The MODUS Quarterly

Issue #10

July 1990

Modula-2 News for MODUS, the Modula-2 Users Association

CONTENT

Problems? Missing a MODUS Quarterly issue? Do you want to subscribe? Contact your administrator at one of the address given below.

Subscription, Administration and Publishing

USA: Stan Osborne
MODUS (America)
P.O. Box 51778
Palo Alto, CA 94303-0721
USA

Europe: Aline Sigrist
MODUS (Europe)
Chemin de Gort 3
CH-1801 Mt-Pèlerin

Editor, MODUS Quarterly

K.N. King
Department of Mathematics
Georgia State University
University Plaza
Atlanta, GA 30303
(404) 651-2245

Submisions for publication

Send all submissions to the editor. Camer-ready copy is strongly encouraged; however, dot-matrix copy is usually unacceptable. If camera-ready copy cannot be furnished, articles may be submitted on floppy disk (IBM PC only, either 5-1/4" or 3-1/2") or by electronic mail. Articles submitted electronically must not require subsequent formatting. Files must be either plain ASCII, in PostScript, or in Microsoft Word format.

The *MODUS Quarterly* welcomes working papers, notes about work in progress, and examples of source code.

Please indicate that publication of your submission permitted. Correspondence not for publication should be PROMINENTLY so marked.

Editorial

MODUS Update

Astute members of MODUS may have noticed the large (i.e., two-year) gap between the publication of *MODUS Quarterly* #9 and *MQ* #10. Reasons for this gap include a lack of submissions (I've received only a handful of articles and letters over the last two years) and organizational difficulties within MODUS. (Procrastination on the part of the editor could be cited as another factor, but hey, who's writing this editorial, anyway?)

With the creation of a new board of directors and the appointment of Stan Osborne as administrator and *MQ* publisher, we've made a big step toward solving our organizational problems. The problem of insufficient material will persist, however, unless members take the time to share their ideas and enthusiasm.

Many thanks to those who submitted letters and articles for this issue. And to the rest of you: get moving! Finish that article that's been sitting on your desk for the past year. Share that nifty code that you've developed. Tell us your opinion of the new draft standard.

Status of Modula-2

With two years having passed since the last issue of the *MODUS Quarterly*, it seems like a good time to assess the status of Modula-2, both in the U.S. and worldwide.

In the U.S., Modula-2 continues to gain popularity, but not at the pace anticipated a few years ago. Although Modula-2 is now widely used in academia, it hasn't yet replaced Pascal as the introductory language of choice.

Commercial use of Modula-2 in the U.S. isn't widespread but neither is it as small as most people think. With several excellent DOS and OS/2 compilers available, Modula-2 is attracting an ever-growing number of microcomputer software developers. More surprisingly, Modula-2 enjoys a certain underground popularity among some prominent U.S. corporations. For example, General Motors is allegedly using an internally developed compiler for numerous mainframe applications. Unfortunately, companies that use Modula-2 often shun publicity, perhaps for fear of embarrassment (after all, their competitors use C!) or—as true Modulans believe—because they don't want the competition to know about the edge that Modula-2 gives them.

There are several reasons for Modula-2's slow growth. Unlike Ada, C, and C++, Modula-2 doesn't enjoy the backing of a large corporation or government organization. Major compiler vendors, notably Borland and Microsoft, have given it the cold shoulder. Pascal compilers now incorporate some of Modula-2's features, reducing the incentive for Pascal programmers to switch to Modula-2. And, of course, Modula-2 has suffered from the lack of a standard.

Ada continues to overshadow Modula-2, although the latter language occasionally gets revenge. I've heard of Ada compilers written in Modula-2. I've also heard of military contractors who write software in Modula-2 and then translate it to Ada before delivery.

During the next few years, Ada is bound to suffer from DoD budget cuts, perhaps creating new opportunities for Modula-2.

Fortunately, Modula-2 is doing much better outside the U.S. In the United Kingdom, Modula-2 has replaced Pascal as the primary undergraduate teaching language at most universities, and commercial interest in the language is strong. A one-day Modula-2 exhibition in London last May attracted 18 exhibitors and around 250 attendees, and the British Computer Society now has a Modula-2 special interest group.

Britain isn't the only country in which Modula-2 is popular. According to major compiler vendors, the language is prospering in German-speaking Europe, Scandinavia, Australia, and New Zealand, and is starting to make inroads in Canada and even Japan.

By at least one measure—the number of books available on the language—Modula-2 is a success. According to Real Time Associates in England, there are now over 100 Modula-2 books in print!

Do you know of an interesting use of Modula-2 in industry? Do you have a report on the status of Modula-2 outside the U.S.? Write an article—or just a letter—and we'll print it.

Standardization Update

Since issue #9, standardization activity has continued at a vigorous pace. Here's an update on the activities of ISO/IEC JTC1/SC22/WG13, the working group that is preparing the international standard for Modula-2:

August 1988: WG13 meets at Timberline Lodge on Mt. Hood in Oregon.

July 1989: WG13 meets at the University of Linz in Linz, Austria.

October 1989: The first draft proposed standard (DP) for Modula-2 is sent to national bodies for comment and balloting. The purpose of the ballot is to determine whether the DP should become a draft international standard (DIS) or be sent back to WG13 for revision.

April 1990: The ballot results are announced. The DP fails to gain approval as a DIS, with the U.S., France, West Germany, the Netherlands, and the United Kingdom voting against the DP. Seven counties support the DP; eight countries don't vote.

June 1990: WG13 meets at the Open University in Milton Keynes, England, to respond to comments on the first DP and resolve issues related to the preparation of the second DP. The I/O library is hotly contested; a three-person subcommittee is appointed to revise it. One surprise: COMPLEX is added to Modula-2 as a pervasive type.

November 1990: The second DP is scheduled for release. Another round of balloting begins.

July 1991: WG13 is tentatively scheduled to meet at Tübingen University in Blaubeuren, Germany. If the second DP gains DIS status, the meeting will focus on future extensions to Modula-2. If the DP isn't approved, the meeting will instead concentrate on the production of a third DP.

Upcoming Events

Two major Modula-2 conferences are currently on the calendar; both will be held in the U.K. Wales will host the First European Modula-2 Conference during December 17–18, 1990 (the Call for Papers appears in this issue). The Second International Modula-2 Conference is set for Loughborough, England, in September 1991.

About This Issue

If you'd like to get a copy of the first draft proposed standard for Modula-2, use one of the order forms in this issue. The DP can be ordered either from the IEEE or from MODUS.

The DP was reviewed by P1151—the U.S. Modula-2 working group—at its March 1990 meeting. The complete U.S. response appears on pages 36–40.

As always, your comments, ideas, and (of course) submissions are welcome. Write to me at the address shown on the inside front cover or send E-mail.

KNK

FIRST EUROPEAN MODULA-2 CONFERENCE

THE POLYTECHNIC OF WALES

17-18 December 1990

CALL FOR PAPERS

The First European Modula-2 Conference is being organised by the Department of Computer Studies at The Polytechnic of Wales. The objective of the conference will be to enable those interested in the the Modula-2 language and its environments to discuss and exchange ideas on recent developments in commercial, industrial and educational fields.

The Conference will comprise of presentation and discussion of submitted papers together with the opportunity for delegates to discuss and view Modula-2 software with vendors.

A list of topics will be:-

Software Engineering
Industrial Applications
Real-Time
Systems and Program Teaching
Object-Orientated Design
Standards for Industry and Commerce

Papers on recent work in these or other current Modula-2 issues are invited. Papers describing in detail case-study implementations using Modula-2 will be welcome.

Three copies of an extended abstract should be sent to:-

Dr M Al-Akaidi
Department of Computer Studies
The Polytechnic of Wales
Pontypridd, CF37 IDL
Mid Glamorgan, U K
Telephone: (0443)480480.FAX:(0443) 480558;
Email: malakaidi@uk.ac.pow.genvax

Dates:		
	Abstracts required by	31 August 1990
	Notifications to authors	28 September 1990
	Final papers required	20 October 1990
	Conference	17-18 December 1990

Programme Committee:
Dr M Al-Akaidi; Steve Collins (RTA); D E Eyres.

The Conference will be held at The Polytechnic of Wales which is located 11 miles north of Cardiff, and is easily accessible by road, rail and air. Accommodation will be available on the campus.

MODUS
P.O. Box 51778
Palo Alto, California
USA 94303-0721

10 August 1990

Dear MODUS Readers,

This letter is to provide you with current information about the Modula-2 Users' Association (MODUS).

My name is Stan Osborne. I am the new "American Administrator" for the Association. Like my predecessors I volunteered for this responsibility. It is my hope that we will be able to publish an issue of MODUS Quarterly every three months.

MODUS was formed to provide a forum for communication between all parties interested in the Modula-2 language. The primary function of MODUS is to publish the MODUS Quarterly. If my memory is correct, MODUS has also sponsored two conferences.

As many of you are already aware, MODUS has not fulfilled its primary function during the last two years. This is no longer the case. Issue #10 of the MODUS Quarterly is ready and being printed. Work has started on getting material for Issue #11. All past and present members are being notified by mail.

Within the next two years the international effort to produce the first Modula-2 standard will be completed. The MODUS Quarterly is one way for you to learn more about the standard and how it affects the future of Modula-2.

You can help with reactivating the Association by:

- submitting a membership application
- sending in letters and articles for publication
- telling other Modula-2 users about MODUS

My sincere thanks to you in advance for your help with the above. If you have questions, comments, criticisms, or complaints about the operation of the Association, please don't hesitate to tell me about them. I am especially interested in suggestions you might have about how any aspect of MODUS can be improved.

Sincerely,

Stan Osborne
American Administrator

Kevin P. Kleinfelter
Management Science America
3445 Peachtree Road NE
Atlanta, GA 30326-1276

To the editor:

I have been working on a project involving System V UNIX on a 3B2, MSDOS, OS/2, and AIX-PS/2. Code should be portable from one machine to another. Unfortunately, this seems to mandate the use of C. Although there are multiple sources of DOS and OS/2 compilers for Modula-2, we have been unable to locate a Modula-2 compiler for AIX-PS/2 or the 3B2. If any reader of this journal can provide information on availability of compilers for these environments, I would be very interested.

Modula-2 is a wonderful language, but until it is as available as C, I will be forced to stick with C. Companies already selling Modula-2 take note: you could sell more compilers for the environments you ALREADY support, if you would just support more environments.

Editor MODUS QUARTERLY
K. N. KING

Department of Mathematics and Computer Science
Georgia State University

Dear Kim,

In issue #9 of MODUS QUARTERLY you introduced the ideas of publishing articles about the use of M2 in industry and of instituting a list of suppliers for M2-related software. To both items I like to contribute in this letter.

