3

On Extending Modula-2 For Building Large, Integrated Systems

Paul Rovner, Roy Levin, John Wick

January 11, 1985

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

SRC Members

Pete Benoit
Andrew Birrell
Andrei Broder
Kenneth Brooks
Mark Brown
John DeTreville
Leo Guibas
Toni Guttman
Jim Horning
Steve Jeske
Karen Kolling
Butler Lampson
Ed Lazowska,
visiting scientist
Roy Levin
Greg Nelson
Carol Peters
Phil Petit
Søren Prehn
visiting scientist
Chuck Price
Lyle Ramshaw
Paul Rovner
Richard Schedler
Mike Schroeder
Larry Stewart
Jorge Stolfi
Garret Swart
Chuck Thacker
Mary-Claire van Leunen
John Wick

Robert W. Taylor, Director

DEC's System Research Center is a new group, still recruiting its initial members and laying plans for long-term work. The business and technology objectives of DEC require a strong commitment to research. We join two other corporate research groups in meeting that commitment.

SRC's role is to design, build, and use new digital systems five to ten years before they become commonplace. Our purpose is to advance both the state of knowledge and the state of the art.

SRC will create and use real systems in order to investigate their properties. Interesting systems are too complex to be evaluated purely in the abstract. Our strategy is to build prototypes, use them as daily tools, and feed the experience back into the design of better tools and more relevant theories. Most of the major advances in information systems have come through this strategy, including time-sharing, the Arpanet, and distributed personal computing.

Among the areas in which SRC will be building prototypes during the next several years are applications of high-performance personal computing, distributed computing, communications, databases, programming environments, system-building tools, design automation, specification technology, and tightly coupled multiprocessors.

We will also do work of a more formal and mathematical flavor; some of us will be constructing theories, developing algorithms, and proving theorems as well as designing systems and writing programs. Some of SRC's theory work will be in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. In other cases, we expect to explore new ground motivated by problems that arise in our systems research.

DEC has a commitment to open research. The improved understanding that comes with widespread exposure seems more valuable than any transient competitive advantage. SRC will freely report results at conferences and in professional journals. We will encourage visits by university researchers and conduct collaborative research. We will actively seek users for our prototype systems. To facilitate interchange, we will develop systems that run on hardware available to universities and work out ways of making our software available for academic use.

On Extending Modula-2 For Building Large, Integrated Systems

Paul Rovner, Roy Levin, John Wick

Acknowledgements: Several people other than the authors contributed to the ideas presented herein. Butler Lampson and Andrew Birrell made major design contributions. Others at SRC offered valuable suggestions as the work progressed. Comments from Greg Nelson and Jim Horning on the presentation led to substantial improvements. Under a joint development agreement, Mick Jordan and Jim Mitchell of Acorn Research Centre contributed to the material on program formatting conventions.

Authors' abstract:

Modula-2 has been chosen as SRC's primary programming language for the next few years. This report addresses some of the problems of using Modula-2 for building large, integrated systems. The report has three sections: Section 1 outlines a set of extensions to the language. (The extended language is called Modula-2+.) Section 2 (with Appendix B) provides a complete description of the Modula-2+ type-checking rules. Section 3 offers some guidelines for programming in Modula-2+.

Our implementation of Modula-2+ is based on the Modula-2 compiler written by Mike Powell at the DEC Western Research Laboratory. Our extensions include features for exceptions and finalization, garbage collection, and concurrency.

Paul Rovner, Roy Levin, John Wick

Capsule review:

This paper is an informal overview of the programming language Modula-2+, an extension of Modula-2. Most of the new features are in the three areas of signalling and handling exceptions, controlling concurrency, and providing a type-safe discipline for managing storage.

The value of exception-handling follows from the fact that many useful abstractions cannot reasonably be implemented in their full generality. For example, the natural numbers are a useful abstraction, but they can't all be represented in a finite-state machine. When a program specifies an abstract operation that exceeds a particular implementations' capacity, some exceptional action must be taken, even if only to print "computation aborted dur to arithmetic overflow." When a single language is used to code systems with many layers of abstraction, it is attractive to provide a control structure for signalling and handling these circumstances. Hence Modula-2+'s exception facility.

Apparently the Modula-2 abstraction called "process" fails to hide the nature of its first implementation by a multiprocessor. Therefore, Modula-2+ introduces a more suitable abstraction for a thread of control (called "Thread").

