PROGRAMMING LANGUAGES AND SYSTEM DESIGN
J. Bormann (editor)
Elsevier Science Publishers B.V. (North-Holland)

Comparing Pascal and Modula-2 as Systems Programming Languages

Pieter H. Hartel

Vakgroep Informatica
Universiteit van Amsterdam
Plantage Muidergracht 6, 1018 TV Amsterdam

ABSTRACT

The high-level programming languages Pascal [JEN76] and Modula-2 [WIR82] are evaluated as tools for system programming. The construction of operating system utilities in Pascal is the focal point of the first part of the paper. Pascal is shown to be adequate for this limited class of applications, on the condition that the program development system provides enough support. The basis of the development system described here is formed by the POST library, a model of Control Data Corporation's Cyber operating system NOS/BE, written in Pascal.

Although the class of problems addressed can be solved with Pascal, a language like Modula-2 with features such as separate compilation and low-level programming support is better suited to serve as an operating system implementation language. To illustrate this, some Pascal programming examples were rewritten.

The reader is assumed to have a working Pascal and Modula-2.

1. Operating system implementation languages

Assembly language has been the systems programmers favourite for many years. Large operating systems, such as IBM's OS/360 and CDC's SCOPE and NOS are almost entirely written in assembly language. However, it is recognised that the use of suitable high level languages help to increase programmer productivity and facilitate debugging [BRO75]. These ideas have been exploited successfully in recent years [RIT75, RIT78].

The average systems programmer in a computer center is not involved in the design and implementation of large operating systems, but earns his daily bread modifying existing systems. These, when delivered by the manufacturer, often fail to meet the customers specifications, thereby introducing the need for so called local modifications. Making these modifications frequently involve major programming efforts, as was the case at the CERN computer centre, Geneva, Switzerland, where the author participated in the work presented here.

Modern ideas represent an operating system as a number of layers with functionality increasing outwards, all centered around a relatively simple kernel. The discussion in this part of the paper focuses on the highest level containing the operating system utilities. These programs list the users file directory, manipulate program libraries etc. Although such programs interact with various parts of the operating system, some of which may be hard to access, the amount of code in the interface is usually modest and can be used relatively easily, once modelled conveniently.

A high level programming language such as Pascal provides sufficient mechanisms to achieve the data abstractions required for operating systems modelling. For the design and implementation of well structured operating systems utilities, the algorithmic abstraction facilities of this language are sufficient. Since most operating systems are largely table driven, advanced facilities for algorithmic abstractions are of relatively little importance for the description of the program-operating system interface. The type structuring facilities of Pascal, and in particular the **record** and **array** constructors make it a suitable modelling language. The fact, that languages such as Fortran do not provide adequate data-abstraction mechanisms disqualifies them as suitable systems programming languages.

2. Pascal for Operating System software

Pascal offers some interesting features, which make it attractive for programming system utilities of modest complexity:

- Constant declarations allow for parameterising of program modules, a feature improving maintainability.
- The basic data types; the pointer, set and enumeration types and the type constructors allow the programmer to define precisely how his data objects are structured. At the same time it allows the compiler to perform rigorous type checking, avoiding some of the runtime overhead necessary for other languages.
- Block structure and good control structures greatly improve the readability of Pascal programs. The flow of control in a well structured program is usually obvious from the source text alone. No flow charts and the like are needed to design and understand a Pascal program.
- The language was designed to make writing the compiler easy. Thus, efficient compilers are common. Examples are the 6000 Pascal Release 2 compiler from the ETH at Z'ürich and its successor the 6000 Pascal Release 3 compiler [STR79] from the University of Minnesota. Both implement Pascal on CDC Cyber computers under various Operating Systems. In the work presented here, 6000 Pascal release 3 was used.

2.1. Low-level programming

In the standard implementation of Pascal, only a minimal system interface is provided. Basic input/output operations, and some auxiliary system functions (eg. *date* and *time*) are generally available. The systems programmer needs many more facilities and consequently has to write an elaborate interface himself. Describing such interfaces in Pascal is not always an easy task, mainly because certain table fields may have different meanings in different contexts. The solution normally involves violating the spirit of the language by defeating the type checking mechanism. For instance a certain memory location is described as an integer value on one occasion, and as an address on another.

In the following three sections, various options will be discussed in detail, with appropriate notes and cautions.

