

Dynamic resource allocation and supervision with the programming language MODULA

I. C. Wand

Department of Computer Science, University of York, Heslington, York YO1 5DD

The programming language MODULA was designed by Wirth to be suitable for programming in areas previously the preserve of Assembly code. Examples of these areas are process control systems, computerised laboratory equipment and input/output device drivers. Experiments with MODULA have shown that the language is quite successful for coding single programs that run on dedicated computer systems, but that it is less successful in applications that require dynamic resource management, both of storage and of processor time.

In this paper we define extensions to MODULA that enable the language to be used for the coding of storage allocators and process supervisors. At the same time an exception handling mechanism is introduced for the convenient handling of the error situations that arise during such programming. The exception handling mechanism turns out to have other uses, such as variable initialisation and process exit code. The consequences of an exception handling mechanism suitable for interactions between processes are examined.

Finally, some estimate is made of the further complexity introduced into an existing simple language, both into its definition and into the compiler. Possible areas of use are mentioned.

(Received December 1978)

MODULA is a simple programming language proposed by Wirth (1977a) for use in programming areas still dominated by Assembly code, such as process control, computerised laboratory equipment and input/output device drivers. Coding exercises in MODULA by Wirth (1977b) and others (Holden and Wand, 1978) have shown that the language is quite successful in these areas of application and is particularly good for the expression of single programs that will run on dedicated computing equipment.

However the language does have a number of drawbacks if it is to be used for the coding of operating systems kernels or for the writing of any software component where *supervision* plays an important part. In particular these difficulties centre upon

- (a) a process supervising other processes
- (b) a storage allocation process and
- (c) the general handling of error conditions at both the process and module level.

It is interesting to note that these are all requirements of the so-called 'Ironman' language (1977).

This paper examines extensions to MODULA which attempt to overcome these drawbacks. A new storage mechanism is proposed based upon a declared type called a *region* which is very similar to the implicit *heap* in PASCAL. For references into such storage, PASCAL-like pointers are proposed. In addition it is proposed that processes and their storage must be allocated within a region, although it is not suggested that **device processes** should be included in this new mechanism. Thereafter references to ordinary processes must be made via pointers, and, furthermore, when a process is *run* it can be executed for a given number of signal occurrences (usually originating from a **device process**) under the supervision of a parent process. By this mechanism supervisory processes can be constructed which can then make sure that a process can only run for a limited period and can subsequently terminate that process if necessary. An exception mechanism is proposed for transmitting error conditions; however the mechanism can be used quite generally.

We suggest that these extensions will cause moderate complications in the language definition and in the compiler. The major run time complication is in the storage allocation mechanism which implies a PASCAL-like *heap* within every region, although with the simplification that any region is allocated within the storage space of the parent process. In

conclusion we suggest that the facilities proposed will enable MODULA-like language to be used for the coding of basic operating system components.

Pointers

In its present form MODULA takes a very static view of storage. If some upper limit on recursion depth is assumed, the storage requirements of all processes can be calculated at compile time and hence the mapping of most storage can be predicted. Clearly such a rigid scheme is not acceptable in

- (a) operating systems, where the number of processes and their storage requirements may vary from time to time, and
- (b) compilers, where internal data structures, such as the dictionary, need a varying number of components—many of differing size.

The usual technique of referencing data structures whose number and size vary on allocation is by *pointers*. For this reason we propose that PASCAL-like pointers should be introduced into MODULA, that is (and using the notation of the PASCAL User Manual, Jensen and Wirth, 1975)

type *<identifier>* = \uparrow *<type identifier>*

denotes a pointer type identifier that can only be used in the subsequent declarations of pointers. These pointers can only reference instances of that type allocated within a *region* (see below). Legitimate type identifiers that can be used in such a **type** declaration are

integer, Boolean, bits, char, signal, region, process
and exception

together with user declared types. Note that type declarations and procedure parameter lists are the only contexts in which the type *process* can be used. There are no restrictions on the contexts where objects of type *region* and *exception* can be declared.

