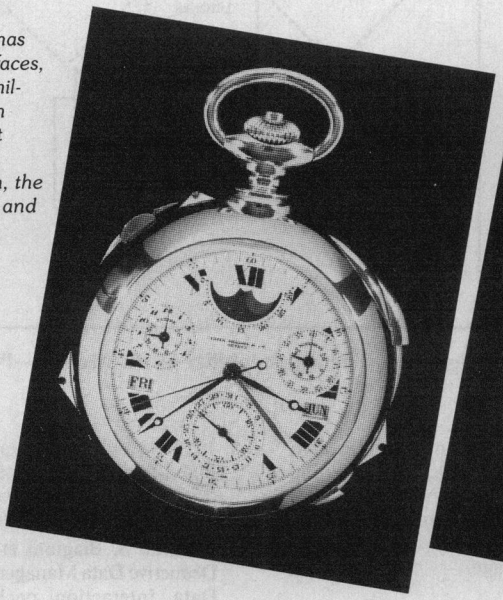
Extending Modula-2 to Build Large, Integrated Systems

Watches now can talk, sing, beep, and calculate (far right). But the most complicated watch ever made is the Patek Philippe "superwatch." Crafted in 1932, it has 870 parts and two faces, and is worth $1.5 million. The ultimate in multifunctionality, it tracks the time, the phases of the moon, the sunrise and sunset, and the heavens above.



The Time Museum

The Time Museum

Paul Rovner, DEC Systems Research Center

Designed to build both system and application software for large systems, Modula-2+ supports exception handling, automatic storage management, and concurrency for multiprocessors without compromising the integrity of Modula-2.

A team at DEC's Systems Research Center is developing software for Firefly, a new personal workstation. Firefly, a multiprocessor computer with shared memory, is being used to build large applications that share data structures and code. It will require powerful programming tools to support several people working in cooperation.

Naturally, this requirement affects our choice of a programming language. We need a single language to develop system software and explore prototype applications on the Firefly. In addition, we need a language that reduces the cost of producing reliable, high-quality software in the domains of programming systems, distributed systems, personal workstations, and related applications.

We chose Modula-2, even though it had limitations and required extensions to provide the full range of capabilities we needed. We called the extended language Modula-2+. This article describes both the limitations of Modula-2 for our task and the changes we made to overcome these limitations. We present the collection of language changes and show how they are adequate for our needs. Implementation details, however, are outside the scope of this article.

Why Modula-2?

We studied the pros and cons of Ada,[1] Concurrent Euclid,[2] and CLU,[3] in addition to Modula-2. We rejected Ada because it is too large and complex. We rejected Concurrent Euclid because, in the areas of primary interest to us — interfaces

0740-7459/86/1100/0046/$01.00 ©1986 IEEE

and procedure values — it is dominated by Modula-2. The choice between CLU and Modula-2 was more difficult. In the end, performance concerns prevailed. CLU's semantics are similar to Lisp, hence we could expect to pay many of the same runtime costs as for Lisp programs. Modula-2, on the other hand, is more like C, hence we could expect to produce code that is as good as the code produced for C.

Modula-2, therefore, came closest to meeting our needs, striking a balance between simplicity and functionality. We had both pragmatic and technical considerations, of course.

Pragmatically, the widespread and growing interest in Modula-2 makes it easier to collaborate with others on problems of mutual interest. The availability of a new Modula-2 compiler written by Mike Powell at the DEC Western Research Laboratory also influenced our decision.[4] Technically, Modula-2 is relatively small, simple, and easy to work with. It also provides a well-integrated combination of features essential to the development of large systems. Modula-2

- enables strong typing with static checking
- separates the specification of an abstraction from its implementation
- supports systems programming via low-level facilities
- supports concurrency, and
- provides procedure-valued variables.

Although these features make it attractive, Modula-2 does not provide additional features in three important areas: exception handling, automatic storage management, and concurrency for a true multiprocessor.[5] Because it has few constraints in these areas, however, we were able to provide extensions without compromising the integrity of the language.

Language design, especially when done by accretion, is difficult. Its simplicity makes Modula-2 remarkably seductive (provocative, even) to those inclined to improve and extend their programming tools. It is easy to get sidetracked into adding ill-advised features, easy to overestimate the value of a feature that is simple to implement, easy to underestimate the impact of a change on the integrity and overall complexity of the original design. But it is difficult to distinguish high-payoff features from low-payoff ones without trying them out, sometimes at great expense. There is no substitute for experience.

Based on our experience and mindful of the dangers, we designed and implemented language extensions to Modula-2. The impact of the design extensions on the core of Modula-2 is small. It was our goal to stay compatible with the underlying spirit of Modula-2, as described in Wirth.[6] For more than 18 months, a dozen programmers have used the Modula-2+ programming system, running on both Ultrix and on the Firefly, to build the Firefly software.

Based on our experience with similar systems,[7,8] we do not anticipate big surprises from the use of Modula-2+. On the other hand, we have used it ourselves for only a short time and are well aware that the actual use of a complex system is often an educational (sometimes humbling) experience for its designers. For example, we have found it worthwhile over the past year to make more extensions in the areas of interfaces and opaque types than originally planned. The emphasis in the design described here has more to do with our experience on similar systems, not Modula-2+, but this will change with time.

It was our goal to stay compatible with the underlying spirit of Modula-2, as described by Wirth.

Exception handling

Facilities for exception handling make it easier to specify a program more completely, and provide standard methods for solving a common class of programming problems. These tools are particularly important in large programs. The Modula-2+ exception handling facilities provide standard methods for dealing with these common programming problems:

(1) How to report the failure of the implementation of an abstraction that cannot reasonably be implemented in its full generality. For example, limited resources or an inadequate algorithm may cause insufficient storage or numerical overflow errors.

(2) How to report that the parameters the client supplied do not conform to the invariants the package requires.

(3) How to gracefully back out of a computational path that fails for some reason, before trying another or moving on. An example is a simple, recursive-descent parser or pretty printer that is called from an interactive read/eval/print loop. The program must parse a given string until the string is found to be either acceptable or ill-formed. If the string is defective, it is desirable to announce failure and return to the top-level control loop.

Another example of the third problem is a system of experimental programs being developed concurrently by a group of people. Valuable progress can be made if one new package is temporarily bullet-proofed against minor problems created by another new, slightly buggy package.

Goals. How a program behaves in unusual situations is an essential part of its specification. Clear specification is most naturally achieved by outlining expected behavior separately from the list of the problems or unusual cases that might arise. Explicit provision in the language for decomposing a program into a normal case part and an exception handler for the exceptional cases improves predictability, robustness, and reliability.