Middle of 1986 I was engaged by the Austrian Federal Publishing Corp. (Osterreichische Staatsdruckerei) to start up a production of CD-ROM publications of jurisdictional literature. The project should begin with the Constitutional Law of Austria together with all related verdicts since 1909, then continue with an index of valid novellations and is currently in the state of finishing the third publication, the Decisions of the Supreme Court since 1945. Planned are about 25 more items of similar nature. Target systems are 80286 computers, with a graphic screen (Hercules or EGA standard) and a magnetic hard disk.

Being a rather new technology at this time, the use of optical disks required access to hardware control. On the other hand, building up a data retrieval system for mainly unstructured data in the size range of 100 to 400 MB of course requires a high level language with certain capabilities.
Other assumptions were that the database user should not be limited in any way accessing the documents, which means structured search as well as free input of any words, optionally linked together with relational operators.
After some short evaluation I rejected my first idea of using ADA and luckily ended up with Modula-2, system Logitech. In 1986 this compiler was the only one worth considering, and even now that several good products are on the market, Logitech's version 3.03 still seems to be the only one really suitable for larger projects.
By the time this letter will be published, those last statements may be outdated already, as I can see some very good systems on the way. Some weeks from now I will be able to do some tests with the Taylor-compiler, which from the technical specifications sounds promising.

The first two completed projects proved that the decision for M2 in general was right, especially talking about the regular updates of the CD-ROM publications and about the reusability of certain program components. In the updates data structures changed significantly, yet the adaption works on the software for the new releases turned out to be in the range of 1 1/2 man-month only (anybody ever tried to update a 150.000 source-line C program one year after first completion?).
In a similar way the developing time for the third production (Supreme Court) was reduced to about 65 percent compared to that of the first production (Constitutional Laws). Both figures are the best argument for using a Modula-2 system in a series of related software products.

Another, quite different, argument for M2 against other general purpose languages was the technical nature of the access to optical disks. The advantage of CD-ROM's for data storage is their capacity and certainly not their "speed" of data retrieval. Compared to magnetic disk storage the access time can be 5 to 8 times slower, when for some reasons a drive reset is required, that factor goes up one magnitude.
To ensure a decent performance of the retrieval system, some work has to be done already during data input by the user, commonly called multitasking.

Finally, due to the very high ergonomical requirements every program ended up with some 240K code in size, which needs a reliable overlay-system. To show the complexity of the retrieval software, I would like to list some of the features:
- boolean search links via and/or/not/adjacent - operators,
- left- and/or right-truncated search parameters,
- synonym recognition
- intermediate result lists with full document information,
- output to screen, printer and magnetic storage, graphic window system
- text processing capabilities
- context sensitive help
- mouse or keyboard operated
- optimal hardware tolerance

Of course, when the projects where first considered, I did not start programming retrieval software right away. The first thing was to produce an adequate library, which soon turned out to become a subsystem rather than a library. This subsystem is currently marketed separately from the CD productions under the name M2TOS (Modula-2 Task Organized Subsystem), and is already in use in several major software producing companies in Austria. Its price is rather high but developers seem to like it.
Modula-2 purists might say that some of the components of this system would still suggest the use of ADA instead of Modula-2, and that some of the things implemented cannot really be realized by using Modula-2 only. In fact, they can, if one considers the extensive use of CODE-statements not too bad as a programming style. The modules which are programmed this way are well packed and hidden, so to the application programmer the subsystem looks like M2 still. The main objective in this development was a practical approach rather than an academical.

Some components of the M2TOS - package:

- real-time kernel with self-initializing tasks, message system, task protection.
- re-entrant BIOS substitution
- run-time loading of drivers
- graphics library for different standards, access several graphic controllers from the same program (run-time loaded drivers).
- text output is done in graphic mode only, via an XWindows-like window system.

- low-level support for most CD-drives, high-level support of external CD-drivers and for MSCDEX
- virtual memory access and handling, both EMS and high RAM
- virtual data types (arrays without size limits)
- keyboard and mouse trap handler (macros up to the size of sub-programs can be logically linked to a key or to a gadget)
- text processing features with an object oriented approach
- printer spooler task

I think, from the range of items covered, it can be concluded that the package is strictly "developers only", and rather not useful for occassional users of Modula-2, but on the other hand provides help for almost everything that could come up in application programming.

It might be remarkable that, in a development system initially designed for doing database retrieval systems, there is not even an indexed file system included. Fact is that the approach of several 100MB data cannot be done in an dBase-like manner anyway, so this part of the retrieval software is developed and optimized separately for each single production. In the projects mentioned above, this section comprises approximately one fourth of the total code size.

It was suggested several times to adapt the package to other compilers and to port it to other machines.
As to adaptions, I first intended to make a version for the JPI-system, which unfortunately proved not powerful enough, especially with using overlays and run-time loading of code. Their coroutine transfer is also a bit too slow to work on more than 4 tasks at a time (some of my programs run on 8 tasks, M2TOS at least theoretically provides for 16).
A more promising adaption project (of minor priority, simply because there is no actual market yet) is a version for the Taylor Modula-2, which I hope to commence this year still.
Porting to other machines was even more frequently requested. Currently I am working on a release for Unix machines (PCS, 68xxx).

Initially I mentioned that this letter will be both a project study and a product announcement. The section project study of course was not covered to the extent the topic would deserve, but I hope it became obvious that M2 as a systems design language has its strong position. In case there should be interest in the topic, I am ready to supply a more detailed paper on implementation details of both M2TOS and the CD-ROM projects. For specific questions please use the mailing address below.

Yours sincerely

Ing. Peter Seewann
pHs software engineering
neusetzgasse 8
a-1100 vienna

vienna, october 1988

A Modula-2 Update Utility

Larry Irwin

Computer Science Department
Ohio University
Athens, OH 45701

irwin@ace.cs.ohiou.edu ...!att!oucsace!irwin
LIRWIN@OUACCVMB.BITNET

1. Introduction

Because of separate compilation, when a definition module is changed, any module which depends on it must be re-compiled, which leads to further re-compilations. With even a small program, the network of dependencies can be confusing.

The dependencies among modules can be represented as a directed, acyclic graph (DAG) with files as vertices, moreover there is hierarchical order in the graph. More structure comes about because of the four-member family of files for each non-foreign, non-main module name, that is, the two source files of definition and implementation and their corresponding compiled versions; and these files are related as

X.DEF -> X.SYM -> X.OBJ <- X.MOD.

While imports are mentioned in the source file, they have their effect from the corresponding compiled file. In effect, an OBJ file imports from SYM files in its own and other families. Dependency is of two kinds: 1) because of effective import (explicitly or implicitly) and 2) a compiled file depends on its associated source file.

Each file has an associated date. Thus when a source file is revised by editing, it obtains a new date. The update problem is then to re-compile the revised source file and all its descendents. The problem is solved by a depth-first traversal of the hierarchical DAG with post-order compilation. The graph can be built on-the-fly by starting with the main module and following the import paths depth-first, taking into account also the files implied in a family. A file is up-to-date when its date is later than every file it depends on.

The update problem can be solved by a simpler, bottom-up, breadth-first traversal of hierarchical levels, when one already has the graph in-hand (because it was saved from a previous update). But this requires that the levels of the hierarchy be numbered. Parnas [1] has defined a level numbering scheme: the level number of a file is one more than the maximum level number of files it depends on, or else it is 0 (when it is absolutely dependent). Source files are level 0.

2. Example

Consider the following up-to-date, small (but not so simple) program of three module families:

File	Date	Level	Depends on
M0.OBJ	350	3	M1.SYM, M0.MOD
M1.OBJ	353	3	M1.SYM, M1.MOD
M2.OBJ	359	3	M1.SYM, M2.SYM, M2.MOD
M1.SYM	340	2	M2.SYM, M1.DEF
M2.SYM	330	1	M2.DEF

The source files have not been put in the File column. The reader is requested to draw the corresponding graph with arrows pointing at the dependent file. For example, an arrow goes from M1.SYM to M0.OBJ, and an arrow goes from M0.MOD to M0.OBJ.

If a change is made to M2.DEF, then all of the source files must be re-compiled, moreover M1.DEF must be re-compiled before M2.MOD. If a change is made to M1.DEF, then only M1.DEF and all the MOD files must be re-compiled. As always in Modula-2, if a change is made to any MOD file, then only that file needs to be re-compiled.

3. The Utility

The utility was implemented in Hamburg Modula-2 for VAX/VMS as a project in the Software Engineering Project course at Ohio and is available at modest cost on a PC floppy disk containing the source code, installation files and a user reference manual. The utility is composed of 13 modules. The essential module encapsulating the graph could be used for any similar update problem, e.g., spreadsheet cells. Some modules encapsulate features which may change during porting, such as directory search, operating system services or a different compiler. A representation of the graph and its node contents are saved in a so-called MUG file, so that unchanged source files need not be re-read. Having specified the main module name, most of the time the user needs only to enter the MU command with no parameters in order to bring a collection of modules up-to-date.

Reference

1. D. L. Parnas, "On a Buzzword: Hierarchical Structure", Proceedings of IFIP Congress 74, Amsterdam. North Holland, 1974.

"Language Independent" Programming

Dr. Dennis S. Martin
Department of Computing Sciences
University of Scranton
Scranton, PA 18510
BITNET:MARTIN@SCRANTON

Abstract.

How much of programming depends on the nuances of syntax in a language? At the University of Scranton, we have conducted a very successful experiment in "language independent" programming using a Language Sensitive Editor for Pascal, Modula-2, and Ada. We have observed that beginning students are not confused and have not traded proficiency in tool-using for understanding. They make a very easy transition from Pascal to Modula-2 to Ada, adding concepts of programming instead of being overwhelmed with the details of the syntax of new languages. There has been an increase in productivity and quality in their software system development.

Case Study: VAX Language Sensitive Editor [2]

Our students start the major with a one semester introduction to computer science using Pascal. Modula-2 is used for most of the remainder of the coursework in the major and we use Ada in the programming languages course in the junior year. We are using the VAX Language Sensitive Editor (LSE) running on a VAX/VMS system to support these languages (among others). This editor can be user-customized for VAX supported languages and allows easy development of support files for other languages. VAX Pascal is a very powerful, non-standard version of the language and we chose to modify the editor for a more standard Pascal. Modula-2 is not a VAX supported language and we developed our own file for the language. For Ada, we have only modified the documentation facilities.

The VAX LSE uses Extended Backus-Naur Form (EBNF) [1] to generate programs. In addition to final code, the source file contains placeholders, which will be replaced by appropriate code. Placeholders have distinguishing delimiters, such as %{ }% or %[ ]%. These are standard printable characters which are syntactically meaningless in the language so the file is always printable with the placeholders easy to observe. The choice of delimiters determines if a placeholder is required or optional. A placeholder is either typed over or expanded. A terminal placeholder is a single element, such as an identifier, and must be typed over. Placeholders are expanded using control keys. A placeholder expands to a template of code, perhaps after choosing the template from a menu. Some placeholders, such as

%[statement_list]%.., automatically duplicate with proper indenting. Punctuation is inserted automatically.