Compile-time checking in Modula-2 guarantees the absence of a class of errors but gives no warning of the error of deallocating storage while references to it remain. Modula-2+ guarantees the absence of these dangerous errors by a combination of run-time and compile-time checking: A garbage collector provides automatic deallocation at run-time, and checking at compile-time enforces a discipline on the use of references that guarantees the validity of the storage system's invariants.

The authors assume that the reader is already familiar with Modula-2. Most of the paper is a general overview, but it drops to the detail necessary for a reference manual in the appendixes, which describe the changes made to the syntax, the type-checking rules, and formatting conventions.

Greg Nelson

Contents

DEC's System Research Center is currently developing software and hardware to provide a powerful base for work in programming systems, distributed systems and personal workstations. We hope to significantly reduce the cost of producing reliable, high-quality software.

Though good software design remains an art, it is easier today than it used to be, due largely to the emergence of systematic techniques for structuring programs. These include improved methods for identifying the interfaces between the component parts of a program, and new programming languages that provide better chosen and better integrated features, particularly those that support explicit interfaces.

Modula-2 has been chosen as SRC's primary programming language for the next few years, both for implementing system software and for exploring prototype applications. It strikes a reasonable balance between simplicity and functionality. While some of the design choices are arguable (e.g., POW84-1), it does provide a well-integrated combination of features from the following domains:

* Strong typing with static checking
* Separate specification of interfaces and their implementations
* Data abstraction (via "opaque" types)
* Systems programming (via "low-level" facilities)
* Concurrency
* Procedure-valued variables

Though we did find it necessary to extend Modula-2 in the ways outlined below, we believe after looking at the available alternatives that it comes closest to serving our needs. A comparative analysis, including both the technical and non-technical reasoning that led to our choice of Modula-2, is beyond the scope of this paper, but would make a good topic for another report.

This report addresses some of the problems of using Modula-2 for building large, integrated systems. It should be of interest both to those who study programming languages in general and Modula-2 in particular, and to those who build such systems. The discussion assumes a familiarity with Modula-2 as described in the book "Programming in Modula-2," by Niklaus Wirth (WIR82).

The report has three sections. Section 1 outlines a set of extensions to the language and to its compiler and runtime library. The extended language is called Modula-2+. Each extension is introduced in the context of our goals and requirements, then presented in detail with examples. A reader interested only in acquiring an overview might read the introductory material in each subsection but skip the detailed presentations of language changes. A grammar for the extensions appears in Appendix A.

Section 2 (with Appendix B) provides a complete description of the Modula-2+ type-checking rules. This is included here because it is otherwise unavailable (the Modula-2 book is vague on this subject) and we find it to be important for effective use of the language. Others learning or using Modula-2 should also find it valuable.

Section 3 offers some guidelines for programming in Modula-2+. Appendix C contains a complete set of conventions for formatting Modula-2+ programs and interfaces: indentation, spelling and punctuation rules are presented, as well as guidelines for placing comments. Again, this material is included here because we find it to be important for effective use of the language, not conveniently available elsewhere, and likely to be of interest to others learning or using Modula-2.

Our implementation of Modula-2+ is based on the Modula-2 compiler written by Mike Powell at the DEC Western Research Laboratory (DECWRL). This is an optimizing compiler with good code quality and a straightforward implementation (POW84-2). In its author's words,

> "The design philosophy of the compiler is 'best simple'. Whenever possible, design decisions were made to favor the simplest alternative that got us most of what we wanted."

1. Language Extensions

This section provides a brief description of our extensions to Modula-2, omitting some of the less important details. A grammar for the extensions appears in Appendix A.

Our primary purpose in embarking on a project to extend Modula-2 was to provide what we believed to be essential features with minimal disruption to the language and the compiler. It was our explicit goal to stay compatible with the underlying "spirit" of both the language and the compiler. It was NOT our goal to build a "PL/I" of Modula-2 systems. It was a ground rule for the DECWRL compiler that it support "standard" Modula-2 programs. The Modula-2+ compiler has the same constraint.

We considered additional ideas for upward-compatible extensions, but adopted them only if they would certainly have high payoff for clients, low impact on the language and a simple implementation. The DECWRL compiler had several such extensions, made in the same spirit. These included removal of restrictions on the type and size of function procedure results and on the size of sets, optional runtime checking, optional treatment of CARDINAL as a subrange of INTEGER, a new standard type for double-precision REALs and facilities for linking to programs written in other languages.

