Guest Editor's Introduction

Modula-2 Experiments Will Help Future Language Designs

Robert P. Cook, University of Virginia

This special issue of IEEE Software focuses on the future of programming rather than on past accomplishments or arguments about the superiority of one language over another. Because of its small size and the widespread dissemination of its compilers, Modula-2 has been adopted as the foundation for a number of experimental systems. The insights gained from those experiments will prove valuable to both current designers and future users of programming systems, no matter what the language.

This issue includes discussions of a range of problem areas — including programming environments, portability, language design, Prolog, and interface encoding techniques — that transcend the interests of only the Modula-2 community. The articles were selected for their technical contributions and because they serve as indicators for the future of programming.

Modula-2 is a beginning, not an end. To emphasize that, consider three questions whose answers may determine the directions of language design into the 21st century.

(1) Will it ever be feasible to reuse existing code (procedures and modules) in new programs, and, if so, to what extent will composition techniques replace traditional programming?

(2) Will interface specification languages eventually supersedes implementation languages in importance?

(3) How much progress has been made in the last 25 years on resolving the notation versus mechanism controversy?

Code reusability. The issue here is how to control the exploding cost of software production. New software is developed in response to new requirements, to technological advances, and to the development of better algorithms.

All software is a hierarchy of modules, with physical devices forming the lowest layer and application programs forming the highest layer. Obviously, if an innovation at any layer required redeveloping the module hierarchy above and/or below that layer, our ability to introduce new technology would be stifled by the prohibitive cost of producing the software.

As the number of modules grows and the hierarchy expands, it is natural to hypothesize that new programs may be largely developed by composing existing programs. Some of the problems encountered in trying to reach that goal, as the article by Nazim Madhavji and colleagues points out, are keeping track of the database of existing modules, or module fragments, and providing a support environment for the composition process.

If this hypothesis is true, it may radically change the way programs are created in the future. However, it also holds the promise of a dramatic reduction in software production costs. Experience with operating systems that support composition operators applied to single-function modules hints that this might be the trend.
Interface specifications. Consider the following interface specification and ask yourself the questions “What is the effect of changing the interface versus changing the implementation?” and “Does it matter what the implementation language is?”

DEFINITION MODULE FamiliarOperatingSystem;
TYPE FileDescriptors = ARRAY [0..1] OF CARDINAL;
LongCharArray = ARRAY [0..9999] OF CHAR;
Path = POINTER TO LongCharArray;
PROCEDURE Fork(): INTEGER;
PROCEDURE Kill(processId, signal : INTEGER) : INTEGER;
PROCEDURE Link(to, from : Path): INTEGER;
PROCEDURE Pipe(VAR inAndOut: FileDescriptors): INTEGER;
END FamiliarOperatingSystem.

First, the effects of changing an operating system's interface can be disastrous even though its implementation may be changed (hopefully correctly) hundreds of times as it is ported to new architectures or as improvements are made.

Second, the implementation language is irrelevant; anything that works will suffice. The conclusions are inescapable: interfaces have long lifetimes relative to languages and languages have long lifetimes relative to implementations.

To increase the utility of an operating system's interface, there are two possibilities.

First, one can define an interface that is portable across a wide range of existing operating systems. This is the approach taken in the OSSI article by Edoardo Biagioni and his coauthors. The result is that programs developed above the level of OSSI in the hierarchy remain immune to the effects of changes in operating system technology.

The second possibility is to improve operating system technology to the point that existing operating systems can be efficiently emulated by a common kernel.

Interface specification languages are in their infancy. For example, there have been a number of complaints about the lack of a Write statement with a variable-length argument list in Modula-2. The defect occurs because of the lack of flexibility in interface specification. One conclusion is that more powerful interface-specification languages will reduce the amount of new programming in existing languages, since more powerful interfaces will complement composition techniques.

The article by Jurg Gutknecht presents an encoding scheme for interface specifications that is extendable in the sense that it is designed to support future extensions to Modula-2 and that it can be used for languages other than Modula-2. An encoding standard for interfaces is important because it allows interfaces in different languages to be compared for compatibility.

Furthermore, it is possible to encode interfaces for languages that do not support definition modules, thus widening the module hierarchy. For example, interfaces for existing Fortran programs could be generated automatically. Interface and program encodings are essential to the development of symbolic debuggers and program development environments.

Notation versus mechanism. Modula-2 is unique in that it represents a contraction over Pascal. In fact, many people are unaware that it has shrunk yet again. With, together with participants from several firms that had implemented Modula-2, issued a revisions and amendments document in 1983. For example, the requirement for an Export list in a definition module as well as the process type for concurrent programming were deleted.