A component can almost always be expressed without including explicit code at each procedure call to deal with exceptions. Inserting a check at each call is not only awkward and inefficient, but if, by mistake, the check is not made, the exceptional result will go unnoticed.

It is more natural to associate an exception handler with a sequence of statements and to let the handler deal with exceptions that arise from any one of the statements. Language support for exception handling is a natural way to program for the normal and exceptional cases independently — code that deals with exceptional cases is outside the normal flow of control.

A related, common programming idiom is a natural temporal framework: first initialize, then execute the main body, then

A semantic model for exception handling and finalization

Exception handling in Modula-2+ is based on a semantic model similar to CLU and Ada. The choice of exception-handling semantics for Modula-2+ was also influenced by our experiences with Mesa, though the Mesa design is based on a quite different model.[11]

A brief sketch of the Modula-2+ exception-handling semantics will clarify the meaning of the language extensions. Our semantic model is a conventional operational one in which a (Modula-2+) computation consists of the sequential evaluation of the expressions and statements (phrases) in a program, according to a precise set of rules. Only the rules relevant to exception-handling are sketched.

Let the evaluation of each expression in the language yield either a value (of the expression's type) or an exception. Let the evaluation of each statement return either a value denoting normal completion or an exception. If evaluation of a phrase yields a normal value (not an exception), then the computation proceeds normally. If evaluation yields an exception, then evaluation of the enclosing phrase terminates, yielding the exception as its value, unless the enclosing phrase is a TRY statement, an EXIT statement, or a RETURN statement.

In a TRY statement, if a handler for the exception is given, the handler is evaluated and its result is taken as the result of the TRY statement. If there is no handler for the exception, the exception is taken as the result of the TRY statement.

Applying the standard procedure RAISE is the only way to explicitly yield an exception. The parameters to RAISE are an exception name and its argument, if any.

Except for the two special exceptions, exitRaised and returnRaised, the semantics of EXIT, RETURN, and TRY ... FINALLY can be described with the rewrite rules below. LOOP statements and procedure bodies are also rewritten.

The special exception exitRaised is used only in the rewrite rules for the LOOP, EXIT, and TRY ... FINALLY statements, is not handled by ELSE, and cannot be used explicitly by client programs. Similarly, returnRaised is used only in the rewrite rules for procedure bodies and the RETURN and TRY ... FINALLY statements, is not handled by ELSE, and cannot be used explicitly by client programs.

Notation: < > encloses a syntactic class of some sort; [] encloses an optional term.

Rewrite rules

1. LOOP rewrite.

```
LOOP (* Modula-2+ LOOP semantics *)
  <statements>
END;
```

means

```
TRY
  LOOP
    (* Modula-2 LOOP semantics; don't rewrite recursively *)
    <statements>
  END;
EXCEPT
  exitRaised:
END;
```

2. EXIT means RAISE(exitRaised).

3. Procedure body rewrite.

```
BEGIN (* Modula-2+ procedure body semantics *)
  <statements>
END
```

means

```
[ VAR v: <result type of proc, if any>;]
EXCEPTION returnRaised [ (<result type of proc, if any>) ] ;
BEGIN (* Don't rewrite this recursively *)
  TRY
    <statements>
  EXCEPT
    returnRaised [ (v) ] :
  Terminate the call on the procedure normally
    [ , yielding the value of v ]
  END;
END;
```

4. RETURN [e] means RAISE(returnRaised [, e])

5. The semantics of the TRY ... FINALLY statement can be described in terms of the TRY ... EXCEPT statement by rewriting it as follows:

```
TRY <statements-1> FINALLY <statements-2> END;
```

means

```
TRY <statements-1> EXCEPT
| exitRaised : <statements-2>; RAISE(exitRaised);
| returnRaised [ (v) ] : <statements-2>; RAISE(returnRaised [, v ]);
ELSE(e)
  <statements-2>;
  RAISE(e.exception, e.argument);
    (* This re-raises the exception represented by e *)
END;
<statements-2>;
```

finalize. A concurrent example might be: first acquire a lock, then examine or change a shared data structure, then release the lock. A tiresome and often overlooked programming problem arises if the main body exits abnormally (via EXIT, RETURN, or exceptional result). But finalization must happen in any case. Coding your way out of such a mess is awkward and error-prone, so language support for this control structure is important. Our finalization feature turned out to be a simple extension of the exception-handling mechanism.

Good debugging facilities are also crucial to the rapid development of large experimental systems. With exception handling, programming bugs and oversights manifest themselves as unhandled exceptions. We needed an implementation that would recognize unhandled exceptions and pass them to the debugger without destroying the context in which the exception was raised. In an environment that isn't equipped with a debugger, the problem of what to do when a program raises an unhandled exception is really the problem of what to do when a program encounters a bug and crashes.

Finally, and most important, our design had to be implemented at reasonable cost and with insignificant runtime overhead for normal execution. In particular, we required the execution cost of opening and closing handler and finalization scopes to be negligible. A satisfactory implementation is in place but is not discussed in this article.

Language forms. Exception handling in Modula-2+ is based on a semantic model similar to the ones for CLU and Ada. See the box above for a formal description of the Modula-2+ semantic model. Exceptions are declared by specifying a name and at most one parameter:

```
EXCEPTION Overflow;
EXCEPTION InvalidCharacter(CHAR);
```

The parameter can be any type that is allowed as the result of a function procedure and that has a similar meaning, hence the restriction of a single parameter. The design would be simpler if no parameters were allowed, but our experience with Mesa and Cedar convinced us that we needed to specify at least one parameter. A similar argument can be made to justify why Modula-2 allows only one result of a function procedure.

Exceptions cannot be assigned to variables or passed as parameters, except as the first argument of RAISE, as explained later in this article.

Exception names obey normal scope rules and are treated like constants. For example, an exception declared in a recursive procedure has the same definition at each invocation. The compiler assigns a unique internal code to each exception declaration, used to identify the exception at runtime.

RAISE and TRY. An exception is raised via a new standard procedure, RAISE:

```
RAISE(exception, argument)
```

if the exception is takes a parameter, or

```
RAISE(exception)
```

otherwise.

Exceptions propagate upward through nested dynamic contexts, looking for a handler. An exception is handled by

```
TRY
  try-body
EXCEPT
| exception-1(variable-name-1):
  handler-body-1
| exception-2(variable-name-2):
  handler-body-2
| ELSE(variable-name-n) handler-body-n
END;
```

The TRY ... EXCEPT statement identifies a statement sequence (the try body) and a list of handlers for exceptions that might be raised by its execution. The code following EXCEPT is similar to a CASE statement.