An item of a given type can be allocated within a specific region using the standard procedure *new*.

new (*r*, *p*) allocates a new variable within the region *r* and assigns the pointer reference of that variable to the pointer variable *p*.

new (*r*, *p*, *x*) allocates the storage space for the process *x* within the region *r* and assigns the pointer reference of *x* to pointer variable *p*. Thereafter all references

to this instance of process x will be made via p . Note that any parameters used by x must be made in the call to *new*, for example

new ($r, p, x(a, b)$)

If a process is referenced using *new*, the *initialise* exception is raised after the storage for the process is allocated. Any code supplied for the *exception* handler (see below) within the allocated process is executed at this point. If there is insufficient storage available for the variable or process to be allocated, the exception *insufficientstorage* is raised.

Any item, process or otherwise, that is referenced by a pointer can be erased from its region using *dispose*.

dispose (p)

will perform the 'inverse' of *new* (r, p). If the item referenced by p is a process, the *terminate* exception is raised in that process before its storage is reclaimed.

Note that it is not proposed in this paper to introduce PASCAL variant records, although clearly to satisfy (b) above some kind of facility of this type might be desirable.

Regions

The basic type *region* is proposed so that the working space for individual processes, or groups of processes, together with any other variables, can be allocated within a given storage area. A region is declared in the usual way, for example

var r : *region*;

Space for a particular *region* is allocated within the storage area of the parent process using the standard procedure *generate*

generate (r, i) allocate a region r of size i in the storage area of the parent process. i is interpreted in an implementation dependent fashion and could, for example, be the number of bytes or words required by r .

The effect of calling *generate* more than once, naming the same region, is to make further allocation impossible within any region other than the most recently generated, although reference to previous regions via existing pointers can continue. If insufficient storage is available the exception *insufficient-storage* is raised.

Note that *regions* can be declared and passed as procedure or process parameters in just the same way as any other type; however no other operations (including assignment) are defined for *regions*.

A standard function

howmuchleft (r)

is available. It returns in implementation dependent units and in a manner consistent with *generate*, the amount of storage left in region r .

Processes

In MODULA it is assumed that the initialisation and running of a process are one and the same action. It is proposed here that these two actions should be separated and that, as alluded to above, the initialisation should be carried out at the same time as the allocation of the process in its region. Therefore in the following process

```
process  $p$ ;
var  $i$ : integer;
upon initialise do  $i := 1$  end;
begin
  {main body of process}
end  $p$ ;
```

the initialisation of i would be carried out when the instance of the process is allocated using *new*. The separation of initialisation and the subsequent running of a process is considered

necessary so that initialisation of any local variables can take place at the same time as storage allocation and therefore is independent of any subsequent scheduling of the process.

In the same manner, the *terminate* condition is raised when a process is the target of *dispose*, so that any tidying up operations can be carried out by a process before it is removed from its region. If a process does not supply initialisation or termination code, a process heading of the form

```
upon initialise do end;
terminate do end;
```

is assumed.

Note that there are no restrictions on the points in a program where *new* may be called with a process as its target—this removes the restriction in the present language in which process calls can only be made in the main program. It is still proposed that processes should only be declared at level 0, that is, they cannot be nested or be local to procedures. However it is suggested that process parameters should only be of type **const**. This will remove any problems that may arise with parameters having shorter lifetimes than the process to which they are passed.

The scheduling of processes

A new standard procedure *run* is proposed

procedure *run* (pn : p ; t : *signal*; n : *integer*): *code*;

where

type $p = \uparrow$ *process*;

and

type *code* = (*finished*, *completed*, *waiting*);

Procedure *run* makes that process instance described by pn runnable for a period of time such that the signal t is sent n times. The process will be made unrunnable if it waits upon a signal within the period defined by t being sent n times. The procedure *run* returns

finished if it runs to the end of the period, or

completed if it completes during the period, or

waiting if it waits on a signal during the period.