The notation used by a programming language is commonly described by its syntax, expressed in Backus-Naur form. The effects of using a notation are defined by its semantic description. The semantics for a notation may involve compile-time checking, a translation or mapping, and the use of runtime mechanisms. The following example of an accept statement in a message-passing system illustrates the difference between notation and mechanism.
Notation:

```
ACCEPT Put(c : CHAR);
    (*delay until a message arrives*)
    buffer[iBuf] := c;
    (*add c to tail of circular queue*)
END Put;
    (*at this point, caller continues*)
iBuf := (iBuf MOD QSIZE)+1;
```

Translation:

```
VAR putPORT : RECORD
    p : PORT;
    message : MessageType;
END; (*putPort*)

Receive(putPort.p)
    (*delay until a message arrives*)
    buffer[iBuf] := putPort.message.c;
    (*add c to tail of circular queue*)
RendezvousOver(putPort.p);
    (*at this point, caller continues*)
iBuf := (iBuf MOD QSIZE)+1;
```

Runtime mechanism:

```
DEFINITION MODULE Ports;
    (*implements the message-passing semantics*)
TYPE Port;
PROCEDURE InitPort(VAR p : Port);
PROCEDURE Send(VAR p : Port);
PROCEDURE Receive (VAR p : Port);
PROCEDURE RendezvousOver (VAR p : Port);
END Ports.
```

In criticizing the notation that Modula-2 lacks compared to other languages, one should weigh the benefits of the notation against what it costs to provide it. For any given mechanism, there are many equally likely notations to make that mechanism accessible to programmers.

During the 1970's, for example, there were literally hundreds of papers on concurrent programming languages. During the 1980's, there have been at least that many papers on distributed programming languages. But for all the papers generated in the 1970's, not a single one of the concurrent programming notations is commonly used.

However, many of the mechanisms, such as signals and message passing, are widespread. Wirth has demonstrated that it is simple to construct modules that provide coroutine, signals, and message-passing services.

Most existing mechanisms can be made available to a Modula-2 programmer either by using Gutknecht's encoding scheme to create an interface for assembly language code or by using code procedures to generate in-line machine code. Interfaces to programs in other languages can also be generated by the same techniques.

Going one step further, the article by Carlo Muller discusses how to access the mechanisms of Prolog from Modula-2. The key is the construction of an appropriate interface. It would seem to be equally feasible to provide interfaces for the mechanisms of other languages, such as Lisp, Snobol, and APL.

The article by Paul Rovner proposes new extensions to the Modula-2 language and presents some of the arguments for their inclusion. Some of my criteria are:

---

**Modula-2 is a beginning, not an end. It raises three questions that could affect language-design directions into the 21st century.**

---

- It has been said that language structures thought. To what extent does the new notation affect your ability to solve a problem or to express its solution in a concise and understandable manner?
- Is the notation more concise than invoking the translated mechanism via procedure calls?
- Does the compiler provide error-checking that would not be available with the mechanism?
- What is the benefit of having all programmers use the same notation? (This can also be achieved by standardizing interfaces.)
- What is the cost of increasing the compiler's complexity?
- Does the notation improve the ability to maintain the code?
- Does the notation decrease the probability of coding errors?
- Does the notation make the resulting code more portable?
- What is the notation's likely frequency of use? (Remember call-by-name in Algol-60? Elegant but unused.)

The set of available mechanisms may also grow faster than our ability to invent useful notations and have them be accepted into common usage. The challenge for the future is to create programming environments that support notation extendability. Some progress has already been made toward this goal with the advent of structured editors and advances in compiler technology, but much work remains.

---

Acknowledgments

The introduction to this special issue on Modula-2 would be incomplete without thanking the authors of the submitted articles and the 70 referees who deserve the credit for this special issue. I also appreciate the efforts of Editor-in-Chief Bruce Shriver, who put in as much work as I did, and probably more. A thank you is also in order for the Turing award scientist Niklaus Wirth, who made all this possible. Thank you, Prof. Wirth. Now what's next?

References


---

Robert P. Cook is an associate professor of computer science at the University of Virginia, where his research interests include distributed real-time systems, high-level language architecture, and programming environments. Before joining the university in 1983, he was an assistant professor of computer science at the University of Wisconsin.

Cook received BE, MS, and PhD degrees in computer science from Vanderbilt University. He is a member of the IEEE Computer Society and the ACM.

His address is Dept. of Computer Science, Thornton Hall, University of Virginia, Charlottesville, VA 22903.