If, during the execution of the try body, one of the listed exceptions is raised, execution ceases, and control passes to the corresponding handler body (a statement sequence). If the exception raised does not match any listed, and an ELSE clause is present, the ELSE handler body receives control. If no ELSE clause is present, a handler is sought in the statically enclosing context (a TRY statement may be nested in a try body). If no handler is found there, the search continues in the context of the calling procedure. If no handler can be found, the computation is suspended and the debugger receives control.

Before a handler body is executed, all dynamic contexts between the raiser and the try body (inclusive) are finalized: the stack is unwound, register values are restored, and explicit finalization actions are invoked. When the handler body has finished executing, control passes to the statement following the TRY construct, exactly where it would have gone had the try body not encountered an exception.

The economics of garbage collectors is improving rapidly, as memory and processors become cheaper.

Handler variables are optional. If the exception takes an argument and the handler needs it, the exception name in the handler arm is followed by the name of a variable. The exception parameter must be assignable to this variable. The value that RAISE passes as the exception parameter is assigned to the variable when the handler body is invoked. If the handler does not need the exception parameter, the variable is omitted, in which case the exception parameter (if any) is discarded.

Our decision to require predeclared handler variables was influenced by the design of the FOR statement, which requires predeclared index variables. We decided that a design that introduced a new name and name scope as formal handler parameters was too radical and had too few advantages to justify.

By convention, the only exceptions handled by ELSE are those caused by programming errors. Low-level system failures or other catastrophes (such as memory parity errors or device failures) require different treatment. ELSE handlers usually do not appear in application programs, though they are sometimes used to bullet-proof experimental code. Instead, they are typically used in code that, for some reason, does not want the debugger to field errors (a robust read/eval/print loop, for example). ELSE handlers must be used with care; thoughtless use can mask programming errors.

The parameter type of an ELSE handler is SYSTEM.FailArg, a new standard type that represents an exception and its argument, if any. If the parameter is not needed by the ELSE handler, the variable is omitted, in which case the identity of the exception and the value of its parameter will be lost. The SYSTEM module does provide operations to examine the components of a FailArg, but they are not described in this article.

Procedure types. To provide clients with reliable documentation of the exceptions that can be raised, procedure headings list them in a RAISES clause:

```
PROCEDURE P (actuals): ret
    RAISES {ex-1, ex-2};
  decls
  BEGIN
    body
  END P;
```

At runtime, this behaves in the same way as

```
PROCEDURE P(actuals): ret;
  decls
  VAR variable-1:
        <parameter type of ex-1>;
      variable-2:
        <parameter type of ex-2>;
    vFArg: SYSTEM.FailArg;
  BEGIN
  TRY
    body
  EXCEPT
  | ex-1(variable-1):
    RAISE(ex-1,variable-1);
  | ex-2(variable-2):
    RAISE(ex-2,variable-2);
  | SYSTEM.Fail(vFArg):
    RAISE(SYSTEM.Fail,vFArg);
  | ELSE(vFArg)
    RAISE(SYSTEM.Fail,vFArg);
  END;
  END P;
```

Thus, the RAISES clause affects both the procedure type and the body — using it in definition modules is a valuable specification technique. Each RAISES clause implicitly includes SYSTEM.Fail, a standard

exception that takes a SYSTEM.FailArg as its parameter. An empty list means that only SYSTEM.Fail can be raised.

The RAISES clause is optional. If no RAISES clause appears, any exception may be raised by the procedure. In this respect Modula-2+ differs from CLU, which automatically converts an exception not explicitly mentioned to its version of SYSTEM.Fail. We judged this aspect of CLU to be too restrictive for our purposes.

By virtue of the ELSE handler, all other exceptions are handled by converting them to SYSTEM.Fail. By convention, SYSTEM.Fail is used to report programming errors. Application programs normally do not handle SYSTEM.Fail. Instead, SYSTEM.Fail is intercepted by a debugging facility in a way that preserves the context in which the exception propagated. This is a special feature of the implementation, not of the language, and is not part of the semantics of exceptions.

Finalization. This feature is a simple extension of the exception-handling mechanism:

```
TRY
  try-body
FINALLY
  cleanup-body
END;
```

The try body executes, then the cleanup body. The try body can be terminated normally, or as a side effect of an EXIT or RETURN statement, or by raising an exception. For example:

```
TRY
  s1;
  TRY s2; s3 FINALLY s4; s5 END;
  s6
EXCEPT
  e: s7
END;
```

If no exceptions are raised, the sequence of execution is *s1, s2, s3, s4, s5,* and *s6*. If exception *e* is raised in *s1*, the sequence is *s1* (partially), *s7*. If exception *e* is raised in *s2*, the sequence is *s1, s2* (partially), *s4, s5,* and *s7*.

If finalization occurs as the result of an exception, and the cleanup body raises an exception, the original exception is lost. The meaning of this construct can be expressed in terms of the other exception handling primitives, as shown by rewrite rule 5 in the box on p. 48.

Automatic storage management

Managing storage is primarily an exercise in bookkeeping. Automating this drudgework through garbage collection has a high payoff, both in the reliability of the product and in freeing the designer/programmer to deal with more important details.

Storage safety, the guarantee that memory will not be corrupted by errant code, is fundamental to correct programs. By definition, a safe program will not violate this guarantee. This means that a safe program will not alter storage to which it does not have proper access (accessing beyond the end of an array and storing through an invalid pointer are two examples). A safe but buggy program might get the wrong answer, but it will not cause an independent program that shares the same address space to do so.

Else handlers must be used with care: thoughtless use can mask programming errors.

Garbage collection is a class of techniques that automatically detects storage no longer accessible and returns it to the pool of free storage. Storage safety and garbage collection are interdependent: Programs with garbage collectors need not employ dangerous, low-level primitives to manage storage and hence are a lot less likely to have bugs that corrupt memory. And garbage collection works only for programs that do not have such bugs.

Experience with Lisp, Smalltalk, Mesa, and Cedar[8,9] has clearly shown that programming tools for the rapid development of large programs[5] that guarantee safe storage have a great advantage over ones that do not. Yet use of garbage collection with languages like Modula-2 is not widespread, because it is considered too expensive in terms of system complexity, memory space, and processor cycles.

However, the economics of garbage collectors is improving rapidly, as memory and processors become cheaper. Also, garbage collection is inherently parallel, so a concurrent machine like the Firefly is a particularly attractive opportunity to decrease these costs still further.

Safe features. Our goal was to provide a safe subset of Modula-2 that was sufficiently powerful to handle most programs conveniently and efficiently. First we identified those Modula-2 language features and library procedures that, if misused, could corrupt memory. If a given program uses none of these, it is safe. Next we provided, where feasible, a safe alternative for each unsafe facility. Simple static analysis, augmented by efficient runtime checks, suffices to guarantee a program is safe.