The parent process, that is the process calling *run*, is halted until the call of *run* is completed, upon which the process continues at the statement following the call to *run*.

An example scheduler is now presented. The scheduler relies upon a **device module** clock which sends a signal every line frequency cycle. The scheduler **process** simply runs the runnable processes held in a circular queue for a multiple of the line frequency cycle. No effort is made to handle error situations arising in the supervised process.

A process scheduler

device module *clock*[6];

{The Device Module clock provides the signal tick which is 'sent' every line frequency cycle.}

define *tick*;

var *tick*: *signal*;

process *kw11*[400B];

var *csr*[177546B]: *bits*;

begin

loop

csr[6] := *true*;

doio;

csr[6] := *false*;

while *awaited*(*tick*) **do** *send*(*tick*) **end**

end

end;

```

begin
  kwl1l
end clock;

interface module queue;
  {'queue' maintains a crude list of runnable processes.
  Processes are inserted into the queue using 'setup'.
  The next process to be run is found using 'next' and
  a process is removed from the queue using 'erase'.}
  define setup, next, erase;
  type p = ↑ process;
  var i, index: integer;
  bn: array 0:31 of Boolean;
  pn: array 0:31 of p;
  a: region;

  procedure setup(pp: process; var pr: p): Boolean;
  var i: integer;
  upon insufficientstorage do setup := false end;
  begin
    i := 0;
    loop
      when i > 31 do setup := false exit;
      when not bn[i]
        do
          new(r, pn[i], pp);
          pr := pn[i];
          bn[i] := true;
          setup := true
        exit;
      inc(i)
    end
  end setup;

  procedure next(var pr: p): Boolean;
  var i: integer;
  begin
    i := index;
    loop
      i := (i + 1) mod 32;
      when bn[i] do pr := pn[i]; next := true; index := i exit;
      when i = index do next := false exit
    end
  end next;

  procedure erase(pr: p);
  var i: integer;
  begin
    i := 0;
    loop
      when i > 31 do exit;
      when p = pn[i] do bn[i] := false; dispose(pr) exit;
      inc(i)
    end
  end erase;

  begin
    i := 0;
    repeat
      bn[i] := false; inc(i)
    until i > 31;
    index := 0;
    generate(a, 1024*100) {100K storage say}
  end queue;

  process scheduler;
  {The scheduler process runs the next available process
  for 2 clock ticks. If no runnable process is available
  the process waits until the next line cycle starts and then
  looks for a runnable process again.}

```

```

  use tick, next, erase;
  const numberofticks = 2;
  var pn: ↑ process;
  begin
    loop
      if next(pn)
        then
          if run(pn, tick, numberofticks) = completed
            then erase(pn)
          end
          else wait(tick)
        end
      end
    end
  end scheduler;

```

Exception handling

It is clearly desirable to incorporate an exception handling mechanism into a language like MODULA. Such a mechanism could be used for:

- the handling of rarely occurring error situations (see stack module example below),
- the handling of process termination statements which may be necessary when a process is prematurely terminated from outside (using *dispose* in the above proposal), and
- a more structured approach towards handling conditions that arise infrequently.

Exceptions occur synchronously and should be distinguished from interrupts which occur asynchronously and in MODULA are handled by **device processes**. As we have already shown the mechanism can also be used when a process is initialised or terminated for any special action that may be required at these times.

Exception handling mechanisms have been examined in detail by Goodenough (1975) and Levin (1977), and a simple scheme for error handling has been proposed by Bron *et al.* (1976). The scheme proposed here is relatively straightforward and follows Levin in suggesting that the exception is defined (normally) with the abstract type it is monitoring (see stack example below) and the handler, which is set up declaratively, is then associated with the use of the abstract type.

The proposed scheme is based upon the declaration of instances of type *exception*, together with their associated handlers. Normally these will be associated with an abstract type and will define errors or situations that occur rarely during their execution. An exception can only be raised *and* handled within the same process; this rule is broken by the *initialise* and *terminate* exceptions which are raised as a result of calling the standard procedures *new* and *dispose*. A standard procedure *raise* is defined

raise(*e*) propagate the exception *e* from the context of the call along the dynamic calling sequence of the current process.

