it is not enough to be able to limit a subuser's access to data by passing him only a subset of the defined primitive operations on that data. These operations in themselves may be too powerful, and only certain aspects of their behaviour should be given.

Let us consider the problem of sorting an array of confidential records. We want to be able to invoke a sort procedure in such a way that it will not be possible for the sort procedure to examine or copy individual records. At the very least, it must be possible for the sort procedure to compare the key values of two array elements and to exchange two arbitrary array elements. In order to exchange two array elements, both the primitive operation to access an array element and the primitive operation to redefine an array element must be used. If a programmer can exchange two elements in an array through the use of these primitive operations, then he can also exchange elements between the given array and a rigged private array. A solution to the problem is to define a non-primitive *swap* operation which will exchange two elements of a single array. This non-primitive operation may be permitted, rather than the more powerful set of primitive operations which must be used to implement swap.

In section 2, a facet is defined to be a set of operations[2] for a data type. In general, given one view of data, a facet can be used to define another (possibly more restricted) view. It is possible for a program to pass a facet of a data instance (i.e., the data instance with the restriction that only facet operations may be applied to it) as a parameter to a procedure and thereby precisely control the procedure's use of the data instance.

For purposes of illustration, programming language features for defining and using facets are introduced through a number of examples. We do not attempt a full or formal definition of the language used. We base our notation on Pascal (Jensen and Wirth, 1974; Wirth, 1971).

In section 3 we consider facets of compound data instances. By packaging parameters into a single facet definition, a user can control operations involving more than one data instance. For example, a user may permit information to be moved from one data instance to another specific data instance without allowing information to be moved to an arbitrary data instance. Section 4 addresses some issues which, for the sake of clarity of presentation, are only lightly touched on in the earlier text.

The problem of generalizing iteration to user defined data types (e.g., so that it is possible to iterate over nodes of a tree as well as elements of an array) has been considered by others (Liskov *et al.*, 1977; Shaw, Wulf and London, 1977). In section 5 we show how generalized iteration can be supported through the use of facets, without any special case extensions to a language. We also consider briefly the interaction between facets and parameterized types (Gries and Gehani, 1977; Liskov *et al.*, 1977; Wulf *et al.*, 1976).

The language features discussed here are currently being implemented. Some implementation considerations are mentioned in section 6. Section 7 is the conclusion.

```
procedure sort (var V: sortable <T> ; n : integer);
    var i, j : integer;
    begin
        for i := 1 to n-1 do
            for j := 1 to n-i do
                if not T.in_order (V,j,j+1) then
                    T.swap (V,j,j+1)
    end;
```

Figure 1: A generic bubble sort.

```
specification of sortable requirement for T;
        function in_order (T; integer; integer): boolean ;
        procedure swap (var T; integer; integer);
```

Figure 2: Specification of the *sortable* requirement.

## 2. FACETS OF SINGLE OBJECTS

If it is known that particular operations are defined for a formal parameter, then it may be possible to write a generic procedure which will process the parameter using only these operations. Instances of any type supporting the operations may be processed.

Using a notation similar to that of Gries and Gehani (1977), Figure 1 shows a generic procedure to sort a vector, $V$, having n elements. In effect, the type of $V$ is passed to the formal parameter $T$. We qualify the unknown type, $T$, with *sortable* to indicate that the actual type corresponding to $T$ must satisfy the *sortable* requirement, which is defined in Figure 2. A requirement specifies the set of operations and the format of each. No semantic information is currently given[3].

If *index_vector* is an abstract data type supporting the primitive operations *in_order* and *swap* such that, for any instance $x$ of *index_vector*,

2. We consider all operations to take the form of function and procedure invocations. The ideas presented here can be applied in environments supporting other types of operations.

3. It is anticipated that semantic information will eventually be added using algebraic axioms. These would have to be satisfied by any facet purporting to satisfy a requirement.

```
type employee_record = record
                         key : integer;
                         name : text
                       end;
     index_vector = array [1..100] of↑employee_record;
define keyorder = sortable facet of index_vector having
    function in_order (vec: keyorder; i,j: integer):boolean;
     begin in_order :=rep vec[i]↑.key <= rep vec[j]↑.key end;
    procedure swap (var vec: keyorder; i,j: integer);
     var temp: ↑employee_record;
     begin
        temp := rep vec [i];
        rep vec [i] := rep vec [j];
        rep vec[j] := temp
     end;
```

**Figure 3: Definition of a *sortable* facet.**

*index_vector.in_order (x,i,j)*
returns *true* when the value of the *i th* element of *x* is in the correct order (less than or equal to for a sort into ascending order) with respect to the *j th* element and *false* otherwise, and

*index_vector.swap (x,i,j)*
will exchange the *i th* and *j th* elements of *x*, then

*sort (x,n)*
will sort the first *n* elements of *x*.
Unfortunately, an abstract data type for vectors may not have *in_order* and *swap* defined as primitive operations. However, it is likely that these operations can be defined in terms of lower level primitive operations. Therefore, we can define a restricted view of a type, called a *facet*, as shown in Figure 3. The restricted view is expressed in terms of a set of permitted operations.

If *x* is of type *index_vector* then
*sort (x <keyorder>, 100)*
will cause *x* to be sorted. The restriction, *<keyorder>*, constrains *sort* to the *keyorder* facet of *x*.

Within the definition of the *keyorder* facet, the **rep** operator may be applied to a parameter of type *keyorder* to produce the underlying *index_vector*. Hence the **keyorder** operations may be defined in terms of *index_vector* operations. The **rep** operator may not be used elsewhere. (See §4 for further discussion of **rep**).

We have defined a secure interface between a calling program and a sort procedure. It would not be possible for *sort* to look at or modify any of the records to which elements of *x* point. At most *sort* can determine the permutation required to correctly order *x*. At worst *sort* can return an incorrectly ordered vector, but one with no information added or removed. The same interface could be used for more complicated and more efficient sort procedures[4].

4. We have restricted ourselves to a simple problem where it probably is as easy to define a safe sort procedure as it is to define a safe interface to an unknown sort procedure. However, the method generalizes to more complicated situations.

```
type bankrecord = record
                    account : 0..999999;
                    name : text;
                    balance : integer;
                    status : (credit, overdrawn)
                  end;
   bankfile = array [1..1000] of bankrecord;
define name_order = sortable facet of bankfile having
   function in_order (bf : name_order; i,j : integer) : boolean;
     begin in_order := rep bf [i].name <= rep bf [j] .name end;
   procedure swap (var bf : name_order; i,j : integer);
     var temp : bankrecord;
     begin
        temp := rep bf [i];
        rep bf [i] := rep bf [j];
        rep bf [j] := temp
     end;
define account_order = sortable facet of bankfile having
   function in_order (bf : account_order; i,j : integer) : boolean;
     begin in_order := rep bf [i].account <= rep bf [j].account end;
   procedure swap (var bf : account_order; i,j : integer);
     begin name_order.swap (rep bf <name_order>,i,j) end;
```

**Figure 4: Two *sortable* facets of a common type.**

We emphasize that, unlike basic operations in a data abstraction definition, the operations defined in a facet are user oriented and will be defined in terms of the more primitive operations available for the data type. They can also be redefined for different facets of the same data type which also meet the given requirement.

The first point is important when libraries of data abstractions are being used; the fact that required operations do not exist as primitives for a type does not in itself preclude the presentation, to a routine, of a specific facet of an instance of that type. The second point is important if true generality is to be achieved; there may be several ordering relations by which, for example, an array may be sorted. Both points are essential if full control over access to data is to be maintained.

In Figure 4, two different *sortable* facets are defined for a *bankfile*. The statement

*sort (savings <name_order>, n)*
will sort the first *n* records of the *savings bankfile* on the *name* field, while

*sort (savings <account_order>, n)*
will sort on the *account* field.

```
specification of selectable requirement for T;
    function is_required (T; integer): boolean;
    procedure select (var T; integer; var integer);
define select_debtors = selectable facet of [from, to: bankfile] having
    function is_required (pair: select_debtors; i:integer):boolean;
      begin is_required := rep pair.from[i] . status = overdrawn end;
    procedure select (var pair:select_debtors; i:integer;var j :integer);
      begin
        j := j+1;
        rep pair.to[j] :=rep pair.from[i]
      end;
```

**Figure 5: A facet of a compound object.**

This example is based on an example given in Jones and Liskov (1978). The solution given there would permit a rogue programmer to exchange information between the given *bankfile* and a rigged private *bankfile*. In addition, separate procedures would be required for sorting on different fields.

We note that facets may be used in any situation where it is necessary to map one view of data onto another (possibly more restricted) view. For example, given a point in a plane represented in polar co-ordinates, it would be possible to define a *rectangular* facet, supporting operations *getx*, *setx*, *gety* and *sety* (with obvious semantics), to map the data onto a rectangular co-ordinate system view.

## 3. FACETS OF COMPOUND OBJECTS

In addition to controlling the processing of individual parameters, a user may wish to control the interaction of a combination of parameters.

Suppose we have two *bankfiles* (as defined in Figure 4) named *main* and *red*. We wish to invoke a procedure to copy those *bankrecords* of *main* with *status overdrawn* into *red*. We do not want to permit the procedure to copy *bankrecords* of *main* to any other destination.

The definition of a *select_debtors* facet of a pair of *bankfiles* is given in Figure 5, and a generic procedure which will process the facet is given in Figure 6. The procedure may be invoked by:

*prepare_sub_list ([main, red]<select_debtors>, m,n),*

where *m* is the number of *bankrecords* in *main* and on exit *n* will contain the number of entries in *red*. The square brackets indicate that *main* and *red* are to be treated, for the purpose of the call, as a single unit. We call such a unit a *Compound*. Within the definition of *select_debtors* individual elements of the unit may be selected in the same manner as components of a record structure.

Note that a procedure which processes a unit, such as *prepare_sub_list*, does not need to know (and in fact cannot know) that one of its parameters is compound. Facets of compounds satisfying a particular requirement need not all be facets of compounds with the same number of elements.

## 4. A CLOSER LOOK AT SECURITY

We have used the term 'security' freely in the text, but have not as yet defined it. We now attempt such a definition, though it is worth bearing in mind that this is our *working* definition, not an attempt to force conformity of view on the semantics of the term.

A data object is defined to be *secure* if the owner (creator) of that object has precise control over granting access to it. Such an object must only be accessible in a prescribed manner, the prescriptions being individually tailored according to need-to-know and right-to-know criteria.

Total security is obviously impossible unless no communication takes place at all: once access has been granted we rely on the grantee to protect the information with which he is provided. With these facts in mind let us take an overview of requirements and facets.

Requirements and facets are governed by the same scope rules as procedures. Conceptually, a complete program must be compiled at once: any problems arising through using separate compilation of modules must be resolved in favour of the tight type checking provided by single compilation.

If a subsystem programmer wishes to 'protect' one of his own variables using a facet defined by another programmer (for example, the project leader) then he must have implicit trust in the facet definition. If, however, tight security is required then a new facet must be defined — perhaps as a copy of the original — to bring it under the control of the subsystem.

Whoever owns (that is, defines) a facet, F, of a type, T, has amplification rights (Wulf *et al.*, 1976) within the facet procedures and functions (via the rep operator). These allow him to apply functions on type T to objects whose F facets are supplied. Hence *tight security is only available to the creator of the facet.*

## 5. FURTHER EXAMPLES AND EXTENSIONS

### 5.1 Iterators

In a traditional programming language it is possible to iterate over the elements of an array (e.g.,

```
procedure prepare_sub_list (var lists:selectable <T>
        from_size:integer; var to_size:integer);
    var current:integer;
    begin
        to_size := 0;
        for current := 1 to from_size do
            if T.is_required(lists, current)
                then T.select(lists,current,to_size)
    end;
```

**Figure 6: Procedure Prepare_Sub_List.**

```
for i := 1 to n do a[i] := 0;
```

will zero an array).

Gries and Gehani (1977) have suggested *ad hoc* extensions so that

```
for element in structure do. . . ;
```

may be used to iterate over the elements of *structure* where *structure* may be a set or any of a number of other standard structured objects. Facilities for defining iteration over the components of instances of a user defined abstract type are provided in Alphard (Shaw *et al.*, 1977) and CLU (Liskov *et al.*, 1977). Facets can be used to interface arbitrary data structures to a generic *for* without *ad hoc* extensions to a language, in this case Pascal-Plus (Welsh and Bustard, 1979).

Rather than introduce special primitive forms, let us consider an *envelope,for*. The envelope will have one parameter, an iterable facet of an item. The *requirement iterable* as defined in Figure 7 demands two procedures and one function. The required envelope can now be defined using only these three routines (Figure 8). If we wish to iterate over the elements of a vector we can define an iterable facet for the type combination [vector, integer], where an instance of the record type will act as a cursor during the iteration. Such a facet is shown in Figure 9; in this instance we are processing the elements in reverse lexicographic order. The elements in the vector *vec* can now be printed by the statement

```
instance loop: for ([vec, i] <descending>)
      write (vec[i])  ;
```

A nested loop can also be constructed easily, viz.:

```
instance outer_loop : for ([vec,i] <descending>)
      instance inner_loop : for ([vec,j] <descending>)
            begin

            end;
```

We could, of course, treat iteration as a special case by specifying the requirement *iterable* and the envelope *for* in a language prelude. A special syntactic form similar to those in current languages could then be provided as an alternative to those above, viz.:

```
for i through vec<descending> do
      for j through vec <descending> do
            begin

            end;
```

Such facilities are desirable, but in no way enhance the power of the language features proposed.

A further example is shown in Fig. 10, where iteration over the nodes of a tree (to be processed in symmetric order) is catered for. The 'cursor' in this case is a *refstack* (or stack_of_tree). For convenience we assume that *refstack* is a data structure for which the usual stack procedures are provided. Using the notation introduced above we may now say

```
for cursor1 through tree1 <inorder> do
```

assuming correct declarations for cursor1 and tree1. Further examples, assuming relevant definitions, are

```
specification of iterable requirement for T;

    procedure cursor_init (var T);

    function more  (T) : boolean;

    procedure next (var T);
```

Figure 7. Specification of the *iterable* requirement.

```
envelope for (s:iterable<T>);

 begin

    T.cursor_init(s);

    while T.more(s) do

      begin

       ***;

        T.next(s)

      end

end;
```

Figure 8. Declaration of a generic *for*.

```
for cursor1 through tree1 <preorder> do
      for cursor2 through tree2 <postorder> do
            begin

            end;
```

### 5.2 Private Types

Facet descriptions of *compound* objects are required above by the need for a cursor local to the section of code using the facet. In Ada (1979) the idea of a private type, where only assignment and equality operators are defined, is introduced. Particularly for recursion it is very useful to have a routine provide storage for data which is otherwise under the control of a facet: consider tree traversals or a Quicksort algorithm. This can be accommodated in facets in a simple and consistent manner by allowing type definitions as well as procedure and function definitions within a facet declaration. A procedure provided with a facet F defining type t can then declare local variables of the above form by the declaration

```
var a,b: F.t;
```

where a and b may only be employed as parameters to procedures defined in F, and in the forms

```
(a = b)
 a:= b;
```

Discussion is left to a subsequent paper.

```
define descending = iterable facet of [v: vector, i: integer] having
    procedure cursor_init(var a: descending);
     begin
      rep a.i: = rep a. v. top
     end;
    function more (a:descending) : boolean;
     begin
      more := rep a.i≥ rep a. v. bot
     end;
    procedure next(var a:descending);
     begin
      rep a.i := rep a.i-1
     end;
```

**Figure 9: Definition of the *descending iterable* facet for a vector.**

### 5.3 Parameterized Types

Language facilities for parameterized types have been widely advocated (Gries and Gehani, 1977; Liskov *et al.*, 1977; Wulf *et al.*, 1976). Facets can be used in an environment which supports parameterized types.

Figure 11 depicts a modification to Fig. 10 in order to allow the use of a generic type *tree*. If *b* is a *tree (integer)* then we can process the values of the nodes of *b*, in symmetric order, by

**for** *cursor* **through** *b* <*inorder*> **do**

In this case *cursor* must be of type stack_of_tree(u).

## 6. IMPLEMENTATION CONSIDERATIONS

There are several ways to implement generic procedures of the type considered in section 2.

A compiler can produce a separate object code procedure for each combination of actual parameter types used in invocations of a generic procedure. We note that if generic procedures were not provided, a user would have to produce a separate source code procedure for each combination of parameter types.

Alternatively, for each actual parameter corresponding to a generic formal parameter, we can pass both the data item and the operations with which to process it. This would reduce speed slightly, but could significantly reduce the amount of object code.

In some situations, a third approach may warrant consideration. If a generic procedure is invoked by a program running on one machine but is to be executed on another, possibly untrustworthy, machine, then it may not be desirable to pass secure data at all. Instead, the procedure may be required to request the invoking machine to perform all desired operations on the secure data. This is not likely to lead to a very efficient implementation. However, one machine is allowed to keep its data securely at home, the other machine is allowed to keep its confidential software secret, and yet the two can interact.

```
type tree = ↑cell;
     cell = record left, right: tree;
                   value: integer
            end;
define inorder = iterable facet of [t: tree, s: refstack] having
  procedure cursor-init (var a:inorder);
    var temp : tree ;
    begin with rep a do begin
      init (s); temp :=t;
      while temp < > nil do
        begin push (s, temp) ; temp:=temp ↑ . left end
    end end;
  function more (a:inorder):boolean;
   begin
    more := not empty(rep a.s)
   end;
  procedure next (var a:inorder);
   var temp:tree;
   begin with rep a do begin
     temp := top(s)↑.right;
     pop(s)
     while temp< >nil do
     begin
       push(s, temp);
       temp := temp ↑.left
     end
  end end;
```

**Figure 10: Definition of the *inorder iterable* facet for a tree.**

The implementation of facilities to support parameterized types has been considered elsewhere (Gries and Gehani, 1977). No particular problems appear to arise from the interaction of facets and parameterized types.

Compound objects, as discussed in section 3, can be passed as records of pointers. A record will have one field for each object in the package.

```
type tree (u: type) = ↑cell (u) ;

    cell (u:type) = record left, right: tree (u);
                        value (u)
                    end;

define inorder = iterable facet of [t:tree (u),s:stack (tree (u))] having

  type refstack = stack (tree (u));

  procedure cursor_init (var a:inorder);

          .
          .
          .
          .

  procedure next (var a : inorder) ;

    var temp : tree (u) ;

          .
          .
          .
          .
```

**Figure 11: Definition of a general tree and part of its *inorder iterable* facet.**

## 7. CONCLUSION

A facet is an alternative (often restricted) view of a data type. The view is expressed as a set of permitted operations. The extensions to a language needed to provide the facet facility are seen to be minimal. The resultant increase in the power of the language is, however, significant.

Security can be greatly enhanced by using facets to define safe interfaces between procedures.

Even where security is not important, facets are useful in mapping one view of data onto another. In particular, facets may be used to define a uniform restricted view of several types. Generic procedures can be written in terms of the uniform view. For example, a generic *for* may be written to iterate over components of a compound data structure. In some cases, several different forms of iteration for a single type may be desired (e.g., preorder and postorder for trees). A facet may be defined for each form of iteration required. Each facet defines a different interface to the generic *for*.

Many other facilities are desirable, though they do not enhance the basic power of the proposed features. These include new combining forms for facets and the parameterization of facets. We are currently investigating the use of facets in a parallel environment.

## ACKNOWLEDGEMENT

The authors would like to thank Dr M.R. Sleep for various suggestions made after reading an early draft of this paper. In particular, it was Dr Sleep who suggested that when facets are used in a network environment data need not be transported to an untrustworthy machine.

## REFERENCES

ADA (1979), Preliminary Ada reference manual, *ACM SIGPLAN Notices 14*, 6 Part A (June, 1979).

DAHL, O.-J., and HOARE, C.A.R. (1972), Hierarchical program structures, in *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, New York, 1972, pp. 175-220.

DAHL, O.-J., MYHRHAUG, B., and NYGAARD, K. (1970), The Simula 67 Common Base Language, Norwegian Computing Ctr., Oslo, 1970.

GESCHKE, C., and MITCHELL, J. (1975), On the problem of Uniform references to data structures. *IEEE Trans: Software Eng. SE-1*, 2 (June 1975), 207-219.

GRIES, D., and GEHANI, N. (1977), Some ideas on data types in high-level languages. *Comm. ACM 20*, 6 (June 1977), 414-420.

GUTTAG, J.V., HOROWITZ, E., and MUSSER, D.R. (1978), Abstract data types and software validation. *Comm ACM 21*, 12 (Dec. 1978), 1048-1064.

JENSEN, K., and WIRTH, N. (1974), PASCAL User Manual and Report. Springer-Verlag, Berlin, 1974.

JONES, A.K., and LISKOV, B.H. (1978), A language extension for expressing constraints on data access. *Comm. ACM 21*, 5 (May 1978), 358-367.

LISKOV, B., SNYDER, A., ATKINSON, R., and SCHAFFERT, C. (1977), Abstraction mechanisms in CLU. *Comm. ACM 20*, 8 (Aug. 1977), 564-576.

SHAW, M., WULF, W.A., and LONDON, R.L. (1977), Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Comm. ACM 20*, 8 (Aug. 1977), 553-564.

WELSH, J., and BUSTARD, D.W. (1979), Pascal-Plus — Another Language for Modular Multiprogramming. *Software Practice and Experience 9*, 11 (Nov. 1979), 947-958.

WIRTH, N. (1971), The programming language Pascal. *Acta Informatica 1* (1971), 35-63.

WULF, W.A., LONDON, R.L., and SHAW, M. (1976), An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Eng. SE-2*, 4 (Dec. 1976), 253-265.

## BIOGRAPHICAL NOTE

*F. Warren Burton received a B.S. degree in Applied Mathematics and an M.A. in Mathematics from Colorado University, both in 1972, and a Ph.D. in Computing from the University of East Anglia in 1977.*

*He was an Assistant Professor of Computer Science at Michigan Technological University for two and a half years, and joined the Computing Studies faculty of the University of East Anglia in January of 1979. His research interests include data structures, programming languages and computational geometry.*

*Brian Lings received a B.Sc. (Hon) degree in Pure Mathematics in 1971 and a Ph.D. in Computer Science in 1975, both from the University of East Anglia. He joined the Computer Science department at the University of Queensland, Australia, as a lecturer in 1975. His research interests include programming languages and data base management systems.*

# The Minimal Directed Spanning Graph for Combinatorial Optimization

Selim G. Akl*

This paper introduces a graph theoretic structure for combinatorial optimization: the minimal directed spanning graph. The new structure — a generalization of the minimal spanning tree for directed graphs — is used to design an approximation algorithm for the asymmetric travelling salesman problem. Experiments with the algorithm are described which suggest that future studies of the applicability of the new structure to the solution of other combinatorial optimization problems might prove worthwhile.