A particularly convenient method of programming, illustrated below, is to type in and expand a special placeholder called a token. Tokens are language keywords, such as IF or FOR, which are typed in and expanded to a template. This is much faster than following a chain of menus and it is particularly useful when editing an existing file. Tokens may be typed in without regard to case and, when expanded, are corrected to the proper case.

A new file contains only a root placeholder, such as %{program_unit}% for Pascal or %{compilation_unit}% for Modula-2. Expanding %{compilation_unit}% in Modula-2, for example, produces the following menu:

```
-> %{program_module}%
   %{definition_module}%
   %{implementation_module}%
```

The cursor control keys move the arrow to choose the appropriate type of compilation unit. The chosen placeholder then expands into the template for the chosen type of unit. This template contains placeholders for the unit name, comments, library units imported, declarations, and body. All are properly indented with appropriate blank lines.

The LSE command COMPILE causes a copy of the source file to be automatically saved and compiled. If the compilation is unsuccessful, an error file is created. The programmer then can REVIEW the error file. This consists of an automatic process to split the screen into two windows, one containing the source file, the other the error file. The programmer reads an error in the error window then switches to the indicted location in the source file and corrects the error. This process is repeated until the program compiles successfully.

Modula-2 is a library based language with type checking across module boundaries for the information needed to properly use imported types, variables, functions, and procedures. The LSE allows the programmer to use windowing to look at the definition modules of imported units. This saves much time.

We have tried to provide a common "look" for all three languages. Our LSE version of Pascal was designed with a built-in "Modula-2 style" and both the Pascal and Modula-2 editors use a few nice Ada features. Modula-2 is a case-sensitive language. Our Pascal editor follows Modula-2 case rules in code that it generates. This produces much more readable Pascal code. Ada is not case-sensitive but is generated by the VAX LSE using consistent case rules. Unfortunately, it uses C case rules with reserved words lower case. We have not found this a problem. The consistency of case, supported by a case-correcting LSE, is sufficient.

Modula-2 requires end delimiters on IF, FOR, and WHILE. Ada requires "tagged" end delimiters (IF ... END IF, FOR ... END LOOP). We always generate tagged BEGIN ... END (* ... *) pairs in the Pascal statements. This eliminates one of the most persistent problem areas in Pascal coding. We add comment tags (END (* IF *), END (* LOOP *)) in Modula-2. We use similar indenting styles and rules for the use of blank lines for all three languages.

Consider the expansion of the token Pascal placeholder IF. Its template is

```
IF %{expression}% THEN BEGIN
    %{statement_list}%..
%[END ELSE IF %{expression}% THEN BEGIN %{statement_list}%..]%..
%[END ELSE %{statement_list}%..]%
END (* IF *);
```

The placeholder

```
%[END ELSE IF %{expression}% THEN BEGIN %{statement_list}%..]%
```

is optional. It may be expanded or deleted. If expanded, it becomes the template

```
END ELSE IF %{expression}% THEN BEGIN
          %{statement_list}%..
```

The ellipsis (..) indicates that the placeholder will be optionally repeated.

This produces code in the following style:

```
IF SexCode = 'M' THEN BEGIN
    WriteLn ( 'Student is male.' )
END ELSE IF SexCode = 'F' THEN BEGIN
    WriteLn ( 'Student is female.' )
END ELSE BEGIN
    WriteLn ( 'Error in Sex Code.' )
END (* IF *);
```

The similar statement in Modula-2 would be generated with very similar keystrokes and would look like

```
IF SexCode = 'M' THEN
    WriteString ( 'Student is male.' );
    WriteLn
ELSIF SexCode = 'F' THEN
    WriteString ( 'Student is female.' );
    WriteLn
ELSE
    WriteString ( 'Error in Sex Code.' );
    WriteLn;
END (* IF *);
```

Using the LSE, we can mimic features not in a language. For example, following Ada, expanding the parameter placeholder yields a menu of parameter types in both Modula-2 and Pascal of in, out, and in-out. This corresponds to the data flow rather than to the implementation-inspired variable/value distinction of these languages. In parameters become value parameters and the others become VAR parameters with the words in, out, and in-out remaining in the comments. The data flow associated with a procedure, an important design consideration, is exhibited.

A very important part of good design is to force the programmer to articulate the reasoning behind the code. This is accomplished when the programmer learns to develop useful internal documentation for a program. Extensive documentation is not necessary but good documentation is. The usual excuses for poor or non-existent documentation include not knowing what documentation is needed nor where it is needed. Another consideration is the amount of time that it takes to type documentation. These objections are answered by carefully chosen documentation templates automatically inserted, with titles, at the appropriate places in the code.

At the simplest level, documentation consists of a clear explanation of input, processing, and output, that is, what information is available, what the unit or procedure is supposed to accomplish, and the results desired. For example, our unit comments template expands to about a screen of text. It includes places for author, title, unit type, date, abstract, unit input, unit output, and modification history. Text automatically wraps to the next line, properly indented. Comments for procedures and functions are also standardized, requiring data flow, pre-conditions, post-conditions, implicit arguments, implicit results, and a functional description. Another critical area for documentation is in loops. Documentation must show under what conditions an exit from the loop will occur.

Conclusion.

As computer professionals, we need to use more computer-intensive tools to enhance programming. Good tools, used properly, can be effective in transforming good algorithms into good code, free from both syntactic and logical mistakes, in an efficient manner. With an LSE, the programmer uses fewer keystrokes, producing more code per time period. The programmer can concentrate on concepts rather than syntax, letting the tools provide the correct syntax, facilitating better quality code and reducing the need for extensive syntactic corrections. With good tools, it is easier to use better but less familiar language features. Finally, the programmer is encouraged to adhere to documentation standards, significantly reducing maintenance costs. The emphasis in programming must shift from syntax to concepts.

Because the LSE is user modifiable, we will continue to adapt

our templates as our understanding of what we want to accomplish increases.

References.

[1] Beidler and Jackowitz, Modula-2, PWS Publishers (1986).

[2] Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer, Digital Electronic Corporation, 1987.

An Extensible User Interface Toolkit in the PEM Environment

Frode L. Ødegård & Tomas Felner

Modula-2 CASE Systems A/S
Maridalsveien 139, N-0461 Oslo, Norway

e-mail: frode@m2cs.uu.no & tomas@m2cs.uu.no

1. Introduction

With *PEM* [M2CS89], Modula-2 CASE Systems (M2CS) provides an interactive software engineering environment for large-scale *Modula-2* [Wirth88] projects. PEM tries to bring the benefits of systems such as *Interlisp-D* [Barstow84] and *Smalltalk* [Goldberg83] to the Modula-2 user. PEM includes an object-oriented database, an object-oriented script language, a program synthesizer (editor, interpreter, debugger), an incremental compilation system, support for meta-programming, version and dependency management and a hypertext system.

PEM is unique in the sense that it supports exploratory programming and rapid prototyping as well as cross development (for embedded systems). The PEM system runs on a Sun workstation and offers a window and mouse-oriented user interface.

We have chosen the NeWS window system from Sun Microsystems as a basis for our user interface toolkit. This paper explains why. It also introduces *event projection* as a technique we have employed with great success in dealing with an extensible display server.

2. Primary Goals

Since the modules were to be used by outside programmers as well as M2CS employees, they had to satisfy the usual demands for reusable components (simple and clear semantics, ease of use, high implementation quality).

The toolkit also had to be flexible and easy to extend.

3. NeWS: an Extensible Display Server

NeWS [Gosling89, Sun87a, Sun87b] was developed by James Gosling et al. at Sun Microsystems and introduces a new and very exciting approach to display server design. NeWS is an interpreter for the *PostScript* programming language [Adobe85a, Adobe85b] with extensions for light-weight processes, events, display operations and object-oriented programming.

PostScript is a general-purpose, stack-oriented language with a device-independent graphics model; in many ways it is a high-level variant of Forth. Pointers are not supported (PostScript has dictionaries instead) and garbage collection is built into the language interpreter.

Non-extensible display servers employ fixed, low-level protocols resulting in complex client-side toolkits. The X [Scheifler86] window system is a good example. Another problem with non-extensible display servers can be the amount of packets exchanged between the server and its clients.

NeWS was designed as an extensible display server to solve these problems. It has a real programming language as its protocol and can work well over a modem connection for some applications.

NeWS clients can be written entirely in PostScript and downloaded to the server. NeWS programs can open files and even communicate over the network. On this level, clients are usually machine independent and are distributed in source form. There are many public domain utilities available, including multiple window classes, a Smalltalk-like class

browser and a powerful, visual debugger.

NeWS clients can also be compiled programs communicating with the server on a suitable level of abstraction. Such clients will typically transfer application-specific PostScript code (procedures, classes) to the server at startup. When the client wants to do a complex graphics operation, it sends PostScript code to call a downloaded procedure. When a client downloads PostScript code, NeWS allows the client to decide whether the code is to be kept for further sessions. This permits reusable components (such as window classes) to be *filed in* in a Smalltalk-like manner.

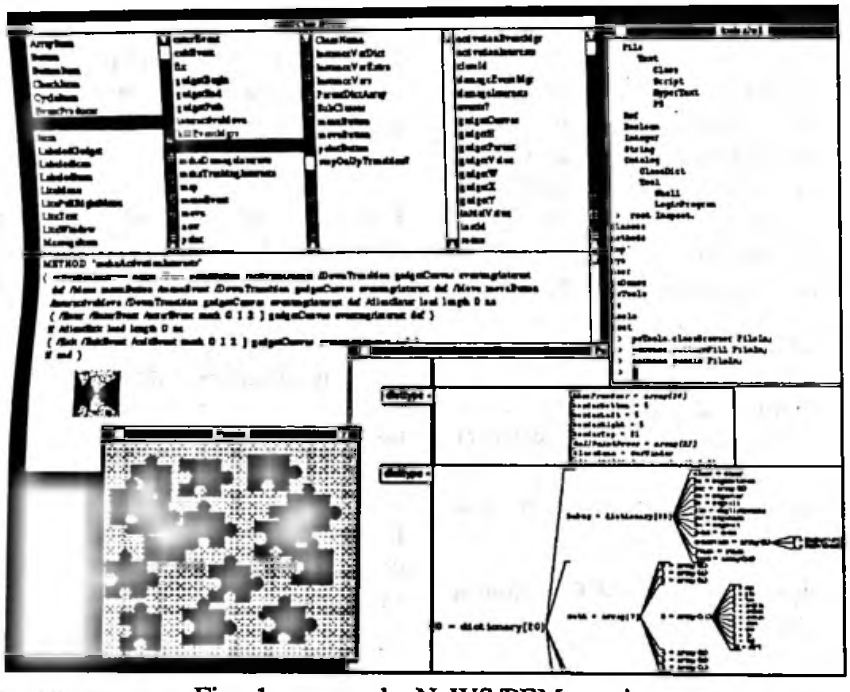

Fig. 1: a sample NeWS/PEM session

PostScript code can handle events locally. For application-specific code, event information can be sent to the client. An example would be a client which opens a window with two buttons labeled "Modula" and "Pascal". The PostScript code would handle the low-level events like depressing a mouse button and would send the string "Modula" to the client if the corresponding button was clicked on by the user.