2.1.1. The variant record

Pascal does not provide a general type cast facility, though the variant record declaration may be used to perform type conversion. This is illustrated in figure 2.1. After assignment of *1* to the *adr* field of the *janus* record, the contents of virtual location *1* may be altered through assignment to the integer pointed at by the record field *ptr*.

```
{$T- switch runtime pointer checking off}
var
  janus: record case boolean of
    true: (adr: integer);
    false: (ptr: ^integer)
  end;
begin
  janus.adr := 1;
  janus.ptr^ := -1; { "PP call error" }
```

Figure 2.1 : The variant record

In this example, the type conversion being performed remains clear. The careful programmer resists the temptation of separating the record declaration and the statements where the conversion is performed.

2.1.2. External procedure declarations

Using separately compiled procedures and functions also allows the programmer to defeat the type checking mechanism, by making the actual procedure or function heading differ from the external declaration. This is far more dangerous than the variant record declaration, since both declarations will not be part of the same program text, and the fact that this conversion was used will be far from obvious. This facility should only be used with the utmost care. For example a procedure to copy a file may be declared as in figure 2.2.

```
procedure copy (var f, t: text); extern;
```

Figure 2.2 : External declaration - usage

The actual procedure text to access the fields of the file control blocks for fast block transfers of data is illustrated in figure 2.3.

2.1.3. Packing of data

Pascal allows data in **record** and **array** structures to be **packed**. The compiler will compute the minimal size in bits for each member of a **packed record** or for the elements of a **packed array**. The elements of a **packed** structure may or may not then be fitted together as closely as possible. This is of great value, when fields of tables

```
type
  fcb = record
    ...
  end { File Control Block };
procedure copy (var f, t: fcb);
var
  ...
begin
  ...
end;
```

Figure 2.3 : External declaration - implementation

in the operating system are not aligned on word boundaries. As an example, in figure 2.4 a 32-bit field is defined in terms of 8-bit bytes.

```
type
  byte = 0 .. 255;
var
  word: packed array [1 .. 4] of byte;
  i: integer;
begin
  for i := 1 to 4 do
    word[i] := i * i;
```

Figure 2.4 : Alignment of array elements

Using this facility, the programmer must be aware of the storage allocation scheme used by the compiler. In general, this is not good programming practice, since programs will become dependent on a particular version of a compiler.

3. POST Operating System model

If a high level language is used to interface to an operating system, the various tables used by the operating system must be manipulated from the high level language whereas operating system functions must be callable from the high level language as well. In other words, a model of the operating system in the high level language is required.

The author has described the NOS/BE tables and user-system communication areas in Pascal as **record** and **array** declarations. The POST (Pascal for Operating System Tasks) library is the collection of these declarations, together with many related **procedure** and **function** declarations.

Since the Pascal language lacks a facility for separate compilation, the source text of the required declarations must be textually inserted in a Pascal program via the 6000 Pascal Release 3 include facility [STR79]. For the implementation of operating systems utilities, which are inherently of modest size, the lack of support for separate compilation is felt only as an inconvenience. Type checking across program and library records is automatic and secure (unless disabled deliberately) since the complete source text is still compiled all at once. A minor disadvantage is that compilation time increases significantly.

The POST source library includes constant, type, variable, procedure and function declarations. The POST object library contains a (small) number of compiled Pascal and assembler procedures. The reason for supplying as many Pascal procedures in source form as possible, is to allow for compiler checking of calls and parameters for type compatibility.

3.1. POST constants

The constant declarations in the POST library typically define table lengths and the size of communication areas in terms of computer words. Also the size of the computer word is defined in terms of bits, bytes, characters etc. Portability of operating systems utilities for NOS/BE is of limited concern.

Program maintenance becomes easier, since the change of a table size requires one source line to be modified, as well as recompilation of all programs that depend on that particular value. Since no facility for separate compilation exists in standard Pascal, the dependencies of programs on the various elements of the operating system model must be carefully administered. This can either be done by hand, or through the use of auxiliary programs.