Short-form title: The Minimal Directed Spanning Graph.

Keywords and phrases: approximation algorithm, asymmetric travelling salesman problem, bipartite matching, combinatorial optimization, minimal directed spanning graph.

CR categories: 5.25, 5.39, 8.3

## 1. INTRODUCTION

The travelling salesman problem (TSP) is the problem facing a salesman who has to visit each of a number of cities exactly once and return to his point of departure while minimizing the cost of his trip. The TSP belongs to the infamous class of NP — hard problems[24] for which no polynomial-time algorithm is known. Existing algorithms have running times which grow exponentially with the number of cities[4,22]. For large problems this is impractical and *approximation algorithms* have been devised to yield a satisfactory — but not necessarily optimal — answer[9,20] The reader unfamiliar with the TSP should consult the excellent surveys in References 5 and 12. Some actual applications where the problem arises are discussed in Reference 19.

One interesting feature of the majority of published algorithms for the TSP is that they rely primarily on the symmetry of the cost matrix (i.e., the cost of going from city a to city b is equal to the cost of going from city b to city a). On the other hand, algorithms for general problems — which do not assume symmetry — behave very badly on symmetric cases[6,7]. In this paper, we describe an approximation algorithm especially designed for the directed (i.e., asymmetric) TSP. The algorithm is intimately related to the one for the symmetric case appearing in Reference 9 and uses the same basic principles.

In Section 2 the terminology is introduced and a graph-theoretic concept upon which our algorithm is based is defined. The algorithm is stated and analyzed in Section 3. Section 4 is devoted to the discussion of a topic related to our present study — the weighted bipartite matching problem. The results of an empirical analysis of the TSP algorithm are reported in Section 5.

## 2. DEFINITIONS

One way of expressing a TSP is through the use of a graph where nodes represent the cities and arcs the routes connecting them. For this reason, various concepts from graph theory have greatly influenced the work on approximation algorithms for the TSP. These include the minimal spanning tree, the optimal perfect matching, the Euler circuit and the Hamilton circuit. In particular, the contribution of the minimal spanning tree was twofold: it provided good estimates for the optimal tour[7,12], as well as efficient algorithms for an approximate solution[9,25]. Let us therefore recall some of these useful concepts from graph theory. The terminology we shall use is mostly from Reference 18 to which the reader is referred for the more fundamental definitions.

1. A *tree* is a connected graph which contains no cycles. Given an undirected connected graph G, a partial graph $G' \subseteq G$ which is a tree connecting together all nodes is called a *spanning tree.*

2. In a directed graph the number of arcs leaving a node is called the *out-degree* of that node. The *in-degree* of a node is the number of arcs entering that node. If for a given node the in-degree is larger than the out-degree we say that the node has an *out-degree deficiency* whose value is the in-degree minus the out-degree. The *in-degree deficiency* is defined similarly. A graph is *balanced* when for each node the in-degree equals the out-degree.

3. A *directed tree* is either rooted to a node or from a node. A tree rooted from a node is a tree in which the in-degree of that node is zero and the in-degree of each of the other nodes is at most one. A tree rooted to a node is a tree in which the out-degree of that node is zero and the out-degree of the other nodes is at most one. A *directed spanning tree* (rooted to or from a node) is a directed graph whose underlying undirected graph is a spanning tree.

4. A graph in which a number $w_{ij}$ is associated with every arc (i,j) in the graph is called a *weighted graph* and the number $w_{ij}$ is called the *weight* of arc (i,j). A *minimal spanning tree* (MST) is that spanning tree with the minimum sum of arc-weights. The *minimal directed spanning tree* (MDST) is defined similarly. The *cost matrix* of an n-city TSP is an nxn matrix whose (i,j)th entry is



*The author is with the Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada. *This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant NSERC-A3336. Manuscript received 16 October, 1979. Revised 18 August, 1980.*

the cost of going from city i to city j. This cost is equal to $w_{ij}$ the weight of arc (i,j) in the corresponding directed graph.

5. An *Euler circuit* of a directed graph is a circuit such that every arc of the graph appears on it exactly once. A directed graph whose underlying undirected graph is connected possesses an Euler circuit if and only if it is balanced. Such a graph is termed *Eulerian.*

6. A *Hamilton circuit* of a directed graph is a circuit such that every node of the graph appears on it exactly once. The solution to the TSP defined on a directed graph is a minimum-weight Hamilton circuit.

7. A *bipartite graph* is a graph whose nodes can be partitioned into two sets $T_1$ and $T_2$ such that no two nodes in $T_1$ or in $T_2$ are adjacent (i.e., all arcs extend "between $T_1$ and $T_2$"). A subset of the arcs of a bipartite graph is said to be a *matching* if no two arcs in it are incident to the same node. A matching which 'covers' all nodes is said to be complete.

8. A directed graph is said to be *complete* if for any two of its nodes i and j there exists an arc connecting i to j. A complete directed graph on n nodes has therefore n(n-1) arcs.

9. Finally, we introduce a new definition of our own: a *minimal directed spanning graph* (MDSG) is a partial graph of a complete directed and weighted graph on n nodes which has minimum weight and whose underlying undirected graph is connected and acyclic. This is equivalent to saying that the underlying graph is an MST in which arc (i,j) is such that $w_{ij}=\min(w_{ij},w_{ji})$, where (i,j) and (j,i) are arcs of the complete graph. It should be noted that the MDSG differs from the MDST in that it:

(a) is unrooted, and
(b) has no restrictions imposed on the in- or out-degree of its nodes.

This concept of an MDSG will be used in the next section to develop an approximation algorithm for the asymmetric TSP.

## 3. AN ALGORITHM FOR THE DIRECTED TSP

Given a complete directed and weighted graph G with n nodes, the algorithm below computes a nearly optimal solution to the TSP defined on G. For clarity of presentation, step-by-step comments follow the algorithm.

### 3.1 Algorithm S

**Step 1:** Obtain an MDSG.

**Step 2:** Add a set of arcs to the MDSG in order to make the directed graph thus obtained Eulerian.

**Step 3:** Find an Euler circuit in this directed graph.

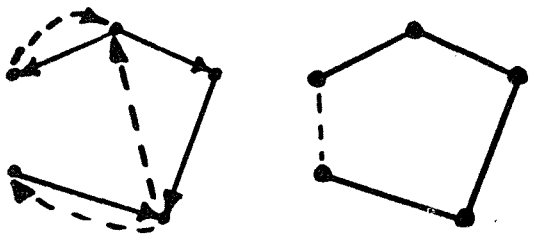
**Step 4:** This Euler circuit can be used (in a manner described below) to derive several Hamilton circuits: among all such Hamilton circuits, choose the one with minimum weight. Stop.

### 3.2 Comments

**Step 1:** If we replace every $w_{ij}$ by $\min(w_{ij},w_{ji})$ in the cost matrix, and apply an MST algorithm[23] on the new matrix we obviously get an MST with the same weight as the MDSG. It is important to store along with every entry in the cost matrix the direction of the edge having that cost, i.e., i→j or j→i.

**Step 2:** Since $\Sigma[(\text{in - degree deficiencies}) - (\text{out-degree deficiencies})]=0$, the addition to the MDSG of arcs leaving nodes with out-degree deficiency and entering nodes with in-degree deficiency yields a balanced graph which is also connected and hence Eulerian. Again, as in Step 1, the arc directions must be kept. Note that choosing an arbitrary set of arcs to obtain a balanced graph is a straightforward matter. An algorithm for obtaining a 'good' set of arcs is discussed in the next section.

**Step 3:** A simple algorithm will generate the Euler circuit[8]. Here we note the importance of keeping the arc directions as mentioned in Step 1 and 2. This is clear from the example in Fig. 1: the Euler circuits resulting from graphs (a) and (b) in Fig. 1 will be quite different.



—— MDSG obtained in Step 1
– – – Arcs added in Step 2
(a)

—— MST obtained in Step 1
– – – Arc added in Step 2
(b)

**Figure 1**

**Step 4:** The method of getting a Hamilton circuit from the Euler circuit is simple. Assume the Euler circuit is $(n_1,n_2,\ldots,n_{l-1},n_l)$ where the $n_i$'s represent the nodes of the graph and are not necessarily distinct. Since an Euler circuit visits every node at least once, one can build a Hamilton circuit by: starting at node $n_1$, moving to the right and introducing a node in the Hamilton circuit only if it appears for the first time. In order to generate *all* Hamilton circuits obtainable from the Euler circuit, two copies of the Euler circuit are placed contiguously, $n_1,n_2,\ldots,n_{l-1},n_l,n_1,n_2,\ldots,n_{l-1},n_l$, and the method just described is applied repeatedly $l$ times, every time using the next node in $(n_1,n_2,\ldots,n_l)$ as the starting node.

### 3.3 Complexity

In Step 1, the MDSG can be obtained in $O(n^2)$ time if — as mentioned above — a modified-MST algorithm is used. A straightforward implementation of Step 2 will run in linear time: create a list of nodes with out-degree deficiency and a list of nodes with in-degree deficiency; then

TABLE 1

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | ∞ | 7 | 65 | 68 | 34 | 81 |
| 2 | 19 | ∞ | 22 | 27 | 59 | 29 |
| 3 | 14 | 43 | ∞ | 62 | 77 | 65 |
| 4 | 76 | 53 | 64 | ∞ | 6 | 51 |
| 5 | 39 | 58 | 38 | 27 | ∞ | 13 |
| 6 | 46 | 67 | 27 | 11 | 38 | ∞ |
| | 1 | 2 | 3 | 4 | 5 | 6 |

match a node from the first list with the first available node on the second list. Similarly, Step 3 is O(n): the MDSG has (n-1) edges; at most as many edges will be added in Step 2 and the resulting Eulerian graph will have O(n) edges which can be traversed in linear time. Step 4 is obviously $O(n^2)$. The overall time-complexity for the algorithm is therefore quadratic in n. The memory requirement is also $O(n^2)$ when the cost matrix is stored in core.

### 3.4 Example

We now illustrate the operation of the algorithm by trying it on a small problem published in the literature[6]. For the cost matrix of Table 1 the best known TSP tour is (1,2,4,5,6,3,1) with cost 94 units.

The step-by-step solution using algorithm S is as follows:

**Step 1:** Obtain an MDSG; this is shown in Fig. 2.

**Step 2:** Nodes with out-degree deficiency = {4,5}.
Nodes with in-degree deficiency = {3,6}.
The set of additional arcs is {(4,3), (5,6)}
The Eulerian graph is shown in Fig. 2.

**Step 3:** An Euler circuit of the graph in Fig. 3 is (1,2,4,5,6,4,3,1).

**Step 4:** There are two Hamilton circuits obtainable from the Euler circuit.
1) (1,2,4,5,6,3,1) with cost 94 units.
2) (5,6,4,3,1,2,5) with cost 168 units.
The first circuit is chosen as the answer.

## 4. AN APPROXIMATION ALGORITHM FOR THE WEIGHTED BIPARTITE MATCHING PROBLEM

It is not difficult to observe that a large number of arcs in the final answer yielded by algorithm S will be contributed by the MDSG and the set of arcs added in Step 2. It is therefore a good idea to try to make the weight of the set of arcs obtained in Step 2 as small as possible. By taking a number of copies of each node equal to its in- or out-degree deficiency the problem is easily reduced to the weighted bipartite matching problem[18]:

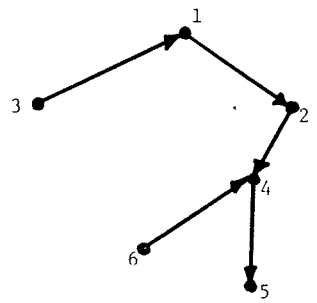



Figure 2

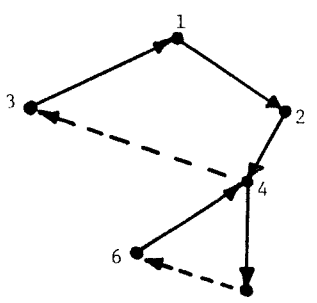



Figure 3

Given an arc-weighted bipartite graph, find a complete matching for which the sum of the weights of the arcs is minimum. There is an elegant solution to this problem which runs in $O(n^3)$ time[16]. For large values of n, however, this would probably be prohibitive. In practice, it is important that we keep the complexity of our algorithm within the quadratic bound. An $O(n^2)$ approximation algorithm for the weighted bipartite matching problem which yields a good but not necessarily optimal answer should be considered (an exact $O(n^2)$ algorithm is not known to exist).

Let N and T be the lists of nodes with in- and out-degree deficiency respectively after Step 1 of algorithm S. Let N' and T' be the lists obtained when N and T are augmented as follows: if a node has an in-degree (out-degree) deficiency equal to x, then (x-1) copies of that node are added to its list. Note that N' and T' have the same cardinality. Now consider the complete bipartite graph consisting of the two sets of nodes N' and T' and a set of arcs X such that each node in T' is connected to all nodes in N'. The algorithm below will yield an approximate solution to the matching problem. We should point out that the algorithm is of the 'greedy' type[11] and is based on the same ideas as the one for the nonbipartite matching problem described in Reference 2.

> "Select a node i $\epsilon$ T' at random. Choose the arc (i,j) — where j $\epsilon$ N' — of minimum weight incident to i and add it to the matching. Delete nodes i and j and all incident arcs. Repeat until all nodes have been matched."

This algorithm is simple to implement and experiments showed it to be quite efficient. It suffers, however, from a serious drawback: in some instances "greed does not pay" and the algorithm is often forced to make very bad choices towards the end of its task. A modification of this algorithm is now presented in an attempt to cure this weakness. The idea is to try to match, early enough in the procedure, those nodes to which some heavily-weighted arcs are incident, thereby reducing the number of bad choices at the final stages.

### 4.1 Algorithm M

**Step 1:** For each node i $\epsilon$ T' find $w_i$ the sum of weights of the arcs in X incident at i.

**Step 2:** Find the node j $\epsilon$ T' for which $w_j \geqslant w_k$ for all k $\epsilon$ T' (resolve ties arbitrarily).

**Step 3:** Choose the arc (j,i) such that $w_{ji} \leqslant w_{jk}$ where i,k $\epsilon$ N', and i and k are not yet in the matching.

**Step 4:** Arc (j,i) is added to the matching, $w_j$ is set to zero and $w_k$ is replaced by $w_k - w_{ki}$ for all unmatched k $\epsilon$ T'.

**Step 5:** If any nodes are still unmatched go to Step 2. Else stop.

The set of arcs obtained at exit from this algorithm is the one to be used in Step 2 of algorithm S.

### 4.2 Complexity

Each of Steps 2, 3 and 4 consists (at most) of n operations and is executed n times. The algorithm has therefore a complexity of $O(n^2)$.

### 4.3 Example

For the cost matrix of Table 2 the best known directed TSP tour is (1,3,9,4,8,5,10,6,7,2,1) with cost 146

TABLE 2

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ∞ | 24 | 18 | 22 | 31 | 19 | 33 | 25 | 30 | 26 |
| 2 | 15 | ∞ | 19 | 27 | 26 | 32 | 25 | 31 | 28 | 18 |
| 3 | 22 | 23 | ∞ | 23 | 16 | 29 | 27 | 18 | 16 | 27 |
| 4 | 24 | 31 | 18 | ∞ | 19 | 13 | 28 | 9 | 19 | 27 |
| 5 | 23 | 18 | 34 | 20 | ∞ | 31 | 24 | 15 | 25 | 8 |
| 6 | 24 | 12 | 17 | 15 | 10 | ∞ | 11 | 16 | 21 | 31 |
| 7 | 28 | 15 | 27 | 35 | 19 | 18 | ∞ | 21 | 21 | 19 |
| 8 | 13 | 24 | 18 | 13 | 13 | 22 | 25 | ∞ | 29 | 24 |
| 9 | 17 | 21 | 18 | 24 | 27 | 24 | 34 | 31 | ∞ | 18 |
| 10 | 18 | 19 | 29 | 16 | 23 | 17 | 18 | 31 | 23 | ∞ |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

units[17].

The behaviour of algorithm S (using algorithm M in Step 2) on this cost matrix is outlined below.

**Step 1:** Obtain an MDSG; this is shown in Fig. 4.

**Step 2:** The set of additional arcs is given by {(1,6),(2,3),(5,8),(5,4),(7,6),(9,3),(10,6)}

**Step 3:** Obtain an Euler circuit in the graph of Fig. 5; this is given by (1,6,2,3,9,3,5,4,8,5,10,6,7,6,5,8,1).

**Step 4:** The minimum-weight Hamilton circuit obtainable from the Euler circuit in Step 3 is (1,2,3,9,4,8,5,10,6,7,1) with cost 169 units.

## 5. EXPERIMENTS

In dealing with a class of combinatorial problems — like the TSP — for which all known exact algorithms have a running time that grows exponentially with the size of the input, it is often useful to estimate the expected solution, or put some bounds on it. This estimate (or bound) can serve several purposes:

(1) In some distribution management problems it is sometimes necessary to estimate the expected distance that would be involved in supplying customers — when the exact locations of the customers are not known in advance — in order to decide, for example, upon the number and locations of depots.

(2) The branch-and-bound approach[17] uses lower bounds to eliminate from further consideration whole parts of the decision tree that would otherwise have to be investigated.

(3) Finally, and most important for our purpose, when an approximation algorithm is tested, a lower bound serves as a reference point against which near-optimal solutions are compared.

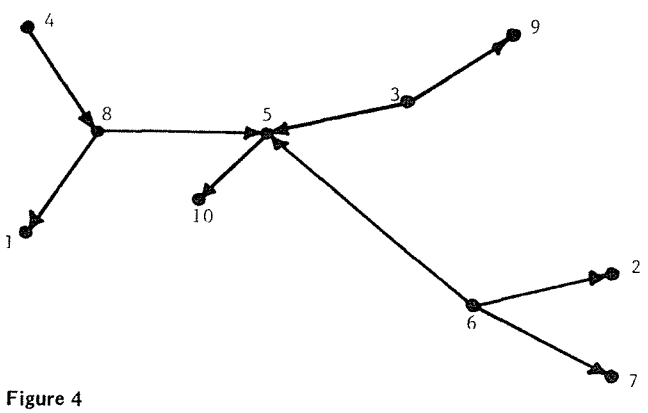



Figure 4

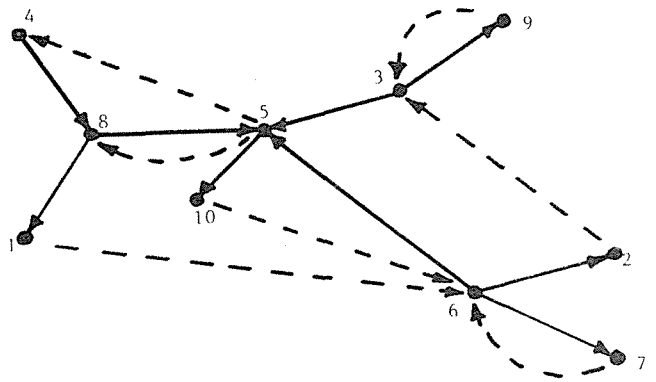



Figure 5

In the case of the symmetric TSP, a variety of estimates of (and bounds on) an optimal tour have been derived[3,7,12,13,14]. This, and the fact that a set of standard problems of up to a few hundred cities appear in the literature[15,21], that can be used for comparison purposes, usually make the task of evaluating an approximation algorithm for the symmetric TSP a relatively easy one.

Quite surprisingly, very few asymmetric cost-matrices for the TSP have been published and even these are for trivial values of the number of cities[1,6,10,17,22,26,27]. Furthermore, in contrast with the symmetric case, it is quite complicated to derive an estimate of the solution, or put some bound on it, for general directed TSPs. Of course, a variety of lower bounds can be obtained for a particular instance of the TSP by solving the corresponding assignment problem[6,7]. These, however, usually involve non-trivial computations which would defeat the purpose of a fast approximation algorithm.

In this section we propose to empirically estimate the quality of the answer provided by algorithm S. In order to do so we shall need — as noted above — a reference point against which our approximate solution is to be compared. It is interesting to observe that a lower bound on the weight of the optimal tour for a directed TSP is the weight of the MDSG. (To see this, remove the arc of largest weight from the optimal tour: what is left is a directed spanning graph whose weight is larger than or equal to that of the MDSG.) The computation of this bound is not only simple but also a basic step of the algorithm we are trying to evaluate.

Algorithm S was tested on random asymmetric cost matrices. Matrix entries were selecred uniformly from the interval (0,1) by a uniform random number generator. Let

n = number of cities,
S = weight of the approximate solution provided by algorithm S,
MDSG = weight of the MDSG
and R = S/MDSG.

Table 3 summarizes the result of a Monte Carlo experiment where E(R) and SD(R) are average value and standard deviation of R computed over 100 randomly generated problems with asymmetric cost matrices.

TABLE 3

| n | E(R) | SD(R) |
|---|---|---|
| 50 | 1.38 | 0.04 |
| 100 | 1.30 | 0.04 |
| 150 | 1.21 | 0.03 |
| 200 | 1.16 | 0.03 |

As Table 3 shows, the answer provided by algorithm S for $n \leqslant 200$ is on the average no worse than 3/2 times the lower bound.

In conclusion, we mention that various algorithms using the ideas presented in this paper are currently being developed to address a number of related combinatorial optimization problems.