Reusability is easily achieved for PostScript code, since NeWS provides object-oriented extensions. However, dealing with events in a consistent manner is almost impossible on the client side without a set of reusable modules. If events are not propagated to the client in a consistent manner, reusability on the PostScript side will be limited in large projects.

4. Event Projection

When designing the toolkit, the first step was finding a flexible and consistent scheme for event handling. We soon realized that there would be events on different levels of abstraction. For example, a PostScript button class would want to tell the client about each button click, while an icon editor could operate entirely on the server side and only inform the client when the user wanted to save the bitmap (after having clicked on a button). Furthermore, it was clear that different amounts (and types) of information would be supplied for different types of events. For the icon editor it would be desirable to send the whole bitmap to the client side as part of the event information when the user clicks on the "save" button.

Starting on the PostScript side we implemented a class *UIEvent* with a method to send event information (represented in a dynamic-length PostScript array) to the client side. A scanner was then written on the client side to read event information from the server.

- a handler in the client module called by the *EventListener*
- a handler installed by a PEM applications programmer called by a gadget module

For each level of abstraction, an event may or may not be propagated to the next

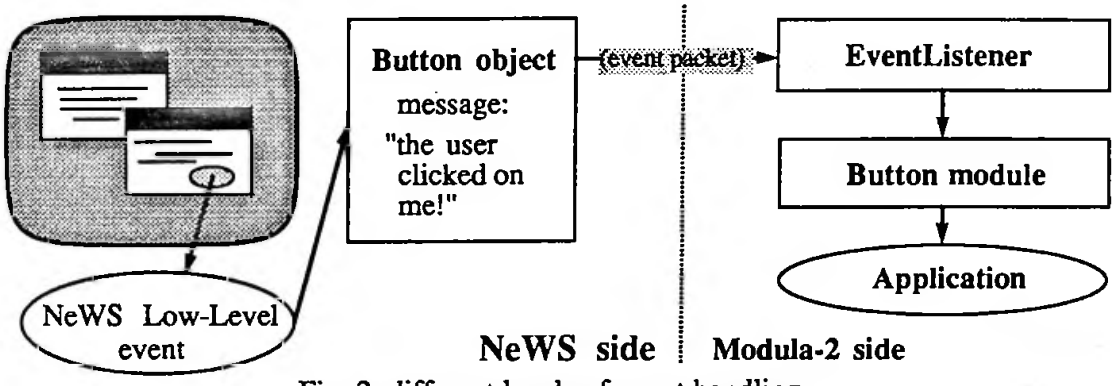

Fig. 2: different levels of event handling

An *event packet* sent from the server consists of a two numbers identifying the PostScript object responsible (a button, for example) and a list of additional items. When the event packet is read on the client side, a handler procedure for the event is found (typically in the module *Button* in this example) and called.

Since the module reading event packets cannot make assumptions about the optional items, the *stream connection* is propagated to the handler procedure which subsequently reads any remaining items in the event packet as well as the termination symbol.

The handler then determines whether to update data structures and/or call a *user handler* (provided by the module which imports from the toolkit).

To summarize, events can occur on many different levels of abstraction:

- a primitive mouse or keyboard event which activates the NeWS event handling code
- a "mouse clicked in your area" event propagated to a higher-level PostScript object (such as a button)
- an event packet sent to the Modula-2 side

level. Given E_n as the set of events which may be propagated to a level n+1 from level n, we have

$$\mathbf{card}\ E_n \geq \mathbf{card}\ E_{n+1}.$$

Since we usually have

$$\mathbf{card}\ E_n > \mathbf{card}\ E_{n+1}$$

or even

$$\mathbf{card}\ E_n >> \mathbf{card}\ E_{n+1},$$

we use the term *event projection* instead of *event propagation*. The most obvious example is when complex event handlers reside on the server side and don't need to inform the client side at all. On convenience, higher-level event information can be sent to the client side.

5. General Architecture

PEM user view the toolkit as a collection of Modula-2 modules for building user interfaces. There is a common window module and multiple *gadget* modules.

A gadget is an item which can be placed inside a window. Gadget modules import from the common window module.

The window module and all gadget modules register themselves with EventListener to have relevant event information delivered to their own internal handler.

EventListener talks to NeWS via another module called *DisplayServer*. A gadget module uses a direct stream connection obtained from DisplayServer to download its PostScript code at startup (provided it isn't already resident).

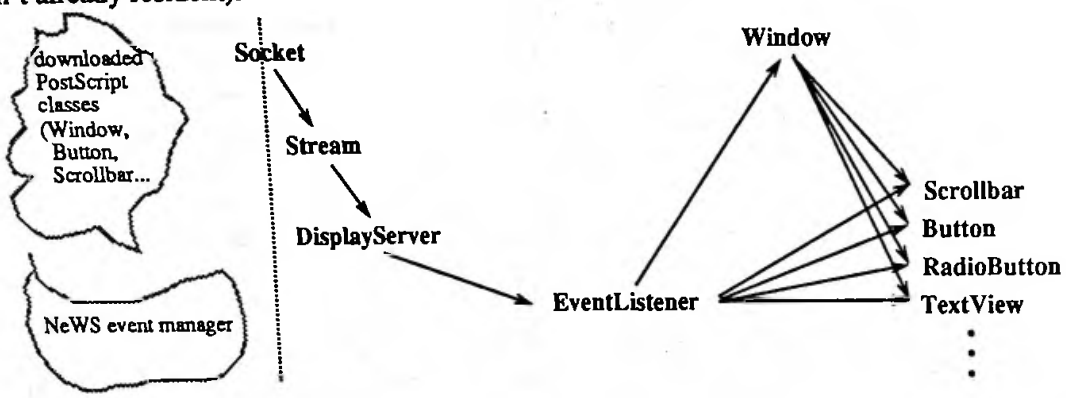

Fig. 3: architecture of the PEM user interface toolkit

New tools and modules can use a direct stream connection as an alternative to using existing toolkit modules. This is provided as an option to be used only when reusability must be sacrificed for the greatest flexibility.

6. Gadgets

Gadget modules usually have the same name as the PostScript class they interface to. The module body will verify that the necessary PostScript class has been downloaded

Most gadget modules will allow their clients to simply await an event as an alternative to installing a handler procedure for simple events.

Gadgets are easily added by subclassing existing PostScript classes and writing Modula-2 modules which interface to them.

More and more high-level gadgets are being written. A ChessBoard gadget could look like a new type of stream to PEM programmers; a stream on which chess moves are read and written in a textual manner. Given such a gadget, the programmer can easily build a window with a ChessBoard gadget, a "quit" button, a "help" button, etc. Implementing PEM's graphics components as gadgets instead of specialized windows thus results in a higher degree of reusability as well as increased flexibility.

7. Implementation Issues

PEM includes a set of modules supporting *streams*. A stream object supports device-independent I/O operations. There are multiple *driver modules* for streams (e.g. File, MemoryStream, Socket). The Socket module is used to obtain a stream connection to the NeWS server. This driver supports asynchronous I/O, which is closely coupled with PEM's light-weight process library.

An EventListener object executes as a separate light-weight process which is activated when NeWS sends data. It is possible to start several EventListeners, by default a separate listener is started for each display server used.

The toolkit allows the user to open windows on several display servers. We are also studying the possibility of moving a window from one display server to another. This would allow the user to move from one workstation to another and have running applications move with him/her.

References

[Adobe85a] Adobe Systems, Inc., PostScript Language Reference Manual, Adobe Systems, Inc., Addison-Wesley, 1985

[Adobe85b] Adobe Systems, Inc., PostScript Language Tutorial and Cookbook, Adobe Systems, Inc., Addison-Wesley, 1985

[Barstow84] Barstow, D. R., Shrobe, H. E., Sandewall, E., Interactive Programming Environments, McGraw-Hill, 1984

[Goldberg83] Goldberg, A., Robson, D., Smalltalk-80: The Language and its Implementation, Addison-Wesley, May 1983

[Gosling89] Gosling, J., Rosenthal, D. S. H., Arden, M., The NeWS Book, Springer Verlag, 1989

[M2CS89] Ødegård, F. L., PEM - A Programming Environment for Modula-2, Modula-2 CASE Systems A.S, 1989

[Scheifler86] Scheifler, R. W., Gettys, J., The X Window System. ACM Transactions on Graphics, 5(2), April 1986

[Sun87a] Sun Microsystems, Inc., NeWS 1.1 Manual, Sun Microsystems, Inc., PN 800-2146-10, 1987

[Sun87b] Sun Microsystems, Inc., NeWS Technical Overview, Sun Microsystems, Inc., PN 800-1498-05, 1987

[Wirth88] Wirth, N., Programming in Modula-2, 4th edition, Springer-Verlag, July 1988

Two Limitations of Modula-2

Rodney M. Bates
1513 Blue Spruce Road
Derby Kansas 67037
(316) 788-7566

Compared to Ada, I find Modula-2 refreshingly clean and simple. However, there are two somewhat similar programming problems I have encountered where the language is a little too simple. It forces me to resort to some ugly hacking which I consider worse than the added language complexity it would take to solve the problems more cleanly. These are real programming situations I am now facing in Modula-2.

In the first problem, I want to define a module which implements some data structure and which exports an iterator to traverse the structure, calling a procedure formal named Visit for every element. For example:

```
DEFINITION MODULE TreeMod;

  TYPE ElementTyp = ...;
  TYPE TreeTyp = ...;  (* probably  opaque *)

  (* various tree  manipulation procedures... *)

  TYPE VisitProc = PROCEDURE ( ElementTyp );

  PROCEDURE Traverse ( Tree : TreeTyp ; Visit : VisitProc );

END TreeMod;
```

The internal data structure is sufficiently complex that the *only* reasonable way to traverse it is with some kind of recursive traversal, with calls on Visit located in several places in the code and these executed at many levels of depth of recursion. When I use Traverse, in almost all cases, I need to pass it a Visit procedure whose body can somehow access various variables and parameters accessible at the point where the call on Traverse is found. With the restriction that procedure actuals can only be declared at the outermost level, the only way this can happen is to use global variables. But I need to do this in code which can be executed by multiple processes, so globals won't work.