Based on our experience with similar systems in the past, we do not anticipate surprises from the use of Modula-2+. But we have used it ourselves only for a short time, and we are well aware that the actual use of a complex system is often an educational and sometimes a humbling experience for its designers. The reader should note that the balance of emphasis in the design depends more on our experience with similar systems than with this one, and should anticipate a critical report on our actual experiences with Modula-2+ a year or so hence.

1.1 Exceptions and Finalization

Introduction

The behavior of an abstraction when its implementation fails is an essential part of its specification. Clarity of such a specification is most naturally achieved by outlining its expected behavior separately from a list of the problems that might arise. Explicit provision in the language for decomposing a program into a normal case part and a "handler" for the exceptional cases improves predictability, robustness and

reliability.

Normal use of a component can almost always be expressed without including explicit code at each procedure call to deal with exceptions. Though it is possible to define each procedure to return a value that (by convention) identifies an exceptional result, it is awkward and inefficient to insert a check at every call. And if by mistake the check is not made, the exceptional result will go unnoticed.

Rather, it is more natural to associate a "handler" with a sequence of statements. The handler deals with exceptions that arise from any one of them. This enables the statement sequence to be written for the normal case; code to deal with exceptional cases can be placed outside the normal flow of control.

A related (and common) programming idiom identifies a natural temporal framework: first "initialization," then the main body, then "finalization." An example in a concurrent program 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 (e.g., via EXIT, RETURN, or exceptional result): the finalization action must usually be taken in any case. Usually, coding one's way out of such a mess is awkward and error-prone. Language support for this common control structure should accompany changes for exception-handling. With a little care, implementation of finalization features can be a simple extension of the exception-handling mechanism.

Good debugging facilities are needed to support rapid development of large experimental systems. When exception-handling is used, programming bugs and oversights are manifest as unhandled exceptions. The implementation must recognize unhandled exceptions and pass them to the debugger without destroying the context in which the exception was raised.

Finally, and most important, we required a design that could be implemented at reasonable cost and with negligible runtime overhead for normal execution. For example, the execution cost of introducing handler and finalization scopes had to be negligible. (A satisfactory implementation is in place but is not discussed here.)

Semantics

Exception-handling in Modula-2+ is based on a "termination" model, similar in some ways to the ones in Ada (ADA82) and in CLU (LIS81). The choice of exception-handling semantics for

Modula-2+ was also influenced by our experiences with Mesa (MIT79). See (LEV77) for a good survey of related issues and techniques.

Handlers are attached to statement-sequences via the TRY statement, described below. When an exception is raised, a handler for it that is attached to the enclosing statement-sequence is sought. If none is found, the next enclosing statement-sequence is examined, and so on, until either a handler is found or the boundary of the current procedure is reached, in which case the search continues in the context of the calling procedure. Finalization actions are performed as they are encountered. When a handler is found, control is passed to it. The handler effectively "takes the place" of the statement-sequence to which it is attached.

If a handler is not found before the root of the stack is reached, the debugger is invoked. In this unusual case, the implementation of RAISE presents the debugger with the execution context as it was when RAISE was invoked: e.g., the stack is intact and finalization actions have not been performed.

Implicit finalization actions are associated with each procedure (e.g., restoration of the caller's context). Explicit finalization actions may be attached to statement-sequences via the TRY statement. While executing, such an action operates in the same context as the statement-sequence to which it is attached. Abnormal termination of such an action, e.g., by raising another exception, has the same effect as if the statement-sequence terminated in the same way.

Procedures and procedure types may be defined to raise only exceptions from a specified set, plus a standard exception, System.Fail. At execution time, other exceptions that would emerge from such a procedure are automatically converted to System.Fail. The argument to System.Fail identifies the original exception and its argument, if any.

Language Forms

Exceptions are declared by specifying a name and at most one parameter. E.g.,

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

The parameter can be of any type that is allowed as the result of a function procedure.

Exception names obey standard scope rules, but are treated like

constants rather than variables. For example, an exception declared in a recursive procedure does not get a new definition for each invocation. Rather, a unique internal code is assigned by the compiler to each exception declaration; this code is used to identify the exception at runtime.

Exceptions are said to "propagate" upward through nested dynamic contexts from the "raiser," looking for a "handler." If a handler is found, the exception is said to be "handled" by it.

An exception is raised by

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

if the exception is declared to take a parameter, or by

```
RAISE(exception)
```

otherwise.

An exception is handled by

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

The code following EXCEPT is similar to a CASE statement. If, during the execution of the statement-sequence, one of the listed exceptions is raised, execution of the statement-sequence ceases and control passes to the corresponding handler-body (a statement-sequence). If the exception raised doesn't match any of those listed and an ELSE clause is present, the ELSE handler-body receives control. If no ELSE clause is present, propagation continues in the enclosing context.