3.2. POST types

The **type** declarations come in various flavours. There are some types that describe strings of bits in terms of basic types as illustrated in figure 3.1.

```
bit10 = 0 .. 1023;
rclflgval = (norecall, recall);
char7 = packed array [1 .. 7] of char;
set6 = set of 0 .. 5;
```

Figure 3.1 : General purpose types

Figure 3.2 gives an example of a type which is more specific to the environment. It is the definition of a "SCOPE Logical file name" as used in the SCOPE 3 and NOS/BE operating systems.

```
left7 = packed record case integer of
  0: (tag: bit42);
  1: (c1: char; b36: bit36);
  2: (c2: char2; b30: bit30);
  3: (c3: char3; b24: bit24);
  4: (c4: char4; b18: bit18);
  5: (c5: char5; b12: bit12);
  6: (c6: char6; b6: bit6);
  7: (c7: char7)
end;
```

Figure 3.2 : String of 7 characters

A variable of type *left7* occupies only 42 (consecutive) bits. It may be cleared out by assigning a zero value to the *tag* field of the structure. Leading (sub)strings of the file name, which is left justified within the field, may easily be extracted or inserted.

The declaration of *left7* demonstrates how design flaws in an operating system lead to complication. The system uses a 6-bit character code (Display code), but makes the character with the representation zero inaccessible in normal text *).

Many tables are described in the POST library, such as the *FDB* (**F**ile **D**efinition **B**lock), the *actb* (ACT communication area), the *PFDentry* (**P**ermanent **F**ile **D**irectory entry) and the *rweb* (RWE communication area). The latter will serve as an example.

The peripheral processor (PP) programs on a CDC Cyber perform most system functions. These programs execute in one of the up to 20 peripheral processors, mainly used for I/O. An example is RWE (a**R**e **WE**), which returns the status of an interactive terminal to the program currently assigned to that terminal. Its communication area consists of one (60-bit) word. The layout of the various fields is shown in figure 3.3.

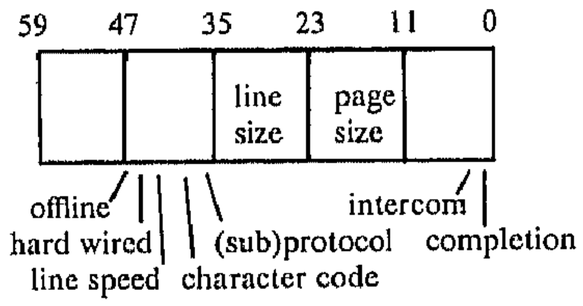

Figure 3.3 : RWE communication area

The description of the (one word) table in Pascal is given in figure 3.4:

3.3. POST procedure and function declarations

The **procedure** and **function** declarations in the POST library provide linked list processing, date and time conversions, operating system interfacing and some limited string handling capabilities.

Figure 3.5 gives a simple example of such an operating system interface procedure. The **function** *interactive* returns a boolean result, depending on the mode of operation of the calling program.

---

*) This problem does not arise when the 64-character set is used.

```
{ Character codes }
rweccval =
( rweASCII,    {  0     ASCII }
  rweExtBCD,   {  1     External BCD }
  rweDis       {  2     Display code }
);

{ Line protocol values }
rwelprval =
( rwelpr0,
  rwemode3,    {  1     Mode-3 protocol }
  rwemode4,    {  2     Mode-4 }
  rwemode2,    {  3     Mode-2 }
  rwewb        {  4     Wide band }
);


{ rwe parameter block }
rweb = packed record case integer of
  0: (tag:          integer);
  1: (
        u1:    bit12;   { unused field }
    offline:   bit;     { bit set if terminal offline }
    hardwired: bit;     { bit set if not dial-up }
    linespeed: bit3;
    charcode:  rweccval;
    subprot:   1..2;    { sub protocol }
    protocol:  rwelprval;
    linesize:  bit12;   { character per line }
    pagesize:  bit12;   { lines per page }
        u2:    bit10;
    intercom:  bit;
    compl:     bit
    )
end; { rweb }
```

Figure 3.4 : RWE model

```
procedure system1 (pp: char3; r: rclflgval; var par: integer); extern;

function interactive: boolean;

{************************************
  function to return true if we are
called from an interactive terminal.
************************************}

var
  par:      rweb;
begin
  with par do
    begin
      tag := 0;
      system1 ('rwe', recall, tag);
      interactive := intercom = 1
    end
end; { interactive }
```

Figure 3.5 : Function interactive

The **procedure** *system1* implements a monitor call, which passes the address of the communication area to the peripheral processor program.