A cheap approach is to package the necessary data in a state record and pass the record around. This can be done crudely without change to the language by giving both Traverse and VisitProc an extra parameter of some relatively universal and pointerish type such as SYSTEM.ADDRESS and using this to pass the pointer to the state record around. Since different clients of Traverse will in general have different state records, type conversions are necessary. At least the use of the pointer instead of the state record itself means it is the same size for all clients.

A second approach would be to weaken the rules about

procedure parameters so they can have static environments as part of their values. For example:

```
PROCEDURE PrintMatchingElements
  ( Tree : TreeMod.TreeTyp ; MatchValue : CARDINAL );

  PROCEDURE Visit ( Element : TreeMod.ElementTyp );

  BEGIN
    IF TreeMod.Value  ( Element )
       = MatchValue (* a non-local reference *)
    THEN
      PrintElement ( Element )
    END
  END Visit;

BEGIN
  TreeMod.Traverse
    ( Tree,
      Visit (* an illegal actual parameter *)
    )
END PrintMatches;
```

Visit needs to be inside PrintMatchingElements in order to refer to MatchValue. But, by existing rules, this makes Visit an illegal parameter for the call on Traverse, since it is not declared at the outermost level.

In Pascal, this is allowed. It works because Pascal has no procedure variables. If the rule in Modula-2 were simply eliminated, it would be possible for procedures to refer to non-local variables which didn't exist. For example:

```
MODULE M;

  VAR ProcVar : PROC;

  PROCEDURE P;

    VAR PV1 ,  PV2 : INTEGER;

    PROCEDURE Q;

    BEGIN
      PV1 := PV2
    END Q;

  BEGIN (* P *)
    ProcVar := Q
  END P;

BEGIN (* M *)
  P;
  ProcVar
END M;
```

The call ProcVar is a call on Q, which uses PV1 and PV2. But by now, P has returned, so these variables don't exist. What happens cannot be explained without using machine level concepts, and these will in general, be different for each machine/compiler combination. The value of PV2 will be garbage, and something unpredictable will get stepped on by the assignment to PV1.

It is a well known consequence of the usual scope rules that this kind of problem is impossible when there are only procedure parameters but no procedure variables. It is possible to come up with one or more systems of rules, each of which allows procedure variables, doesn't restrict their values to outermost procedures, prevents the kind of problem shown in the example, and is statically enforceable. Unfortunately, these get pretty complex. The appendix gives one system which I believe does all this.

However, this is obviously far too complicated. A system such as this would be more consistent with the philosophy of Ada than with that of Modula-2. A much simpler alternative which still solves the problem I have combines but does not intermix the system of Modula-2 and that of Pascal.

There are two kinds of procedure types. One is exactly the set of procedure types currently in Modula-2, with the same rules. In particular, values must be procedures declared at the outermost level.

Procedure types of the second kind are strictly incompatible with those of the first kind. No procedure variables of these types are allowed, only parameters. However, actual parameters can be declared at any level.

Syntactically, this distinction can be made conveniently. The former kind of procedure type is usable as a parameter only if it has a type name, which is used in the formal declaration:

```
...
TYPE BoolProc = PROCEDURE ( BOOLEAN );
...
PROCEDURE P ( PFormal : BoolProc );
  (* BoolProc is a type name *)
```

Here, PFormal is compatible with other variables of type BoolProc, but all such values must be outermost procedures.

The new kind of procedure type is allowed only as the type of a formal, so it is denoted by putting the entire type definition (not just a type name) right into the formal declaration:

```
...
PROCEDURE Q ( QFormal : PROCEDURE ( BOOLEAN ) );
  (* PROCEDURE ( BOOLEAN ) is a type definition *)
  ...
```

Here, QFormal is incompatible with any procedure variable. However, a caller of Q can supply an actual parameter which does not have to be outermost.

My second programming problem is similar, except that now the user of Traverse also must navigate (in my case, it must

build) another complex structure which again can reasonably be done only with a recursive traversal. The two structures are different shapes and thus a single glorious recursive algorithm to both traverse the tree and build the other structure can't be constructed. Even if it could, that would mean abandoning the abstraction of separate algorithms.

Thus the obvious thing to do is to use coroutines. But every time a SYSTEM.TRANSFER is done from one to the other, some values must be passed. Again these can only be passed in globals, and again, this pair of coroutines must be executable by multiple processes (each with its own pair), thus globals won't work.

The approach I am now using, which works with the language as is, is to use a state record and pass its pointer in global variables but protect the exchanges between a related pair of coroutines with mutual exclusion synchronization. Fortunately, it is possible to arrange to do this only once when the coroutine pair is created, rather than every time a SYSTEM.TRANSFER is done.

A more reasonable solution would be to give SYSTEM.NEWPROCESS and SYSTEM.TRANSFER each an additional parameter for passing around state record pointers. These procedures are really part of the language itself, since they cannot reasonably be written in Modula-2. Putting skin procedures around SYSTEM.NEWPROCESS and SYSTEM.TRANSFER does not help; it only provides a place to put whatever hack is used. Thus a language change really is required.

The extra parameters would be a very simple language change, and I am proposing it as a solution to my second problem. For SYSTEM.TRANSFER, the extra parameter is VAR. Whatever value the caller supplies, the coroutine transferred to receives. This implies that when a return to the original caller of TRANSFER finally occurs, a new value of this parameter is returned, supplied by whatever other coroutine transferred to the first one.

For SYSTEM.NEWPROCESS, the new parameter is a constant parameter. The type of NEWPROCESS's formal P is changed to a procedure type which accepts one VAR parameter of type SYSTEM.ADDRESS. When the new coroutine starts, this formal parameter will be equal to the value of the new parameter to NEWPROCESS.

```
DEFINITION MODULE SYSTEM ...

...
  TYPE PROCESSPROC = PROCEDURE ( VAR ADDRESS );

  PROCEDURE NEWPROCESS
    ( P : PROCESSPROC;
      A : ADDRESS;
      n : CARDINAL;
      STATE : ADDRESS
      VAR p1 : ADDRESS;
    );
```

```
PROCEDURE TRANSFER
  ( VAR p1 , p2 : ADDRESS ; VAR STATE : ADDRESS );

...
```

Even with the suggested change to SYSTEM.NEWPROCESS and SYSTEM.TRANSFER, I regard the state record approach to be fairly unpleasant. Intuitively, I feel that packaging state into a record and explicitly passing it around was one of those techniques which was appropriate perhaps twenty years ago, but which the concept of local, automatic variables was supposed to obviate. More concretely, it requires the deliberate circumvention of the type rules, with corresponding loss of static safely, in a setting which has nothing to do with low level programming, machine dependencies, etc.

My language change proposal for the second problem merely makes it possible to use the hack analogous to the one I have objected to as a solution to the first problem. I don't know of anything cleaner, however.

I realize than many regard complicating the language as a terrible sin. I too believe in keeping things simple as possible. What I don't believe in is simplifying the language at the cost of greatly complicating programs written in it, in ugly and dangerous ways. I think these proposals represent good tradeoffs between language complexity and program complexity.

Appendix

This system allows procedure variables, doesn't restrict their values to outermost procedures, prevents references to *nonexistent* local variables, and is statically enforceable.

1. For every procedure variable and every constant procedure formal parameter, define its "innermost value scope" as the scope where the variable/formal is declared.

2. For every VAR procedure formal, define its innermost value scope as the scope where the containing procedure is declared, i.e. one level out from the scope where the formal itself is declared.

3. It is an invariant that the value of a procedure variable/parameter is a procedure declared in its innermost value scope or in a scope outer to it.

4. All assignments of procedure values VL := VR have a compile-time check that the innermost value scope of VL is equal to or inner to that of VR. Passing an actual to a constant formal is a form of assignment, for purposes of this rule.

5. When passing to a VAR formal, the actual and formal must have exactly the same innermost value scope.

6. When calling the value of a procedure variable or formal, PV, actuals passed to constant procedure formals of PV will have to be declared at the outermost level (since PV could be outermost) and VAR parameters must be disallowed altogether. A consequence of this is that a procedure variable declared at other than the outermost level, of a type with a VAR procedure parameter can never be called, and thus might be made illegal to even declare.

7. These rules must be extended from variables and parameters of procedure type to those which contain procedure types. The definition of innermost value scope is extended to variables/formals of records and arrays which contain procedure types. The assignment and parameter passing restrictions are applied to these additional types. The innermost value scope of a component is the same as that of its containing array/record.

8. These rules must also be extended to cover heap objects which are or contain procedure types. The definition of innermost value scope is extended to variables/formals of pointer types which point to objecs of such types. The assignment and parameter passing restrictions are applied both to these pointer types and to objects and their components accessed by them. The innermost value scope of a heap object is the same as that of the pointer used to locate it.

Aside from its complexity, another drawback in this is that the representation of procedure values now must have both a code pointer and an environment. This would be incompatible with pointers to procedures in, for example, C, in case one wanted to pass these between languages. But this is type-unsafe anyway. Moreover, C necessarily must have a parameter passing convention which doesn't match the reasonable (and universally used) convention for Modula-2, which usually means there must be compilers which understand multiple conventions and directives to tell them which to use. Surely this procedure value representation problem can be solved with no more ugliness than is required anyway. For example, a compiler directive might call for certain procedure types to be represented without the environment, reverting to the rule that their values must be outermost procedures.

The integrated method has been adopted in several environments such as the Cornell Program Synthesizer[7], InterLisp[8], Gandalf[9], Smalltalk[10], etc.

This paper describes the design of a Modula-2 integrated environment and the implementation of its kernel on IBM PC/AT.

2. Modula-2 Integrated Environment and its Kernel

We have been making researches on Modula-2 environment for several years. Modula-2 is a hopeful system programming language presented by professor N.Wirth. It provides module facilities which is very essential to large software[11].

At present, the environment includes a Modula-2 compiler, a linker, a run time debugger, a syntax directed editor[12], a Modula-2 design tool based on PDL[13] and several automatic transformers between different high level languages[14,15]. As is discussed above, the environment is lack of integration. So we are determined to develop an integrated environment which will support the integration of the tools in the current envrionment.

2.1 Overview of the integrated environment

The four goals in our mind when we design the integrated environment are

1). providing interactive multiple-window user interface

Interactive user interface allows the user to detect the wrong operations early so as to prevent the unnecessary actions of the environement. By means of the multiple windows the screen can be used very efficiently and the user may get more information from a single screen.

2). inheriting the tools in the original environment

The new environment should be compatible with the original environment and can inherit the software in the original environment easily.

3).using a software information data base to manage tools and projects.

All the tools in the environment can share the data base and be integrated loosely by the data base.

4). using a kernel environment to support the integration of the coding phase and the debugging phase.

Since the coding phase and the debugging phase are most often repeated during the software development process, it is significant to make the two phases integrated tightly in a kernel environment.