Of course, some programs — such as a storage allocator or a garbage collector — cannot be shown to be safe in this way. Furthermore, our restrictions are inappropriate for other programs, including those that require access to low-level facilities or have stringent performance requirements.

Well-behaved REFs. The use of POINTERS is restricted in safe programs, so we added a well-behaved form of POINTER, REF. REFs behave much like POINTERS, except that the storage they address is never explicitly freed by the programmer. REF can nearly always be used instead of POINTER, and is declared:

```
TYPE Node = REF NodeRep;
TYPE NodeRep = RECORD
  next, prev: Node;
  (* other fields *)
END;
```

Like POINTERS, REFs are created by NEW. That is, the first parameter to NEW may be a variable of type REF *T*. The standard constant NIL is assignable to a REF. REFs may be embedded in RECORDS (as above) and ARRAYS. Such structures are called REF-containing. Every REF variable is initialized to NIL.

SAFE modules. Safety checking is enabled in implementation modules with the keyword SAFE:

A concurrency example.

This small program illustrates how concurrency constructs are used to solve a classic problem. It defines two threads of control: a producer and a consumer. The producer generates values; the consumer uses them. The two threads communicate via a shared buffer of limited size, managed as a circular buffer. Only one thread at a time may access the buffer, and each thread must synchronize its activities with the other.

```
SAFE MODULE ProCon; (* Producer/Consumer *)
IMPORT Thread;

CONST BufferMax = 10;

TYPE
  BufferIndex = [0..BufferMax-1];
  BufferObject = RECORD
    mutex: Thread.Mutex; (* for mutual exclusion *)
      nonFull, nonEmpty: Thread.Condition; (* for synchronization *)
    in, out: BufferIndex;
    contents: ARRAY BufferIndex OF INTEGER;
  END;
  Buffer = REF BufferObject;


 (* Insert this value into the buffer *)
PROCEDURE Produce(buffer: Buffer; i: INTEGER);
  BEGIN
    LOCK buffer^.mutex DO (* guarantee exclusive access *)
      (* if the buffer is full, wait for free space *)
      WHILE (buffer^.in+1) MOD BufferMax = buffer^.out DO
        Thread.Wait(buffer^.mutex, buffer^.nonFull);
      END;
      (* store the value and increment the input index *)
      buffer^.contents[buffer^.in] := i;
      buffer^.in := (buffer^.in+1) MOD BufferMax;
      (* announce the presence of a new value in the buffer *)
      Thread.Broadcast(buffer^.nonEmpty);
    END;
  END Produce;


 (* Get the next value from the buffer *)
PROCEDURE Consume(buffer: Buffer): INTEGER;
  VAR i: INTEGER;
  BEGIN
    LOCK buffer^.mutex DO (* guarantee exclusive access *)
      (* if the buffer is empty, wait for a new value *)
      WHILE buffer^.in = buffer^.out DO
        Thread.Wait(buffer^.mutex, buffer^.nonEmpty);
      END;
      (* get the next value and increment the output index *)
      i := buffer^.contents[buffer^.out];
      buffer^.out := (buffer^.out+1) MOD BufferMax;
      (* announce the presence of a free buffer space *)
      Thread.Broadcast(buffer^.nonFull);
    END;
    RETURN i;
  END Consume;

(* This is the root of the producer thread. The procedure type
must match the (generic) Forkee type because the procedure is
passed as an argument to Thread.Fork *)
PROCEDURE Producer(x: REFANY): REFANY;
  VAR
    buffer: Buffer;
    i: INTEGER;
  BEGIN
    (* x is known to be a Buffer *)
    buffer := NARROW(x, Buffer);
    (* Generate 10000 values *)
    FOR i := 1 TO 10000 DO Produce(buffer, i) END;
      Produce(buffer, 0); (* generate the last value *)
    RETURN NIL;
  END Producer;

    (* The root of the consumer thread *)
PROCEDURE Consumer(x: REFANY): REFANY;
  VAR
    buffer: Buffer;
    i: INTEGER;
  BEGIN
    (* x is known to be a Buffer *)
    buffer := NARROW(x, Buffer);
    (* Get and use values until the last one is seen *)
    REPEAT i := Consume(buffer) UNTIL i = 0;
    RETURN NIL;
  END Consumer;

(* Main Program *)
VAR
  b: Buffer; (* The shared buffer *)
  p, c: Thread.T; (* The two threads *)
  dummy: REFANY;

BEGIN
  NEW(b); (* Create the shared buffer *)
  WITH b^ DO (* Initialize the shared buffer *)
    in := 0; out := 0;
    Thread.InitCondition(nonEmpty);
    Thread.InitCondition(nonFull);
    Thread.InitMutex(mutex);
  END;

  (* Create and start the two threads *)
  p := Thread.Fork(Producer, b);
  c := Thread.Fork(Consumer, b);

  (* Await completion *)
  dummy := Thread.Join(p);
  dummy := Thread.Join(c);
END ProCon.
```

```
SAFE IMPLEMENTATION MODULE
  Thread;
SAFE MODULE ThreadClient;
```

And in definition modules as:

```
SAFE DEFINITION MODULE Thread;
```

The SAFE prefix means the implementation module is guaranteed safe by the compiler, and it is safe to import procedures and variables from the interface. When a program corrupts memory, the modules not guaranteed by the compiler are the prime suspects.

The compiler ensures that a SAFE implementation is indeed safe by enforcing several rules:

- Dynamic array bounds checking is enabled.
- Execution stack overflow checks are enabled.
- Dynamic NIL checking (detecting attempts to dereference a REF whose value is NIL) is enabled.
- VAR and PROCEDURE imports to SAFE modules come from SAFE interfaces.
- SAFE modules may not dereference a POINTER to yield a REF-containing value or VAR.
- SAFE modules may not perform any assignment through a POINTER.
- SAFE modules may not apply a type-transfer function to obtain a REF-containing type.
- A variant record with REF-containing arms has its variant tags set at the time it is allocated (via extra parameters to NEW), and the tags cannot be changed.
- Dynamic variant tag checking for REF-containing fields is enabled.

Type system extensions. In Modula-2, it is common practice to use a type-transfer function or other loophole (SYSTEM.WORD, SYSTEM.ADDRESS) to make a value of a par-

ticular type (*T*) acceptable to generic operations, then another loophole to make the resulting generic value acceptable as a *T*. Though dangerous, this is an acceptable technique in unsafe modules. But loopholes that could legitimize counterfeit REFS are illegal in SAFE modules.

Fortunately, by providing a limited form of runtime type discrimination for REF types, our program goes a long way toward eliminating most unsafe loopholes. And the execution cost of this runtime test is small, comparable to checking the tag in a variant record.

