PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

A CRITIQUE OF MODULA

Andrew Richardson

Department of Computer Science
University of New South Wales

ABSTRACT

MODULA and MODULA-2 are the latest major languages designed by Professor Niklaus Wirth (Wirth,1977 and Wirth,1978). They are both claimed to be high level languages suitable for the programming of dedicated computer systems, with emphasis on process control systems and device drivers. A critique of MODULA, the earlier of the two languages, is presented in this paper. The emphasis is on the "useability" of MODULA, and whether it achieves its stated goals. A compiler for MODULA has been written in BCPL by two members of the University of York, J.Holden and I.C.Wand (Cottam,1978), and this is the compiler used by the author.

1. A BRIEF DESCRIPTION OF MODULA

Areas of programming such as process control systems, device drivers and computerised equipment have long been the almost exclusive domain of Assembly code. MODULA has been designed in an attempt to reduce this domination by providing a high level language that can perform efficiently in these areas.

The language itself has a number of apparent virtues : it is a concise language (suitable for implementation on small computers); it is a "high level" language; and it is based on the now well known language PASCAL, thus being already partly familiar to many new users. It has not implemented a number of the features of PASCAL, however, as the size of the language has been been kept at a minimum to enable implementation on small machines. Whether Wirth has selected the correct features to include and omit is a matter of some contention and these differences are now examined in more detail.

1.1 PASCAL FEATURES NOT IMPLEMENTED IN MODULA

As has been said, MODULA is based closely on PASCAL. Some of the main features of PASCAL that have been omitted from MODULA are :

* Pointers/Dynamic Storage Allocation,
* Input/output facilities,
* File manipulation,
* Variant records,
* Real arithmetic,
* The "GO TO" statement,
* The "FOR" statement,
* Set types, and

* Subrange types.

By far the most annoying omission in my experience was the lack of dynamic storage allocation, and hence the absence of pointers. Pointer-based structures have the big advantage of being able to vary in size, as storage allocation takes place at run-time rather than compile-time. This property was sorely missed, one case in point being the writing of a program to scroll a GT40 display. In such a program it is necessary to represent the screen as some form of character buffer. Storage had to be allocated for the theoretical maximum number of characters that could fit on the screen, although in practice this maximum is virtually never attained. On the other hand, if some pointer facility had been available storage could have been allocated as required. It appears that the overhead involved in introducing a pointer facility would be more than compensated for by the increased efficiency of programs, especially in small computers where core space is at a premium.

The omission of set types is another decision that I would question. Although the type "bits" (which represents an array of boolean variables) can often be used in place of sets, there are still occasions when sets are very useful. For example, when a case statement is used all possible alternatives must be specified (there is no default facility in a case statement in MODULA). The easiest way to ensure this is to have a set containing all the possible alternatives and to enter the case statement only if the expression to be matched is contained in the set. The type "bits" would usually not be suitable for such a function.

Variant records and subrange types are convenient but not essential, and their omission did not cause any substantial problems.

Other omissions that could be annoying are the "GO TO" and "FOR" statements. The former can be useful when a program has a number of error conditions that require quick termination, and the latter is a very simple way of executing a loop a set number of times. The "FOR" statement is especially useful in situations where a number of arrays are being manipulated. Again, both of these instructions, although useful, are not essential and substitutes are readily available.

The three remaining omissions listed, input/output facilities, file handling capabilities and real arithmetic are logical, as MODULA is a language that would generally be used at a level where such facilities are being implemented rather than used. File manipulation facilities, for example, are not really needed when, as Wirth puts it, "the typical application of MODULA is regarded as the design of systems that implement rather than use such a file facility" (Wirth,1977). File manipulation also assumes a good deal of run-time support, which may not always be available.

With the exception of pointer-based data structures and possibly set types, the omissions have been well decided, bearing in mind that this is a language where ease of implementation and compactness are prime objectives.

The method of terminating blocks has also been changed. Commands such as "IF", "WHILE", "CASE", etc must be explicitly terminated. This is an improvement over PASCAL as it obviates the need for a compound statement and generally makes programs easier to read. For example, the PASCAL code:

```
WHILE x<y DO
    BEGIN
        a := b+c;
        inc(x)
    END;
```

becomes in MODULA

```
WHILE x<y DO
    a := b+c;
    inc(x)
END;
```

This explicit termination rule also clarifies programs containing multiply nested "IF" statements, as the "ELSE" part of each "IF" expression is clearly associated with its intended partner.