The whole integrated environment can be described by Fig.1.

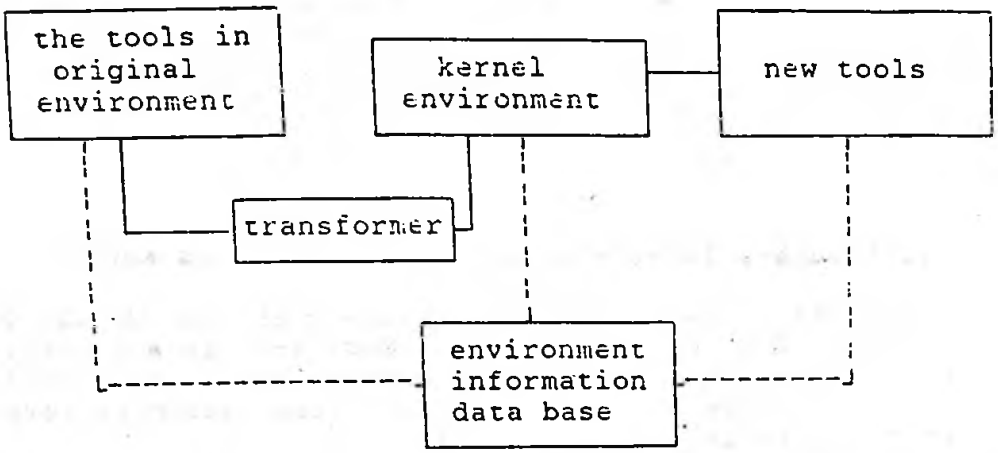

Fig.1 Modula-2 integrated environment

The tools in the environment can be either integrated loosely by the environment information data base or integrated tightly by sharing the representation of Modula-2 program which is defined in the kernel environment.

2.2 The kernel environment

The kernel environment implements the integration of the editor and the debugger. It can also support the incremental compiling and separate debugging of Modula-2 modules.

The kernel environment can be described by Fig.2.

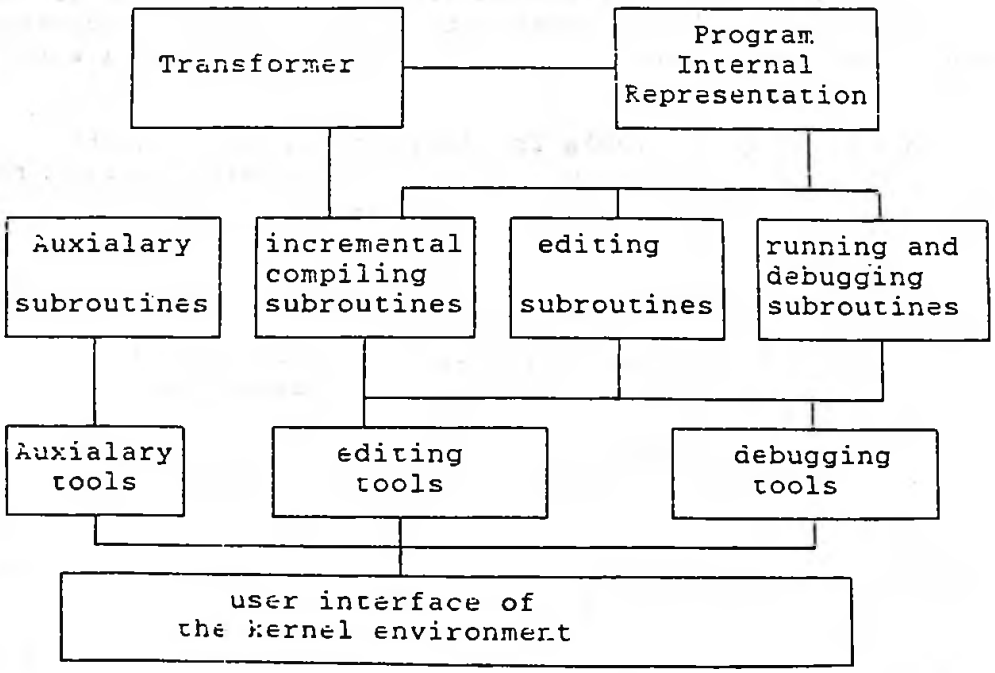

Fig. 2 the kernel environment

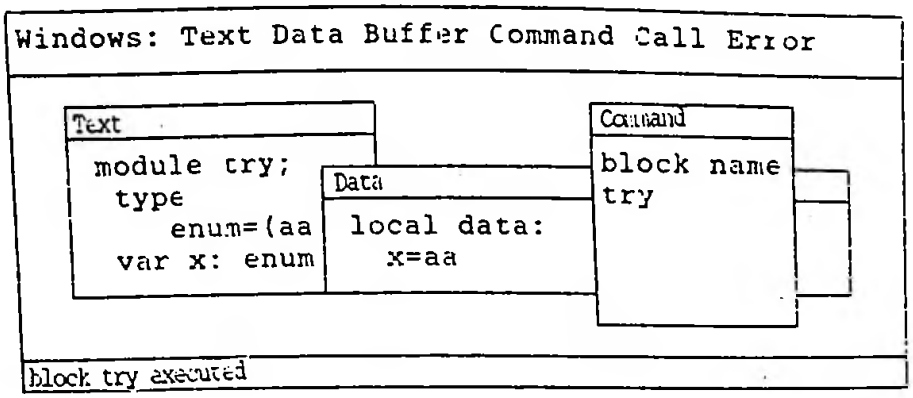

Fig. 4 the screen of the kernel environment

The kernel environment supports six windows which are text window, data window, call window, buffer window, command window, and error window. These windows are used to display program text, to display the data when debugging,to display the call relations of the procedures, to edit programs,to display command prompt and to display error information respectively.

There are four classes of commands in the kernel environment. Each class of commands is provided with a single menu. These commands are:

1). Editing commands

This class of commands are used to edit the program in full screen mode or in syntax directed mode. A program can be loaded into buffer window and then edited. Some editing commands can lead to the incremental code modification automatically, i.e., after editing the program can be immediately debugged without recompilation. For example, you may delete a variable declaration clause, which will lead to the deletion of all the statement related to the variables declared in the declaration clause.

2). Debugging commands

This class of commands are used to debug the program. The user may set or reset break points at any place of the program and can execute the program until a break point appears or continue executing the program from a break point to another. The debugging interface also supports single step execution. At any time when the program pauses, the data of all the procedures or modules can be checked. The data are displayed in data window and may also be modified when it is necessary.

Separate debugging is supported. As soon as a module is compiled into the tree representation(see Fig.3). The module can be debugged even though the other imported modules haven't been programmed. However, in such a case, the user should define the execution environment of the module by setting the data value in data window to simulate the real execution environment.

3). File Commmands

This class of commands are used to operate the files including listing directory, showing a file, erasing a file, renaming a file etc.

4). Other commands

This class of commands include calling a Modula-2 program or an execution file and window operations etc.

Above all, the interface of the kernel environment can be programmed by the user. The user may define his/her own interface in a configuration file. The color of each window, the size of editor buffer and the other attributes of the interface can be redefined by the user.

4. The Incremental compilation facility

In the original environment when a part of a module is modified the whole module must be recompiled. For a large module the recompilatin is usually much time-consuming. The kernel environment supports incremental compilation of Modula-2 module, i.e. as soon as some part of the module is changed the kernel environment will adapt the program tree accordingly, so that the user can continue executing or debugging of the module. The incremental compilation facility obviously improves the efficiency of software development. In fact it is also the key to the integration of the editor and the debugger. Because of the incremental compilation the environment eliminates the difference between editing status and debugging status.

In the program tree the description of each syntax unit is local to several nodes so that there will not be inconsistent description when modifying the tree.

Incremental compilation is implemented by a group of incremental compiling subroutines including the scanner, the scope handler, semantic analyser etc. When some syntax unit is modified by the editor the kernel environment will first set the text buffer to the proper position so that the scanner could scan symbols at the right position and then create suitable scope status. By analysing the semantics of the syntax unit the kernel environment can determine the related nodes of the syntax unit in the program tree. Finally the kernel environment calls the related incremental compiling subroutines.

5. Separate debugging of modules

Modula-2 language provides module facilities, but the original environment could only support separate compilation of modules and could not support separate debugging of modules. As a result, any module cannot be debugged unitl the related modules are all implemented.

The kernel environment can support not only traditional debugging but also separate debugging of modules.

The execution of Modula-2 module is in the way of interpreting program tree. The kernel environment uses two stacks to represent the execution status of a user program. One is the traditional procedure calling stack and the other is the statement environment stack.

Modula-2 is a kind of block language and the order among the statements is complex. For the following program segment

```
.........
I:=1 ;
T:=TRUE;
WHILE T DO
  IF I=3 THEN T:=FALSE
  END;
  I:=I+1
END;
J:=J+1;
.........
```

if the user sets a break point at the statement 'I:=I+1' then the succeed statement of it may either be the statement 'J:=J+1' or be the while statement which contains the statement itself. The statement environment stack is used to record the kind of the statement being executed and the succeed statement of it. By means of the statement environment stack the kernel environment can resume the break point correctly.

In the kernel environment the single procedure or the single module can be debugged separately.

Suppose the module to be debugged is M which imports modules $M_1, M_2, \ldots, M_n$. The user can either debug the single module M by simulating the execution environment of M or debug the modules $M+\{M_{i1}, M_{i2}, \ldots, M_{ik}\}$ where $\{M_{i1}, M_{i2}, \ldots, M_{ik}\}$ is a subset of $\{M_1, M_2, \ldots M_n\}$.

In order to improve the efficiency of debugging the kernel environment provides the interface with the codes generated by the compiler in the original environment. So the correct modules of {M1,M2,...,Mn} can be compiled by the original compiler to generate efficient codes. With the interface provided by the kernel environment the new debugger can access the data and call the procedures in those modules correctly.

The separate debugging of procedures is implemented by simulating the execution environment of the procedures.

6. Conclusions

At present we only implemented the kernel environment of the whole Modula-2 integrated environment. The kernel environment supports incremental compiling and separate debugging of Modula-2 program in an integrated way. It is written in Modula-2 itself. There are altogether above ten thousand lines of Modula-2 programs.

The software information data base has not been implemented and this is the future work we are going to do. After the data base is implemented the whole software life cycle can be supported in an integrated way. We are also going to add a configuration management system and a version management system to the environment so that the project documents can be managed efficiently.

The Formal U.S. Response to ISO on the Draft Proposal for Modula-2

The IEEE working group charged with representing the interests of the United States to WG13 of SC22 of JTC1 of the ISO/IEC requests that the United States register its vote **against** the publication of DP10514 (SC22/N738, referred to here as D106) as a Draft International Standard for the following reasons:

0. We believe that the US community would not accept the current Draft Proposal (DP) because it neither codifies established practice, nor gives adequate justification for its differences therefrom. In 1987 the U.S. committee drafted this statement.