Before execution of a handler-body, all dynamic contexts between the raiser and the statement-sequence (inclusive) are "finalized": i.e., the stack is unwound, register values are restored, and explicit finalization actions are invoked (see page 8 below). Upon completion of the handler-body, control passes to the statement following the TRY construct, that is, to the same place it would have passed if the statement-sequence hadn't encountered an exception.

The exception name in a handler arm may be followed by the name of a variable declared in an enclosing scope. The exception

parameter must be assignable to this variable. The value passed by RAISE as the parameter to the exception is assigned to the indicated variable when the handler is entered. A variable need not be specified if it is not needed in the handler-body.

By convention, the only exceptions handled by ELSE are those caused by programming errors. Low-level system failures or other catastrophes, e.g., memory parity errors or device failure, require different treatment. ELSE handlers do not usually appear in application programs, though they are sometimes useful for bullet-proofing experimental code. Rather, their application is in code that for some reason does not want the debugger to field errors, e.g., for a read/eval/print loop. Such a handler might print a message identifying the exception and its argument and then continue at the top level. ELSE handlers must be used with care; thoughtless use can mask programming errors.

The parameter type of an ELSE handler is System.FailArg, a type that represents an exception and its argument, if any. Operations are provided in the System module for examining the components of a FailArg, but they will not be described here.

Syntactic Shorthands

(1) TRY statement-sequence PASSING {ex-1, ex-2} END;

means

```
TRY
  statement-sequence
EXCEPT
| ex-1(v1): RAISE(ex-1,v1);
| ex-1(v2): RAISE(ex-2,v2);
| ELSE(v3)  RAISE(System.Fail,v3);
END;
```

In this construct, any of the listed exceptions raised during the execution of the statement-sequence pass through to the enclosing context.

(NOTE: To support debugging, such simple "pass-through" handlers are treated specially by the compiler: termination is postponed until a non-trivial handler is found.)

All other exceptions are handled by converting them to System.Fail, which is included implicitly in every PASSING clause (by virtue of the ELSE). The phrase "PASSING {}" means that only Fail can be raised.

By convention, System.Fail is used to report programming errors. Normally, application programs do not handle System.Fail.

Rather, any unhandled exception, including System.Fail, is intercepted by a debugging facility in a way that preserves the contexts through which the exception propagated.

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

This behaves the same at runtime as

```
PROCEDURE P(actuals): ret;
  <decls>
  TRY <body> PASSING {ex-1, ex-2} END;
```

but the inclusion of the RAISES clause in procedure types provides a client with reliable documentation of the exceptions that can be raised. Thus, the RAISES clause affects both the procedure type and the body. Use of the RAISES clause in definition modules is a valuable specification technique.

As with PASSING, each RAISES clause implicitly includes System.Fail, and an empty list means that only Fail can be raised. If no RAISES clause appears, any exception may be raised by the procedure.

Finalization

```
TRY
  block-statement-sequence
FINALLY
  clean-up-statement-sequence
END;
```

In this construct, the block-statement-sequence is executed and, upon its termination, the clean-up-statement-sequence is executed. Termination of the block-statement-sequence occurs either normally, or by executing an EXIT or RETURN statement, or by raising an exception. For example, consider the following:

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

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

in s2, the sequence will be s1; s2 (partially); s4; s5; s7.

If finalization occurs as the result of an exception, and the clean-up-statement-sequence raises an exception, the original exception is lost.

1.2 Safety

Previous experience with LISP(TEI78), Smalltalk(GOL83), Mesa(LAM80, MIT79) and Cedar(DEU80, TEI84) has taught us that the use of a single virtual address space enables the development of highly integrated applications that share packages and present a standard, ubiquitous user interface. Systems of concurrent applications that share data structures and programs are especially well-suited for development in a common address space. Multiple, loosely coupled address spaces provide additional options for structuring software, but do not provide a consistently better alternative for such systems.

Efficient, non-disruptive garbage collection is a key requirement in a shared address space. In a setting where multiple complex and experimental programs that share memory are being developed together, and by different people, it is crucial to guard against programming blunders that smash memory. In addition, our experience indicates that it is possible to achieve a dramatic decrease in the fraction of programming costs due to storage management.

We define a "safe" program as one that will not violate the invariants of the storage allocator, i.e., it will not cause memory to be smashed. Operationally, this means that a safe program will not alter storage that has not been properly allocated (e.g., by accessing "off the end" of an array or storing through an invalid pointer). A safe program can still get the wrong answer, but it will not cause an independent program sharing the same address space to do so.