REFANY. REFANY is a new type for declaring variables that hold a value of any REF type. For example,

```
TYPE
  R = RECORD i, j: INTEGER END;
  S = RECORD m, n: INTEGER END;
  T = REF R;
  W = REF S;
VAR
  x: T;
  y: W;
  ref: REFANY;
ref := x; (* legal *)
ref := y; (* legal *)
x := y; (* illegal *)

x := ref;
  (* illegal, but see TYPECASE, below *)
```

Assignment of a REF <type> to a REFANY is always legal and generates no extra code, but to go the other way requires an explicit runtime type discrimination, using either the NARROW function or the TYPECASE statement.

NARROW. Often, the programmer knows the REF type of a REFANY value and needs to coerce the REFANY to this type in a safe way. NARROW is a standard procedure provided for this common case.

```
PROCEDURE NARROW(r: REFANY; t:
  TYPE): t;
```

The first parameter to NARROW must be a REFANY and the second must be the name of a REF type. If the first argument to NARROW is NIL, NARROW will return NIL. Otherwise, if the type of the value of *r* is *t*, NARROW will return this value. If the type is not *t*, NARROW will raise the exception SYSTEM.NarrowFault.

The choice of semantics for NIL was based both on its function as a polymorphic REF value and on our unsatisfactory experience with another design, wherein an error is raised for NIL. We found that raising an error for NIL often caused programmers to insert redundant explicit NIL checks before invocations of NARROW, when in fact the natural place for the explicit NIL check is after the NARROW.

TYPECASE. Often, the actual REF type may be known beforehand to be one of several. A construct similar to the CASE statement is provided for this:

```
TYPECASE ref OF
| T(x): statement-sequence-for-type-T
| W(y): statement-sequence-for-type-W
| ELSE statement-sequence-for-others
END;
```

The expression following TYPECASE must evaluate to a REFANY. If it is NIL, the first case arm will be selected. This design is consistent with the treatment of NIL by NARROW.

At runtime, the type of the value of ref is determined (call it type *X*). The types named on the left sides of the case arms are then compared for equality with *X*. For example, *X* is compared with *T* for the first arm above, and with *W* for the second arm. If a match is found, REF is assigned to the parenthesized variable (if any) and the corresponding statement sequence is executed. If no match is found, the ELSE clause, if present, is executed. If there is no ELSE clause, a runtime error is generated (the CASE statement has a similar definition). The parenthesized variable may be omitted, but if it is present, it must be assignable to a value of the type preceding it. Such variables must be predeclared.

Opaque REFs. In a definition module, Modula-2 allows the programmer to define an abstract type and its associated operations without giving implementation details of either the type's concrete representation or the code that implements the operations. Such types serve to isolate clients from implementation details, while preserving the compiler's ability to do static checking.

Such types are called opaque, but unfortunately they are somewhat translucent: the concrete representation is usually a pointer, a subrange, or a standard ordinal type, and must occupy at most one WORD. Indeed, Modula-2 allows structures containing an opaque type to be created, assigned, copied, and so on. The assumption is that a simple assignment operation suffices for opaque values. This is certainly not the case for REFS.

Extending the language to relax the restrictions on opaque types and tighten the implementor's control over the proliferation of opaque values would have some merit independent of our other extensions, but the clincher is the interaction between REFS and opaque types. Concrete types must be allowed to be REFS. But REFS require a nonstandard assignment operation because our garbage collector is based on reference counting.

An interface defining an opaque REF type is written:

```
TYPE T = REF;
```

The name opaque REF type is used to designate a REF to an unknown type. The client of the opaque REF type cannot manufacture an object of the unknown type and cannot dereference an opaque REF to obtain one.

The concrete definition of the opaque REF type must appear in the corresponding implementation module. Any REF type is allowed, including a REF type defined in another interface. Legal concrete definitions are, for example:

```
TYPE T = REF Rec;
TYPE T = Other.Ref;
```

where Other.Ref must be a REF or opaque REF type.

An opaque type is presented in the interface (definition) part of a module and its concrete definition is given in the corresponding implementation module:

```
DEFINITION MODULE FileSystem;
TYPE
  File = REF;
  Error = (noError, notFound,
           alreadyOpen ...);
  Buffer = ARRAY [1..100] OF CHAR;
PROCEDURE Open
  (name: INTEGER; VAR file: File) :
  Error;
PROCEDURE Close(file: File): Error;
PROCEDURE Read
  (from: File; numBytes: CARDINAL;
  VAR buf:Buffer): Error;
...
END FileSystem.

IMPLEMENTATION MODULE
```

```
FileSystem;
TYPE FileRec = RECORD
  ...
END;
TYPE File = REF FileRec;
PROCEDURE Open (name: ...): File;
...
PROCEDURE Close(file: File);
...
PROCEDURE Read
  (from: File; numBytes: CARDINAL;
      VAR buf:Buffer): Error;
...
END FileSystem.
```

Open arrays. Modula-2 allows arrays to be manipulated when the number of elements is not known until runtime, but these open arrays must be formal parameters of procedures. The programmer is not allowed to create new open array variables or allocate open array objects, but in practice he often needs to manage storage just this way. The Modula-2 programmer could use loopholes, Storage.ALLOCATE, and address arithmetic to overcome this limitation in unsafe programs, but not in safe programs. To eliminate the need for this unsafe circumvention:

(1) Modula-2+ allows open arrays as types:

```
TYPE EArray = ARRAY OF ElementType;
```

(2) Open array types can be used in the declaration of REF types:

```
TYPE EArrayRef = REF EArray;
```

(3) NEW, when used with variables of type EArrayRef, takes the number of elements as its second parameter. If the number of elements specified is zero, NEW returns a REF to an open array with zero elements:

```
VAR a1, a2: EArrayRef;
NEW(a1, 613); a2 := NIL;
IF a2 = NIL THEN
  NEW(a2, HIGH(a1^)+1);
END;
```

In Modula-2+, an open array can be designated either with the name of an open array parameter or the dereferencing operator (^) following a REF. An open array designator can occur in three contexts: as an actual array parameter corresponding to an open array formal parameter, as an array designator preceding a bracketed subscript expression, or as an actual parameter to the standard procedure HIGH:

```
PROCEDURE Equal(a1, a2: EArrayRef):
    BOOLEAN;
  VAR i: CARDINAL;
  BEGIN
   IF a1 = a2
    THEN RETURN TRUE END;
   IF HIGH(a1^) # HIGH(a2^)
    THEN RETURN FALSE END;
   FOR i := 0 TO HIGH(a1^) DO
    IF a1^[i] # a2^[i]
     THEN RETURN FALSE END;
   END;
   RETURN TRUE
  END Equal;
```

Concurrency

Features that describe and control concurrency are crucial in large, complex programs that use a multiprocessor. Proper synchronization and mutual exclusion are subtle and error-prone even for small programs, but are much more difficult for large ones. Modula-2 is adequate if the implementation is based on coroutines, but not if it uses multiprocessors.