1.2 NEW FEATURES OF MODULA

1.2.1 MODULES

By far the biggest innovation introduced in MODULA has been the module (as the name suggests). Modules are self-contained units containing constants, variables, data types, procedures and processes local to the module. None of these items can be accessed outside their own module unless they are "exported" by being put in the DEFINE list of their module. Another module can then "import" these items by putting them into its own USE list.

There are three types of module: standard Modules, Interface Modules and Device Modules. Standard Modules have the property mentioned above, but no other special properties. Interface Modules have mutual exclusion properties which will be discussed in the section on signals and processes. Device Modules contain all the machine-dependent facilities and will be discussed more fully later.

Whether modules are sufficiently flexible is an interesting question. The main drawback of the present module design is that duplication of code is a possible necessity. Classes such as those in SIMULA (CDC,1975) and Concurrent Pascal (Brinch Hansen,1975) escape this problem but they are more complex and would increase the size of the compiler. Wirth appears to have made a reasonable compromise between facilities and implementation size in this area. The present modules are very clear and a definite aid to the setting out of an ordered and tidy program.

1.2.2 SIGNALS, PROCESSES AND INTERFACE MODULES

The major multi-programming feature in MODULA is the process. As far as the programmer is concerned, processes can be taken as executing in parallel with one another. Processes can be activated by more than one source, and are in general reentrant. They cannot be nested, and must therefore only occur at the outermost level of the program. Processes are controlled by three operations : "wait", "send" and "awaited". Each of these operations has an argument of type "signal", and operates only on that argument. For example,

```
send(signal1);
```

"Send" will wake up a process that was waiting for the argument signal, "wait" will put a process into the waiting state and "awaited" is a boolean variable that is true when its argument signal is being waited for somewhere in the program.

When several processes share common variables, it can be awkward to have more than one process using (and especially changing) those variables at the same time. Interface Modules have been introduced to prevent this occurring, and provide a queueing facility to ensure mutual exclusion. Several processes in an Interface Module can be waiting at once, and the process currently running does not relinquish control until it executes a "send" or a "wait" statement, at which stage the next process in the queue takes over control. There has been some criticism of this set-up (Holden and Wand, 1978), the main point of contention being that control should not be relinquished on a "send" command. There is merit in this argument from a programmer's point of view, but scheduling within an interface module would become much more complex if this suggestion were adopted, as Holden and Wand point out. No reasonable and efficient method of scheduling has been suggested that would cope with such a modified arrangement, so it appears logical to stick to the present system (which is quite acceptable).

1.2.3 DEVICE MODULES

Device Modules contain nearly all the machine-dependent areas of the language. This means that implementing MODULA on different computers will not mean rewriting the whole compiler, but chiefly only the parts of the compiler dealing with Device Modules. (The exception is the type "bits". "Bits" can be either eight or sixteen bits long, depending on the machine being used.) The separation of machine dependent areas can also be an aid to the writing of well set out programs.

The processes in a Device Module are different to standard processes, and are referred to as Device Processes. These have an ability to receive hardware interrupts through the "doio" command. This command makes the process wait for a hardware interrupt from an interrupt vector address specified in the Device Process heading. Mutual exclusion is performed in a Device Module by specifying the machine processor priority level at which the processes contained in the Device Module will run. This priority is specified in the heading of the module.

The second main feature of a Device Module is the means provided for accessing the hardware interfaces of peripherals. In a PDP-11 computer, these interfaces are represented by device registers at fixed addresses. MODULA lets the user access these registers by associating variables in a Device Module with these hardware addresses. For example, the statement

```
VAR dpc[172000B] : integer;
```

will assign the variable "dpc" to the memory location 172000B.

In the interests of minimising both code and the interaction between Device Processes and other areas of the program, Wirth has imposed the following restrictions:

(i) Device Processes are not reentrant (unlike other processes). This restriction has been shown to have only a slight effect on the efficiency of code generated on a PDP-11 (Holden and Wand, 1978).