An exception handler is declared at the head of a procedure or process in the following manner

upon <exception name> **do** <statement sequence> **end**

Once an exception handler has been entered and its body of statement completed, then

- if the handler was declared at the head of a procedure, exit is made to the statement following the call of the procedure, or
- if the handler was declared at the head of a process, then the *terminate* exception is raised in that process.

An example of the use of an exception handler

To illustrate an application of the proposed exception handling

mechanism, consider the following example of the abstract type *stack*, together with its manipulating procedures. Note that the exceptions themselves are part of the abstract type but that the handler must be supplied by the user of the type.

The abstract type 'integerstack'

```

module integerstack;
{An abstract type stack which represents a stack of integers.
 If stack overflow or underflow occurs, then the
 associated exception is raised.
 Note that parameterised types would give both a stack of any
 size as well as stack elements of arbitrary data types}
define push, pop, set, stack;
const max = 1024;
type stack = record
    sk: array 0: max-1 of integer;
    index: integer;
    underflow, overflow: exception
end;

procedure set(var x: stack);
begin
    x.index := 0
end set;

procedure push(var x: stack; i: integer);
begin
    if x.index = max then raise(x.overflow)
    else
        x.sk[x.index] := i;
        inc(x.index)
    end
end push;

procedure pop(var x: stack): integer;
begin
    if x.index = 0 then raise(x.underflow)
    else
        dec(x.index);
        pop := x.sk[x.index]
    end
end pop;

begin
end integerstack;

```

A use of 'integerstack'

The exception handling mechanism suggested here is supplied in the declaration part of a Module, Process or Procedure and once entered control cannot pass back into the statement part of the interrupted construct. When the last statement of the exception handling unit has been executed, the execution of the Module, Process or Procedure is completed.

The use of the abstract type stack could be as follows:

```

module useintegerstack;
use push, pop, set, stack;
...
var s: stack;
...
upon
    s.overflow do ... deal with error ... end;
    s.underflow do ... deal with error ... end
begin
    ... statements using set, push, pop etc ...
end useintegerstack;

```

Visibility

The above use of 'integerstack' breaks the visibility rules of MODULA. The present rules are

- (a) that only the identity of a type is known outside the module in which it is defined. This means that all structural details, such as record field names and the components of scalar types, are unknown outside the defining module; and
- (b) all local variables exported from a module are read-only or, in the case of a signal, wait-only.

We propose here that it should be possible selectively to export (and subsequently import) specific structural information. A possible notation for the above example might be

```

module integerstack;
define push, pop, set, stack(underflow, overflow);
...
end integerstack;
module useintegerstack;
use push, pop, set, stack(underflow, overflow);
...
end useintegerstack;

```

where the record fields or the components of scalar types must be specifically listed in both the **define** and **use** lists.

Of course once the structural details of types are made available to the user of that type, then the specific details of implementation may become available to the user. However two things will guard against misuse

- (a) exported objects are still read-only, and
- (b) only components specifically exported will be known.

Note that this proposal certainly lacks the flexibility of GYVE pointers (Schwartz and Shaw, 1976) or the completeness of the Euclid (Lampson *et al.*, 1977) visibility rules. However it is suggested that the simple rules of MODULA, together with the modification suggested here, will deal with the majority of practical cases.

Levin's storage allocation problem

When discussing the viability of various exception handling schemes, Levin (1977) posed the problem of a general storage allocator which was responsible for the administration of a storage pool used by an indeterminate number of processes. He suggested that the 'natural' way to program a solution to this problem would be for the allocator to try and satisfy the request from its own resources; if it could not do so, an exception would be raised asking other processes in the system to release storage. When the processes had released some storage, the allocator would then try to satisfy the original request. The attraction of using an exception for communication is that it is unnecessary for the allocator to know either the number of processes or their names.