## REFERENCES


1. ACKOFF, R.L. and SASIENI, M.W., Fundamentals of Operations Research, Wiley, New York, 1968.
2. AVIS, D., and AKL, S.G., An Empirical Study of Heuristics for the Weighted Matching Problem, unpublished manuscript.
3. BEARDWOOD, J., HALTON, H.H., and HAMMERSLEY, J.M., The Shortest Path Through Many Points, *Proceedings of the Cambridge Philosophical Society*, 55, 1959, pp. 299-327.
4. BELLMAN, R., Dynamic Programming Treatment of the Travelling Salesman Problem, *Journal of the ACM*, 9, 1962, pp. 61-63.
5. BELLMORE, M., and NEMHAUSER, G.L., The Travelling Salesman Problem: A Survey, Operations Research, 16, 1968, pp. 538-558.
6. BELLMORE, M., and MALONE, J.C., Pathology of Travelling Salesman Subtour-Elimination Algorithms, Operations Research, 19, 1971, pp. 278-307.
7. CHRISTOFIDES, N., Bounds for the Travelling Salesman Problem, Operations Research, 20, 1972, pp. 1044-1056.
8. CHRISTOFIDES, N., Graph Theory: An Algorithmic Approach, Academic Press, New York, 1975.
9. CHRISTOFIDES, N., Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem, *Management Sciences Research Report* No. 388, Carnegie Mellon University, February 1976.
10. CONWAY, R.W., MAXWELL, W.L., and MILLER, L.W., Theory of Scheduling, Addison Wesley, Reading, Massachusetta, 1967.
11. EDMONDS, J., Matroids and the Greedy Algorithm, *Mathematical Programming*, 1, 1971, pp. 127-136.
12. EILON, S., WATSON-GANDY, C.D.T., and CHRISTOFIDES, N., Distribution Management, Griffin, London, 1971.
13. FEW, L., The Shortest Path and the Shortest Road Through n Points, *Mathematika*, 2, 1955, pp. 141-144.
14. HAMMERSLEY, J.M., and HANDSCOMB, D.C., Monte Carlo Methods, Methuen, London, 1964.
15. KARG, R.L., and THOMPSON, G.L., A Heuristic Approach to Solving Travelling Salesman Problems, *Management Science*, Vol. 10, No. 2, 1964, pp. 225-248.
16. KUHN, H.W., The Hungarian Method for the Assignment Problem, *Naval Res. Logist. Quart.*, 2, 1955, pp. 83-97.
17. LAWLER, E.L., and WOOD, D.E., Branch-and-Bound: A Survey, Operations Research, 14, 1966, pp. 699-719.
18. LAWLER, E.L., Combinatorial Optimization: Networks and Matroids, Holt, Rinehart and Winston, New York, 1976.
19. LENSTRA, J.K., and RINNOOY KAN, A.H., Some Simple Applications of the Travelling Salesman Problem, *Operational Research Quarterly*, Vol. 26, No. 4, 1975, pp. 717-733.
20. LIN, S., Computer Solutions of the Travelling Salesman Problem, *Bell System Technical Journal*, Vol. 44, 1965, pp. 2245-2269.
21. LIN, S., and KERNIGHAN, B.W., An Effective Heuristic Algorithm for the Travelling Salesman Problem, Operations Research, 21, 1973, pp. 498-516.
22. LITTLE, J.D.C., MURTY, K.G., SWEENEY, D.W., and KAREL, C., An Algorithm for the Travelling Salesman Problem, Operations Research, 11, 1963, pp. 972-989.
23. PRIM, R.C., Shortest Connection Networks and Some Generalizations, *Bell System Technical Journal*, 36, 1957, pp. 1389-1401.
24. REINGOLD, E.M., NEIVERGELT, J., and DEO, N., Combinatorial Algorithms Theory and Practice, Prentice Hall, Englewood Cliffs, New Jersey, 1977.
25. ROSENKRANTZ, D.J., STEARNS, R.E., and LEWIS, R.M., An Analysis of Several Heuristics for the Travelling Salesman Problem, *SIAM Journal on Computing*, Vol. 6, No. 3, 1977, pp. 563-581.
26. SASIENI, M., YASPAN, A., and FRIEDMAN, L., Operations Research, Wiley, New York, 1959.
27. WAGNER, H.M., Principles of Operations Research, Prentice Hall, Englewood Cliffs, New Jersey, 1969.


## BIOGRAPHICAL NOTE

*Selim G. Akl received the B.Sc. and M.Sc. degrees in Electrical Engineering in 1971 and 1975 respectively, both from the University of Alexandria and the Ph.D. degree in Computer Science from McGill University in 1978. He is currently an Assistant Professor of Computing and Information Science at Queen's University in Kingston. His research interests are primarily in the area of algorithm design and analysis, in particular for problems in combinatorics, artificial intelligence and pattern recognition. Dr Akl is a founding member of the Canadian Applied Mathematics Society, a member of the Canadian Information Processing Society, the IEEE Computer Society and the ACM.*

# Marginal Totals for Multidimensional Arrays

John Burr*

Efficient procedures are derived for computing arrays of marginal totals S from any given multidimensional array X, for use when the number of dimensions in X or S is unknown at the time of writing the program. Generalized transposition of X is included as a special case.
Keywords: marginal totals, multidimensional array, transposition.
CR category: 4.22

## 1. INTRODUCTION

This paper develops two efficient procedures for computing arrays of marginal totals S from any given multidimensional array X. This problem is of special interest in those cases where the number of dimensions in X or S, that is, the number of subscripts needed to identify any one element of the array, is unknown at the time of writing the program. The same procedure can be used to effect generalized transposition of X, which is merely the special case when S has the same number of dimensions as X.

The first procedure runs sequentially through the elements of X, and this will be the natural choice when main memory can hold S but is too small to hold the whole of X. The second procedure runs sequentially through the elements of S, that is, it sums all those elements of X that contribute to a given element of S before setting to work on the next element of S. This will be the natural choice if the elements of S are to be used without necessarily being stored, for example, if we only wish to know the sum of the squares of the elements of S. When memory size is not a constraint, the second procedure will usually execute a little faster than the first.

This problem occurs in survey analysis, and in the analysis of variance of complete factorial experiments. I developed these procedures for use in my own programs 15 years ago, and I feel sure that other programmers must be using them also, but I have never seen them in published form. There is some slight overlap with the work of Guttmann (1976) and Meyer (1978) on multiple sums of a function with many arguments.

## 2. TERMINOLOGY AND NOTATION

I propose to borrow two words, *factors* and *levels*, from the terminology of factorial experiments. My aim is to avoid the ambiguity associated with the term "dimension", by replacing it with a term having more concrete associations. As an example, an agricultural field trial may be designed to compare yields of wheat with four different strengths of application of a fertilizer. We then say that "Fertilizer" is a factor with four levels. The same experiment might also employ "Depth of ploughing" as a factor with two levels, "Harvest time" as a factor with three levels, "Varieties" (of wheat) as a factor with five levels, and "Blocks" as a factor with two levels. In a complete factorial experiment, all possible combinations of levels occur, so that the design described above would be called a five-factor experiment involving

$$4 \times 2 \times 3 \times 5 \times 2 = 240 \text{ observations.}$$

Notice that the five varieties will probably be labelled 1, 2, 3, 4 and 5 in some arbitrary order, the labels having no quantitative significance, but we nevertheless say that this factor "occurs at five levels". It is convenient to abbreviate each factor name by its initial letter, so that the five-dimensional array X may be called FDHVB. If we now refer to an array of marginal totals (or means) by the name FHV, it is at once apparent that this will be a three-dimensional array obtained by summing (or averaging) over all levels of D and B. In the sequel, I shall sometimes refer to the factors (or dimensions) of X by the names A, B, C, D, . . . in that order.

In the algorithms and program fragments in this paper, all variables and arrays, other than formal parameters, are treated as global. V is a real one-dimensional array large enough for the physical storage of both X and S, and T is a real variable. (Alternatively, V, X, S and T could be all integer, or all double precision, or all complex, but for definiteness I shall call them real.) All other variables and arrays are of type integer, and most of these are described below.

| | | |
|---|---|---|
| nfx | = | number of factors in X. |
| nfs | = | number of factors in S. |
| nfns | = | number of factors not in S |
| | = | nfx − nfs. |
| k | = | index ranging from 1 to nfx. |
| lev(k) | = | number of levels of the kth factor in X. |
| jmax(k) | = | lev(k) − 1. |
| j(k) | = | kth subscript value in X, with range 0 to jmax(k). |
| nx | = | number of elements in X |
| | = | product of lev(1), . . . , lev(nfx). |
| ns | = | number of elements in S. |
| locx | = | starting point for X in the array V. |
| locs | = | starting point for S in the array V. |
| jx | = | index ranging from locx to locx + nx − 1. |
| js | = | index ranging from locs to locs + ns − 1. |

I have adopted a convention compatible with Fortran, such that as we scan the elements of X from left to right starting at V(locx), where every j(k) is initially zero,



*Department of Computing Science, University of New England, Armidale, NSW, 2351. *Manuscript received 25 July 1980.*

the values of j(1) will cycle most rapidly, the values of j(2) will cycle less rapidly than those of j(1) but more rapidly than those of j(3), and so on. Hence the general element of X, with subscripts j(1), . . . , j(nfx), will be physically stored at V(jx), where

$$jx = locx + mx(1)*j(1) + \ldots + mx(nfx)*j(nfx)$$

and the multipliers mx(k) are defined by:

$$mx(1) = 1,$$
$$mx(k) = mx(k-1)*lev(k-1), \quad k = 2, \ldots, nfx.$$

Other integer arrays will be described as the need arises.

To improve readability, I sometimes use hyphens within procedure names, and I use closing delimiters endif and endfor to avoid the ugly proliferation of the delimiters begin . . . end.

## 3. SUMMING SEQUENTIALLY IN X

If we knew the number of factors in X to be 4, for example, we could use the following loops to generate marginal totals in S, after some suitable initialization (to be described below).

```
jx ← locx;
for j(4) ← 0 to jmax(4) do
  for j(3) ← 0 to jmax(3) do
    for j(2) ← 0 to jmax(2) do
      for j(1) ← 0 to jmax(1) do
        js ← locs + ms(1)*j(1) + ms(2)*j(2)
                  + ms(3)*j(3) + ms(4)*j(4);
        V(js) ← V(js) + V(jx); jx ← jx + 1
      endfor
    endfor
  endfor
endfor
```

The elements of S, that is, the ns elements of V starting at V(locs), must be initially set to zero, and the multipliers ms must be initialized in the manner described below. I give first a simple example, then the general case.

Suppose that the factor name of X is ABCD and the factor name of S is CAD, which means that we are not only summing over the levels of B, but also transposing so that C becomes the most rapidly varying factor in S. Here ms(3) is 1, since js must be increased by 1 whenever j(3) increases by 1. As we scan the elements of S from left to right, we see that j(1), the subscript for A, only changes after j(3) has run through its full range, that is, after lev(3) steps. Hence ms(1) must equal lev(3). We can determine in like fashion that ms(4) must equal lev(3)*lev(1). Finally, ms(2) must be zero, since if two sets of subscript values for the array X differ only in the second subscript j(2), then the corresponding elements of X must map into the same element of S.

To generalize and automate the above, we need a concise way of numerically coding the description of S. I propose to use an array fs (acronym for factors of S) with nfs elements. The ith element fs(i) will contain an integer k denoting that the ith factor in the name of S is the same as the kth factor in the name of X. Hence, in the example given above, we have:

```
fs(1) = 3,      ms(3) = 1
fs(2) = 1,      ms(1) = ms(3)*lev(3)
fs(3) = 4,      ms(4) = ms(1)*lev(1)
```

This leads easily to the general algorithm for setting up the array ms:

```
procedure setup-ms;
begin
      for k ← 1 to nfx do ms(k) ← 0 endfor;
      m ← 1;
      for i ← 1 to nfs do
            k ← fs(i); ms(k) ← m; m ← m*lev(k)
      endfor
end
```

The next step toward a general marginal-total algorithm is to expand the four "for" and "endfor" statements, as shown below:

```
L4: j(4) ← 0;
L3: j(3) ← 0;
L2: j(2) ← 0;
L1: j(1) ← 0;
L0: . . . . . .
      if j(1) < jmax(1) then j(1) ← j(1) + 1; go to L0 endif;
      if j(2) < jmax(2) then j(2) ← j(2) + 1; go to L1 endif;
      if j(3) < jmax(3) then j(3) ← j(3) + 1; go to L2 endif;
      if j(4) < jmax(4) then j(4) ← j(4) + 1; go to L3 endif
```

This is easily generalized to the case where nfx, the number of factors in X is not necessarily 4:

```
      k ← nfx + 1;
L10: k ← k − 1; j(k) ← 0;
L20: if k > 1 then go to L10 endif;
      . . . . . .
L30: if j(k) < jmax(k) then
            j(k) ← j(k) + 1; go to L20 endif;
      k ← k + 1; if k ≤ nfx then go to L30 endif
```

It remains only to improve efficiency by removing the lengthy calculation of js from the innermost loop. We can achieve this, and at the same time remove all the multiplications, by observing that whenever j(k) is incremented by 1, js needs to be increased by ms(k), and whenever j(k) is reduced from jmax(k) to zero, js must be decremented by a suitably initialized array element that I shall call jsdec(k). Since the purpose of this decrement is to undo the effect of jmax(k) increments each of size ms(k), it is clear that the value of jsdec(k) must be ms(k)*jmax(k). This decrementing can be done just before the statement k ← k +1 in the last line of the program fragment shown above. Also, if we insert the statement j(k) ← 0 in the same place, then the two lines labelled L10 and L20 can be removed from the main loop, with a further gain in speed of execution. The final version of the algorithm is shown below.

```
procedure margin-1;  (**sequential in X**)
begin
      setup-ms;
      for k ← 1 to nfx do
            j(k) ← 0; jsdec(k) ← ms(k)*jmax(k)
      endfor;
      for js ← locs to locs + ns − 1 do V(js) ← 0 endfor;
      jx ← locx; js ← locs;
            (**summation loop begins**)
L1:   V(js) ← V(js) + V(jx); jx ← jx +1; k ← 1;
```

```
L2:   if j(k) < jmax(k) then
            j(k) ← j(k) + 1; js ← js + ms(k);
            go to L1 endif;
      j(k) ← 0; js ← js − jsdec(k); k ← k + 1;
      if k ⩽ nfx then go to L2 endif
end
```

## 4. SUMMING SEQUENTIALLY IN S

Consider first the example from the last section, where the factor names of X and S are ABCD and CAD.

```
js ← locs;
for j(4) ← 0 to jmax(4) do
  for j(1) ← 0 to jmax(1) do
    for j(3) ← 0 to jmax(3) do
      T ← 0;
        for j(2) ← 0 to jmax(2) do
          jx ← locx + mx(2)*j(2) + mx(3)*j(3)
                    + mx(1)*j(1) + mx(4)*j(4);
          T ← T + V(jx)
        endfor;
      V(js) ← T; js ← js +1
    endfor
  endfor
endfor
```

In preparation for generalizing this, I introduce an array named perm whose first four elements, in this example, will be 2, 3, 1 and 4. In general, perm will have nfx elements, the last nfs of which point to the nfs factors occurring in the name of S, while the first nfx-nfs elements point to the factors of X that do not appear in S. For setting up the array perm, it is useful to have a boolean function named member(v, n, k) that returns the value true or false accordingly as k is or is not a member of the set v(1), . . . , v(n):

```
boolean function member(v, n, k);
begin
  for i ← 1 to n do
    if v(i) = k then return true endif
  endfor;
  return false
end

procedure setup-perm;
begin il ← 0; i2 ← 0; nfns ← nfx − nfs;
  for k ← 1 to nfx do
    if member(fs, nfs, k) then
      i2 ← i2 + 1; perm(nfns + i2) ← fs(i2)
    else il ← il + 1; perm(il) ← k endif
  endfor
end
```

I now introduce an array h such that h(k) = j(perm(k)), so that the nfx elements of h are a permutation of the subscripts j(1), . . . , j(nfx). The corresponding permutations of the upper bounds, increments and decrements will be denoted by hmax, mh and hxdec. This leads to the general algorithm shown below. Note that the procedures transpose-x and sum-x could easily be combined, since the first is only a special case of the second, but this would lead to some loss in clarity and an increase in execution time.

```
procedure margin-2;  (**sequential in S**)
begin  setup-perm;
  for k ← 1 to nfx do
    h(k) ← 0; i ← perm(k); mh(k) ← mx(i);
    hmax(k) ← jmax(i); hxdec(k) ← hmax(k)*mh(k)
  endfor;
  jx ← locx; js ← locs;
  if nfs = nfx then transpose-x else sum-x endif
end

procedure sum-x;
begin   T ← 0;
L1:   T ← T + V(jx); k ← 1;
L2:   if h(k) < hmax(k) then
            h(k) ← h(k) + 1; jx ← jx + mh(k);
            go to L1 endif;
      if k = nfns then
            V(js) ← T; T ← 0; js ← js + 1 endif;
      h(k) ← 0; jx ← jx − hxdec(k); k ← k + 1;
      if k ⩽ nfx then go to L2 endif
end

procedure transpose-x;  (**degenerate form of sum-x**)
begin
L1:   V(js) ← V(jx); js ← js + 1; k ← 1;
L2:   if h(k) < hmax(k) then
            h(k) ← h(k) + 1; jx ← jx + mh(k);
            go to L1 endif;
      h(k) ← 0; jx ← jx − hxdec(k); k ← k + 1;
      if k ⩽ nfx then go to L2 endif
end
```

## 5. IMPLEMENTATION

After completing the procedures margin-1 and margin-2 in the forms shown above, I translated them into Fortran subroutines and successfully tested them on the University's DEC 2060. Execution time in microseconds was found to be approximately

$$72 + K*nx,$$

where the constant 72 is accounted for by the CALL and RETURN statements, and the multiplier K ranges from about 9 to 16 microseconds depending principally on the value of hmax(1) or jmax(1). The procedure margin-2 was generally about five to 15 percent faster than margin-1, except when the values of lev(1), . . . , lev(nfx) were only 2 or 3, when margin-2 was sometimes a little slower.

## REFERENCES

GUTTMANN, A.J. (1976), Multi-dimensional summations in Fortran. *Software – Practice and Experience*, Vol. 6, p. 221.

MEYER, B. (1978), A note on computing multiple sums. *Software – Practice and Experience*, Vol. 8, p. 3.

## BIOGRAPHICAL NOTE

*John Burr obtained his M.Sc. in Mathematics from the University of Queensland in 1954, and his Ph.D. in Mathematical Statistics from the University of New England in 1963. He has been Professor of Computing Science at the University of New England since 1973. His chief field of interest is applications software.*

# Distributed Computing and its Competitors

L.M. Casey*

With the falling cost of processors and memory modules there has been a rising interest in their interconnection to form larger systems. This paper surveys the factors affecting general performance for the following three architectural options:
(i) High speed 'single' main processor.
(ii) Multiple processors connected to a common memory.
(iii) Multiple computers connected by a high speed (short distance) communication link.
Both hardware and software issues are covered.
Keywords and Phrases: Distributed computing, multiprocessor, performance, concurrency.
CR Categories: 4.32

## 1. INTRODUCTION

The rising performance and cost effectiveness of microprocessors will lead in the future to the widespread use of private dedicated computers. However, there will still be a large demand for shared computing facilities such as those provided by many of today's mainframes. Shared centralised databases and expensive high bandwidth peripherals provide an incentive for sharing computing power. Individual microprocessor systems may not be able to provide users with as good response times as a more powerful shared facility. There are many half-hidden costs in operating and maintaining diverse, dispersed, small systems. Mindful of these, an organisation may wish to keep its computing centralised.

Multiple microprocessors could be used in the provision of the larger scale computing facility. Indeed, the most economic of future systems may be those on a single undiced wafer, the individual (good) chips on the wafer being interconnected to make the complete system. This paper surveys the hardware and software issues involved in three possible types of system. These three types are:

### 1. The Uniprocessor System

This is the conventional computer. The central processing unit may be constructed with conventional random logic, gate arrays and/or bit slices. Within the central processing unit there may be more than one arithmetic and logic unit. The essential characteristics of a uniprocessor system is that it executes a single stream of instructions.

### 2. The Multiprocessor System

A multiprocessor system has a number of processing units, each of which executes its own stream of instructions. All processors share a common primary memory. This is often called tight coupling. A multiprocessor system has a single operating system.



### 3. The Distributed System

In a distributed computer system a number of separate computers interact with each other by means of a fast communications subsystem. It is often called a loosely coupled system. Each computer processes its own instruction stream. Given the requirement of speedy communication it is unlikely that the computers will be geographically dispersed. Indeed in the near future the communication subsystem could be a bus laid out on a single board. Concomitant with the hardware structure the operating system must be constructed in such a way as to present the user with a single unified system. This attribute is frequently missing from systems labelled 'distributed' (Enslow, 1978).

This paper concentrates on the provision of general purpose computing power, the type of computing provided for time shared systems by today's mainframes and larger minis. Many of the points made are also applicable to other environments such as process control. Any one of the above three types of system should be replaceable by another without impacting the user interface (changes in performance excepted). The substitution of uniprocessors by multiprocessors is a common occurrence. Replacement of either by distributed systems has yet to be fully demonstrated. It is an area of active research. Nevertheless quite a lot can be said about the characteristics a successful distributed system would exhibit.

Which type of system is to be preferred is chiefly a matter of cost effectiveness, tempered by considerations of reliability, availability and ease of expansion.

Since they consist of identical 'building blocks' multiprocessor and distributed systems have an obvious potential for reliability and high availability. This potential has been demonstrated in systems such as Pluribus (Ornstein, Crowther, Kraley, Bressler, Michel and Heart, 1975) and the commercially available Tandem 16.

The reasons for the superior cost effectiveness of microprocessors over mainframes are well known. Basically the volume of production of microprocessor systems gives them lower per unit costs for design, manufacture, checkout and (if produced!) software. Conversely the low volume of mainframe sales means that the same tech-

*The author is with the DSIR Physics and Engineering Laboratory, Lower Hutt, New Zealand. Manuscript received 13 June 1980.

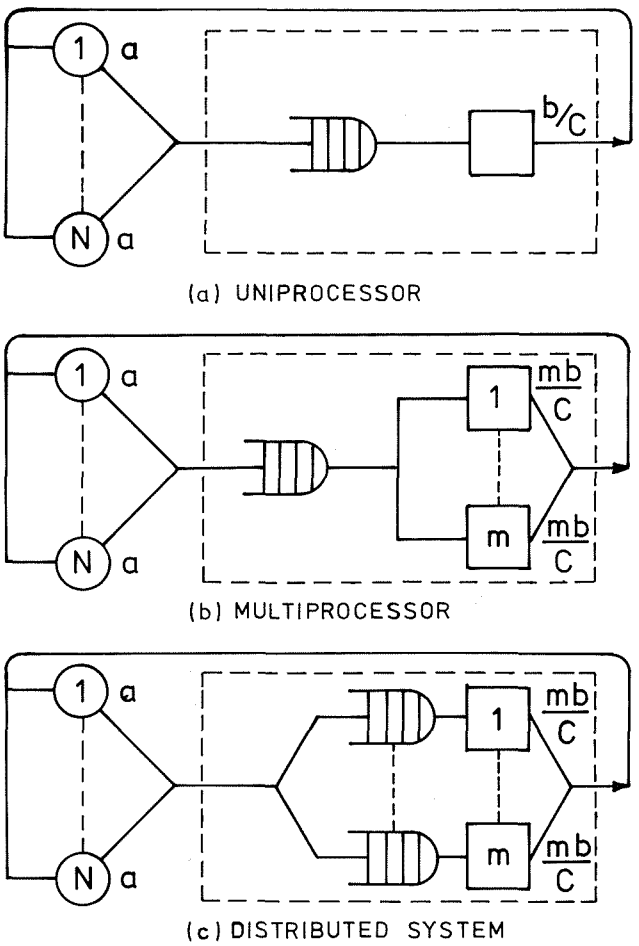



Figure 1

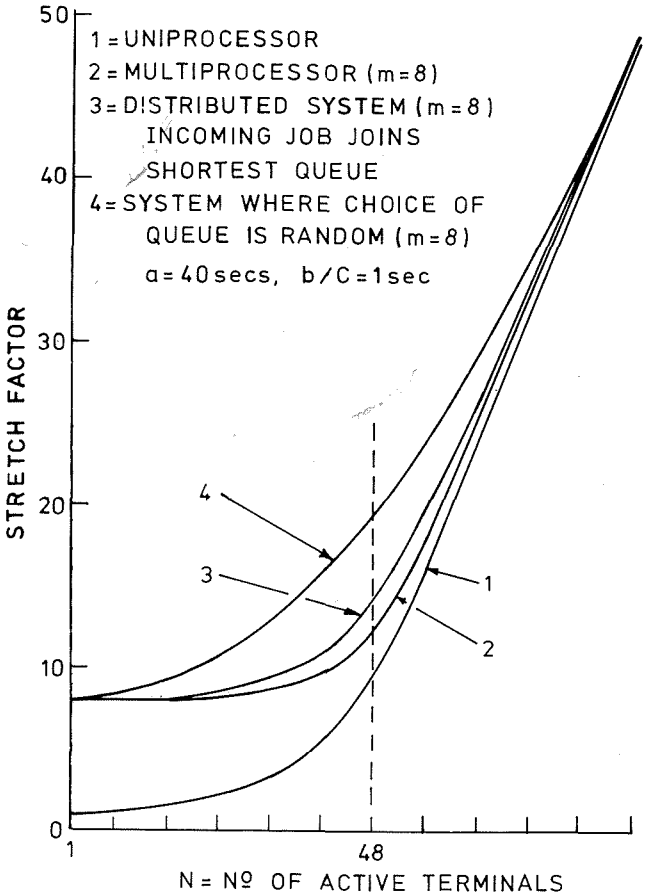



Figure 2

nology has to be retained over a longer period if manufacturers are to recoup their investment. Even when the latest LSI technology is applied to mainframes, the complexity and speed of their processors means that its full benefits cannot be reaped. (For a full discussion of these issues see Borgerson 1976, Casey 1977, Tjaden and Cohn 1979).