The general philosophy of the committee and what the committee felt to be the charter for Modula-2 standardization:

1) Clarify imprecisions and contradictions in the language as we know it
2) Avoid removing language features unless necessary
3) Avoid changing language features without good reason
4) Avoid semantic changes that are not associated with syntactic changes
5) Minimize language extensions
6) 'Deprecate' obsolete features rather than removing them (i.e., have compilers accept these features while issuing warning messages, and warn users that such features may not be supported in the long term)
7) Flag dangerous programming practices, often by importing from SYSTEM.

We are disappointed that WG13 has been unable to state its goals or philosophy.

1. The document is incomplete, has many open 'to-do' items, small errors, etc. We are extremely concerned that there has been no formal (e.g. automated) verification of the VDM.

2. Although the definition of Modula-2 should be fully and separately supported both by VDM and by English text, we find that in the D106 neither method adequately defines the language. D106 uses English text too sparingly and the text is often incomplete and inadequate. The dialect of VDM used in D106 is still undefined, leaving the DP in the illogical position of providing an undefined definition of Modula-2. We cannot support the definition given by the D106 since we cannot know what the definition means. A reference to an out-of-print book [Jones80] is not a legitimate substitute for a definition, especially as that book does not describe the dialect of the VDM used in D106; neither is a reference to a moving target such as the new VDM-SL NWI. We agree that having the VDM is a good idea, provided the formalism has a complete, stable, and publicly available definition, either included in the DP or referenced by it. If not so defined, the formalism must be removed.

3. Alternative tokens §5.5. We still hold our position of August 1988 that the alternative symbols for square brackets ("(.", ".)") create unnecessary syntactic ambiguity to no positive gain. We reiterate our proposal that "(!" and "!)" be used instead.

4. Requirements clauses §4.11. The various minima suggested in §4.11 are unnecessary, technically problematical, and have the potential of delaying the standards process if adopted. We propose that all suggested minima be eliminated except for the single requirement that SET be large enough to cover CHAR.

5. Value constructors §6.7.5. There is general consensus against array and record non-constant value constructors. They add no new functionality at significant cost both to the definition and to compiler complexity. Many would be willing to undo Wirth's dynamic set value constructors, if that were necessary to get rid of array and record value constructors.

6. Comment bodies §5.8.1. We believe the intent of the language comment facility is
   a. Comments shall not change the meaning of the program, and
   b. As far as possible, one should be able to convert any piece of legal program text into comment by enclosing it in "`(* ... *)`" brackets, regardless of the presence in that text of comments, compiler directives, strings, or other constructs.

It is important that the DP clearly state this intent, even if it cannot properly be captured in a syntax specification, and that the implementor be encouraged to honor this intent.

7. Compiler Directives §5.8.2. We note several problems with compiler directives:
   a. A "$" is a national currency symbol, and is not therefore appropriate as the default directive symbol.
   b. Arbitrary white space before the "$" unnecessarily complicates scanning the directive.
   c. The nesting interaction between directives and normal comments is likely to cause problems.

We reiterate our suggestion of 1988 that "<*" and "*>" be used as bracketing symbols. It should also be stated that compiler directives can be commented out.

8. Machine Addresses in Variable Declarations §6.2.5. We request that `MACHINEADDRESS` be a record type in accordance with the proposal presented at Linz by Keith Hopper of New Zealand. In D106, `MACHINEADDRESS` is a function; since function calls cannot appear in constant expressions, this precludes the intended use of `MACHINEADDRESS` as a way to specify the address of a variable.

9. Constant Expressions §6.7.7. This section requires constant real expressions to yield the same result whether computed at compile time or at run time. Although we note that a significant subset of the potential users of Modula-2 insist on this property, it causes a severe problem on machines where real arithmetic is dynamically changeable. For example, on a machine with IEEE-754 floating point, the declaration,

```
CONST HalfPi = PI/2.0;
```

will compute a value dependent on the current rounding mode of the floating-point operations. Since this mode can be changed during program execution, the computed value cannot in general be decided at compile time. While this example could in principle be dealt with by deferring evaluation of the constant until execution time, the problem is insuperable in cases such as

```
TYPE A = ARRAY[1..TRUNC(expr)] OF INTEGER;
```

where the value is *required* at compile time but cannot then be computed. This problem should be addressed and resolved.

10. Exceptions §6.12, §7.3, §8.2, Annex G. D106 does not include a single complete model for adding an exception-handling capability to Modula-2. Since most of the proposed libraries presuppose that such a capability is defined, the failure to reach closure on this issue has an impact on both the definition and the standardization process. The evaluation criteria offered in N328 §3 appear sound to us, and in the light of these criteria we observe
   a. Any proposal meeting these criteria is likely to involve a language change too great to be admissible at this stage in the standardization process, and
   b. Any proposal not involving a language change is unlikely to offer sufficient benefit to justify its inclusion.

Accordingly, we propose that all references to exception modules be removed from the DP. If WG13 believes that one final attempt should be made to revise this facility, then we suggest that a very simple and primitive `setjmp/longjmp` model, such as D70/N247 proposal 2, if not overly embellished, may provide an approach that can yield an acceptable minimal proposal in the time available.

11. CAST ≠ bitsize §7.1.3.4. The definition of CAST should be changed so that
   a. The effect of CAST on values of ambiguous sizes (i.e.the absract types $\mathbb{Z}$, $\mathbb{R}$) is implementation-dependent,
   b. The CAST of a typed value into a type of different size is implementation-dependent, and
   c. A CAST of a value designator whose alignment is "incorrect" for the target type is implementation-dependent.

12. Lexis of Identifiers §5.3. If low-line ("_") is to be permitted as a component of an identifier, then it should be permitted wherever a letter is permitted. The lexis should not be made more complicated merely to enforce one body's view of good taste.

13. Termination §7.1.2. We reiterate the US position of 1989 as described in P155, dated 4 August 89. Termination as drafted is unnecessarily complex, provides little additional benefit over P155, and has undesirable interactions with exception handling and coroutines.

14. Forward reference pointer types. The problem exhibited by the following apparently legal code fragment should be resolved:

```
TYPE T=Q;
PROCEDURE a;
  VAR v: ^T;
  PROCEDURE b;
  BEGIN
    v^:=x; (* what code generated by one-pass compiler? *)
  END b;
  TYPE T=R;
END a;
```

15. SYSTEM module §7.1. In abstracting a storage model underlying SYSTEM, the D106 lets the abstract model pervade the definition module. Thus what should have been simple access to machine primitives has become complex operations on pieces of storage that are rarely directly supported by the native hardware. The DP should not require module SYSTEM to export constant and type identifiers other than the following:

```
CONST
  LOCSPERWORD = (* implementation defined *);
  BITSPERLOC = (* implementation defined *);
  BITSPERWORD = BITSPERLOC*LOCSPERWORD;
TYPE
  LOC; (* an opaque type equivalent to SET OF [0..BITSPERLOC-1] *)
  WORD; (* an opaque type equivalent to SET OF [0..BITSPERWORD-1] *)
  BITSET = SET OF [0..BITSPERWORD-1];
  ADDRESS = POINTER TO LOC;
  MACHINEADDRESS = RECORD (* implementation defined *) END;
    (* required by KH proposal *)
```

The identifier ADDRESSVALUE should be removed, as CAST is sufficient. Furthermore, the system function procedures SHIFT and ROTATE should operate on and return values of type WORD so defined:

```
PROCEDURE SHIFT  (value: WORD; amount: INTEGER): WORD;
PROCEDURE ROTATE (value: WORD; amount: INTEGER): WORD;
```

As decided by WG13, we request that MACHINEADDRESS be a record type to be used in fixing a hardware address value in the manner proposed by Keith Hopper (see #8 above) as follows:

```
VAR v [MACHINEADDRESS{MediumModel,0FEH,0D00DH}]: INTEGER;
```

16. String constant catenation §5.5.2.5. A different symbol is needed for string literal catenation, as "||" causes an ambiguity with case list separators; we suggest overloading the binary operator "+". There is also a problem with using the null string to denote the string termination character. The expression "'a'+''+'b'" should be the same as "'ab'" and not insert a string terminator in the middle.

17. Type transfer Annex E, p.16 (see CAST §7.1.3.4). We were unable to find in D106 the type transfer of PIM (*Programming in Modula-2* by Wirth). WG13 agreed to retain this feature but deprecate it. We suggest that, similar to NEW and DISPOSE, the old form of type transfer might require that SYSTEM.CAST be visible.

18. INTEGER and CARDINAL §6.9.1. We were unable to find in D106 the relationship between the ranges of INTEGER and CARDINAL and the bounding constants known as K1, K2, and K3.

19. WG13 agreed that all changes from PIM would be noted in the document, and we observe that many are. We require that all changes from and clarifications to PIM be noted. We also request that in each future draft, all additions, changes, and deletions from the previous draft be marked with change bars or other appropriate mechanism, to facilitate proper and efficient evaluation of the draft proposals.

20. Coroutines §7.2. While we can support moving coroutines to a separate module, we cannot support the syntactic and semantic changes made from PIM as they add no functionality and break existing code.

21. I/O Library §9.2. Noting the lack of goals or rationale, as well as the size, complexity, and novelty of the proposed I/O library, we cannot support the proposal in D106. WG13 requested a small and simple I/O library be produced. We reiterate that request. We note that D75 of August 1988, which tried to address that request, has not been allowed to mature.

22. Strings Module §9.4. The Strings module uses the look-ahead philosophy that WG13 purged from the I/O library after much debate. The current module is a radical departure from all previous implementations, complicates code, and is of no clear benefit to the programmer.

23. Module protection §6.1.11; Procedure protection Annex G. Module protection, priority, and associated syntax and semantics should be removed from the Draft Proposal because:

a. They are an extension to the language defined in PIM,
b. They assume a non-universal machine model,
c. They assume particular process and synchronization models, and
d. They will cause serious problems when multi-processor Modula-2 is designed.

24. LowReal and LowLong §8.3.1, §8.3.2. The modules LowReal and LowLong currently specify a set of constants that define various parameters of the floating-point implementation. These values become invalid if SetMode() is ever called. A much more usable definition would be:

```
TYPE
  FPInfoValues = (FPIEEE, FPISO, FPRounds, FPGUnderflow, FPException);
  FPInfo = SET OF FPInfoValues;

PROCEDURE GetFPInfo(): FPInfo;
  (* returned value is valid until next call of SetMode() *)
  (* Comment: may want to expand FPException *)
```

25. Concurrent Programming Modules §9.3. The D106 has significant flaws:
    a. It mixes event handling with basic process operations.
    b. Important operations on process queues are not in this module, namely, Lock, EnterQueue, SuspendMe, and a way to release a lock.
    c. In the Semaphores module the definition is for general semaphores, not counting semaphores (the two are distinct), and the operator set is incomplete (e.g. no indivisible VP()).