The language proposals made in this paper will not solve Levin's problem, in particular

- (a) the process that raised the exception cannot be resumed after the exception has been serviced, and
- (b) the procedure *generate* can only respond to requests using pointers of known type. A general allocator, perhaps a procedure, would have to respond to requests to allocate an object of arbitrary type (and size). Clearly a relaxation of the type checking mechanism would be necessary to satisfy this requirement.

The following language extensions are suggested to solve Levin's problem

- (a) the pointer type *any* is introduced for use in parameter lists. It will satisfy any actual parameter of type pointer, that is


```

type any = ↑ <any type>;

```

 and,
- (b) a statement **resume** can be included in the exception handling unit following **do** and will cause the program context, where the exception was raised, to be resumed.

The following program module is a solution to Levin's problem

```
interface module allocator;
define grab, return, release;

var x: region;
    allocated: Boolean;
    release: exception;

upon insufficientstorage
do raise(release);
    if howmuchleft(x)=0 then allocated := false end;
    resume
end;

procedure grab(var r: any): Boolean;
begin
    allocated := true;
    new(x, r);
    grab := allocated
end grab;

procedure return(r: any);
begin
    dispose(r)
end return;

begin
    generate(x, 10000)
end allocator;
```

All users of the storage allocator should provide an exception handler to free storage in response to the exception 'release'. The exception handler would have the form

```
upon release
do
    ...
    return(something);
    resume
end;
```

The mechanism used above is considerably less general than that proposed by Levin (1977); in particular, when an exception is raised, parameters cannot be passed to the handler, nor can values be returned by the handler—except via global variables. In addition the set of processes that can handle the exception must be considerably wider than that outlined so far. Exceptions must be propagated across process boundaries and the set of viable handlers for the exception must be expanded to include processes that are not on the active calling sequence of the current process. Also, if several processes have handlers for the same exception, one must resolve the question of how many of the handlers are invoked—should the first valid handler suffice? For the solution of Levin's problem we require

- (a) all handlers that know its name can receive the exception, irrespective of whether or not they are on the calling sequence of the process that raised the exception, and
- (b) when the exception is raised, all handlers for that exception are invoked in an undefined order.

Clearly the mechanism could be made more complicated by indicating, when the exception is raised, the subset of valid handlers that can respond. A further complication, which is not discussed here, is the effect of a handler, not in the current calling sequence of the raising process, that does not contain a **resume** statement. Furthermore what happens if one of a group of handlers does not contain a **resume** statement? Are the handlers that contain **resume** statements executed before that

handler or is the order of execution random?

Conclusions

We estimate that the language proposals (excluding the exception handling required for Levin's problem) made in this paper will increase the length of the defining document for MODULA by approximately 20%. The major complication will be the definition of the semantics of three new types, the operations allowed upon them, exception handling (basically a new control sequence) and the small changes in the visibility rules. Of particular difficulty are the semantics of the standard procedure *new*.

We further estimate that the size of the compiler (Holden and Wand, 1977) will increase by about 20%. A large proportion of this code will be in the second or semantic pass of the compiler. The size of the PDP-11 run time package will probably increase by a factor of three or four, perhaps approaching 512 bytes. The major items that must be added are

- (a) the standard procedure *run*, and
- (b) the heap storage allocation mechanism, including *new* and *dispose*.

However even with the increase in size given above, the compiler and its run time system will still be quite small by the standard set by other languages in this area. At the present time we have no plans to implement the language features discussed in this paper.

Both the definition and implementation of a resume statement, as required for a solution of Levin's problem, presents considerable difficulty. A similar facility in PL/I was very difficult to define; certain ON conditions could resume their invoking context, others could not. The problem is somewhat easier here as the exceptions can only be generated by software (they cannot be generated, for example, by hardware overflow conditions), although in other respects it is more difficult as many handlers may be invoked as a result of a single raised exception condition (which may cross process boundaries). For the solution of Levin's problem, the exception must be propagated to handlers that may not be on the calling sequence of the process raising the exception. This will give rise to difficulties in locating and invoking the set of valid handlers. No solution to this problem is given in this paper, either in the definition of the extensions to MODULA or of their implementation. It may be that exception handling, when used in a block-structured language, should remain an essentially simple mechanism and be used for straightforward error handling (as in Ironman, 1977).