The important question is whether performance scales up when microprocessors are incorporated in multiprocessors or distributed systems. Both types of system impact performance in two ways. The architecture itself imposes limitations, which are discussed in the next section. Specific computing tasks carry extra overheads if they are distributed. This is discussed in Section 3.

## 2. ARCHITECTURAL ISSUES

### 2.1 Multiserver Effects

It is a generally accepted queuing theory result that average response times of a single server system are lower than the average response times of a m server system where each server has $1/m^{th}$ of the capacity of the single server (Kleinrock, 1974). While both types of queuing system have the same overall capacity the multiserver system can only utilise it fully when there are at least m jobs to be served. When there is only one job to be processed the single server will deal with it m times faster than (one server of) the multiserver system.

Figure 1 depicts queue structure diagrams for a time sharing system with N active terminals. The mean 'think time' of a terminal user is stipulated as 'a' seconds. The average number of instructions required to service each interaction is 'b'. The uniprocessor (a) is assumed to have a capacity of 'C' operations per second. Here it is assumed that the individual processors of the multiprocessor (b) and the distributed system (c) have an effective processing rate of $1/m^{th}$ of the uniprocessor, where m is the number of processors in the system. Thus the average service time of an interaction is mb/C. In multiprocessors, jobs are selected from a common queue while in a distributed system each computer maintains its own queue.

Sauer and Chandy (1979) have studied several aspects of queuing in multiprocessor systems. Extensive studies for distributed systems have not been performed but some results are presented in (Casey, 1977). By way of an example of the differences between uniprocessors, multiprocessors and distributed systems we present below results of an analytic/simulation study of response times where m = 8.

Let the stretch factor be the factor by which the service time on the uniprocessor of an interaction is multiplied to give its average response time. Thus the minimum stretch factor for the uniprocessor is 1 and for the multiprocessor and distributed system it is m = 8. Figure 2 graphs stretch factors as a function of the number of active terminals. For these curves a = 40 seconds, b/C is

assumed to be 1 second. Scheduling at each processor is assumed to be 'processor sharing', that is, round robin with vanishingly small quantum. Curve 3 gives the stretch factor for a distributed system where each incoming interaction is directed to the computer with the shortest queue. This assumes that each computer can process all types of interaction. (Curve 4, which is discussed in Section 3.3, covers the case when computers are functionally specialised).

Most time sharing systems operate somewhere near their 'saturation' point (Kleinrock, 1974). Forty-eight active terminals is the 'saturation' point for the multiprocessor and distributed systems. For this loading the average response times for an interaction requiring 0.1 seconds of processor time on the uniprocessor, are 1.0, 1.2 and 1.4 seconds for uniprocessor, multiprocessor and distributed system respectively. The differences are hardly going to be noticed by the user. However, when the uniprocessor is lightly loaded with, say, 32 active terminals, a user might be tempted to submit a large task. If the task required 60 seconds of processing time his average response time would be 3.1 minutes. The same task on the distributed system with the same loading would have an average response time of nearly nine minutes. While 3.1 minutes could be classified as interactive working, nine minutes definitely is not.

The above examples illustrate the effect on response times of work transferred from a mainframe uniprocessor to a multiprocessor or distributed system using less powerful processors. Short interactions are virtually unaffected but tasks requiring substantial bursts of processing power will suffer. Subdividing large tasks into parallel sub-tasks offers a potential remedy in some cases. This is discussed further in Section 3.2.

### 2.2 Memory Requirements

In a uniprocessor system the multiprogramming level is normally greater than 1 so that the central processor is not held up by I/O for a particular task (this can be I/O for the task or the I/O involved in swapping the task for another one). For a multiprogramming level of k there has to be enough primary memory to hold k task images or working sets.

For a multiprocessor system with m processors there must be at least enough primary memory to hold m task images or working sets. Otherwise all m processors could not be active simultaneously. Borgerson (1976) suggests that the total multiprogramming level should be $m + k - 1$, where k is the multiprogramming level of the 'equivalent' uniprocessor with the identical I/O subsystem. Recently a more refined estimate has been produced (Tjaden and Cohn, 1979). For systems with m greater than about eight, the multiprogramming level required is close to m.

For a distributed system each computer's primary memory must be of sufficient size to hold at least two task images or working sets if any overlap of I/O and processing is to be obtained. Since the total memory of a distributed system is not contiguous, even more memory must be provided to cater for fragmentation effects.

As well as its task images or working sets, a system must have memory space for the resident portions of its operating system. Again, the distributed system, with a basic kernel in every computer (see Section 3.1), has the greatest overall memory requirements.

The only compensation for a distributed system is that the speed of its memory need not be as fast as that for 'equivalent' uniprocessors and multiprocessors. Also, if it is required, cache memory for each processor in a distributed system can be provided as straightforwardly as in a uniprocessor. Cache operation in a multiprocessor system requires substantial extra hardware to ensure coherence (Censier and Feautrier, 1978).

### 2.3 Contention

The fall-off in performance of multiprocessors due to primary memory contention is well known and has been extensively studied. The problem can be minimised relatively cheaply by splitting memory into many independent modules.

More serious is the contention for the processor to memory pathway. The mechanisms used for the processor-memory switch of multiprocessor systems are:

(i) full interconnection to multiported memories.
(ii) crossbar switch
(iii) bus

All three mechanisms severely limit the expansion potential of multiprocessors. Full interconnection is not really tenable for systems containing more than three or four processors. A crossbar switch limits expansion to its designed capacity. A switch with a large capacity is likely to be very expensive, chiefly because of the large number of external connections it would require. Such a switch would wreck the cost effectiveness of small configurations. There is not a direct limit to expansion using a bus but bus contention ensures that the addition of extra processors becomes less and less cost effective.

For distributed systems the preferred forms of the communications subsystems are rings (Penny and Baghdadi, 1979) and buses, including the Ethernet (Metcalfe and Boggs, 1976) type of bus. As computer sizes shrink a variant of the processor-memory type of bus will probably predominate. The bandwidth requirements of distributed systems are a lot less than those of multiprocessors. Simulation experiments suggest that 2 Mbytes/second is an adequate bandwidth for a load balancing distributed system of 20 computers, each equivalent to one of today's powerful minicomputers (Casey, 1977). Thus for distributed systems the interconnection mechanism is not the expansion limiting factor it is for multiprocessors.

### 2.4 Size of Systems

The above points suggest that for both multiprocessors and distributed systems fewer processors of greater power are preferable to more processors of low power.

When a processor consists of a single chip its cost, relative to a complete system, is miniscule. Any savings made on cheaper, less powerful processor chips would be lost many times over with the extra memory, 'real-estate' and power supply requirements.

## 3. PROGRAMMING ISSUES

### 3.1 Concurrency

That distributed systems have not become very widespread is probably due most to the lack of appropriate software support. For distributed systems, as for others, the main software issue is the handling of concurrency. The special problem of distributed systems is that the tasks that interact may reside in different computers.

Further, reliability and system expansion considerations make system transparency an absolute necessity. The physical location of the entities a programmer is working with must be immaterial to him. Unless this is done, altering the underlying configuration of the system will be a hazardous operation.

The software problems of distributed systems are surmountable if a disciplined approach is taken to programming. Currently two general programming disciplines for handling concurrency are recognised: the message-oriented approach and the procedure-oriented approach (Lauer and Needham, 1978). Both are potentially suitable for distributed systems.

In both approaches each computer in the system is managed by a kernel. Each kernel is aware of the location of all resources in the system or, with the help of other kernels, can locate resources when necessary.

In a message-oriented system resources are associated with processes. A resource is accessed by sending its process a message and waiting for a reply. The function of the kernel is to deliver the messages to the correct destination processes, whether they be in the same computer as the source or not. The message-oriented approach has been adopted in such distributed systems as DCS (Farber and Larson, 1972), DCN (Lay, Mills and Zelkowitz, 1974), and HXDP (Jensen, 1978).

In a procedure-oriented system resources are accessed by procedures. A computation consists of calls to various procedures. The kernel's function in this case is to ensure the orderly entry to and return from procedures including the transmission of parameters. There are at least two implementations in progress of distributed systems that employ the procedure-oriented approach (Casey and Shelness, 1977; Dowson, 1977).

Note that because the underlying mechanism for exchanging data in distributed system is by message passing, it does not follow that the message-oriented approach is superior to the procedure-oriented approach. The messages between computers are at a primitive protocol level. At the higher level it matters little whether messages or parameter lists are being passed about. Each approach leads to a different style of programming. Each makes some features easy to implement while making others awkward. A more detailed analysis is carried out in (Casey, 1978) which concludes that the procedure-oriented approach has overall advantages.

Understanding of what mechanisms are required to handle concurrency in a distributed system is only the first step to producing appropriate programming language constructs. Concurrency is an area of active programming language research. But not all new constructs proposed are feasible for distributed systems. Many proposals require either centralised tables or shared memory locks for their implementation. Progress is promising, however. Many new languages incorporate a 'module' feature. Data within a module is not directly accessible from outside of the module. This will facilitate the siting of different modules in different computers.

Currently, disciplined programming incurs very high overheads, not only in distributed systems but also in multiprocessors and uniprocessors. This is because message passing, monitor entry and so on, is all carried out by software. In multiprocessors and uniprocessors much overhead can be avoided by using undisciplined mechanisms such as priority and spin locks. But this should not be the final solution. Software costs now so dominate hardware costs that hardware should be designed to efficiently support the mechanisms required for robust software. This would benefit uniprocessors, multiprocessors and distributed systems alike.

### 3.2 Task Decomposition

If the performance of a multiprocessor or distributed system on a single task is inadequate then that task may be decomposed into sub-tasks to improve its performance. No general method of converting a sequential task into a set of parallel sub-tasks has been found. It seems that each program needs to be individually examined.

A decomposition into sub-tasks may lessen the high primary memory requirements of multiprocessors and distributed systems (Section 2.3). The decomposition of the Harpy DECAL task developed for Cm* (Jones, Chansler, Durham, Feiler, Scelza, Schwans and Vegdahl, 1978) occupied very little more space than a uniprocessor version. However, splitting a task into co-operating sub-task involves overheads of two kinds.

First there are the overheads of co-operation and interaction. There are the inefficiencies in the implementation of message passing and so on mentioned above. Apart from this, any synchronisation of sub-tasks normally causes one or more processors to idle, waiting for locks to become free or messages to arrive.

The second kind of overhead is the extra instructions required to execute the parallel version of the task compared to the sequential version. Wilkes (1977) identifies a class of algorithms for which the parallel version requires fewer overall instructions than the serial version. But normally more instructions are required. In the Hearsay II system approximately half of the total processor time was used executing extra code that would not have been present in a uniprocessor version (Fennel and Lesser, 1977). The overall performance of the Harpy DECAL task using five LSI-11 processors was equal to that of a PDP 11/40 uniprocessor version. A PDP 11/40 is approximately three times as fast as an LSI-11.

A more disturbing result of both the Hearsay II and Harpy projects was the inability to decompose the tasks to use effectively more than about six processors. More processors could be used but they produced little or no decrease in the time taken to perform the task. Research on task decomposition has a long way to go.

### 3.3 Task Assignment

A multiple computer system where each kind of task was permanently assigned to one computer would suffer the following three drawbacks compared to a distributed system that has dynamic task assignment.

(i) Availability and resilience would be diminished. If one computer fails its tasks could no longer be carried out.

(ii) Poor expansion characteristics would result after major tasks (in terms of utilisation) each have their own processors. Any computer added after that would have very little effect on the system's performance.

(iii) Variations in the load on each computer would mean that some were overloaded while others were idle. This results in poor response times. Curve 4 of figure 2 shows the minimum average stretch factor for a system of eight computers. Here it is assumed

that the overall average execution time of tasks on each computer is the same (= 8 secs) and that the average utilisation of each computer is identical. In practice it would be nigh impossible to distribute tasks so. The stretch factors shown in curve 4 would be considerably worse if, for example, three of the eight computers accounted for 90 per cent of the system's computing load.

The ability to dynamically alter the binding of tasks to computers within a distributed system is a prerequisite to fail-soft operation. Since the basic mechanisms must be provided it is natural to look at moving tasks between computers to balance the load on each. As figure 2 shows the potential gains in performance are quite large, probably more than enough to offset the extra overheads involved.

Again this is an area requiring a lot of research. Several load balancing mechanisms have been proposed. These include a special (duplicated) central hardware device to queue all tasks (Ornstein *et al.*, 1975), bidding (Farber and Larson, 1972), and the broadcasting of current status to other computers (Casey and Shelness, 1977). As an example of the problems still to be faced consider a group of co-operating sub-tasks. When the distributed system is lightly loaded then ideally each sub-task should be located in a different computer, so as to maximise parallel operation. However, when the system is very heavily loaded the preferred assignment may be to have all sub-tasks in the same computer, so as to minimise inter-task communication delays.

Load balancing and, to a lesser extent, fail-soft requirements involve the exchange of status between computers. The number of such messages is likely to grow as the square of the number of computers in the system (Casey, 1977). This explosive growth could be the factor limiting the ultimate size of a distributed system. Either the communications sub-system would become overloaded or the entire processing capacity would be dedicated to handling the status messages.

## 4. CONCLUSIONS

The preceding sections have identified points favourable and unfavourable to each of the three architectures under consideration. It is not possible to conclude that one type of system is superior to the others. Instead, below is a 'wish list' of the attributes or advances each system needs to improve its overall acceptability.

For uniprocessors:

- more reliability
- decreased processor hardware costs

For multiprocessors:

- lower cost for high capacity processor-memory interconnection
- decreased primary memory prices
- advances in task decomposition methodology
- better programming language constructs for concurrency

For distributed systems:

- substantially cheaper primary memory
- more hardware support for managing tasks and their interactions
- better programming language constructs for concurrency
- advances in task decomposition methodology
- practical load balancing strategies.

Even if uniprocessors should prove dominant the research going on into distributed operating systems, concurrency in programming languages and task decomposition will not be wasted. Today's 'super' computers, with their long pipelines, banks of special registers and caches, have little remaining potential for architectural improvement. Yet there is an unsatisfied demand for massive computing power. The only way to meet some of this demand may be to link together 'super' computers.

## REFERENCES


BORGERSON, B.R. (1976), "The viability of multimicroprocessors systems", *IEEE Computer*, Vol. 9, No. 1, January 1976, pp. 26-30.

CASEY, L.M. (1977), "Computer structures for distributed systems", Ph.D. Thesis, CST-2-77, University of Edinburgh.

CASEY, L.M., and SHELNESS, N.S. (1977), "A domain structure for distributed computer systems", Proceedings of the 6th Symposium on Operating System Principles (Purdue), November 1977, pp. 101-108.

CASEY, L.M. (1978), "On kernels for distributed computer systems", Report CSR-27-78, University of Edinburgh, August 1978.

CENSIER, L.M., and FEAUTRIER, P. (1978), "A new solution to coherence problems in multi-cache systems", *IEEE Transactions on Computers*, Vol. C-27, No. 12, December 1978, pp. 1112-1118.

DOWSON, M. (1977), "DEMOS – a multiprocessor computer", Internal Report, Computer Science Division, National Physics Laboratory (Teddington, GB), October 1977.

ENSLOW, P.H. (1978), "What is a 'distributed' data processing system?", *IEEE Computer*, Vol. 11, No. 1, January 1978, pp. 13-21.

FARBER, D.J., and LARSON, K.C. (1972), "The structure of the Distributed Computer System – software", Proceedings of the Symposium on Computer Communications Networks and Teletraffic (New York), April 1972, pp. 539-545.

FENNEL, R.D., and LESSER, V.R. (1977), "Parallelism in artificial intelligence problem solving: a case study of Hearsay II", *IEEE Transactions on Computers*, Vol. C-26, No. 2, February 1977, pp. 98-111.

JENSEN, E.D. (1978), "The Honeywell experimental distributed processor – an overview", *IEEE Computer*, Vol. 11, No. 1, January 1978, pp. 28-38.

JONES, A.K., CHANSLER, R.J., DURHAM, I., FEILER, P.H., SCELZA, P.A., SCHWANS, K., and VEGDAHL (1978), "Programming issues raised by a multiprocessor", *Proceedings of the IEEE*, Vol. 66, No. 2, February 1978, pp. 229-237.

KLEINROCK, L. (1974), "Resource allocation in computer systems and computer-communication networks", Proceedings IFIP Congress 74 (Stockholm), 1974, pp. 11-18.

LAUER, H.C., and NEEDHAM, R.M. (1978), "On the duality of operating system structures", Proceedings Second International Symposium on Operating Systems, IRIA, October 1978, reprinted in *ACM Operating Systems Review*, Vol. 13, No. 2, April 1979, pp. 3-19.

LAY, W.M., MILLS, D.L., and ZELKOWITZ, M.V. (1974), "Operating systems architecture for a distributed computer network", Computer Networks: Conference of IEEE Computer Society and NBS (Gaithersburg), May 1974, pp. 39-44.

METCALFE, R.M., and BOGGS, D.R. (1976), "Ethernet: distributed packet switching for local computer networks", *Communications of the ACM*, Vol. 19, No. 7, July 1976, pp. 395-404.

ORNSTEIN, S.M., CROWTHER, W.R., KRALEY, M.F., BRESSLER, R.D., MICHEL, A., and HEART, F.E. (1975), "Pluribus – a reliable multiprocessor", *AFIPS Proc. Nat. Comp. Conf.*, Vol. 44, 1975, pp. 551-559.

PENNY, B.K., and BAGHDADI, A.A. (1979), "Survey of computer communications loop networks", *Computer Communications*, Vol. 2, No. 4, August 1979, pp. 165-180 and Vol. 2, No. 5, October 1979, pp. 224-241.

SAUER, C.H., and CHANDY, K.M. (1979), "The impact of distributions and disciplines on multiple processor systems", *Communications of the ACM*, Vol. 22, No. 1, January 1979, pp. 25-33.

TJADEN, G., and COHN, M. (1979), "Some considerations in the design of mainframe processors with microprocessor technology", *IEEE Computer*, Vol. 12, No. 8, August 1979, pp. 68-74.

WILKES, M.V. (1977), "Beyond today's computers", Proceedings IFIP Congress 77 (Toronto), 1977, pp. 1-5.

## BIOGRAPHICAL NOTE

*Liam Casey is a scientist in the Computer Research Section of the Physics and Engineering Laboratory, DSIR, New Zealand. He started his career in the Applied Mathematics Division of DSIR. In 1974 he went to Edinburgh University where he obtained a Ph.D in Computer Science. He continued his Ph.D research into distributed computing as a Post Doctoral fellow for 18 months before returning to New Zealand in October 1978 to his current position. His research interests centre on distributed computing, and include performance modelling, load balancing and language questions.*

# Program Control by State Transition Tables

Peter Juliff*

The use of state transition tables as a means of program control provides a programming methodology which is easy to construct, leads to concise source code and allows for ease of maintenance without alteration to procedural code.
CR Categories: 4.0

## INTRODUCTION

The use of state transition tables as a means of controlling the execution of a program is a much neglected topic. This is surprising given that this technique helps to achieve two important criteria in program design:

(a) simplicity and elegance of the source code, and
(b) ease of subsequent amendment.

Whilst some use of state transition tables has been made in specialist areas such as software controlled telephony where state switching is a familiar engineering concept, they have been largely ignored by programmers working in more conventional environments. It is the aim of this paper to explain their construction and operation as drivers of individual sections of a program or of the overall program logic.

## RATIONALE

Working from the premise that:

(a) ease of ongoing amendment is one of the most important factors to be considered in program design, and
(b) program amendments which may be implemented by the alteration of data are more desirable than those which require the alteration of procedural code,

then any technique which facilitates this end is a major contributor to program flexibility. The use of a state transition table may be confined to one individual module within a program and may be part of the local data owned by that module, or it may apply to the control of the entire algorithm. The aim of its construction is to enable modifications to the program's behaviour to be effected with little or no alteration of the procedural code.

## PROGRAMS AS STATE ORIENTED ALGORITHMS

A program may be considered as an algorithm which progresses from its current state to one of many other possible states depending on changes in its operating environment. The arrival at any one particular state depends on the prior state and the occurrence of an event.

The tools required to harness these changes to control the operation of a program are:

(a) the ability to devise a table which specifies the states in which a program may operate and the events which cause a change from one state to any other,
(b) the ability to define a data structure which embodies this state table, and
(c) the availability of a computed branch instruction to implement the program control,
e.g., a Computed GO TO in FORTRAN,
a CASE statement in ALGOL, PL/1 and PASCAL,
a GO TO . . . DEPENDING ON . . . in COBOL, or
an ON statement in BASIC.

The following examples are shown using COBOL because of the ease of defining data and the explicit nature of its syntax.

### Example 1

A program accepts data from an operator at a terminal. One operation involves the acceptance of a monetary value which may be of a variety of formats and which will be deposited, left justified, in an eight-character field with unused character positions to the right of the received field containing spaces.