Modula-2 is adequate if the implementation is based on coroutines, but not if it uses multiprocessors.

The underlying primitives we used to change and extend Modula-2 to program a true multiprocessor are similar to those used in Mesa and Cedar.[10]

First a note on terminology: To avoid confusion about the word "process," which has many conflicting connotations, we use the word "thread" to mean a thread of control and its associated stack.

Modula-2 facilities. In its Processes module, Modula-2 provides a small, simple, high-level facility to deal with concurrent, cooperating threads. Mutual exclusion is provided via special implementation modules, called monitors. The system guarantees that only one thread at a time can be executing in any one monitor. The number of monitors, and hence the number of data structures that can be protected by mutual exclusion, must therefore be a load-time constant. This precludes a programming style that depends on the dynamic creation of objects, each one protected against concurrent access. Yet many of our programs require such object-style monitors.

Synchronization is provided in Modula-2 via signals that can be sent and awaited. If no thread is waiting for a particular signal, a Send operation on that signal is ignored. When a thread waiting for a signal resumes, it can assume that the condition causing the signal to be sent is still satisfied. Unfortunately, not enough is said in Wirth's book about the relationship between signals and monitors. Worse, the situation is a lot more complicated than it appears at first: there are subtle semantic dependencies among the Wait and Send operations, monitors, and the scheduler. This design is adequate if the implementation is based on coroutines, but not if it uses multiprocessors.

An important question is not answered in Wirth's book: Must Wait and Send be called only from within a monitor that protects both the shared data and the signal?

If the answer is no, a signal may be sent and discarded just as a thread is about to wait for it. Such signals will be lost, and deadlocked or erroneous execution will ensue.

If the answer is yes, Wait must simultaneously suspend the thread and exit the monitor, then reenter the monitor when the thread is awakened some time later. What happens with Send is less clear, but there are at least two options:

(1) Send awakens a waiting thread, which would not be allowed to enter the monitor until the thread that did the Send exits, so the sender must be careful not to exchange shared data after doing the send and before exiting the monitor.

(2) Send awakens a waiting thread, exits the monitor, reschedules itself to allow the other thread to run, then reenters the monitor.

There are other subtle interactions between monitors and signals, leading to other design questions. In general, using the primitives in the Processes module to program all but the simplest applications for a multiprocessor is tricky and error-prone. For example, programs whose cor-

rectness depends on the proper use of features like scheduling priorities and the "awaited" predicate (at least one process is waiting) are hard to get right.

One basic problem with Modula-2's Process facility is the awkward requirement that the (one) thread awakened by Send be able to assume the readiness of the data it needs. A better solution to this entire set of problems is to have client programs test for the readiness of their data when they are awakened by Send, and wait again if the data is not ready.

The design of Modula-2+ can be implemented simply and efficiently for a multiprocessor, without introducing restrictions based on scheduling priority. A few programming idioms cover most of the tricky cases, including passing a parameter to the root procedure of a new thread, synchronizing one thread with the termination (perhaps the result) of another, invoking WAIT from inside a monitor, and identifying critical sections.

The Thread module. Most of the concurrent programming facilities in Modula-2+ are in the Thread module. Its design incorporates three important notions: Thread, Mutex, and Condition.

The opaque REF type *T* is defined in the Thread module to represent a thread of control:

```
TYPE T = REF;
```

A thread of control is created by Fork:

```
TYPE Forkee =
  PROCEDURE(REFANY): REFANY;
PROCEDURE
  Fork(proc: Forkee; arg: REFANY): T;
```

Fork creates a new thread of control within the caller's address space and causes it to call the given Forkee (proc), with the parameter arg. No information about the genesis of the new thread is retained. Moreover, the new thread gets its own execution stack, which is rooted at the call on proc. This means, for example, that an unhandled exception affects only the thread in which it is raised, causing that thread to enter the debugger. Other threads are not directly affected, though their subsequent behavior may depend on the damaged thread.

When the Forkee returns, its result may be acquired by Join:

```
PROCEDURE Join(thread: T): REFANY;
```

Join synchronizes with the specified Thread, waiting until the Forkee returns. Threads are themselves collectible objects (Thread.T is a REF type). A Thread that has returned from the Forkee has no client-accessible references and is not referenced by an outstanding Join will be claimed by the garbage collector.

Threads are explicitly synchronized with a Mutex:

```
TYPE Mutex;
PROCEDURE
  Acquire(VAR mutex: Mutex);
  (* If the mutex is free (not marked
  "inUse"), mark it "inUse" and continue. Otherwise, wait for it to become
  free. Do this atomically with respect to
  other threads *)
PROCEDURE
  Release(VAR mutex: Mutex);

(* Clear the "inUse" mark on the mutex
  and continue. This may cause another
  thread waiting in Acquire to be
  awakened and to acquire the mutex *)
```

Threads use Conditions to notify one another of state changes. For example, in the producer/consumer example in the box on p. 51, a Condition is used by the producer to awaken the consumer when the shared buffer becomes nonempty.

```
TYPE Condition;
PROCEDURE Wait(VAR mutex: Mutex;
  VAR condition: Condition);
PROCEDURE
  Broadcast(VAR condition: Condition);
PROCEDURE
  Signal(VAR condition: Condition);
```

A Condition, always used in connection with some mutex, has three operations: Wait, Broadcast, and Signal.

Wait is called from a thread that holds the mutex. It causes the thread to block, waiting for the specified condition to be notified and the mutex to be released. When the thread is awakened, it will attempt to reacquire the mutex. Since it may be competing with other threads for the mutex, it may have to block again.

Broadcast and Signal are usually called from a thread inside the mutex. Broadcast wakes every thread that has previously called Wait and has not yet been awakened. Signal wakes one or more such threads. (Normally one, except in rare cases when a critical race may awake more than one. It is not worth the loss in runtime efficiency to handle such cases.) Signal is more efficient for programs that do not require waking multiple threads, but when in doubt, Broadcast should be used.

As an optimization, Signal or Broadcast may be called after releasing the mutex. In that case, Broadcast wakes all threads that had called Wait before this thread acquired the mutex, and Signal wakes one or more such threads. Either operation may also wake zero or more threads that have called Wait after this thread released the mutex. This optimization can help prevent a newly awakened thread from immediately blocking on the mutex.

The semantics we have adopted can be implemented efficiently. They imply that return of control from a Wait is merely a hint that some action may need to be taken by the thread, it does not guarantee (as in some monitor-based communication schemes) that action is required. Therefore, Waits should occur in WHILE loops. If programs are written to treat return from a Wait as a hint, extra wakeups can affect performance, but not correctness.