We wanted to use simple static analysis to prove (informally) that a given program is safe, and we wanted such analysis to apply to as many programs as possible. Our approach had two parts:

1. We identified those Modula-2 language features and library procedures that if misused could smash memory. If a given program uses none of these, it is safe.

2. Where feasible, we provided a safe alternative for each such facility unless very few programs would need it.

The discussion below and in the rest of section 1 outlines a small collection of extensions and restrictions to Modula-2 that provide a language of sufficient power and convenience for writing most programs. Simple static analysis is adequate for proving such programs to be safe.

Of course, some programs cannot be shown to be safe in this way, e.g., a storage allocator or a garbage collector. Others for which such restrictions are sometimes inappropriate include ones requiring access to low-level facilities and ones with stringent performance requirements. Rather, these must be assumed to be safe. Experience indicates that there are few such programs. As usual, they must be written and maintained with special care.

REFs

The use of POINTERs is restricted in safe programs. We have added to the language a well-behaved form of POINTER called REF, which can nearly always be used instead. REFs behave much like POINTERs, except that the storage they address is never explicitly freed by the programmer. REFs are declared as follows:

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

Like POINTERs, REFs are created by NEW, i.e., 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" or "RC". Every REF variable is initialized to NIL.

Static checking for safety is enabled in an implementation or program module via use of the keyword SAFE. For example,

```
SAFE IMPLEMENTATION MODULE Threads;
SAFE MODULE ThreadsClient;
```

Definition modules can also be marked SAFE, e.g.,

```
SAFE DEFINITION MODULE Threads;
```

This indicates that the corresponding implementation module is safe, hence it is safe to import procedures and variables from the interface. If the corresponding implementation module begins

with the keyword SAFE then it is guaranteed safe by the compiler; otherwise, it is safe on the assumption that the implementor knows what he is doing. If a core-smash bug occurs, implementation modules not checked by the compiler are the prime candidates for scrutiny.

The compiler ensures that a SAFE implementation is actually safe by enforcing the following rules:

** Array bounds checking is enabled in SAFE modules.

** NIL checking (detecting attempts to dereference a REF whose value is NIL) is enabled in SAFE modules.

** VAR and PROCEDURE imports to SAFE modules must come from SAFE interfaces.

** SAFE modules may not dereference a POINTER to yield an RC value or VAR.

** SAFE modules may not perform any assignment through a POINTER.

** SAFE modules may not apply LOOPHOLE or a type-transfer function to obtain an RC type. The types Address and Word are not compatible with REFs (see section 2 on type-checking).

** 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 subsequently changed. Variant tag checking for RC fields is always enabled in safe modules.

1.3 Runtime Types

There are times, in using languages that provide strong typing and static checking, when a programmer needs to circumvent the restrictions imposed by the type system. Many such cases are attributable to one basic inadequacy, namely the lack of what is called "subclassing" in Simula (BIR73) and Smalltalk, i.e., the ability to define a new type as a specialization of an existing one: the applicable operations for the new type include the operations of the existing type, plus some new ones. A familiar example is REF (or POINTER) types: many common operations (e.g., assignment, list processing) need not distinguish a "REF T" from a "REF S."

In standard 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 particular 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. Happily, provision of a limited form of runtime type-checking for REF types, described below, goes a long way toward eliminating most unsafe loopholes. And the execution cost is small: it is 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 checking code; to go the other way requires an explicit runtime test, using the following construct:

```
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, and must not be NIL. If it is NIL, a runtime error will be generated.

> We considered a design wherein the ELSE clause would catch NIL. The pros and cons of the two sets of features appeared to balance; we chose this design because the implementation was simpler and the compiled code is faster in the normal case.

At runtime, the type of the object addressed by "ref" is determined (call this type X). Then, in turn, the types pointed to by the types named on the left-hand sides of the case arms are compared for equality with X. For example, X is compared with R for the first arm above, and with S 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. Note that the parenthesized variable may be omitted, but if it is present, it must be assignment-compatible with a value of the type preceding it.

It is often the case that the programmer knows the actual type of a REFANY, and needs to coerce the REFANY to its actual (REF) type in a safe way. TYPECASE can be used for this, but is somewhat verbose:

```
TYPECASE ref OF
| T(x): <empty statement sequence>
| ELSE  <programming error>;
END;
```

NARROW is a standard procedure provided for this common case.

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

For example,