The rules governing the format of this value are:

(i) there may be a leading minus sign,
(ii) there may be leading spaces and the field may be entirely blank,
(iii) the field will terminate on either the filling of the entire eight-character field or on the encountering of a non-leading space,
(iv) if the value consists of dollars only, the decimal point may be omitted, but if a decimal point is encountered it must be followed by at least one further digit,
(vi) a maximum of four digits for dollars and two digits for cents.

To summarise:

| Valid Values | Invalid Values |
|---|---|
| −12.34ΔΔ | −12.ΔΔΔΔ |
| ΔΔ−12ΔΔΔ | 123−ΔΔΔΔ |
| .12ΔΔΔΔΔ | .123ΔΔΔΔ |
| −.12ΔΔΔΔ | 12345ΔΔΔ |
| 1234.56Δ | 12.345ΔΔ |

A state transition table specifying the permissible formats of this field would be constructed as:



*The author is with the Department of Data Processing, Prahran CAE, Prahran, Victoria, 3181.* *Manuscript received 8 July 1980, and revised 14 August 1980.*

| Current State | New State Depending on Next Character: Minus Sign | Digit | Dec. Point | Space | End of Field | Else |
|---|---|---|---|---|---|---|
| 1. Before 1st character. | 2 | 3 | 4 | 1 | 7 | 6 |
| 2. Just after minus sign. | 6 | 3 | 4 | 6 | 6 | 6 |
| 3. Digits before dec. point. | 6 | 3 | 4 | 7 | 7 | 6 |
| 4. Just after dec. point. | 6 | 5 | 6 | 6 | 6 | 6 |
| 5. Digits after dec. point. | 6 | 5 | 6 | 7 | 7 | 6 |
| 6. Error discovered | | | | | | |
| 7. Finished | | | | | | |

The program module charged with the task of validating and assembling this value will commence in State 1 and will progress through the other states depending on the content of the input field. Each new state will be selected on the basis of the current state (pointing to the row in the table) and the next input character encountered (pointing to the column in the table.

Typical input contents and the states traversed in the algorithm would be:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Input Characters | | – | 1 | 2 | . | 3 | 4 | Δ | Δ |
| State | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 7 | |
| Input Characters | | Δ | Δ | 1 | 2 | 3 | 4 | Δ | Δ |
| State | 1 | 1 | 1 | 3 | 3 | 3 | 3 | 7 | |
| Input Characters | | 1 | 2 | . | Δ | Δ | Δ | Δ | Δ |
| State | 1 | 3 | 3 | 4 | 6 | | | | |

Each state reached may involve work to do on the input field, e.g., accumulating an amount of dollars or cents, or may merely operate to shift to another row in the table and hence follow a potentially different set of rules for the selection of the next state.

Excerpts of COBOL coding to implement this technique would appear as:

```
WORKING STORAGE SECTION.
.
.
.
01 SOURCE-FIELD.
   03  SOURCE-CHARACTER          PIC X      OCCURS 8
                                            INDEXED BY SOURCE-INX.

01 DESTINATION-FIELD.
   03  DOLLARS                  PIC 9(4).
   03  CENTS.
       05 CENT                   PIC 9      OCCURS 2
                                            INDEXED BY CENT-INX.

01 MONETARY-VALUE                PIC S9(4)V9(2)
   REDEFINES DESTINATION-FIELD.
01 CONSTRAINTS.
   03  SOURCE-SIZE              PIC 9      VALUE 8.
   03  DOLLAR-SIZE              PIC 9      VALUE 4.
   03  CENTS-SIZE               PIC 9      VALUE 2.
01 SWITCHES.
   03  FINISHED                 PIC 9.
   03  VALUE-OK                 PIC 9.
   03  NEGATIVE-VALUE           PIC 9.
01 SWITCH-VALUES.
   03  TRUE                     PIC 9      VALUE 1.
   03  FALSE                    PIC 9      VALUE 0.
01 ONE-CHARACTER.
   03  ONE-DIGIT                PIC 9.
01 CHARACTER-COUNTERS.
   03  DOLLAR-COUNTER           PIC 9.
   03  CENT-COUNTER             PIC 9.
01 STATE-TABLE.
   03  FILLER                   PIC 9(6)   VALUE 234176.
   03  FILLER                   PIC 9(6)   VALUE 634666.
   03  FILLER                   PIC 9(6)   VALUE 634776.
   03  FILLER                   PIC 9(6)   VALUE 656666.
   03  FILLER                   PIC 9(6)   VALUE 656776.
01 STATE-TABLE-AGAIN
   REDEFINES STATE-TABLE.
   03  STATE-ROW                OCCURS 5 INDEXED BY ROW-INX.
       05  STATE-COLUMN         OCCURS 6 INDEXED BY COL-INX.
           07  NEW-STATE        PIC 9.
01 CURRENT-STATE                 PIC 9.

PROCEDURE DIVISION.
.
.
.
   ACCEPT SOURCE-FIELD.
   MOVE FALSE TO FINISHED
                 NEGATIVE-VALUE.
   MOVE TRUE TO VALUE-OK.
```

```
    SET SOURCE-INX
        CENT-INX TO 1.
    MOVE ZERO TO MONETARY-VALUE.
                 DOLLAR-COUNTER
                 CENT-COUNTER.
    MOVE 1 TO CURRENT-STATE.
    PERFORM 100-GET-VALUE UNTIL FINISHED = TRUE.
    IF VALUE-OK = TRUE
    AND  NEGATIVE-VALUE = TRUE
         MULTIPLY -1 BY MONETARY VALUE.

.
.
.
.
100-GET-VALUE SECTION.
101. IF SOURCE-INX > SOURCE-SIZE
         MOVE 5 TO COL-INX
     ELSE
        MOVE SOURCE-CHARACTER (SOURCE-INX) TO ONE-CHARACTER
        SET SOURCE-INX UP BY 1
        IF ONE-CHARACTER = "-"
           MOVE 1 TO COL-INX
     ELSE
        IF ONE-CHARACTER IS NUMERIC
           MOVE 2 TO COL-INX
     ELSE
        IF ONE-CHARACTER = "."
           MOVE 3 TO COL-INX
        ELSE
           IF ONE-CHARACTER = SPACE
              MOVE 4 TO COL-INX
           ELSE
              MOVE 6 TO COL-INX.
     MOVE CURRENT-STATE TO ROW-INX.
     MOVE NEW-STATE (ROW-INX  COL-INX) TO CURRENT STATE.
     GO TO 109 102 103 109 105 106 107
        DEPENDING ON CURRENT-STATE.
102.
     MOVE TRUE TO NEGATIVE-VALUE.
        GO TO 109.
103.
     IF DOLLAR-COUNTER IS NOT < DOLLARS-SIZE
        GO TO 106
     ELSE
        COMPUTE DOLLARS = 10 * DOLLARS + ONE-DIGIT
        GO TO 109.
105.
     IF CENT-COUNTER IS NOT < CENTS-SIZE
        GO TO 106
     ELSE
        MOVE ONE-DIGIT TO CENT (CENT-INX)
        SET CENT-INX UP BY 1
        ADD 1 TO CENT-COUNTER
        GO TO 109.
106.
     MOVE FALSE TO VALUE-OK.
107.
     MOVE TRUE TO FINISHED.
109.
     EXIT.
```

Changes in the rules governing the format of the value may be implemented by altering the content of the state transition table.

Consider the following:

(i) leading spaces are no longer allowed,

(ii) all values must contain a decimal point, however there needs be no following cents digits if the field contains only dollars and no preceding dollars digits if the field contains only cents,

(iii) the field must not be entirely blank.

The state table, altered to reflect these new rules, would appear as:

| | | | | | |
|---|---|---|---|---|---|
| 2 | 3 | 4 | 6 | 6 | 6 |
| 6 | 3 | 4 | 6 | 6 | 6 |
| 6 | 3 | 4 | 6 | 6 | 6 |
| 6 | 5 | 6 | 7 | 7 | 6 |
| 6 | 5 | 6 | 7 | 7 | 6 |

Altering the contents of STATE-TABLE to incorporate these new values would effect the desired amendment to the algorithm without any alteration being required to the procedural code.

**Example 2**

A record in a cataloguing system contains a Part Number field of the format a/b/c. The requirements of each portion are:

a: numeric, 1 to 3 digits, must not be omitted
b: numeric, 1 to 4 digits, must not be omitted
c: either: numeric, 1 to 4 digits
or: alphabetic, 1 to 4 letters
but: not alphanumeric, must not be omitted.

The field terminates at the first space or upon exhaustion of its 13 permissible characters.

The field is to be checked for validity of format and, if valid, assembled in a field of format ABC where:

A: 3 digits, right justified, leading zero fill
B: 4 digits, right justified, leading zero fill
C: 4 characters, left justified, trailing space fill

Excerpts of COBOL coding to process this field are shown below, including the state table in comment format for documentation purposes.

```
WORKING-STORAGE SECTION.
.
.
.
01 INPUT-PART-NO.
   03  INPUT-CHARACTER              PIC X      OCCURS 13
                                               INDEXED BY INPUT-INX.
01 OUTPUT-PART-NO.
   03  A                            PIC 9(3)
   03  B                            PIC 9(4).
   03  C.
       05  C-CHARACTER              PIC X      OCCURS 4
                                               INDEXED BY C-INX.
01 CONSTRAINTS.
   03  INPUT-SIZE                   PIC 9(2)   VALUE 13.
   03  A-SIZE                       PIC 9      VALUE  3.
   03  B-SIZE                       PIC 9      VALUE  4.
   03  C-SIZE                       PIC 9      VALUE  4.
   03  DELIMITER                    PIC X      VALUE "/".
01 SWITCHES.
   03  FINISHED                     PIC 9.
   03  PART-NO-OK                   PIC 9.
01 SWITCH-VALUES.
   03  TRUE                         PIC 9      VALUE 1.
   03  FALSE                        PIC 9      VALUE 0.
01 ONE-CHARACTER.
   03  ONE-DIGIT                    PIC 9.
01 CHARACTER-COUNTERS.
   03  A-COUNTER                    PIC 9.
   03  B-COUNTER                    PIC 9.
*
*  STATE TRANSITION TABLE FOR PART NUMBER
*                                        ----- INPUT CHARACTER -----
*  CURRENT STATE                 DIGIT  SPACE  LETTER  "/"  END OF  ELSE
*                                                            FIELD
*  1.  BEFORE 1ST CHAR.            2      8      8      8      8      8
*  2.  DIGITS BEFORE 1ST "/"       2      8      8      3      8      8
*  3.  JUST AFTER 1ST "/"          4      8      8      8      8      8
*  4.  DIGITS BEFORE 2ND "/"       4      8      8      5      8      8
*  5.  JUST AFTER 2nd "/"          6      8      7      8      8      8
*  6.  DIGITS AFTER 2ND "/"        6      9      8      8      9      8
*  7.  LETTERS AFTER 2ND "/"       8      9      7      8      9      8
*  8.  ERROR
*  9.  FINISHED                                 <NEW STATE>
*
01 STATE-TABLE.
   03  FILLER                       PIC 9(6)   VALUE 288888.
   03  FILLER                       PIC 9(6)   VALUE 288388.
   03  FILLER                       PIC 9(6)   VALUE 488888.
   03  FILLER                       PIC 9(6)   VALUE 488588.
   03  FILLER                       PIC 9(6)   VALUE 687888.
   03  FILLER                       PIC 9(6)   VALUE 698898.
   03  FILLER                       PIC 9(6)   VALUE 897898.

01  STATE-TABLE-AGAIN
    REDEFINES STATE-TABLE.
    03 STATE-ROW                                OCCURS 7 INDEXED BY ROW-INX.
       05 STATE-COLUMN                          OCCURS 6 INDEXED BY COL-INX.
          07 NEW-STATE                          PIC 9
01  CURRENT-STATE                               PIC 9.
```

```
PROCEDURE DIVISION.
.
.
.
    MOVE FALSE TO FINISHED.
    MOVE TRUE TO PART-NO-OK.
    SET  INPUT-INX.
         C-INX TO 1.
    MOVE ZERO TO  A   A-COUNTER
                  B   B-COUNTER.
    MOVE SPACES TO C.
    MOVE 1      TO CURRENT-STATE.
    PERFORM   100-FIX-PART-NUMBER
              UNTIL FINISHED = TRUE.
    IF PART-NO-OK ....
    ELSE ....
100-FIX-PART-NUMBER SECTION.
101.
    IF  INPUT-INX >INPUT SIZE
       MOVE 5 TO COL-INX
    ELSE
       MOVE INPUT-CHARACTER (INPUT-INX) TO ONE-CHARACTER
       SET INPUT-INX UP BY 1
       IF ONE-CHARACTER IS NUMERIC
          MOVE 1 TO COL-INX
       ELSE
          IF ONE-CHARACTER = SPACE
             MOVE 2 TO COL-INX
          ELSE
             IF ONE-CHARACTER IS ALPHABETIC
                MOVE 3 TO COL-INX
             IF  ONE-CHARACTER = DELIMITER
                MOVE 4 TO COL-INX
             ELSE
                MOVE 6 TO COL-INX.
    MOVE CURRENT-STATE TO ROW-INX.
    MOVE NEW-STATE (ROW-INX   COL-INX) TO CURRENT-STATE.
    GO TO  110 102-A 110 104-B 110
           106-C 106-C 108-ERROR 109-DONE
           DEPENDING ON CURRENT-STATE.
102-A.
    IF A-COUNTER IS NOT <A-SIZE
       GO TO 108-ERROR
    ELSE
       COMPUTE A = 10 * A + ONE-DIGIT
       ADD 1 TO A-COUNTER
       GO TO 110.
                    ,
                    ,

104-B.
    IF B-COUNTER IS NOT <B-SIZE
       GO TO 108-ERROR
    ELSE
       COMPUTE B = 10 * B + ONE-DIGIT
       ADD 1 TO B-COUNTER
       GO TO 110
106-C.
    IF C-INX >C-SIZE
       GO TO 108-ERROR
    ELSE
    MOVE ONE CHARACTER TO C-CHARACTER (C-INX)
    SET C-INX UP BY 1
    GO TO 110.
108-ERROR.
    MOVE FALSE TO PART-NO-OK.
109-DONE.
    MOVE TRUE TO FINISHED.
110.
    EXIT
```

Once again, a number of significant changes to the permissible format of the Part Number could be accommodated by an alteration of values in the table, e.g.,

(i) permit leading spaces prior to the 1st character,

(ii) permit any, or all, of a, b and c to be omitted, i.e., provide for a, a/b, a//c, /b, /b/c, //c.

The state table necessary to provide for these new formats would be:

| | | | | | |
|---|---|---|---|---|---|
| 2 | 1 | 8 | 3 | 9 | 8 |
| 2 | 9 | 8 | 3 | 9 | 8 |
| 4 | 9 | 8 | 5 | 9 | 8 |
| 4 | 9 | 8 | 5 | 9 | 8 |
| 6 | 9 | 7 | 8 | 9 | 8 |
| 6 | 9 | 8 | 8 | 9 | 8 |
| 8 | 9 | 7 | 8 | 9 | 8 |

As in Example 1, the above alteration of the contents of STATE-TABLE would effect the necessary changes in the algorithm without any alteration to the procedural code.

Example 3.

The previous two examples have related to the control of an individual program module. The following example illustrates the use of a state switching driver to control the processing path of an entire program.

The example chosen is a standard sequential father/son master file update where each of the input files may exist in one of three possible conditions:

| | |
|---|---|
| No record currently in buffer | State: Vacant |
| Record read and available to process | : Waiting |
| End-of-file reached | : Ended. |

The processing algorithm may then be in any of nine possible states depending on the respective status of the master and transaction files:

| Program State | Input File States | |
|---|---|---|
| | Master | Transition |
| 1 | Vacant | Vacant |
| 2 | Vacant | Waiting |
| 3 | Vacant | Ended |
| 4 | Waiting | Vacant |
| 5 | Waiting | Waiting |
| 6 | Waiting | Ended |
| 7 | Ended | Vacant |
| 8 | Ended | Waiting |
| 9 | Ended , | Ended. |

It is presumed that transactions will be either Additions of new master records, Changes to existing master records or Deletions of existing master records.

The skeletal algorithm to drive the program's execution would be of the format:

```
Update:
   Set Current State = 1;
   Repeat until FINISHED = TRUE:
         Depending on Current State, Perform:
               1: Procedure A
               2: Procedure B
               3: Procedure B
               4: Procedure A
               5: Procedure C
               6: Procedure D
               7: Procedure A
               8: Procedure E
               9: Set FINISHED = TRUE.
Procedure A:
   Read Transaction Record.
   If end-of-file   State = State +2
   Else             State = State +1.

Procedure B:
   Read Master Record.
   If end-of-file   State = State +6
   Else             State = State +3.

Procedure C:
   If Master Key <  Transaction Key,
      Write Master Record to Output File,
      Read Master Record
      If end-of-file      State = 8
      Else                State = 5
   If Master Key > Transaction Key,
      If addition,        Write Transaction to Output File
      Else                Report Transaction as an Error.
      State = 4.
   If  Master Key =       Transaction Key,
      If addition,        Report Transaction as an Error
                          State = 4.
      If Change,          Amend Master Record
                          State = 4.
      If Deletion,        State = 1.

Procedure D:
   Write Master Record to Output File.
   Read Master Record.
   If end-of-file,        State = 6
   Else                   State = 9.

Procedure E:
   If Addition,    Write Transaction to Output File
   Else            Report Transaction as an Error.
   State = 7.
```

This example is somewhat different from the first two in that, whilst not using a state transition table as such, it adheres to the principle of regarding a program as a state-oriented algorithm and directs control through a Current State variable.

## RELATIONSHIP WITH OTHER CONTROL TECHNIQUES AND OTHER LANGUAGES

It is useful to compare control via a state transition table with alternative methods which also aim to provide program flexibility. It has been suggested that the use of 88-level condition names or decision tables may provide more readily understandable control mechanisms in COBOL.

The limitations of these methods are imposed by the number of potential conditions which must be catered for. From the above examples it can be seen that the number of possible conditions arising from combinations of sequences of data is the product of the number of states and the number of events which may cause a change of state.

Hence, if there was the possibility of five states and six events which could change those states, there would be potentially 30 conditions to which to assign individual names and for which to test if condition names were used and 2" possible rules to cope with in a decision table.

Although the above illustrations have been given in COBOL it should be apparent that, as stated above, the technique would be able to be implemented in any language with the facility to operate on a two-dimensional array and a multiple branching technique.

### References

It is common practice with papers such as this to give a list of references to similar material. Despite recourse to libraries and colleagues I am unable to find any other work which covers this topic.

## CONCLUSION

One of the foremost criteria in the design of any program is the provision of future flexibility. Flexibility is generally easier to achieve via the alteration of data constructs rather than alteration of procedural code.

The use of state transition tables provides coding which is, in general, more concise than that produced by alternative methods of control and more amenable to on-going maintenance.

## BIOGRAPHICAL NOTE

*The author is currently Head of the Department of Data Processing at Prahran College of Advanced Education, Melbourne. Prior to taking up his present position he was Deputy Programming Manager for Health Computing Services and prior to that Senior Lecturer in Information Processing at Caulfield Institute of Technology.*

# Computer Aided Design of Printed Circuit Board Layouts

G.L. Cock*

A large percentage of equipment manufactured for industrial and home use containing electronic equipment includes printed circuit boards (PCBs) on which miniature electronic components are mounted. Computer aided layout methods have been developed which avoid the laborious and time-consuming manual steps that have been necessary to produce these boards and at the same time allow the use of more accurate and more reliable production techniques. This paper describes a computer program that will automatically route wiring paths for printed circuit boards. Conductor path determination is a major step in the production of artwork for PCB manufacture.

Keywords and phrases: Computer aided design, CAD, Printed circuit board.
CR category: 3.24

## 1. INTRODUCTION

During the past decade the complexity of printed circuit boards has been such that manual methods for layout have been unable to cope. More and more manufacturers have been looking towards Computer Aided Design (CAD) where the computer is used as a bookkeeping and drafting tool, or Design Automation (DA) where the computer makes decisions such as where to place components or how to route wire paths. Computer Aided Editing (CAE) is normally used in association with Design Automation and permits the designer to visually modify a layout. Most attempts to produce wire (or conductor) routing programs have been based on Lee's Algorithm (Lee, 1961) or some development of it (Dunne, 1966; Fisk, 1967; Akers, 1967; Whatmough, 1972; Rubin, 1974; Hoel, 1976). Due to the large number of integrated circuit packages used these days the vertical/horizontal method of wire routing has proven the most popular (Zane and Harrell, 1968; Hightower, 1974; Rosa and Lucio, 1979).

The Lee algorithm has some outstanding advantages and disadvantages. If a path exists through the maze the algorithm will find it. If more than one path exists it will find a shortest path. However, because of the exhaustive cellular search techniques the computer storage requirements can be great, and computer run times excessive.

In the vertical/horizontal method paths are laid down on two planes on a regular grid (usually with 0.05 in. spacing); one plane carries mainly horizontal lines and the other mainly vertical. Only the co-ordinates of the ends of the lines are stored, and this is a great improvement upon Lee's method of storing the contents of every basic cell in the maze. The search through this structure, which links together associated pieces of information, is quicker than Lee's exhaustive cellular search. The penalty paid for the savings in storage and time is that some routes may not be completed.

The program described in this paper incorporates both methods of path routing, but from the practical viewpoint the vertical/horizontal method is the more acceptable. The program is called WREPCL — Wiring RoutinE for Printed Circuit Layouts.



## 2. PRACTICAL REQUIREMENTS

Electronics covers a very wide field and some PCBs are so specialised that they cannot be laid out by computer aided methods, however the majority of PCBs does not fall into this category. Computer methods must satisfy the general functions shown by Figure 1.

A PCB may consist of a front and rear layer only (termed a "double-sided" board), or it may consist of a sandwich of layers (termed a "multi-layer" board). The majority of PCBs are "double-sided".

The component placement stage requires that the components be placed on the board in accordance with accepted standards and practices. In addition, components are normally placed so that wiring lengths are minimized.

Conductors can have various widths. Wide conductors are used to supply power to some components and generally occupy predetermined paths on the board. The thinner conductors (signal conductors) are generally "free routing", that is, they may be placed in any available space on the board.