The implementations of Thread.Mutex and Thread.Condition are identical. The two types are provided to emphasize the important conceptual distinctions between the two notions.

The LOCK statement. The Modula-2+ syntax for mutual exclusion is:

```
LOCK mutex DO statement-sequence END;
```

where mutex is a designator of type Mutex. This construct is equivalent to:

```
VAR &t: POINTER TO Thread.Mutex;
  (* &t is a new, unique name *)
&t := SYSTEM.ADR(mutex);
  (* evaluate mutex once *)
Thread.Acquire(&t^);
TRY
  statement-sequence
FINALLY
  Thread.Release(&t^)
END;
```

It was not strictly necessary to provide the LOCK statement as a shorthand for the code above, but its convenience and clarity have proven to be extremely valuable. This statement is used by the vast majority of the concurrent programs we have written.

Modules and interfaces

Modula-2 presents a model of how parts of a system should fit together. The principal idea is that it is good to separate the specification of an abstraction from its implementation. A definition module specifies an abstract interface, and an implementation module its implementation, supplying procedure bodies and concrete type definitions. Modula-2 requires that an implementation module export exactly one interface and an interface be exported by exactly one module. Unfortunately, larger packages are often best organized with multiple implementation modules that collectively export multiple interfaces.

A less restrictive package structure could be achieved without extending the language by using intermediate modules to do nothing more than import and call procedures. Managing these extraneous implementation modules and useless procedures, however, is too expensive. For example, it is often very expensive to restructure a large implementation module in the midst of its development when the need for an auxiliary interface arises. We were forced to revisit this issue after finding that our initial design, purposely modest, was inadequate.

Our initial design extended the declaration forms for variables, procedures, and exceptions, making them more like the declaration forms for constants and types. Thus, an imported item could be given as the definition of the new identifier:

```
TYPE Mutex = Concurrency.Semaphore;
PROCEDURE
  Acquire(m: Mutex) = Concurrency.P;
VAR nThreads:
  INTEGER = Concurrency.nThreads;
```

A suitable arrangement of client interfaces was achieved by regrouping exports from implementation modules into veneer interfaces, which consisted entirely of definitions of the form, n = mod.n.

Unfortunately, we had more problems with this simple approach than we anticipated. Changing definition modules, even for private use only, requires an analysis to identify the affected client modules reliably. Software version management tools help substantially, but the cost of maintaining consistency in a complex system remains high.

Worse still, use of the pass-through technique led to deep trees of import dependencies among interfaces. The most public interfaces, and hence the programs that imported them, would often depend on the lowest-level private interfaces. The Thread interface, for example, passed to its clients the definitions of Acquire and Release from the Concurrency interface, the low-level interface that defined all concurrency-related operations. To support operations for other clients, Concurrency imported Machine, the interface that defines the VAX machine architecture. This included all details of the machine state (register values, processor status) plus various utility types and procedures for dealing with it. Thus, even the smallest program with a LOCK statement required analysis whenever any change was made to any one of these interfaces.

The semantics we have adopted can be implemented efficiently.

Gratuitous import dependencies meant that more analysis than necessary was required to satisfy our version management tools, and this took longer because there was extraneous work to do. This situation would have worsened as we built larger systems.

A more suitable design was needed. We chose to allow a single implementation module to partition its exports into several complete interfaces. We also considered a further generalization to allow one module to export only some of the items in an interface, as in Mesa. Although this facilitates the direct collaboration among several modules to implement an entire interface, we rejected the idea because the same result can be achieved by using the pass-through feature in conjunction with the following feature.

An interesting problem is how to deal with several alternative implementations of the same interface, but this is beyond the scope of our work.

IMPLEMENTS. The new statement:

```
MODULE M IMPLEMENTS A,B,C;
```

is allowed in place of

```
IMPLEMENTATION MODULE M;
```

as the first statement in a program module.

In this new case, the module is compiled as an implementation module, but compilation begins with inclusion of definition module A into the name-lookup context being constructed for M, then the inclusion of B, then of C. Inclusion of a definition module proceeds as follows.

Each name declared in the definition module is ported to the new name-lookup context being constructed for M. Porting a name from one context to another means that subsequent reference to the unqualified name in the new context will identify the item defined with that name in the original context.

Only one implementation module is allowed to implement a given interface.

Using the new form, the existing statement

```
IMPLEMENTATION MODULE Foo;
```

can be expressed as

```
MODULE Foo IMPLEMENTS Foo;
```

For example, we used the IMPLEMENTS feature to split the Concurrency interface into two parts: one with the definitions of Acquire and Release for high-level clients like Thread, and one with items of interest to lower level clients. This broke Thread's dependency on Machine and made system development easier.

Problems with opaque types. The IMPLEMENTS feature eased communication among collaborating modules, but this communication was crippled when an opaque type was used because its concrete representation could be known only in its own implementation module.

For example, a package we developed for efficient text manipulation and stream-oriented I/O required fast individual read and write operations. This meant integrating the implementations of the Text module, the Reader module and the Writer module. A simplified form of these three abstractions is shown in Figure 1.

The problem was that the Reader and Writer implementations required knowl-

edge of the concrete representation of a Text.T.

To support such applications, Modula-2+ allows the concrete definition for an opaque REF type D.T to be given not only in D's implementation module, but in any module A that imports D. The program linker is used to verify that all separately compiled modules that are loaded together define the same concrete type for D.T.