The utility of the proposal made in this paper can only be tested by writing software components in this dialect of MODULA. If the arguments presented in this paper are well conceived then the class of problem to which MODULA can be applied will have been considerably extended. We hope that this will prove to encompass both low level programming, such as device drivers (previously well served by MODULA), and higher level programming (previously well served by Concurrent PASCAL).

Acknowledgement

The work described in this paper was carried out when the author was on sabbatical leave at the IBM Thomas J. Watson Research Center, Yorktown Heights, USA. The support of IBM and the hospitality of colleagues at Yorktown Heights is gratefully acknowledged.

References

- BRON, C. FOKKINGA, M. M. and DE HAAS, A. C. M. (1976). A proposal for Dealing with Abnormal Termination of Programs, Memorandum Number 150, Department of Applied Mathematics, Twente University of Technology.
- GOODENOUGH, J. B. (1975). Exception Handling: Issues and a Proposed Notation, *CACM*, Vol. 18, pp. 683-696.

- HOLDEN, J. and WAND, I. C. (1977). Experience with the Programming Language MODULA, *Proceedings of IFAC/IFIP Real Time Programming Seminar*, Eindhoven, K. Smedema (editor), Pergamon Press.
- HOLDEN, J. and WAND, I. C. (1978). An Assessment of MODULA, *Software—Practice and Experience*. In press.
- JENSEN, K. and WIRTH, N. (1975). *Pascal—User Manual and Report*, Springer-Verlag.
- LAMPSON, B. W., HORNING, J. J., LONDON, R. L., MICHELL, J. G. and POPEK, G. J. (1977). Report on the Programming Language Euclid, *SIGPLAN Notices*, pp. 1-79.
- LEVIN, R. (1977). Program Structures for Exceptional Condition Handling, PhD Thesis, Department of Computer Science, Carnegie-Mellon University.
- SCHWARTZ, J. T. and SHAW, P. (1975). A Brief Survey of the Principal Concepts of GYVE, *GYVE Newsletter*, Number 1, Computer Science Department, Courant Institute of Mathematical Sciences, New York University.
- US DEPARTMENT OF DEFENSE, (1977). *Ironman: Requirements for High Order Computer Programming Languages*.
- WIRTH, N. (1977a). MODULA: A Language for Modular Multiprogramming, *Software—Practice and Experience*, Vol. 7, pp. 3-35.
- WIRTH, N. (1977b). The Use of MODULA, *ibid*, pp. 37-65.

Book reviews

Programming in Standard Fortran 77 by A. Balfour and D. Marwick
1979; 384 pages. (Heinemann, £9.50, £4.50 paper)

The authors claim in their preface that this book will serve either as a text for the novice or as a reference for the more experienced programmer. They also imply that FORTRAN is for use only by scientists and engineers; this feeling is highlighted elsewhere in the book. They state that they are going to encourage the reader to write good FORTRAN programs and to this end devote a large part of Chapter 3 to a quite unnecessary use of an ALGOL-like algorithmic design language even down to the bold faced type for keywords. This language emerges in a contrived way at intervals during the remainder of the book and will probably serve to confuse the beginner and annoy the reader who is using the book as a reference. Once the reader has been told on page 3 that FORTRAN is not an ideal programming language even though it is so widely used he will realise that the authors are ALGOL (Pascal?) men wearing FORTRAN hats.