The component placement and conductor routing stages are most demanding and time-consuming for human beings. If a computer can be used to fully or partially

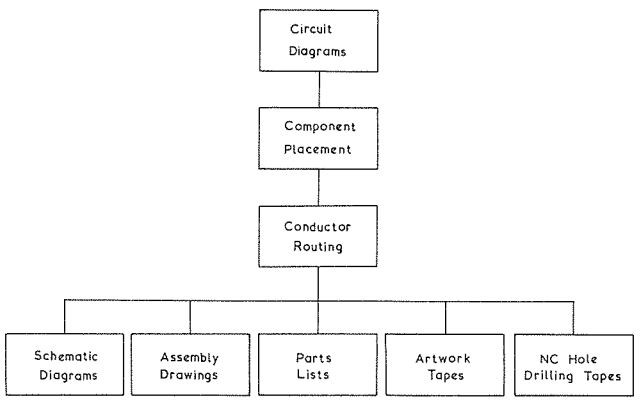



**Figure 1. Layout functions.**

*The author is with the Advanced Engineering Laboratory, Defence Research Centre, Salisbury, South Australia. *Manuscript received 27 March 1980, and revised 23 June 1980.*

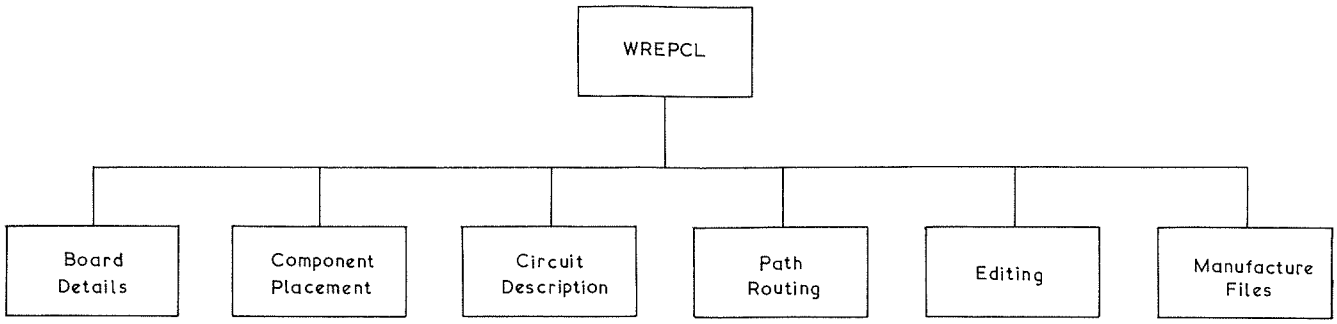



**Figure 2. WREPCL phases.**

automate these stages then considerable time will be saved. The layout designer carries out the interesting layout design steps and leaves the mundane repetitious steps to be performed by the computer.

## 3. PROGRAM REQUIREMENTS

To aid practical acceptance a computer program to aid PCB layout design must allow for manual interaction. Design Automation, which generally completes 95 to 100 per cent of the required conductor routing, requires manual interaction for the post-routing stage so that remaining unconnected paths can be completed. Computer Aided Design allows for manual interaction during the whole layout and routing process.

A basic program must carry out path routing and provide output for a photo-plotter (which produces the artwork) and drilling data for the PCB manufacturing process. A program may incorporate facilities to carry out other functions shown in Figure 1. WREPCL is a Design Automation program incorporating manual interactive component placement, manual and automatic path routing, Computer Aided Editing, off-line photoplotting and drilling data output.

## 4. PROGRAM STRUCTURE

Figure 2 depicts the WREPCL program phases. The routines have been written in Fortran IV. The first three steps involve the input of board details, component placement and circuit description, and convert information to a form (known as the DATA file) suitable for reading by the path routing phase. The majority of the routines in the program is associated with the path routing phase.

When using the vertical/horizontal method, in the path routing phase, a path does often exist, but it is too devious for the search methods to find it. Failures can in this case be corrected easily and quickly during the Editing phase. The line segment data structure can be modified by adding or deleting line segments. The Editing phase also permits the addition of other information such as "text".

The contents of the data structure (known as the LAYOUT file) are processed by routines to create files in a format suitable for reading by artwork production equipment and numerically controlled drilling machinery.

## 5. PATH ROUTING ALGORITHMS

Prior to entering the actual path routing procedures the program establishes an order of wiring priority (e.g., power wiring must precede signal wiring). Those conductor paths that have been determined manually are assigned directly to the line segment data structure.

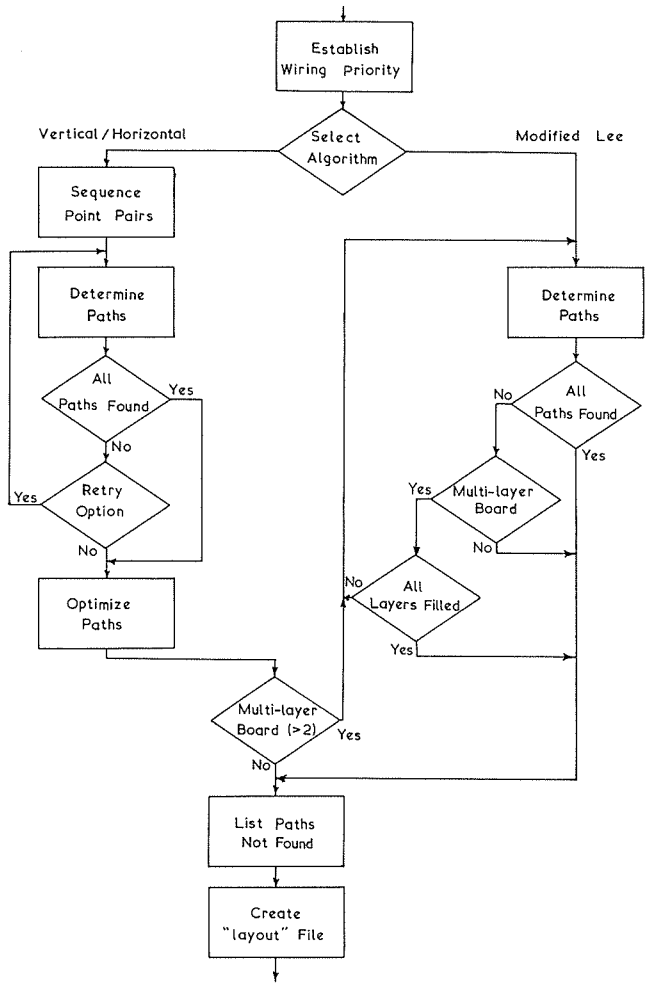



**Figure 3. Flowchart of the algorithms.**

The first step in the vertical/horizontal algorithm is to break each of the nodes (electrically common conductors) into a number of sections, each section forming a connection between a pair of points. A minimum tree algorithm is used to ensure that the connections have the minimum length of wiring, regardless of the order in which they were specified. The order in which the routes are completed is of vital importance if there is to be any chance of completing the board wiring. A high priority is allocated to those connections which have little choice of

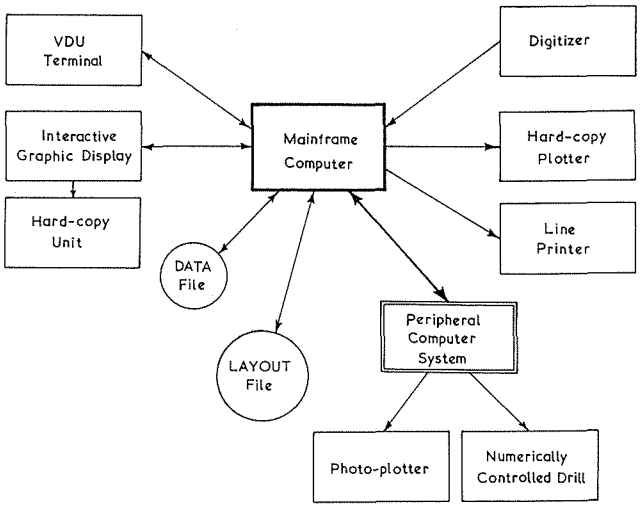



Figure 4. WREPCL operation facilities.

path, that is, the shortest connections are made first.

During the determination of paths in the vertical/horizontal algorithm eight attempts are made to connect two points, four in each direction. With each attempt the initial starting direction is altered. A retries option permits deeper search attempts to be made in order to complete connections that were not made on the first pass.

An optimization routine optimizes paths in order to reduce the number of interconnections between the two sides (layers) of the board and hence simplify the manufacturing requirements. The resultant board may contain vertical and horizontal conductors on both sides of the board provided of course that no paths cross.

When using the modified Lee algorithm interconnections are first attempted in an order that depends on the order in which the wiring data is submitted. For any one node multiple target points are available for interconnections. If previously fitted branches block the way, some earlier branches are removed and an attempt is made to fit the branches in another order before deferring one or more to subsequent layers (if the board is a multi-layer board). The modified Lee algorithm assigns the path between two points to one layer only.

The path routing procedure terminates after all paths have been completed or when the remaining interconnections are too difficult for the search methods to find in a reasonable time (CPU time). It is more economical to place the few uncompleted paths manually during an editing stage.

Although the vertical/horizontal method and Lee's algorithm may, in general, be applied individually to multi-layer boards, WREPCL has been designed to use the vertical/horizontal method for the first two layers and the Lee algorithm for any subsequent layers. This is due to the fact that a minimum amount of software change was required to combine the use of the routing algorithms in this way.

## 6. OPERATION

The WREPCL program interacts with an operator who responds to a series of questions from the computer by inserting design details. Prior to running the program

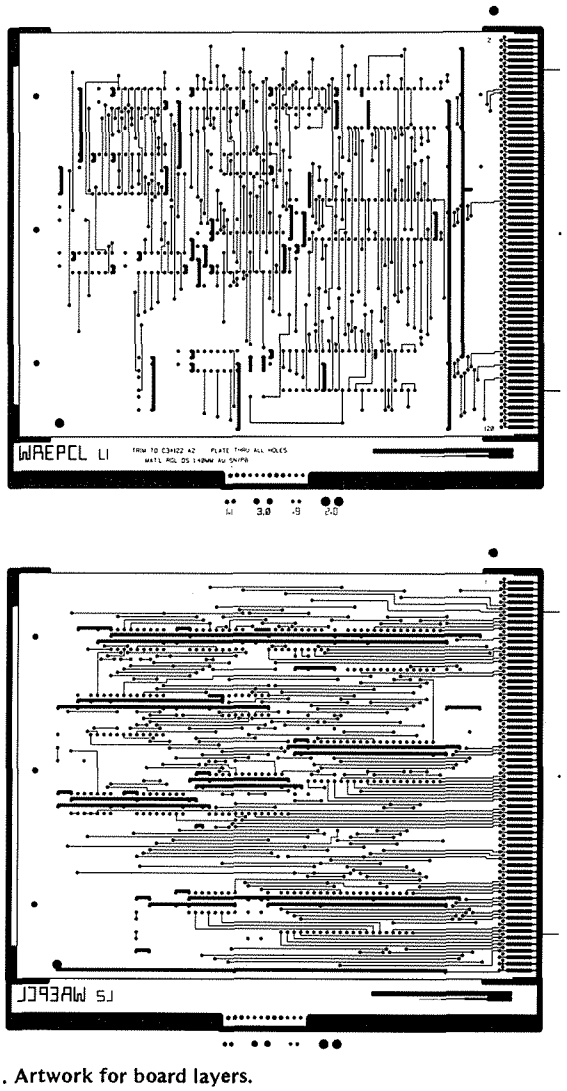



Figure 5. Artwork for board layers.

the layout designer should establish the positions of the components on graph paper and show the desired positions of the power wiring. This helps to streamline the layout operation when using the interactive graphic display terminal or digitizing tablet.

A components library is provided since components with the same number and position of pins occur extensively. Component types are easily placed by merely indicating position on the board and library reference.

Following manual placement of the power wiring (and other desired pre-placed wiring) a list of the desired signal interconnections (nodes) between components is entered into the computer and data verification carried out. The interconnections are then submitted to the automatic path routing procedure. Following routing the operator is informed of the interconnections that could not be placed (normally less than five per cent). A hard-copy plot is obtained at this stage so that the layout designer can study the result for correctness and for available space in which to route paths not placed by the routing algorithm. The path routing phase reads the DATA file and creates a LAYOUT file.

The board layers are then displayed on an inter-

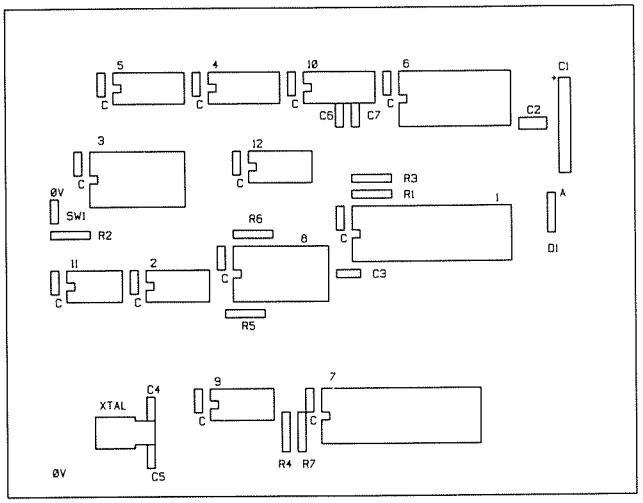



Figure 6. Artwork for overlay.

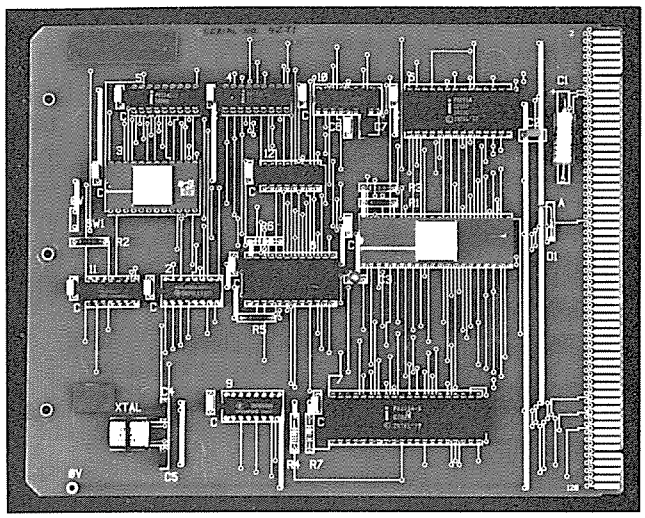



Figure 7. Completed board.

active graphic display terminal. Unplaced interconnections are inserted together with other necessary information required for board identification and manufacture. A component assembly diagram (Overlay) is produced by WREPCL and it may also be displayed and modified. The Editing phase updates or modifies the LAYOUT file. A hard-copy plot is obtained from the LAYOUT file in order for the layout designer to check for completeness and correctness of paths entered during the Editing phase.

On completion of the layout the operator creates the "manufacture" files used by the post-processing peripheral computer system.

## 7. CAPABILITIES

The WREPCL program provides the following major characteristics:

1. Layouts for PCBs up to a size of 380 mm square.
2. Choice of layout grid spacing, namely, 2.54 mm (0.1in), 1.27 mm (0.05in) or 0.64 mm (0.025in).
3. Three sizes of conductor widths.
4. Five pad (land) sizes for component mounting.
5. Incorporates a components library.
6. Optional interactive insertion of components, conductors and text.
7. Provision for future inclusion of automatic component placement and schematic diagram generation.

Figures 5 and 6 show the artwork produced for a board designed with the aid of WREPCL. Figure 7 shows a completed board. CPU time for automatic path routing was 15 seconds using an IBM 370/3033 computer. In general, artwork generation time is halved when using the computer aided design system.

Although WREPCL can lay out paths for multi-layer boards, a multi-layer board has not yet been manufactured.

## 8. CONCLUSIONS

The Design Automation process combined with Computer Aided Editing can be a practical aid to designing PCBs if the results can be obtained readily and economically – that means:

1. Easy access to a computer facility (and interactive graphics).
2. Fast turn-around times, and
3. Low processing costs.

## ACKNOWLEDGEMENT

The author would like to thank the many personnel at the Defence Research Centre, Salisbury, who gave advice or help towards the development of the WREPCL program.

## REFERENCES

AKERS, S.B. (1967), "A modification of Lee's Connection Algorithm", *IEEE Trans. Electron. Comput.*, Vol. EC-16, pp. 97-98.

DUNNE, G.V. (1966), "The Design of Printed Circuit Layouts by Computer", *Proceedings of the Third Australian Computer Conference*, pp. 419-423.

FISK, C.J. (1967), "ACCEL – Automated Circuit Card Etching Layout", *Proc. IEEE*, November 1967, pp. 1971-1982.

HIGHTOWER, D.W. (1974), "The Interconnection Problem: A tutorial", *Computer*, April 1974.

HOEL, J.H. (1976), "Some Variations of Lee's Algorithm", *IEEE Trans. on Computers*, Vol. C-25, pp. 19-24.

LEE, C.Y. (1961), "An Algorithm for Path Connections and its Applications", *Trans. IEEE*, Vol. EC-10, No. 3, pp. 346-365.

ROSA, R.C. and LUCIO, T.P. (1979), "Programs for two-sided printed circuit board design", *Computer Aided Design*, Vol. 11, No. 5, September 1979.

RUBIN, F. (1974), "The Lee Path Connection Algorithm", *IEEE Trans. on Computers*, Vol. C-23, pp. 907-914.

WHATMOUGH, R. (1972), "A method for automatic layout of the conductor paths of printed circuit boards", Dept. of Supply, Weapons Research Establishment, *WRE-Technical Note 112* (AP), August 1972.

ZANE, R. and HARRELL, D.A.W. (1968), "PUZZLE – Computer Aided Design of Printed Circuit Artwork", University of California, UCRL-18172, April 1968, SWM 6533U.

## BIOGRAPHICAL NOTE

*The author is a Senior Technical Officer in Digital Systems Engineering Group, Advanced Engineering Laboratory, Defence Research Centre, Salisbury. After attaining a drafting background and experience in electronic equipment design and development the author has, since 1970, been involved with system design and software development for engineering applications of large and small computers. The author has undertaken studies in electronics and computer programming at the South Australian Institute of Technology.*

# Letters to the Editor

*The Australian Computer Journal welcomes Letters to the Editor for publication, either to comment on the contents of recent issues, or to discuss topics in computing likely to be of interest to the Journal's readers. Letters are not refereed. However, the Editor reserves the right to make changes of a stylistic nature to any letters published, or to return letters for modification should they contain anything regarded as unsuitable for publication.*

## MAY EDITORIAL CRITICISED

I should like to make a few comments on your editorial in the May 1980 issue of the Journal. Whilst I found the first part of the editorial very enlightening, as a "desk-chained program-progressor" I found the latter part rather obnoxious and insulting to the computer practitioner. It is a shame that the views you express are all too typical of those prevailing in the Australian computer science academic community. Your contention that all worthwhile research originates from universities is, I feel, extremely tenuous. As far as I am aware, most of the more important developments in computer science have had their birth in private research laboratories around the world, rather than in computer science departments of universities. I would seriously question just how much university-originated research is really incorporated into "tomorrow's computing systems". Even in Australia I am personally aware of one large company which has carried out quite extensive research as well as the development of products which were not yet known on the world market. Naturally, due to the highly competitive nature of the industry, such work is very seldom publicised. Perhaps this can excuse the arrogance of academics in believing that they are the source of all enlightenment in the computing community.

In order to overcome such misconceptions I believe there should be a much closer co-operation between industry and universities in Australia and would suggest that this would be a field in which the ACS should become active.

*Greg Ward, MACS,*
*AEG-Telefunken,*
*Darmstadt, West Germany*

*Editor's Remarks:*

*Although the Editorial under criticism was written by my predecessor, and indeed I do not share all its sentiments, I must rise to its defence. The Editorial did not claim that academics are the only ones who do research. Rather, the Editorial stated that the value of research has not been given adequate appreciation by most practitioners, a situation which is unfortunately all too true.*

## COMMENT ON TAB ARTICLE

I refer to the essay, "Computer Applications in the Totalizator Industry in Australia" (Vol. 12, No. 3, August, 1980) from Dr D.L. Overheu. These notes of Dr Overheu contain a number of errors of fact and omission relating to the computing activities of the Totalizator Agency Board of NSW.

This Board ordered a dual 360/44 with 10, 2314 disks from IBM Australia in 1969 and the system commenced on-line operations, as scheduled, in August, 1971. Two 2311 disks were installed, on lease, for a brief period in 1970, pending delivery of the 2314 type. The first non-metropolitan or country branches, in the Hunter Region, to achieve on-line selling cutover in 1977, not 1973 as stated in the Journal.

Maintenance of terminals, mini-computers, multiplexers and associated equipment was commenced by our own staff in 1976, however, mainframe maintenance is still carried out by the supplier. Thus, it is not accurate to state that there was "total supplier dependence at NSW/TAB" in 1977. This Board took delivery of 370/135s in 1975/76 not 370/138s as stated. The IBM Series 1 mini-computers are being used for a Master Collator and Display system but they were never envisaged for use in dividend calculation.

It is understood that IBM determined to vacate the totalizator terminal field *after* this Board decided upon the A.W.A. MRT-2 mark sense reader as IBM was a tenderer for the contract. It may not be generally known that the MRT-2 was developed by A.W.A. to the specification of this Board. We have never made "studies . . . of the Alpha language"; however, it is referenced within a quotation in a paper prepared by our staff in 1978.

*A.J. Windross, MACS,*
*Automation and Research Manager*
*NSW TAB*

*AUTHOR'S REPLY:*

Mr Windross's concern for accuracy is appreciated. However, it should be pointed out that gaining factual information about TAB computer developments is extremely difficult[1]. My article makes extensive use of several informed sources. Apart from typographical errors we must therefore assume that these sources have sometimes been in error.

I would emphasise that the article was about general trends that seem to be occurring in the industry and the difficulty of satisfying some of its needs. These trends have been towards some form of distributive processing, often with mini-computers, self-operated cash sale terminals for off-course, but not generally for on-course, and towards the use of higher level programming languages rather than assembly language for system development.

On the particular question of the MRT-2 terminal I have no doubt that had IBM wished to do so they could have gone into a redevelopment of their mark-sense terminal and a competitive marketing situation. My information by personal communication is that AWA spent some time talking with *all* TABs before deciding on the MRT-2 development. This company has considerable experience of its own in totalizator systems and their requirements.

1. However the following further reference has surfaced.
VICKRESS, Frank, 'An approach to design and specification of a real time on-line Data collection and processing system'. Proceedings of Fourth Australian Computer Conference, Adelaide, SA, August 11-15, 1969, VI, p. 43.

*D.L. Overheu,*
*Canberra CAE,*
*Board Member, ACT/TAB*

# Book Reviews

*Infotech State of the Art Report, Micro-Computer Software.* Infotech International Ltd., 1979. £130.
Vol. 1, Analysis and Bibliography. 223 pp; Vol. 2, Invited papers. 282 pp.

Infotech is a UK-based organisation which regularly publishes reports on topics in computing which are judged to be of current interest. These two volumes are about microcomputer software and follow the standard Infotech format in which a number of experts are invited to submit papers, 15 of which make up Volume 2. An editor, in this case R. Dowsing of the University of East Anglia, then takes extracts from these and other appropriate papers and rearranges them together with a commentary to form a comparative analysis of the subject which then forms Volume 1.