Limiting access to private implementation details is achieved by placing the definition of the concrete type in a module that can be accessed only by the collaborating implementors. In the Text/Reader/Writer example, an auxiliary module named TextPrivate was defined and imported only by the three implementation modules Text, Reader and Writer:

```
SAFE DEFINITION MODULE
    TextPrivate;
  TYPE TextRep =
    REF RECORD ... END;
END TextPrivate.

SAFE IMPLEMENTATION MODULE
    Text;
  IMPORT TextPrivate;
  TYPE T = TextPrivate.TextRep;
  ...
END Text.

SAFE IMPLEMENTATION MODULE
    Reader;
  IMPORT Text, TextPrivate;
  TYPE TextT = Text.T;
    TextT = TextPrivate.TextRep;
  ...
END Reader.

SAFE IMPLEMENTATION MODULE
    Writer;
  IMPORT Text, TextPrivate;
  TYPE TextT = Text.T;
    TextT = TextPrivate.TextRep;
  ...
END Writer.
```

```
SAFE DEFINITION MODULE Text;
  TYPE T = REF; (* represents a vector of characters *)
    Index = [LAST(INTEGER)];

  PROCEDURE Cat(t1, t2: T): T RAISES {};
   (* Returns the concatenation of "t1" and "t2". *)
  PROCEDURE Length(source: T): Index RAISES {};
   (* Returns the number of characters in "source." *)

  (* ETCETERA, including facilities for text literals *)
END Text.

SAFE DEFINITION MODULE Reader;
  IMPORT Text;

  TYPE T = REF;
  (* A Reader.T represents the source of a character sequence. The Reader package is
    optimized for fast initialization and fast sequential access to Text.T. Readers are also
    the standard way of accessing other character sources such as files and terminals.
    *)

EXCEPTION EndOfFile(T);

  PROCEDURE FromText(source: Text.T): T RAISES {};
   (* Returns a Reader taking characters from the Text source. *)
  PROCEDURE EOF(rd: T): BOOLEAN;
   (* Returns TRUE iff a call to GetChar would raise EndOfFile. *)
  PROCEDURE GetChar(rd: T): CHAR RAISES {EndOfFile};
  PROCEDURE Find(rd: T; pattern: Text.T): INTEGER;
   (* Finds the first occurrence of pattern, reading forward from the current position of
     rd. If no match is found, Find returns -1 and leaves rd positioned at the end. If a
     match is found, Find returns the index of the first character of the match and
     leaves rd positioned to read the character following the match.*)

  (* ETCETERA, including facilities for formatted input *)
END Reader.

SAFE DEFINITION MODULE Writer;
  IMPORT Text;

  TYPE T = REF;
  (* A Writer.T represents a destination for a character sequence. The Writer package is
    optimized for fast initialization and fast sequential writing of Text.T. Writers are also
    the standard way of directing characters to other destinations such as files, pipes,
    and terminals. *)

  PROCEDURE New(bufferLength: INTEGER): T RAISES {};
   (* Returns a new Text writer. A Text writer accumulates the characters written to it in a
     private buffer. *)
  PROCEDURE ToText(wr: T): Text.T RAISES {Error};
   (* Returns a Text containing the characters written to the Text writer wr. *)
  PROCEDURE PutChar(wr: T; c: CHAR) RAISES {};
  PROCEDURE PutText(wr: T; source: Text.T) RAISES {};

  (* ETCETERA, including facilities for formatted output *)
END Writer.
```

Figure 1. A simplified form of the Text, Reader, and Writer modules.

We are using Modula-2+ at SRC to build both system and application software for a multiprocessor workstation. We have extended the language to include support for exception handling and garbage collection, and to replace the facilities for dealing with concurrency with ones more suitable for a true multiprocessor. Experience has shown these extensions to have no effect on the integrity of the overall design.

The extensions for exception handling include a finalization feature and a change to the type system to enable reliable documentation of the exceptions that can be raised by a given procedure. The implementation supports debugging for unhandled exceptions. It has negligible overhead in the normal case (no exceptions raised).

Though the extensions for storage safety and garbage collection are more radical, they have worked out very well. These techniques are sufficiently mature for wide-

spread use. We recommend them for other systems based on Modula-2 or similar languages.

The changes for concurrency were minor. The Processes module and the ability to specify a module as a monitor were removed, one statement form (LOCK) was added, and the standard Processes module was replaced by the Thread module. The features provided by the LOCK statement and the Thread module enable rapid progress in the development of large concurrent programs. These methods, too, have been tested extensively in real systems and are sufficiently mature for widespread use. □

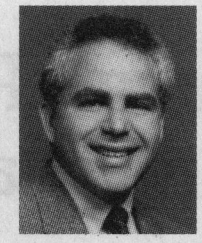
Acknowledgments

The frequent use of the pronoun "we" in the text is intentional: the work was done in a community of colleagues, active users of and contributors to one another's work.

Andrew Birrell, Butler Lampson, and Roy Levin had a large influence on the design. Others at the DEC System Research Center offered valuable suggestions as the work progressed. Contributions to the text of this article were made by Andrew Birrell, Violetta Cavalli-Sforza, Cynthia Hibbard, Roy Levin, and Mary-Claire van Leunen. The presentation was improved by suggestions from the referees and Cynthia Hibbard. Mike Powell provided an initial implementation of Modula-2.

References

1. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815A-1983, US Defense Dept., Washington, DC, Feb. 1983.
2. Richard C. Holt, *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley, Reading, Mass., 1983.
3. Barbara Liskov et al., *CLU Reference Manual*, Lecture Notes in Computer Science 114, Springer-Verlag, New York, 1981.
4. Mike Powell, "A Portable Optimizing Compiler for Modula-2," *Proc. SIGPlan Compiler Construction Conf.*, June 1984.
5. L. Peter Deutsch and Edward A. Taft, *Requirements for an Experimental Programming Environment*, Xerox PARC Technical Report CSL-80-10, Palo Alto, Calif., June 1980.
6. Niklaus Wirth, *Programming in Modula-2, third ed.*, Springer-Verlag, New York, 1985.
7. James G. Mitchell et al., *Mesa Language Manual*, Xerox PARC Technical Report CSL-79-3, Palo Alto, Calif., 1979.
8. Daniel C. Swinehart et al., "The Structure of Cedar," *SIGPlan Notices*, Vol. 20, No. 7, July 1985.
9. Paul Rovner, "On Adding Garbage Collection and Runtime Types to a Strongly Typed, Statically Checked, Concurrent Language," Xerox PARC Technical Report CSL-84-7, Palo Alto, Calif., 1985.
10. Butler W. Lampson and David D. Redell, "Experiences with Processes and Monitors in Mesa," *Comm. ACM*, Vol. 23, No. 2, Feb. 1980.
11. R. Levin, "Program Structures for Exceptional Condition Handling," Dept. Computer Science technical report, Carnegie Mellon Univ., Pittsburgh, June 1977.



Paul Rovner is a researcher at the DEC System Research Center. His current interests are programming systems and related applications of concurrency. His previous position was at Xerox PARC, where he served as a technical leader on the Cedar project. He has held other research positions at BBN and MIT's Lincoln Laboratory.

Rovner was a founding member of the computer science department at the University of Rochester, where he served as an associate professor. He has also worked on problems in distributed computing, automatic speech understanding, automatic data structure selection, and computer graphics. He received his PhD from Harvard University in 1976.

Must Reading

Turn to the Inmac catalog for helpful hints and problem solving advice from our engineers. And for over 2,400 computer supplies and accessories displayed and described in detail. All have our exclusive 45-day trial and minimum 1-year guarantee.

FREE!

Call or write today. Yes... rush me your catalog today.

800-547-5444

inmac™

Name _____
Company _____
Address _____
City _____ State _____ Zip _____
Phone ( ) _____
2465 Augustine Drive, Santa Clara, Ca. 95054

5079

Reader Service Number 4