4. System utilities

At CERN over 30 system utilities have been written in Pascal. The number of lines in each program ranges from 200 to 6000. The average number of 900 lines per program may serve as a measure of the complexity. In less than three man-years a total of 27000 lines of source text were produced.

Some aspects of the CERN AUDIT program will be used as an illustration of the success of the software development method advocated.

4.1. AUDIT

With the NOS/BE package, CDC provides the program AUDIT to list entries in the file directory. This program is written in FORTRAN (479 lines of code) and assembler for the central processor (another 720 lines of code). It also uses a specialised peripheral processor program EPF (2317 lines of assembler code). Altogether, there are 3516 lines of program text (comment lines are not included). Over the years many local modifications (400 lines of text) had crept into the program, in order to satisfy new requirements.

The conversion of the operating system from SCOPE 3.4 to NOS/BE 1.3 offered the opportunity to reconsider the situation. It was decided to write a new AUDIT program in Pascal. The new program consists of 2500 lines of program text.

One may question if a complete rewrite is not rather wasteful, in terms of man power as well as research and experience put into the manufacturers program. The contrary is however true for the following reasons:

- The old program had become messy, difficult to read and to modify. The new program is well structured, clean, easy to read and modify.
- Considering the increased functionality, much more effort would have to be put into modifying the old program, than was required to write the new one.
- The performance of the redesigned AUDIT compares favourably to that of the manufacturers program. The data of figure 4.1 refer to a set of files of a typical user at CERN.

	CDC's AUDIT	CERN's AUDIT
Execution field length (octal words)	40000	37400
CPU time used (seconds)	0.531	2.138
Real time used (seconds)	24	7.5

Figure 4.1 : AUDIT performance

5. Modula-2

The Modula-2 programming language is a successor to Pascal. As opposed to Pascal, the language was designed for systems programming and not for educational purposes. The support for separate compilation and low-level programming are important additions when compared with Pascal.

5.1. Modules

Modules in Modula-2 serve to group intimately related declarations together in a unit of program text and provide a mechanism to hide irrelevant detail.

5.1.1. Visibility

In Modula-2 programs, modules may be used to build a "wall" around parts of programs to hide certain details of the actual implementation from the clients of the module. Unless explicitly exported from the module, objects are hidden from the outside world. Conversely, only objects explicitly imported are visible within the module. The **import/export** specifications of a module provide assistance in program documentation and maintenance. Such facilities, which greatly improve the modularity of programs, cannot be implemented in Pascal. Ultimately procedures could be (mis)used as modules, but then no object declared locally would be visible to the outside world. Moreover no "own" variables could be used. The usefulness of such "modules" would be extremely restricted.

5.1.2. Separate compilation

For the implementation of operating systems utilities, which are inherently of modest size, the support for separate compilation is not strictly required from the programming language. This has been illustrated in chapter 3.

The main advantage of Modula-2 over Pascal is, that it supports separate compilation. A program may be assembled from modules, which interact through separately specified interfaces. In Modula-2 these are called **definition modules**. The implementation of a module may be designed independently of the interface

specification in a so called **implementation module**. If the implementation of a module has to be changed, its clients (ie. the modules using the interface) are not affected by the changes; these modules need not even be recompiled. If on the other hand the interface specification of a module is changed, then the modules using the interface will have to be recompiled. The Modula-2 compiler and linker will issue warnings to this effect. The separate compilation facility greatly improves ease and maintainability of software.

Figure 5.1 shows the programming examples of section 3.2 rewritten in Modula-2 illustrating the use of separately compiled modules.

```
DEFINITION MODULE RWE;

  IMPORT
    NosBe;

  EXPORT QUALIFIED
    ccval, lprval, b, interactive;

  TYPE
    (* Character codes *)
    ccval =
    ( ASCII,   (*  0      ASCII *)
      ExtBCD, (*  1      External BCD *)
      Dis      (*  2      Display code *)
    );

    (* Line protocol values *)
    lprval =
    ( lpr0,
      mode3,   (*  1      Mode-3 protocol *)
      mode4,   (*  2      Mode-4 *)
      mode2,   (*  3      Mode-2 *)
      wb       (*  4      Wide band *)
    );

    (* parameter block *)
    b = RECORD
          u1:          NosBe.bit12;    (* unused field *)
        offline:       NosBe.bit;      (* bit set if terminal offline *)
        hardwired:     NosBe.bit;      (* bit set if not dial-up *)
        linespeed:     NosBe.bit3;
        charcode:      ccval;
        subprot:       [1 .. 2];       (* sub protocol *)
        protocol:      lprval;
        linesize:      NosBe.bit12;    (* character per line *)
        pagesize:      NosBe.bit12;    (* lines per page *)
          u2:          NosBe.bit10;
        intercom:      NosBe.bit;
        compl:         NosBe.bit
    END; (* b *)

  PROCEDURE interactive (): BOOLEAN;

END RWE.
```