This publication is a qualified success. The invited papers in Volume 2 seem at first to be somewhat unco-ordinated. Some (e.g., 'A Simple Microprocessor Task Monitor', by D.M. England) describe simple techniques which have been in use for decades on larger systems, whereas others (e.g., 'A Guide to Communicating Sequential Processes', by S.S. Kuo, M.H. Linch and S. Saadat) describe theoretical work which has not yet been fully implemented on any sized system. One paper, 'Microprogram Assemblers for Bit-Slice Microprocessors', by V.M. Powers and J.H. Hernandez is a comprehensive survey of available products in a particular area, whereas another paper, 'Forth Programming Language for Real-Time Mini and MicroComputer Applications', by E. Rather is a detailed description of a particular proprietary product. 'Microcomputing – The Software Dimension' by P. Hazan is a speculative paper about how microcomputers may develop, whereas 'Totally Portable DP Software' by C. Hawkins describes practical experience gained with a portable COBOL-based system. However, on reflection, it seems that these papers may accurately reflect the current state of confusion over microprocessor software. On the one hand, it can be argued that microcomputer software ought not to be considered a subject in its own right, as the same principles should apply as on larger systems; on the other hand, those struggling to implement applications on limited hardware with inadequate aids know that, for the moment anyway, microcomputer software is different.

Unfortunately, the diversity of the invited papers makes the comparative analysis in Volume 1 rather unsatisfactory; there is just not enough common ground between the papers to make a point-by-point comparison of items useful, in spite of the fact that liberal use has also been made of excerpts from other volumes in the Infotech series. Volume 1 does, however, contain a useful bibliography of some 111 papers.

To conclude, if you are looking for comprehensive coverage of the subject, full of 'how to do it' hints, then these volumes would be a disappointment. If however you can use an up-to-date collection of well written and thoroughly readable papers which explore a number of interesting areas in microcomputer software, then this somewhat expensive publication can be recommended.

*David Rowe,*
*Monash University*

*Microprocessor Applications: International Survey of Practice and Experience.* Infotech International Ltd., 1979. 359 pp. £49.50.

The major part of this publication is precisely what its title suggests: a collection of 20 papers describing a broad spread of microprocessor applications. This volume is not in the standard 'Infotech State of the Art Report' format in that it consists of only one volume and does not have the usual comparative analysis. This is possibly because there is not enough common ground between the applications described in the papers to make a comparative analysis useful. Instead the volume opens with a 42-page editorial introduction titled 'Basics' which is largely devoted to the hardware technology of microprocessors. This is curious as it does little to illuminate the papers where the emphasis is on applications, with the microprocessors themselves barely rating a mention.

The 20 invited papers have been carefully chosen to illustrate a range of applications in which the microprocessors are used as replacements for dedicated logic, rather than as small-scale general-purpose computers. There are three groups of papers: the first describe applications in which the microprocessors are used for monitoring and data collection, e.g., in a nuclear reactor, for meteorological conditions and for human lung measurements. The second group of papers describe applications where the microprocessors are used to control some device, e.g., an autoclave used in pharmaceutical manufacture, a turbo-jet engine and a hospital drip-feed unit. The final group of papers describe rather more complex uses of microprocessors in a number of areas, e.g., medical, telecommunications and broadcasting fields. The emphasis in most of the papers is on the nature of the problems which led to the adoption of microprocessors in each case and the advantages gained in doing so; in most papers there is less emphasis on the details of the implementation. All of the papers are well written and are quite readable.

A refreshing feature of the papers is their truly international flavour; there are papers from Brazil, Eire, France, Holland, Italy, Japan, UK and USA. It seems that microprocessor applications is one of the few areas in computing which avoids the usual American domination.

The volume also contains a bibliography of 87 references and an eight-page glossary of technical terms.

To conclude, this is not a reference book on how to use microprocessors, but it is a useful introduction to the range of applications in which microprocessors may be used to advantage. It is regrettable that the price is so high as similar information is available in a range of other publications.

*David Rowe,*
*Monash University*

E.E. Swartzlander (1979), *Computer Arithmetic*, Dowden, Hutchinson and Ross (distributed by Academic Press). 378 pp. $45.00.

This book is Volume 21 in the Benchmark Series in Electrical Engineering and Computer Science. It contains 43 papers on digital arithmetic as implemented in digital computers. These have been carefully selected by the Editor from papers published over a 30-year period, and are considered as "benchmark" papers in the field of Computer Arithmetic. The papers are reproduced in full, and are presented in seven parts with the following headings: I: Overview, II: Addition and Subtraction, III: Multiplication, IV: Division, V: Logarithms, VI: Elementary Functions and VII: Floating-Point Arithmetic. The papers in each part are preceded by the Editor's comments. These provide an introduction to the historical development of the subject matter, a summary of each paper's contribution to this development and a list of other important and related papers not reproduced in this book. The book's bibliography contains over 250 references.

The Benchmark Series serve three major purposes: firstly to provide a practical point of entry into an area of research, secondly to provide a convenient means of study of areas related to the reader's principal interests, and thirdly to provide a compact collection of the major works on which the reader's present research activities and interests are based. This volume on Computer Arithmetic clearly serves these purposes. In these times of very rapid technological advances in which more and more applications are based on digital techniques (including digital arithmetic), this book will undoubtedly be of immense value. Teachers in tertiary educational establishments will find this book to be not only an excellent reference but also a book on which an advanced course in Electrical Engineering or Computer Science can be based. Designers of digital systems will find this book most useful. It is an excellent book which should enhance the value of all public and private collections.

*D.G. Wong,*
*University of Sydney*

G.V. Rao (1978), *Microprocessors and Microcomputer Systems*, Van Nostrand Reinhold. $24.50.

It feels somewhat strange to review a book in 1980 first published in 1978 and prepared and written in 1977 or earlier. Therein lies the problem with any book in this constantly changing area of advanced technology. As such, it does not, or could not, cover the new world of 16 bit "chips" such as the Intel 8086, Zilog 8000 or Motorola 68000. In another way, however, this superbly produced book is of value even if you may expect to see it in a "remainder sale".

The book consists of some 14 chapters and as is unusual

for this type of book it covers very well the fundamental physics of large scale integration and chip fabrication. At all points in the book liberal use is made of detailed diagrams and tables, with some form of illustration on almost every page.

Chapters 1 to 3 cover the basic electronics of solid state devices although at times the coverage may be sketchy. There are some problems at times where two different concepts or principles are introduced together in these early chapters. For example, in the section (3.7) related to electron-beam addressable memory the concepts of electron-beam mask fabrication, lithographic techniques and ion-implantation are also introduced in a section that is only nine lines long. In this sense the background of the book comes to the fore, i.e., it was created from a set of lecture notes. Chapters 4 to 6 cover memory and peripheral topics, but once again the date of the book is notable as the Intel MCS-85 chip set is covered in a whole section. Chapter 7 is a list of microprocessors available at the time and each gets a small summary paragraph. In most cases this works well but the coverage of "bit-slice" technology, via the AMD-2900 series, suffers by being covered in less than two pages. For software-oriented readers, Chapter 8 provides a sketchy and somewhat disoriented introduction to the software available on microprocessors. However, the description of PASCAL as a language capable of describing data and procedures in "pidgin English" is rather unusual as are many other descriptions of hardware and software elements in the book. Chapter 9 is occupied by a rather pointless reference chart on available microprocessors at the time. Chapter 10 does contain a valuable set of diagrams on basic digital logic. The remaining chapters cover display units, reliability and application.

The short "fly leaf" description of the book claims that the book has been designed to facilitate communication between a number of groups including hardware and software people, marketing and training groups, "students, faculty and laymen as this science enters every facet of the home and industry". About the only thing I can agree with here is the last phrase in relation to the so-called "microcomputer revolution". The book is too fragmented and sketchy for any really serious use and unfortunately it has dated badly because of its extensive use of comparative tables. The most rewarding section of the book is really its first few chapters on the basic physics and electronics of advanced LSI technology. Providing one can ignore such excesses in style as a description of nuclear fusion as a provider of "superenergy" a perusal of the arguments in these chapters is worthwhile.

*W.J. Caelli,*
*Electronic Research Associates, Queanbeyan, NSW*

*ICL Technical Journal*, Volume 1, Issue 2, May 1979

Well, they've done it again! In my review of the first issue of this new journal, I expressed my pleasure at seeing a technical journal that was addressed not only to academic people like the reviewer, but also to the profession at large. My pleasure is reinforced by this second issue.

It contains six papers of tremendous variety. The first, Computers in support of agriculture in developing countries by G.P. Tottle, gives an interesting insight into the problems experienced by developing countries. His conclusion that computers can be used in these areas seems well justified, even though it is quite contrary to conventional wisdom. The paper which interested me most was the second, Software and algorithms for the Distributed-Array Process by R.W. Gostick. The DAP was briefly described in the first issue, and this paper fills out many of the details. As well as describing the DAP, the author presents the solution as four classical problems: matrix multiplication, searching an array, traversing a graph and sorting. Although the description of the algorithms is a little weak, perhaps due to space considerations, the programs themselves are quite clear and illustrate the DAP very nicely. As an educator, I was delighted to see that the algorithms needed for these problems underline the importance of teaching general principles in algorithm design, as well as the currently best techniques. System performance is always an important consideration and the next two papers, Hardware monitoring on the 2900 range by A.J. Boswell and M.W. Brogan, and Network models of system performance by C.M. Berners-Lee, address the topic. The first gives a clear description of the TESDATA 1187 monitor, and examples of its use. The second describes a class of models called FAST (which means Football Analagy of System Throughput). This paper is the only one in the journal which will place excessive demands on the general reader. The fifth paper, Advanced technology in printing: the laser printer by A.J. Keen, describes, as its title suggests, ICL's LPS-14 laser printing system. As one condemned to a 300 lpm printer, I found the description of a 10,000 lpm printer quite fascinating. The final paper, The next frontier: three essays on job control is by David Barron, the only author not on ICL's staff. We are accustomed to entertaining papers by this author — and we are not disappointed here. Perhaps the best way to illustrate both the style and the content is to give two quotations.

> " 'Man is born free but is everywhere in chains', said Rousseau. A recent candidate in a computer-science degree examination put it more succinctly: 'The function of the software is to prevent people using the hardware'."
>
> "They (operating system designers) should not build into the operating system a preconceived idea of how it is going to be used, least of all if (as is most common) they have never been users themselves."

All round, another very impressive issue.

*J.S. Rohl,*
*University of Western Australia*

Bauer, F.L. and Broy, M. (eds) (1979), *Program Construction*. Lecture Notes in Computer Science 69, Springer-Verlag. 651 pp. $US29.70.

The volume presents material from an International Summer School under four headings which categories the main topics and indicate their formal nature.

I The Thinking Programmer: Interplay between Invention and Formal Techniques.
II Program Verification: Proofs, Programs and their Development — The Axiomatic Approach.
III Program Development by Transformation: From Specification to Implementation — The Formal Approach.
IV Special Language Considerations and Formal Tools: Languages as Tools — Interactive Program Construction.

On first reading, sections II and III were found to be the most informative and contained surveys of the two main approaches to program construction. Some of the papers can be read with few prerequisites but to get the most benefit from the volume an appreciation of the notation of mathematical logic, including lambda calculus, McCarthy (1963), and of structured programming, Dijkstra (1976), is recommended. Good references are provided with most papers.

Section II develops the rules for verifying the correctness of a program which are commonly associated with the term "structured programming". It follows with an account of the programming language Euclid which was evolved from Pascal and which incorporates these proof-rules. The section ends with the applications of the method to the development of examples in concurrent programming.

Section III, of 257 pages, introduces a second method of developing correct programs based on transformation rules for converting a piece of program to another form whilst preserving its semantic meaning.

This can be used to express language constructs in terms of more elementary ones for which correctness is readily established. The method is of importance in the formal study of semantics and the implementation of programming languages. The articles provide a survey of transformation methods with simple examples in a coherent form which offers a good starting point for study.

Sections I and IV contain some general articles but mainly provide additional material to expand II and III respectively.

By presenting the two techniques outlined above, the work gives a valuable perspective of current (1978) activity and will repay serious study by students of advanced programming. In particular it should encourage the use of verification in structured programs and secondly makes material on transformation methods more accessible.

## REFERENCES

DIJKSTRA, E.W. (1976), *"A Discipline of Programming"*, Prentice Hall, Englewood Cliffs, N.J.

McCARTHY, J. (1963), *"A Basis for a Mathematical Theory of Computation"*, in P. Braffort, D. Hirschberg (eds), *Computer Programming and Formal Systems*, North-Holland Publishing Co., Amsterdam.

*D.P. Hodgson,*
*Western Australian Institute of Technology*

Deutsh, D.R. (1979), *Modelling and Measurement Techniques for Evaluation of Design Alternatives in the Implementation of Database Management Software*, National Bureau of Standards, Washington, D.C. 231 pp. $7.70.

This text is a published Ph.D. thesis and, in consequence, cannot be classified as light reading for the data processing professional interested in evaluating database software. Its market is to be found mainly in universities and CAEs with computing departments engaged in database research. The author describes a methodology which may be followed in database management systems (DBMS) design in order to arrive at a product which is both performance and cost efficient. A simulation modelling system, called the set processor performance model (SPPM), is described using SPPM, it is possible to design a DBMS and evaluate its potential performance characteristics without actually constructing it.

SPPM consists of over 200 Fortran modules that run on a PDP-10 under the TOPS10 operating system. The author claims that SPPM could be implemented on any similar system with a Fortran compiler. Several pages of the text are devoted to defining what many of the modules do, which I found rather tedious. However, for a research student involved in DBMS simulation these would be important. I was rather disappointed at the author's failure to apply his method in making a comparison of existing commercial DBMS systems. With this criticism in mind, I consider chapter 8 to be the most interesting part of the book, with its discussion of model evaluation. Also discussed in the chapter are methods for validating the performance of a simulation program.

Overall the text would make a solid starting point for a research student wishing to undertake a DBMS project. Over 170 references are cited in the bibliography. Also, the thesis does provide strong evidence that simulation is a powerful tool that the data processing profession can employ to evaluate potential systems, whether or not they are databases.

*D.J. Hubbard,*
*Bendigo College of Advanced Education*

Boyer, R.S. and Moore, J.S. (1979), *A Computational Logic*, ACM Monograph Series. Academic Press, New York, 397 pp. $41.30.

This latest volume in the excellent ACM Monograph Series summarises many years work by the authors into the discovery of techniques for performing proofs by induction. In particular, they describe heuristic techniques for proving properties of recursively defined functions over inductively defined objects. Since examples of inductively defined objects include integers, sequences, lists, trees, expressions and formulas, the techniques described are immediately applicable to proving properties of programs which operate on such objects. More generally, the techniques are applicable to any proofs by mathematical induction.

Although the authors carefully define the logical system they use, the main contribution of their book is the description of the heuristics they use in proofs, and their techniques for using previously proved theorems as lemmas. All their techniques are illustrated through a well-chosen set of examples and have been implemented in a large Lisp program capable of proving all the theorems described in the book.

Examples of such theorems include:

```
(IMPLIES (PLISTP  X)
         (EQUAL (REVERSE (REVERSE X) )   X) )
```

and

```
(EQUAL  (FLATTEN (SWAPTREE A) )
        (REVERSE (FLATTEN  A) ) )
```

where PLISTP, REVERSE, FLATTEN and SWAPTREE are all recursively defined functions over lists or trees. The climax of the book is the description of the proofs of four complex examples: the correctness of a theorem prover for propositional logic, the correctness of a simple optimizing compiler for arithmetic expressions, the correctness of a fast string matching algorithm (written in an imperative programming language), and the unique factorization theorem (any positive integer can be represented as the product of a finite sequence of primes, and any two finite sequences of primes with the same product are permutations of one another).

The main heuristics used by the authors to prove a given formula are the following:

- Simplify the formula by applying axioms, "rewrite" lemmas and function definitions.
- Replace "destructive" functions (e.g., CAR) by "constructive" functions (e.g., CONS).
- Use equalities and then throw them away.
- Generalize the formula by introducing variables for terms that have "played their role".
- Eliminate irrelevant terms from the formula.
- Use induction to split the formula into two or more simpler formulas to be proved.

Each of these heuristics is applied in turn. If any heuristic succeeds in changing its input (without proving it), the whole process is repeated on each of the output formulas. The process terminates when there are no formulas still to be proved (success) or when a formula is recognizably not a theorem or fails to be changed by this process (failure).

Each of these heuristics is carefully described in one or more chapters. In each case the explanation is exemplary: first some simple examples to motivate and informally describe the heuristic, then a precise description of the heuristic, and finally a more complex application of it. The first theorem presented above is used as a running example in these descriptions.

Generally, the presentation and style of the book is excellent. The authors write clearly and simply, motivating all their descriptions with well-chosen examples. That parts of the book are still difficult to understand is a consequence of the complexity of the subject matter. One interesting aspect of the book is that some sections consist entirely of the output of the authors' theorem-proving program which describes its proof attempt in perfectly clear, understandable English (refuting a common criticism that proofs of program properties are always unreadable).

An appendix contains a complete list of function definitions accepted by the system and theorems proved by it. An excellent bibliography and index complete the book. In reading the book fairly carefully, I detected one misprint. A minor quibble is the authors' perpetuation of the Lisp tradition of using CAR and CDR as selectors whatever the abstract data type being considered.

Despite the value of the heuristics presented, and the success of the authors' theorem-proving program, it is clear that much research remains to be done in this field. The following weaknesses in the system presented stand out:

- The necessity for some "bridging" lemmas in the more complex proofs. These lemmas do not actually state useful facts in their own right, but direct the theorem prover's course of action. Such lemmas require too much understanding of the theorem prover by the person using the system.
- The necessity in one proof to define some functions unnaturally (e.g., using [ID X] where ID is the identity function) so that the induction heuristic will choose the "right" induction scheme.
- The inability to generalize constants when required.
- The necessity to state how theorems should subsequently be used as lemmas.

These are all difficult problems to overcome and I look forward to seeing the authors' subsequent attempts to solve them.

In summary, "A Computational Logic" is an excellent description of the techniques the authors have developed for performing inductive proofs in a variety of domains automatically. Readers with no previous experience in automatic theorem proving would find this book a good starting point. Anyone remotely interested in mechanical program verification or the mechanization of mathematical proofs in general should read it, and people seriously interested in these topics will want to have their own copy. Every computer science and mathematics library should certainly have one. I recommend it.

*R.W. Topor,*
*Monash University*

Maynard, J., *Computer Programming Made Simple*, W.H. Allen, London. 2nd ed. 1980. 350 pp. £2.50.

To my mind *Computer Programming Made Simple* is not a simple book. The author states that it will be of interest to students at schools and further education establishments but I believe the average school student would find it generally unappealing. They would not be inspired to read any further than the front cover. It is a thick text book with small type, few diagrams, and several out-of-date photographs. In fact, this is a library book or teacher reference. I have looked at so many books that make similar claims. They are usually written by tertiary level authors for a tertiary level market but attempt to sell to a wider market by stating on the cover that the book would be of interest to schools, too. How-

ever, these two target audiences are not compatible. Although the concepts are clearly explained, the very presentation of the book excludes it from the school market. It would need to be more visually stimulating to attract the average student's attention.

To be critical on a second point, the book is a second edition of a 1972 publication. The publisher claims that the book is completely updated and has an additional 50 pages on microprocessors, home computers, and the BASIC programming language. In reality this statement means that two short chapters on micros and BASIC have been tacked onto the end of the book and two photographs of micros inserted into the photograph section. The section on micros is extremely brief and although it carefully and precisely provides an understanding of the jargon of micro-computing, it goes into too many technical details for a book whose title is *Computer Programming Made Simple*. The section on BASIC is also brief and refers to a fairly "mickey mouse" version of the language.

For an updated edition, comments like:

*"Punched cards and punched paper tape represent between them something over 95 per cent of the input media currently in use in commercial computer installations".* *(p. 15)*

*"Keyboards cannot be considered for inputting a large quantity of data".* *(p. 15)*

*"A typical disk pack can hold about 30 million characters of information".* *(p. 25)*

and on multiprogramming:

*"The number of programs involved . . . would typically be three for a medium-sized computer and up to about 16 for a large computer . . . "* *(p. 45)*

would seem to be incorrect.

On the good side, the book does have an understandable introduction to general computing concepts and definitions of computer terms in part 1 (60 pages) before it starts into its main objective, computer programming, which consists of an extensive introduction to COBOL (150 pages, the major part of the book) plus short sections on FORTRAN and BASIC. There are review questions at the back of the book, along with an index and glossary.

In summary, *Computer Programming Made Simple* is a fairly thorough reference text, extensive in terms of the ground it covers, but slightly out of date and not totally suited to the average school student or to the general public as stated in its publicity.

*John Read and Sandra Wills,*
*Elizabeth Computer Centre,*
*Education Department of Tasmania*

Tucker, A.B., *Text Processing – Algorithms, Languages and Applications*, Academic Press, 1979. 171 pp. $23.

This is a useful little book that kills two birds with one stone. Its avowed purpose is to introduce computer text processing. Useful as this may be, I rather think that the book will be more valuable in another context: to introduce *computers* to an audience which is neither mathematically nor commercially oriented, using text processing tasks as illustrating examples. So if you are looking for a textbook for an introductory programming course for typists, librarians or politicians, give this one a try. It would be far more suitable than alternative books of more conventional orientations.

The first chapter provides a brief introduction to text processing and computer hardware/software. Although very short, the part on computers does manage to get the idea across quite effectively. Chapter 2 gives a reasonably complete discussion of the basic features of PL/1. While the mathematical capabilities of the languages are necessarily omitted, all the control structures, includ subroutines, are introduced and explained in an elementary but clear fashion, together with, of course, the text manipulation features of PL/1. With some augmentation by the teacher this would do very well as an introduction to PL/1. Chapter 3, on SNOBOL, is not quite as good, and requires greater effort on the part of the reader. To take two examples, the list of pattern matching function descriptions on page 91 is a bit obscure, and the deferment of examples to a later section rather leaves the reader up in the air. On pages 96-97 the same symbol S is used to mean three different things all within short space of each other, and it would have been helpful if the author had spaced the examples of page 96 apart and separated out the two definitions of P and PUNC from the pattern matching statements – it took me some time to work out that the definitions are not part of the pattern match. However, a good teacher would no doubt be alert enough to point these problems out in class to save the students' confusion.

Chapter 4 provides brief descriptions of the packages KWIC, FUMULUS, and SCRIPT, and the IBM text editor CMS. The first two are for bibliographical processing, the third for text preparation, and the last is general purpose, which could form the basis of an introduction to on-line computer use for students not specially interested in the previous three packages. Chapter 5 reviews the available literature on common programming languages (the word PASCAL appears three times!), packages, and current research, and refers to a total of 87 books and articles on various aspects of the subject.