Modula-2 Users' Association
MEMBERSHIP APPLICATION

Name : _____

Affiliation : _____

Address : _____

Address : _____

City : _____

State : _____ **Postal Code:** _____ **Country:** _____

Phone : (___)____-_____ **Electronic Addr :** _____

Application as: New Member _____ or Renewal _____

Implementation(s) used : _____

Option:	___	Do NOT print my phone number in any rosters
or:	___	Print ONLY my name and country in any rosters
or:	___	Do NOT release my name on mailing lists

---

* * Membership fee per year (25 US$ or 45 SFr) * *
(Primarily pays for MODUS Quarterly publication costs.)

Members of the USA group who reside outside of
North America, please add $15.00 for air mail postage.

In North and South America, please send check or money order (drawn in US dollars) payable to Modula-2 Users' Association at:

Modula-2 Users' Association
P.O. Box 51778
Palo Alto, CA 94303-0721
United States of America

Otherwise, please send check or bank transfer (in Swiss Francs) payable to Modula-2 Users' Association at:

Modula-2 Users' Association
C/O Aline Sigrist
CH 1801 Le Mont-Pelerin
Switzerland

The Modula-2 Users' Association exists to provide a forum for communication between all parties interested in the Modula-2 Language. The primary function of the association is to publish the MODUS Quarterly. Also the association has occasionally sponsored conferences.

Modula-2 is still a new and developing language; this organization provides implementors and serious users a means to keep informed about the standardization effort, while discussing implementation ideas and peculiarities. For the recreational user, there is information on the status of the language standard, along with examples and ideas for programming in Modula-2. For everyone, there is information on current implementations and the other resources available for obtaining information on the language.

Modula-2 News Issue # 0 October 1984

Purposes, practices and promises for Modula-2 News
Revisions and Amendments to Modula-2, Niklaus Wirth
Specification of Standard Modules, Jirka Hoppe
Modula-2 in the Public Eye (a bibliography), Winsor Brown
Modus Membership list, by name
Modus members's addresses, by location
Modula-2 Implementation Questionnaire

Modula-2 News Issue # 1 January 1985

Review of Gleaves' Modula-2 text by Tom DeMarco
MODUS Paris meeting 20/21 Sep 84, C.A. Blunsdon
Report of M2 Working Group, 8 Nov 84, John Souter
Modula-2 Standard Library Rationale, Randy Bush
Modula-2 Standard Library Definition Modules
Modula-2 Standard Library Documentation, Jon Bondy
Validation of M2 Language Implementations, J. Siegel

MODUS Quarterly # 2 April 1985

Letters, Anderson & Emerson
Opaque Types in Modula-2, C. French & R. Mitchell
Dynamic Module Instantiation, Roger Sumner
The Linking Process in Modula-2, Jeanette Symons
Modula-2 Library Comments, Bob Peterson
Modula Compilers - Where to Get 'em, Larry Smith
Coding War Games Prospectus, Tom DeMarco
M2, An Alternative to C, M. Djavaheri, S. Osborne

MODUS Quarterly # 3 July 1985

Letters, Endicott & Hoffman
Some Thoughts on Modula-2 in "Real Time", Paul Barrow
RajaInOut: simple, safer, I/O for
Logitech/MS-DOS, R. Thiagarajan
Selection of Contentious Problems, Barry Cornelius
Expressions in Modula-2, Brian Wichmann
The Scope Problems Caused by Modules, Barry Cornelius

MODUS Quarterly # 4 November 1985

State of MODUS, George Symons
MODUS Meeting Report, Bob Peterson
A Writer's View of a Programmer's Conference, Sam'l Bassett
Concerns of A programmer, Dennis Cohen
Modifications to the Standard Library
Proposal, R. Nagler & J. Siegel
Proposal, standard library and M2 extension,
Odersky, Sollich, & Weisert
Standard Library of the Unix OS, Morris Djavaheri
The Standard Library for PC's, E. Verhulst
Editorial, Richard Karpinski
Modula-2 Compilation and Beyond, D.G. Foster
Modula-2 Processes - Problems and Suggestions, Roger Henery

MODUS Quarterly # 5 February 1986

Editorial, Richard Karpinski
Exporting a Module Identifier, Barry Cornelius
Letter on multi dimensional open arrays, Niklaus Wirth
Letter on DIV, MOD, /, and REM, Niklaus Wirth
BSI Accepted Change: Multi-dim. open arrays, Willy Steiger
N73: NULL-terminated strings in Modula-2, Ole Poulsen
ISO Ballot Results re BSI Specifying Modula-2
Draft BSI Standard I/O Library for Modula-2, Susan Eisenbach
Portable Language Implementation Project: Design and
Development Rationale, K. Hopper and W.J. Rogers
The ETH-Zuerich Modula-2 for the Macintosh, Chris Jewell
NewStudio: Engineering a Modula-2 Application for the Mac,
A. Davidson, H.B. Herrmann, E.R. Hoffer

MODUS Quarterly # 6 November 1986

Editorial, Richard Karpinski
Letter on opaque types, File type, and SET OF CHAR, P. Williams
Letter on exported identifiers, E. Videki
Why the Plain Vanilla Linkers, J. Gough
Letter re best article & MacModula-2, M. Coren
Significant Changes to the Language Modula-2, Barry Cornelius
All About Strings, Barry Cornelius
Type Conversions in Modula-2, B. Wichmann
Improving the quality of Definition Modules, A. Sale
A Programming Environment for Modula-2, F. Odegard
Academic Modula-2 Survey, L. Mazlack
Compilers for Modula-2 (Zuerich list)
Membership List

MODUS Quarterly # 7 February 1987

Editorial, Richard Karpinski
New Products
Modula-2 Standardisation: A go betweens tale, Welsh & Bailes
Modula-2 VM/CMS, Thomas Habernoll
TCP Implementation in Modula-2, F. Ma & L. D. Wittie
Building an Operating System with Modula-2,
B. Justice, S. Osborne, & V. Wills
Note on Implementing SET OF CHAR, Source Code
for a SetOfChar MODULE, A. Brunnschweiler

MODUS Quarterly # 8 May 1987

Editorial, Richard Karpinski
Letter re unwarranted BSI changes, T. DeMarco
Response to DeMarco letter, R. Karpinski
Letter re standards questions, A. R. Spitzer
Open Letter from a Practicing Programmer, W. Nicholls
Coroutines and Processes, R. Henery
Another look at the FOR statement, B. Cornelius
Automatic export of identifiers from the definition module,
A. H. J. Sale
BSI Modula-2 Working Group Standard Concurrent Programming
Facilities, D. Ward

MODUS Quarterly # 9 October 1987 (July 1988)

Editorial, K. N. King
Letter re Linking and Overlays, A. Layman
Letter re BSI Language Changes, J. Savit
Letter re Thoughts on Modula-2, T. Pittman
Letter re Modula-2 and FORTRAN, C. Tanzer
MODUS Conference 1987, Program and Abstracts
Problems with the Definitions of ORD and VAL, B. Cornelius
Proposed BSI Standard Modula-2 I/O Library
Will Modula-2 be Sucessful? NO!, J. Lancaster
Modula-2 Use in Urban Transportation Vital Control Systems,
R. Lardennois
A Dhrystone Benchmark for PClones, A. Gurski

The above back issues are still in print. MODUS Administrators supply single copies at $7 US or 12 Swiss Francs.

Modula-2 Users' Association
DP10514 ORDER FORM

This is an order for the the latest copy of the Draft Proposed Standard for Modula-2. At this time we expect the next draft (2nd) to be published in late 1990. If you make no indication on this form, you will be sent the current version of the draft.

COST: US $30.00 (Includes postage within North America.)

Add US $20.00 for Air Mail outside of North America.

Mail this form with payment to:

MODUS
P.O. Box 51778
Palo Alto, CA 94303-0721
USA

I have enclosed $_____ for _____ copies.

Name : _____

Affiliation : _____

Address : _____

Address : _____

City : _____

State : _____ **Postal Code:** _____ **Country:** _____

Phone : (___)____-_____ **Electronic Addr :** _____

90.8.5 mq

TO ORDER DP10514 FROM IEEE COMPUTER SOCIETY

COST: $35.00

Mail or Fax this form to:

IEEE Computer Society
Standards Office
1730 Massachusetts Avenue NW
Washington, DC 20036
Attn: Lisa Granoien
Fax (preferred): +1 (201) 562-1571

Order ISO/JTC1/SC22/WG13 Draft of DP10514 - P1151 Modula-2

Name: _____

Company: _____

Address: _____

_____

City/State: _____

Phone: _____

I have enclosed $_____ for _____ copies
(make checks payable to IEEE Computer Society P1151)

OR

Please charge ___ Visa ___ M/C ___ AmEx

Number: _____

Expires: _____

Bank No: _____

Signature: _____

89.12.17 rb

Modula-2 Users' Association
MEMBERSHIP APPLICATION

Name : _____

Affiliation : _____

Address : _____

Address : _____

City : _____

State : _____ **Postal Code:** _____ **Country:** _____

Phone : (___)____-_____ **Electronic Addr :** _____

Application as: New Member _____ or Renewal _____

Implementation(s) used : _____

Option: ___ Do NOT print my phone number in any rosters
or: ___ Print ONLY my name and country in any rosters
or: ___ Do NOT release my name on mailing lists

** Membership fee per year (25 US$ or 45 SFr) **
(Primarily pays for MODUS Quarterly publication costs.)

Members of the USA group who reside outside of North America, please add $15.00 for air mail postage.

In North and South America, please send check or money order (drawn in US dollars) payable to Modula-2 Users' Association at:

Modula-2 Users' Association
P.O. Box 51778
Palo Alto, CA 94303-0721
United States of America

Otherwise, please send check or bank transfer (in Swiss Francs) payable to Modula-2 Users' Association at:

Modula-2 Users' Association
C/O Aline Sigrist
CH 1801 Le Mont-Pelerin
Switzerland

The Modula-2 Users' Association exists to provide a forum for communication between all parties interested in the Modula-2 Language. The primary function of the association is to publish the MODUS Quarterly. Also the association has occasionally sponsored conferences.

Modula-2 is still a new and developing language; this organization provides implementors and serious users a means to keep informed about the standardization effort, while discussing implementation ideas and peculiarities. For the recreational user, there is information on the status of the language standard, along with examples and ideas for programming in Modula-2. For everyone, there is information on current implementations and the other resources available for obtaining information on the language.

MODUS
P.O. Box 51778
Palo Alto, CA 94303
U.S.A.

MODUS Europe
c/o Aline Sigrist
CH-1801 Le Mont-Pèlerin
Switzerland

Return Postage Guaranteed