(ii) Device Processes may not signal each other. This restriction is designed to minimise the switching time for Device Processes, as it means that switching between Device Processes can only occur when a hardware interrupt is received.

(iii) Device Processes may not call non-local procedures. This restriction again reduces switching time and simplifies the scheduling of processes.

The first restriction was no real problem in programming in MODULA, but the last two certainly were. These restrictions considerably reduced the clarity of large programs, with frequent "fiddles" being necessary to get around them. In one program in particular, a handler written to enable the PDP-11/45 to communicate with other computers via a party line, these restrictions were particularly awkward. Such a program relies very heavily on hardware interrupts, and thus requires large and cumbersome Device Modules, as Device Processes cannot call non-local procedures. Signalling was also a problem in this context, owing to the fact that as most of the program consisted of Device Modules most of the processes were Device Processes and thus could not signal each other. One extreme measure was a process (in a standard module) that waited for a signal from a Device Process and then relayed that signal to another Device Process - not very efficient but a step of desperation. These restrictions considerably detract from program clarity and thus must, to some extent, reduce the advantages of programming in MODULA.

There are two options open if this arrangement is to be changed. The first option would be to remove all three restrictions - restrictions (ii) and (iii) because of their serious detrimental effect on the language, and restriction (i) because it has been shown that its lifting would not involve any substantially greater overheads. I realise that this will mean a substantial increase in the complexity of the compiler but these restrictions do seriously detract from the "useability" of MODULA, and so I consider the modifications worth any but the most drastic increase in overheads.

The second option would be to standardise the modules by removing the Device and Interface Modules (as in MODULA-2). I believe that the concept of separating machine dependent areas of the language (and thus machine controlling processes) is a strong aid to the structuring of efficient programs, and therefore I would prefer the former option if it were possible.

2. THE YORK COMPILER

The University of York Compiler (Version 1.00) was released on 20th June, 1978, and runs under the UNIX operating system. It is a four pass compiler written in BCPL and uses a sequential binary stream and in-core storage to communicate between the passes. There is no run-time storage allocation. Output is in the form of PDP-11 Assembly code.

The compiler has one major (and sometimes fatal) fault. An identifier may not be used unless it has previously been declared. In other words you can't mention an identifier in one module that has been defined in a subsequent module, even when it has duly been exported and imported. In even moderately large programs this rule is incredibly annoying. It often ruins the whole concept of modularity introduced in MODULA, and is sometimes insurmountable.

The compiler is satisfactory for compiling small, straightforward programs. It produces good quality code and does not take up large areas of core. Owing to the overlay methods employed in the compiler it is not sharable by users. Thus each user causes a new copy of the compiler to be loaded, resulting in a deterioration in system response time. This state of affairs would not be satisfactory if a number of people were using MODULA at the same time, but owing to the nature of the language and of the machines on which it will most probably be implemented, this problem should not often arise.

When one moves into larger more complex programs the compiler at its present stage of development is not satisfactory, due chiefly to the "declaration before use" rule described above. If this problem were corrected the compiler would be a satisfactory basic implementation of MODULA as presently specified.

3. IMPLEMENTATION RESTRICTIONS

MODULA has been designed with implementation on microprocessors a prime concern. The size of the language has been kept to a level that is acceptable for microprocessors, but there are a number of requirements that any computer on which MODULA is to be implemented must satisfy. The first two requirements are a direct result of MODULA's intended role as a systems programming language.

Firstly there must be a suitable form of hardware interrupts to satisfy the "doio" command. Coupled with this there must also be a facility to set processor priorities to ensure that these interrupts are processed according to the programmer's requirements. Virtually all modern micros have such facilities, and this requirement shouldn't really narrow MODULA's implementation possibilities.

Secondly there must be a way of accessing the hardware interfaces of peripherals. This is essential to provide a method of controlling any peripheral devices that may be attached to the processor. Again, most micros provide methods of access that would be acceptable to a MODULA compiler, although some do not provide as simple a system as that provided by the PDP-11 family of computers.

The third requirement is more directly connected with facilitating implementation than the previous requirements. It is mentioned by Holden and Wand (1978) and deals with the problem of implementing the reentrancy requirements of MODULA. Holden and Wand consider that some form of stack facility is essential if implementation is to be possible. They mention the INTEL 8080 as one example of a microprocessor that suffers in this area. I have had no personal experience of this problem, being involved in using rather than implementing MODULA, and therefore cannot comment in any detail on this restriction. Suffice it to say that a stack facility would greatly improve the chance of efficiently implementing the language.