It is a pity that the considerable material of FORTRAN is not presented from a FORTRAN user's viewpoint, rather than from the stance adopted. Having said this, the body of the text describes most of the features of the language very adequately in an order which is more or less the norm for FORTRAN text books. Most chapters have a number of useful exercises for which solutions appear in an appendix. Input/output is covered in three separate chapters which would probably have been better combined into two. I was surprised that Chapter 7 is not restricted to the list directed forms of the READ and PRINT statement instead of presenting simple and not so simple formats first. The coverage of further input/output later in the book is entirely adequate although no mention is made under the heading 'use of files' of the problems that will be raised by different implementations of the OPEN statement. Much of Chapter 8 (Program structures), particularly a description of statement lines and comments, could appear earlier. With the almost universal availability of terminals the three pages devoted to a program handwritten on coding sheets seems superfluous. Predictably the chapter entitled 'Control statements' follows IF-THEN-ELSE by DO and relegates such useful features as logical IF and computed GOTO to the final chapter without a forward reference. We are given a mere 13 pages entitled 'Arrays and subscripted variables' which makes the mistake of introducing arrays by means of a matrix. Many non-mathematical FORTRAN users have been perplexed by this approach over the years.

A good chapter entitled 'Procedures' is marred by the authors' conclusion that argument lists should always be used in preference to COMMON. Logical and complex facilities are well covered but the chapter on double precision facilities starts with the sweeping statement that 'the majority of programmers will seldom, if ever, require to use double precision arithmetic'. Perhaps the authors have never had cause to use the NAG library on a 16-bit mini. Character handling is given a chapter to itself although the use of more examples here would have had much to recommend it. Towards the end of the book a chapter entitled 'Case studies' discusses a number of non-trivial programs in some detail. This is a good chapter confused by

the large number of different typefaces used by the printer; one is vaguely aware of the problem throughout the book but never more so than here.

There is sufficient awareness today about program portability for Chapter 21 to consist of more than 3½ pages—it is not enough merely to point the reader at a number of references, good though these sources may be. A number of 'other features' are lumped together towards the end of the book. One could perhaps justify this treatment of the arithmetic IF and assigned GOTO, but not of the logical IF and computed GOTO. The IMPLICIT statement should appear with type statements near the start of the book. The authors state that their coverage of ENTRY and alternate RETURN statements is deliberately superficial which is a pity as full coverage of these statements would have fitted logically into the chapter on procedures. Most serious of all was the statement that 'EQUIVALENCE is a facility of dubious value' and the restriction of its description to two pages. The book is concluded by appendices on conflicts with FORTRAN 66 (too brief), a summary of statements and statement order and an unnecessary copy of the syntax charts taken from the ANSI standard. Answers to exercises precede a comprehensive index which will please the reference reader.

In conclusion, certainly the best FORTRAN 77 textbook to appear so far, despite the attempts by the authors to discuss simultaneously structured programming techniques. It is good value particularly in paperback but will probably appeal more to the programmer updating to FORTRAN 77 than to the novice.

D. M. VALLANCE (Salford)

Issues in Data Base Management by Herbert Weber and Anthony Wasserman, 1979; 263 pages. (North-Holland, \$34.25)

I recommend this book to anyone wanting to get up-to-date with current issues in data base management. It provides a selection of some of the particularly relevant papers and panel sessions at the Very Large Data Base conference, Berlin 1978.

To read all the papers presented at that conference would be a very daunting prospect—I know, I was present and tried! However, I believe this book to be readable, particularly if one starts with the comments by panel members prior to studying the principal survey paper of the session. The comments provide a few hooks on which to hang ideas and that is important when trying to catch up with reading late in the evening!

There are five subject areas: data base design; data base software engineering; distributed data base systems; impact of new technologies, and data base security and privacy. The survey papers present the problems and achievements in each area while the panelists' comments take up the more controversial points.

Readers should not expect to find many answers to the issues raised; what they will get is an appreciation of the factors involved and some idea of trade-offs. The fainthearted may decide to keep well away from the data base area! The more courageous reader will have plenty of ammunition to fire at the data base salesman, particularly in the distributed data base or the security field.

PETER H. PROWSE (London)