Figure 5.1 : Definition module RWE

The types necessary to interface the utility program to the system function RWE are specified in the definition module *RWE*, together with the header of the function procedure *interactive*.

As can be seen the interface specification is not concerned with the implementation details. These are given in the implementation module as shown in figure 5.2.

```
IMPLEMENTATION MODULE RWE;

  FROM SYSTEM IMPORT
    ADR;
  IMPORT
    NosBe;

  VAR
    status: b;

  PROCEDURE interactive (): BOOLEAN;
  BEGIN
    RETURN status.intercom = 1
  END interactive;

BEGIN
  status.compl := 0;
  NosBe.system1 ('rwe', NosBe.recall, ADR (status))
END RWE.
```

Figure 5.2 : Implementation module RWE

Once started interactively, a program will remain interactive. As a matter of optimisation, the module in the initialisation section may call the RWE system function to retrieve the terminal characteristics. These will remain valid throughout the execution of the program. The use of the *status* variable for maintaining the terminal characteristics mirrors this invariance. Furthermore, since this variable is visible only from within the implementation module all unauthorised accesses to it are prohibited.

5.2. Low-level programming

The effect of the variant record declaration and its use in section 2.1 may be achieved in two different ways, depending on whether the address is known at compile- or run-time.

5.2.1. Address specification of variables

This method as illustrated in figure 5.3 fixes the storage location of the *MonitorCall* variable to virtual address *1*.

```
MODULE test1;

  VAR
    MonitorCall [1]: INTEGER;

BEGIN
  MonitorCall := -1
END test1.
```

Figure 5.3 : Address specification of a variable

5.2.2. Type casting

Figure 5.4 shows the use of the type cast facility to assign to the variable *janus* the value *1* , such that the object pointed at is virtual location *1*.

This method is the more dangerous of the two, but it provides more flexibility over the method of fixing the storage location of a module variable, since the latter must be constant. Its appearance is much more clear than the Pascal equivalent.

5.3. Alignment of records and arrays

The **packed** property of Pascal may not be attributed to record or array declarations in Modula-2. Sometimes it is regarded as a serious deficiency [SPE82], but this need not be the case if the storage allocation scheme of the compiler is a sensible one. In the POST library all record declarations, except those used for type conversion are bit-aligned (ie. **packed**). Since the variant record declaration involves two or more equally sized fields in terms of storage allocation, the alignment boundary is irrelevant. Therefore, if the compiler would bit-align all

```
MODULE test2;

  TYPE
    IntPtr = POINTER TO INTEGER;
  VAR
    janus: IntPtr;

BEGIN
  janus := IntPtr (1);
  janus^ := -1
END test2.
```

Figure 5.4 : Type casting

record fields by default, the alignment specification can be left out of the language. The bit-alignment strategy would work equally well for arrays, since the only word-aligned (ie. unpacked) arrays in the POST library are character arrays, which are used during string manipulations. Modula-2 provides enough support for handling variable length strings to make word-aligned character arrays superfluous.

6. Conclusions

The use of Pascal or Modula-2 encourages structured programming. This makes programs easier to write, read and debug. It does not necessarily make them slower or less efficient than equivalent assembler or FORTRAN programs.

The strong typing of the languages forces the programmer to describe his data objects precisely and allows the compiler to check thoroughly, that manipulations on these objects are valid.

The systems programmer who needs to manipulate absolute addresses may do so within the framework of the language. Low-level facilities to achieve this should, however, be used with great care. They are in general dangerous, and violate the spirit of the language.