It appears from the above requirements that MODULA could be implemented on most computers without a significant loss in effectiveness or versatility, although some of the presently popular microprocessors would not readily support an efficient implementation (owing to a lack of stack facilities). It would obviously be advisable to check that a micro satisfies the above requirements if an implementation of MODULA is to be attempted.

4. SUGGESTED EXTENSIONS/CHANGES

4.1 POINTERS

As detailed in section 1.1 the lack of pointer-based data structures is not a practical omission. Such data structures can markedly improve the efficiency of a program. It is therefore recommended that run-time storage allocation be introduced so that some form of pointers can be implemented.

4.2 SETS

I would also recommend that sets be introduced to the language, provided that the overheads involved are not too high, as this change is for convenience and program clarity rather than efficiency.

4.3 TYPE CHECKING

The present type checking is very rigid. This is generally an asset but when the hardware interfaces are associated with variables this asset becomes a nuisance and detracts from program clarity. For example, the programmer might generally need the value of device register to be numeric, so he declares his variable as an integer type. However, when just one bit of that integer has to be set, the rigidity of the type checking prevents the program from setting that bit to true, and requires the programmer to add or subtract numbers to set the bit. This can be rather cryptic to someone else trying to follow what the program is doing and also increases the likelihood of mistakes being made.

I would therefore recommend the introduction of a type, more basic than integer and bit types, in which the type checking is substantially relaxed. Perhaps a sensible restriction would be that such a type could only be used where device registers are involved.

4.4 DEVICE PROCESS RESTRICTIONS

As stated in section 1.2.3 I would recommend the removal of the three restrictions on Device Processes. This would greatly improve the flexibility of the language.

5. CONCLUSIONS

MODULA was designed to bring the advantages of high level programming into areas previously dominated by Assembly code. It was also necessary that storage requirements be kept at a minimum, so that implemation would be feasible on small computers. A compromise had to be reached between these two requirements. This compromise has generally been well decided, with the exception of the points set out in section 4. MODULA is well suited to implementation on small computers - compilers are relatively straightforward to write, the storage requirements are small, efficient code can be produced and implementation is possible on most machines.

Programs written in MODULA are far clearer and easier to write than their equivalents in Assembly code, but I would question the restrictions on device processes for the reasons stated in section 1.2.3. These restrictions are the major drawback of the language. Owing to the organisation of processes it is yet to be seen whether very large multiprocess programs could be written in MODULA. On the other hand MODULA appears very well suited to the writing of device handling programs, and it is here that MODULA makes its biggest contributions. The language is certainly a large step forward in this area.

6. BIBLIOGRAPHY

Brinch Hansen,P: "The Programming Language Concurrent Pascal", IEEE Trans. Software Eng., vol. SE-1, pp. 199-207 (1975)

Cottam,I.D.: "Functional Specification of the Modula Compiler", Report Number 13, Department of Computer Science, University of York (1978)

Holden,J. and Wand,I.C.: "Experience with the Programming Language Modula", Report Number 5, Department of Computer Science, University of York (1977)

Holden,J. and Wand,I.C.: "An Assessment of Modula", Report Number 16, Department of Computer Science, University of York (1978)

Jensen,K. and Wirth,N.: "PASCAL User Manual and Report", Springer-Verlag, New York, N.Y. (1975)

Control Data Corporation, "SIMULA Version One Reference Manual", Control Data Cyber 170 Series (1975)

Wand,I.C. and Holden,J.: "MCODE", Report Number 14, Department of Computer Science, University of York (1978)

Wand,I.C.: "Dynamic Resource Allocation and Supervision with the Programming Language Modula", Report Number 15, Department of Computer Science, University of York (1978)

Wirth,N.: "Modula: a Language for Modular Multiprogramming", Software - Practice and Experience Vol 7, 3-35 (1977)

Wirth,N.: "The Use of Modula", ibid 37-65

Wirth,N.: "Design and Implementation of Modula", ibid 67-84

Wirth,N.: "MODULA-2", Institut fur Informatik, ETH Ch-8092, Zurich (1978)