Each chapter has an extensive list of exercises, and selected answers are provided. There are also three appendices on ASCII/EBCDIC codes, tape and disk I/O, and a short glossary of 18 text processing terms. Lastly there is an index, unfortunately too sparse for looking up language constructs. Indeed the whole book could use a bit of expansion. The book has been carefully produced, and *very* carefully proof read. It is, to repeat, a useful little book.

*C.K. Yuen*

Rozenberg, G., and Salomaa, A., *The Mathematical Theory of L Systems*, Academic Press, New York, 1980. 352 pp. $38.

An L system (named after Aristid Lindenmayer, of the Theoretical Biology Group at Utrecht), is a rewriting system, of the general type that we associate with the name of Post, but with each symbol in the generated string rewritten at each discrete time interval, whereas the usual Post productions used in formal language theory rewrite only a single symbol at a time. This property of *parallel rewriting* is shared by cellular automata, but the individual cells in cellular automata merely change state, whereas a symbol in an L system may expand into several symbols, analogous to the multiplication of cells. L systems have a practical interest as models of aspects of the development of organisms, or portions of organisms.

The book under review is by two major researchers in L systems. Grzegorz Rozenberg was a pioneer in the study of L systems, and Arto Salomaa has produced important results in several areas of automata theory and formal language theory, including L systems. As one might expect from two such experienced researchers and authors, this book is both well-written and authoritative. It belongs in every library which pretends to cover theoretical computer science or mathematical biology.

Having given the book some well deserved praise, I must now point out that it is not a book by which the average interested reader would be advised to venture into the field for the first time. Its subject is indeed the *mathematical* theory of L systems, and only the slightest motivation is provided. An earlier book by Rozenberg, in collaboration with Gabor Herman, called *Developmental Systems and Languages* is still the best place to start (with the introduction contributed to that volume by Lindenmayer especially worthwhile reading).

L systems today are divided into a bewildering number of variations, the most mathematically tractable of which are designated by number-letter strings ending in "OL". The "O" merely means that each symbol is rewritten without reference to adjacent symbols (context freeness, in the formal language sense). The L, of course, designates the parallel rewriting mentioned above.

Starting with a formal language in the usual (Chomsky) sense, one can proceed to a "pure grammar", in which the nonterminal symbols are eliminated. A pure grammar to generate $\{a^+\}$ would consist of a "start set" $\{a\}$ and a single production rule $a \rightarrow aa$. Notice that this is a deterministic system, in that there is only one production rule that can be applied. If an ordinary context free grammar is deterministic, it generates only a finite set of strings, but this is not true for pure grammars. Now let us consider the parallel rewriting interpretation: at each instant of time, each a in the string is rewritten as two a's. The result at time $n + 1$ a's, so we say the grammar generates $\{a^n \mid n > 0\}$. Interestingly, this language cannot be generated by any pure grammar, which gives us an example of the power of parallel rewriting systems.

The system described above for generating $\{a^n \mid n > 0\}$ is an example of a DOL system (the D standing for "deterministic"). Since it has no erasing productions, it is called "productive" and thus is a PDOL system. If it were productive but not deterministic, it would be a POL system. One can reintroduce the idea of nonterminals and get what is called an EOL system. One can also generalise by allowing in the grammar one or more *sets* of rewriting rules that can operate on the string, but with rules from only one set operating on a given string at a given time. These systems are called TOL (the T standing for "table") systems. If the rewriting rules are deterministic within each set, then the systems are DTOL. A TOL with nonterminal symbols is an ETOL.

And so it goes on.

The relationship between the various families of languages generated by the systems mentioned are, incidentally, as shown in Fig. 1.

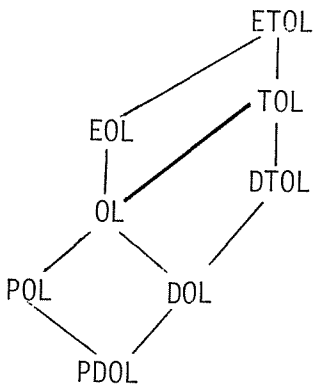



Figure 1

(The lines downward in the diagram indicate inclusion of the lower family in the upper one). There are many other families mentioned in the book, and their profusion tends to be a little confusing. My main reason for preferring the presentation of Herman and Rozenberg is that this vast array of systems is much easier to sort out when presented with at least *some* biological motivation. (This holds true even for the non-biologist, since the motivations are easy to understand).

There are a number of interesting mathematical results in L system theory, however irrespective of motivation. The decideability of the DOL equivalence problem (discussed in Chapter III) is such a result. The material on complexity considerations in Chapter VI is interesting and leads to a number of open problems. The discussion of multi-dimensional L systems (also in Chapter VI) is good mental exercise.

*L.H. Reeker,*
*University of Queensland*

Kahn, P.M. ed., *Computational Probability* Academic Press, 1980. 340 pp. $21.

The book is not a cohesive discussion of "Computational Probability", but the Proceedings of an Actuarial Research Conference held on Computational Probability at Brown University in 1975. Except that they relate to the topic of computational probability, the chapters contained therein are unstructured. Like the proceedings of other such conferences, this is both its strength and weakness. For the informed reader, it gives a broad spectrum view of the subject matter without necessarily saying something new. For the student, it contains the information he requires but not necessarily in a form which can be easily assimilated.

Some of the papers address specific aspects of acturial research (e.g. APL for Actuaries; Reversionary Annuities as Applied to the Evaluation of Law Amendment Factors; Non-life Business and Inflation; Simulation of a Multirisk Collective Model), while others in comparison address quite theoretical questions (e.g. Computational Problems Related to the Galton-Watson Process; Central Limit Analogues for Markov Population Processes; Backward Population Projection by a Generalized Inverse). A few are rather pragmatic (e.g. Some Ideas in Computational Probability; Experimental Computation) with the odd one having only a tangential association with Computational Probability (e.g. Symbolic Information Processing, Numerical Fourier Inversion).

It is certainly not the type of book to which a computer professional or scientist would turn for information about algorithm aspects of computational probability. It does however represent for such an audience an illustration of the range of sophistication over when theoretical topics, such as probability, can range computationally.

*R.S. Anderssen*
*Division of Mathematics and Statistics,*
*CSIRO, Canberra*

apply to large scale computer usage have been applied to user protective packages for micro-users. The packages include application programs and independently available subroutines. All documentation is scope oriented.

The released programs include a linear equation solving package, based on the LINPACK project (SIAM 1979), the ANSI/77 elementary function set in single and double precision form, and a random number generation, simulation, and testing package. The 32 elementary function codes (sin, log, . . .) include a separate power (a to the power b) function. The accuracy and scope of the elementary functions embedded in the delivered Fortrans has been extended.

The linear equations packages, which solve up to 50 by 50 systems, operate on the IBM Series/1 and Radio Shack Model II micro-computer. The elementary functions and random number packages have been certified on the Model II. Release in given environments is followed by movement to other processors.

The software is produced by a consortium of computer scientists, numerical mathematicians, and statisticians which has set the development of robust, tutorial

"Verbatim will provide full technical and engineering backup for the locally produced products," he said.

"We expect to increase our market share in Australia significantly in the current financial year with excess production being shipped overseas.

"Data and word processing applications have been expanding in recent years at more than 30 per cent a year, and the market for flexible disks in Australia has been growing even faster," he said.

## COMPUTER NETWORK FOR AUSTRALIAN PAPER MANUFACTURER

Nine computers are to be supplied to Associated Pulp and Paper Mills (APPM) in Melbourne by Britain's International Computers Limited (ICL) under a $18 million contract.

The computers – eight 'ME29' and one Model '2956' – will be installed during the next two years in APPM's head office in Melbourne, and in the company's four paper mills in New South Wales, Victoria and Tasmania. They will be linked together to form a distributed data processing network using the advanced facilities of ICL's information processing architecture system. This will also provide the enhanced capabilities necessary for the development and operation of a comprehensive manufacturing system for use in the paper mills and extensions to existing administration systems.

ICL will also provide supporting services and software under the contract.

## RPG II ENHANCEMENTS OFFERED BY DATA GENERAL

Data General Australia Pty Ltd has announced several significant enhancements to its AOS and AOS/VS RPG II programming languages.

The enhancements are designed to improve overall program development and maintenance capabilities, as well as to optimise RPG programs for high-speed execution.

AOS RPG II operates under Data General's Advanced Operating System and executes on any ECLIPSE data system with 512KB memory. AOS/VS RPG II operates under the recently-announced AOS/VS operating system and executes on 32-bit ECLIPSE computers.

Both AOS and AOS/VS RPG II include an interface to Data General's INFOS and INFOS II file management systems, allowing users to create and maintain large data bases in an on-line, multi-terminal environment.

AOS and AOS/VS RPG II languages include two compilers – the Data General RPG Interactive Compiler (DG/RIC) and Data General RPG Optimising Compiler (DG/ROC).

DG/RIC runs interactively using an interpreter under AOS or AOS/VS, and provides interactive debugging, formatted dump facilities, and dynamic paging for large programs.

DG/ROC generates optimised machine-level code for the production of fast-executing versions of RPG programs.

software as its task, so as to make recent research advances in computer science availble to micro-computer users.

C. Abaci can be contacted at 101 Dixie Trail, PO Box 5715, Raleigh, N.C. USA.

## DIGITAL ANNOUNCES 124-MEGABYTE, WINCHESTER-TECHNOLOGY DISK UNIT

Simultaneously with its parent company, Digital Equipment in Australia and New Zealand announced in late October, its first Winchester-type disk drive, designed for VAX computer systems.

Called the RM80, the new 124-megabyte unit uses state-of-the-art microprocessor control to achieve high performance and reliability with a low cost per megabyte for program and data storage.

The RM80, designed and manufactured at Digital's Colorado Springs facility, in the US is the company's first disk product to employ Winchester technology, in which read/write heads, platters, and spindle are constructed as a sealed assembly, and which allows the heads to rest on special "landing zones" when power is removed. The design affords improved reliability because of the absence of contamination associated with removal and replacement of disk packs and cartridges. The sealed assembly also permits more precise operating tolerances to allow higher data recording densities.

The RM80 is intended for use with MASSBUS-equipped VAS-11/780 and VAX-11/750 systems. The RM80 is obtainable either as a part of VAX-11/780 and

VAX-11/750 packaged systems or as separate add-on units for currently installed systems. The disk subsystem (drive and controller) is priced at approximately $34,000. Add-on drives are priced at just over $22,000. Deliveries are scheduled to begin in the first half of next year.

The new product attains an average seek time of 25msec and an average access time (seek plus latency) of 33msec. Data transfer rate is 1.2 megabytes per second. The head/disk assembly (HDA) contains four platters with seven data recording surfaces, one surface for servo information, and two read/write heads per data surface. The HDA also incorporates a low-inertia rotary actuator.

The RM80 is available with single or optional dual port access. A mixture of up to eight RM80 and other disk products such as the RM03 and RM05 can operate on a sigle MASSBUS controller.

An integral microprocessor controls all major drive functions including servo adjustment and diagnostic procedures. Microdiagnostics are used to verify drive functions upon startup and to isolate faults to the field-replaceable unit level, thereby minimising repair times.

The new device can be characterised as a midrange disk product incorporating significant enhancements in data integrity, performance, and economy.

The RM80 processes almost twice the storage capacity of Digital's RM03 disk system for about the same price and a lower monthly maintenance charge. The RM80's average seek time, too, is exceptional for a product of its price and capacity.

The new unit offers high data integrity — improved reliability and lower error rates — through microprocessor control.

The self-diagnostic capability, Digital's first in a disk product, eliminates the need for auxiliary test boxes or tools. Furthermore, there are no requirements for scheduled preventive maintenance.

The RM80 is designed to operate under normal office conditions of temperature and humidity.

## PHILIPS ANNOUNCE MAG CARD WORDPROCESSOR CONVERSION

Philips has announced a conversion facility to allow users of IBM magnetic card equipment to upgrade to its P5002 word processing system.

The magnetic card converter can take the data from up to 128 magnetic cards and with minor editing, transfer that data onto one flexible disk.

Mr Barrie Hepworth of Philips Data Systems division said that research undertaken for Philips into word processor buying trends plus estimates of the installed base of IBM magnetic card equipment suggested the conversion facility would offer an easy upgrade to the benefits of a word processing system which offered a visual display facility and flexible disks.

"It is estimated there are more than 3000 IBM magnetic card machines installed in Australia. Our research also indicates that users of this equipment, who are already convinced of the benefits of word processing, will be well disposed to moving into the more flexible disk format."

Mr Hepworth said that not only was the magnetic card limited in terms of adding to, updating and modifying multi-page documents but it presented physical difficulties in terms of filing.

"In turn searching documents, without a visual display is difficult," he said.

The flexible disk word processor means many more documents can be stored on a single disk, updating and text manipulation is easy and searching and retrieval is made simple with a full size video screen. Philips is also hiring out the conversion facility.

## PLOT 50 EASY GRAPHING AIDS ENGINEERING ANALYSIS

The next Tektronix Plot Easy Graphing software allows engineers to create presentation quality graphics to report experimental or analytical data.

Tektronix designed Plot 50 Easy Graphing for users with little previous graphing experience. The program responds to simple command verbs to enter data and to generate graphics on Tektronix 4050 Series Graphic Computing Systems.

Plot 50 Easy Graphing is the second in a series of software products with Standard File Formats, allowing several programs to share the same data. Data generated by, or data entered into, one software product can be accessed by another product in the series.

Easy Graphing generates high-quality, fully labeled graphics to aid comparison, interpretation, and illustration of numerical data. The program includes line graphics to illustrate trends, pie charts to show proportion relationships between parts of a whole, bar graphs to dramatise comparisons of quantities and scatter plots to analyse data points. Command files permit repetitive graphs to be drawn by adding only updated data.

Easy Graphing is a BASIC language version of Tektronix Plot 10 Easy Graphing, which is recognised as the industry standard in graphics software. It has five major components, including a four-phase tutorial program to review graphing concepts for first time users. The Easy Graph program is a question-and-answer session to help users generate graphs without knowledge of the Easy Graphing command language. Help Files are available for any Easy Graphing command, and utility programs are included to copy, list, edit, and duplicate Plot 50 Easy Graphing files.

## LARGEST DISK COMPANY TO MANUFACTURE IN AUSTRALIA

The largest manufacturer of flexible (floppy) disks in the world, Verbatim Corporation of the United States, will open a manufacturing facility in Australia this year.

Verbatim Corporation which has its headquarters in California, offers a complete line of removable, mini-magnetic media for data storage (diskettes, mini-disks, data tape cartridges and digital cassettes) marketed under the Verbatim trade name, as well as a full line for original equipment manufacturers.

The new facility will be at 52-54 River Street, South Yarra.

Mr Brian Johnstone, newly appointed General Manager of Verbatim Australia Pty Ltd, said that production of eight inch flexible disks would start before the end of October, while five and a quarter inch mini-disks would be manufactured in the first quarter of next year.

"The Australian made disks will feature our new 'Datalife' technology," he said.

He said that Verbatim branded flexible disks have been available in Australia for six years through Magnetic Media Services Pty Ltd, who will also distribute the locally produced products.

# THE AUSTRALIAN COMPUTER JOURNAL

Volume 12

February 1980 to November 1980

# 1. CUMULATIVE CONTENTS

*Technical Contributions*

## 2. TITLE INDEX

*(Titles of books reviewed in the journal given in italics.)*



## 3. CONTRIBUTOR INDEX

# 4. CR CATEGORIES INDEX

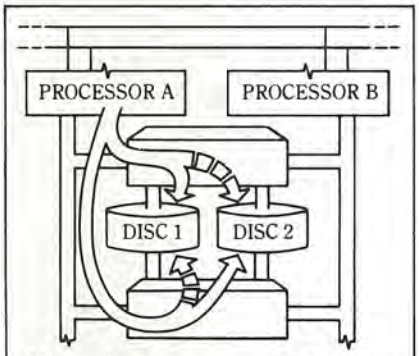
# The World's First NonStop™ DBMS

## Tandem NonStop™ ENCOMPASS

ENCOMPASS is the only DBMS with the benefit of running on a NonStop system. It's also the only high performance, relational data base management system designed from scratch to provide unmatched data integrity in high volume on-line transaction processing environments.

## We made relational fast.

A true data base operating system is the foundation with much of the retrieval techniques designed right in. And the structure of data on disc is optimized to minimize head movement. Memory cache retains most frequently used items in a buffer. In fact, any information in a file that filled a 300M byte disc drive could still be retrieved with an average of one seek. Combined with Tandem's optional mirrored volumes, it all adds up to tremendous speed and throughput with all the benefits of relational structuring.




Mirrored volumes automatically maintain identical copies; write operations occur simultaneously, read operations performed using drive with head closest to data.

## Networking made easy.

Each individual system can be expanded to sixteen processors, with additions of memory, terminals, discs, and there will be no loss whatsoever on the original investment—hardware or software.

The relational nature of ENCOMPASS, along with our networking software, EXPAND, allows a single data base to be distributed over multiple systems. Easily and safely. Up to 255 systems, each with as many as sixteen processors and thousands of terminals, each with unobstructed access to the data base distributed throughout the network.

## Automatic Terminal Management.

Terminal management has been the classic nightmare of on-line data base systems. No more. ENCOMPASS automatically handles complete support for the Tandem 6520 Multi Page Display, Tandem 6510, and IBM 3270 connected by a variety of communication lines including Asynchronous, Byte Synchronous, Multipoint, Point to Point, X25 and SDLC.

Screen formatting, data validation, screen sequencing and data mapping, plus sequencing and control of multiple terminals; these are all handled for the application programmer automatically and at a fraction of the cost in development time and dollars.

## Backout and recovery over a distributed data base.

Consistency of the data base is essential. Multiple files must be capable of being updated simultaneously, even if located across distributed nodes. If for any reason a transaction cannot be 100% completed, this is the one system in the world which can un-do it completely. Automatically.

The system will recover each piece of the transaction from everywhere in the distributed data base. Without cost-killing overhead. A major breakthrough in a network, DBMS. No one else even comes close.

## NonStop™ availability in Hardware and Software.

Because of its unique architecture, the system will keep on running without interruption, without loss or duplication of a transaction-in-process even if a failure occurs in any processor, I/O channel, disc or disc controller.

Tandem NonStop™ architecture provides not only this redundancy in hardware, but the software to take advantage of it, utilizing all available resources.

The NonStop™ system ensures that every update is completed to the data base. And with ENCOMPASS DBMS, NonStop™ operation is automatically built into all of your programs.

## On-line's as easy to program as batch.

One key theme behind the performance and reliability of our NonStop™ DBMS, ENCOMPASS, is the ease of use for programmers, systems designers and users.

It's easy to extend the data base, to expand the system to a network, to manipulate the data. And it's easy to add new resources, new hardware, new software, new files. It's easy to provide controls and security. Easy to work with. Easy to repair. And above all, easy to learn.

You don't need to learn a whole new language with the ENCOMPASS system: OPEN, READ, WRITE are the verbs you've been using all along. Industry standard COBOL, FORTRAN, MUMPS and our own transaction-oriented language TAL provide simple interaction between programs and data base.

## The All-ENCOMPASSing DBMS.

And there's still more. In addition to all this, there's a whole host of other features that the ENCOMPASS data base management system will provide. To name just a few:

- on-line data base backup
- transparent access to distributed data base
- efficient query-report generation
- dynamic and automatic adjustment to varying transaction loads

All that remains for the user is:

- simple batch type application modules.

What could be simpler?

**TANDEM NonStop™ Systems**

**TANDEM COMPUTERS**
DISTRIBUTED IN AUSTRALIA BY:

**MANAGEMENT INFORMATION SYSTEMS PTY. LTD.**

3 Bowen Crescent, Melbourne, Vic. 3004 Telephone (03) 267 4133

22 Atchison Street, St. Leonards, N.S.W. 2065 Telephone (02) 438 4566

S.G.I.O. Building, Cnr. Turbot & Albert Sts, Brisbane, Queensland 4000 Telephone (07) 229 3830

NCAMIS/38

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

Office Bearers. *President:* G.E. Wastie; *Vice-Presidents:* R.C.A. Paterson, A.W. Goldsworthy; *Immediate past president:* A.R. Benson; *National treasurer:* C.S.V. Pratt; *Chief executive officer:* R.W. Rutledge, PO Box N26, Grosvenor Street, Sydney, 2000, telephone (02) 267 5725.

Editorial Committee: *Editor:* C.K. Yuen, CSIRO Division of Computing Research, P.O. Box 1800, Canberra, A.C.T. 2601. *Associate Editors:* J.M. Bennett, T. Pearcey, P.C. Poole, A.Y. Montgomery, J. Lions.

**SUBSCRIPTIONS:** The annual subscription is $15.00. All subscriptions to the Journal are payable in advance and should be sent *(in Australian currency)* to the Australian Computer Society Inc., PO Box N26, Grosvenor Street, Sydney, 2000. A subscription form may be found at the end of the August issue.

**PRICE TO NON-MEMBERS:** There are now 4 issues per annum. The price of individual copies of back issues still available is $2.00. Some 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 640, Crows Nest, N.S.W., 2065. No trade discounts are given, and agents should recover their own handling charges. Special rates apply to members of other Computer Societies and applications should be made to the Society concerned.

**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.

**REPRINTS:** 50 copies of reprints will be provided to authors. Additional reprints can be obtained, according to the scale of charges supplied by the publishers with proofs. Reprints of individual papers may be purchased for 50 cents each from the Printers (Publicity Press).

**PAPERS:** Papers should be submitted to the Editor, authors should consult the notes published in Volume 12, pp. 71-75 (or request a copy from the National Secretariat).

**MEMBERSHIP:** Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries. Canberra: P.O. Box 446, Canberra City, A.C.T., 2601. NSW: Science House, 35-43 Clarence St, Sydney, N.S.W., 2000. Qld: Box 1484, G.P.O., Brisbane, Qld, 4001. S.A.: Box 2423, G.P.O., Adelaide, S.A., 5001. W.A: Box F320, G.P.O. Perth, W.A., 6001. Vic: P.O. Box 98, East Melbourne, Vic, 3002. Tas: P.O. Box 216, Sandy Bay, Tas, 7005.



**Published by:** Associated Business Publications, 28 Chippen Street, Chippendale, N.S.W., 2008. Tel: 699-5601, 699-1154.
*All advertising enquiries should be referred to the above address.*

Printed by: Publicity Press Ltd., 29-31 Meagher Street, Chippendale, N.S.W., 2008.

ISSN 004-8917

# THE Australian Computer Journal

*VOLUME 12, NUMBER 4, NOVEMBER 1980*

## CONTENTS



This Journal is Abstracted or Reviewed by the following services: :

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