A model of the NOS/BE operating system has been constructed in Pascal. Sophisticated system utility programs can be written relatively easily and quickly in Pascal. Modula-2 can be used for a much wider class of operating systems programming problems. A model of the underlying operating system is required in both cases.

The main advantage of Modula-2 over Pascal is, that it supports separate compilation. This facility allows for a high degree of program modularisation. Separation of the interface specification and module implementation may be mirrored in the composition of the operating system design teams.

7. Acknowledgement

I would like to thank my colleagues of the CDC software section at CERN, in particular M. Silvano de Gennaro. I have received much helpful advice from Bob Hertzberger and David Rosenthal of the Faculteire Vakgroep Informatica, Universiteit van Amsterdam, whom I would like to thank especially for their constructive comments on the draft and final versions of the manuscript. Gijs Mos and Guido van Rossum of the Stichting Academisch Rekencentrum Amsterdam contributed to the implementation of the POST library.

References

[BRO75]
Brooks Jr., F. P.
The mythical man-month, Essays on software engineering
Addison-Wesley Publishing Company, Reading, Massachusetts, 1975.

[JEN76]
Jensen, K.; Wirth, N.
Pascal user manual and report
Springer-Verlag, New York, 1978.

[RIT75]
Ritchie, D. M.
C Reference manual
UNIX *) Bell Laboratories, Murray Hill, New Jersey, 1975.

[RIT78]
Ritchie, D. M.
The UNIX time-sharing system
Comm. ACM 17, 7, 1978, pp. 365-375.

[SPE82]
Spector, D.
Ambiguities and Insecurities in Modula-2
ACM Sigplan notices, Vol. 17, 8, 1982, pp. 43-49.

[STR79]
Strait, J. P.; Mickel, A. B.; Easton, J. T.
Pascal 6000 Release 3
University of Minnesota, Minneapolis, 1979

[WIR82]
Wirth, N.
Programming in Modula-2
Springer Verlag, Berlin, 1982.

*) UNIX is a Trademark of Bell laboratories

DISCUSSION

Lindsey, C.H.: Have you any figures, like you showed with the AUDIT example, to show when you implemented it in MODULA-2 that it occupied less space, or is quicker, or anything corresponding to that?

Hartel, P.: I have spent three years on this PASCAL exercise, which was from 1978 to 1981, and after that I have not been doing the same exercise again with MODULA-2. I just showed you some of the syntactical aspects of why MODULA-2 it is better. As a matter of fact, the compilers that are available are not really very good yet. Work will have to be put into improving generated code. I am confident that we will have good compilers at some point in time. So I have no figures. Even if I had figures, they would have been worse.

Persch, G.: I have two questions. The first one. How do you ensure that your records layout is correct? Do you really believe that only the known type-mapping of the compiler is efficient? Do you print it out and compare it with your record layout in the head and how do you think?

Hartel, P.: This is really a very difficult problem, and what I did in this case was: I took the manufacturers manual with the table layouts, sat down and wrote them in PASCAL and hoped they were correct.

Persch, G.: Can you prove it is correct?

Hartel, P.: I have no tools or anything and the bad thing about it is that you have to have some knowledge of how the compiler allocates its storage, and this is, of course, very low level programming.

Persch, G.: And the other thing is one of standardization, is MODULA-2 really unambiguously defined, the semantics? I think is a main criterion for standardization.

Hartel, P.: No, I think, the book as it stands there is rather incomplete. It would have been nice if it had been written down more precisely. Unfortunately, it is not. This is not the case for PASCAL anymore, because there have been these lenghty discussions about standardizing PASCAL, ISO-committees and the like. The same could happen to MODULA-2, I would say. But the language itself is not very precisely, specified.

Stiller, P.: When discussing MODULA you should have a glance at Concurrent PASCAL and EDISON. B. Hansen obviously likes to design languages in some congruent manner. My question is, did you make such a comparison including also EDISON, for instance, and if so, what is your opinion in general?

Hartel, P.: I have not compared MODULA-2 with EDISON nor with Concurrent PASCAL. I have looked a bit at MODULA-2 and ADA, and there, I think, that for the time being MODULA-2 is much simpler. We have compilers which are in size ten thousand lines as opposed

to hundred thousand of lines for the ADA-compiler. So I think for the time being MODULA-2 is the better choise. I think that it is a step in the right direction, it is not the last think to happen in this area. It is workable and it runs and I am happy with it.