```
x := NARROW(ref, T);
```

is equivalent to

```
TYPECASE ref OF
| T(x):
| ELSE RAISE(System.NarrowFault, ref);
END;
```

The first parameter to NARROW must be a REFANY, and the second parameter must be the name of a REF type.

1.4 Opaque Types

In a DEFINITION module, Modula-2 allows the programmer to define an abstract type and to specify 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 a type is called "opaque," but unfortunately it is somewhat translucent: the concrete representation must be either a pointer or a subrange and must occupy one WORD. Indeed, Modula-2 allows variables of an opaque type to be created (e.g., via NEW), assigned, copied, etc. An assumption is evident here, namely that simple assignment and equality operations suffice for opaque values. This is often not the case. Consider EQ and EQUAL in LISP, and reference-counted assignment for some forms of garbage collection.

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 contain REFs. But REFs require a non-standard assignment operation because our garbage collector is based on reference-counting. The following design addresses these problems.

OPAQUE

The opaque type facility defined in standard Modula-2 has been extended to include the new form:

```
TYPE Object = OPAQUE;
```

This form is generally used in conjunction with a REF type as follows:

```
TYPE Handle = REF Object;
```

An OPAQUE type provides no operations to its client, and therefore instances of the type cannot be allocated, assigned, passed as actual parameter, or compared against each other. Nor can it be a field of a record or element of an array, even in a local variable.

The corresponding concrete type, however, is not constrained to be a pointer or a subrange, as in standard Modula-2. (For reasons related to consistency among runtime types, it is constrained to be a "type constructor," however. See section 2.1 for a list of the type constructors in Modula-2+.)

Consider the following example:

```
DEFINITION MODULE Threads;

TYPE Object = OPAQUE;
TYPE Thread = REF Object;

PROCEDURE Fork(PROCEDURE(REFANY): REFANY, REFANY): Thread;

PROCEDURE Join(Thread): REFANY;

(* . . . *)

END Threads.
```

The client of Threads cannot manufacture a Threads.Object and cannot even dereference a Threads.Thread to acquire one. However, the implementation has complete access to Threads.Objects.

```
IMPLEMENTATION MODULE Threads;

TYPE
  Object = RECORD
    next: Thread;
    state: ThreadState;
    (* other fields *)
  END;

TYPE ThreadState = (Unborn, Alive, Dying, Dead);

PROCEDURE Join(t: Thread): REFANY;
  BEGIN
  IF t^.state = Unborn THEN (* . . . *) END;
  (* . . . *)
  END;

(* . . . *)

END Threads.
```

1.5 Concurrency

In the "Processes" module, standard Modula-2 provides a small and simple facility at a high level of abstraction for dealing with concurrent, cooperating threads of control. Unfortunately, for the reasons cited below, it is too limited to support our needs. See (LAM80) for a deeper discussion of these issues.

(A note on terminology: the word "process" carries many conflicting connotations. To avoid confusion, we use the word "thread" instead of "process" to mean a thread of control with its associated stack.)

(1) In standard Modula-2, mutual exclusion is provided via implementation modules that are specially marked as "monitors." The system guarantees that only one thread at a time can be executing in any one of a monitor's procedures. The number of monitors, hence the number of data structures that can be protected via mutual exclusion, must therefore be a load-time constant. A programming style that depends on the dynamic creation of objects, each with independent protection against concurrent access, is precluded. Many of our programs will require such "object-style" monitors.

(2) In standard Modula-2, synchronization is provided 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 is resumed, 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. Though the design is adequate if the implementation is based on co-routines, it is not workable for multi-processors.

A central question is:

> Must Wait and Send be called only from within a monitor that protects both the shared data and the signal?
>
> If not, there is a critical race: a signal may be sent and discarded just as a thread is about to wait for it. Such signals will be lost. Deadlock or erroneous execution will ensue.
>
> If so, Wait must simultaneously suspend the thread and exit the monitor, then re-enter the monitor when the thread is awakened some time later. The semantics of Send are less clear; there are at least the following two options: the simpler design would awaken a waiting thread, which cannot enter the monitor until the thread that does the Send exits. The sender must not change shared data after doing the send and before it exits the monitor. A more complicated design for Wirth's Send would waken a waiting thread, exit the monitor, reschedule itself to allow

the other thread to run, then re-enter the monitor.

Fortunately, there is a better solution to this set of problems if clients are able to deal with "spurious wakeups." The design outlined below can be implemented simply and efficiently for a multi-processor without introducing restrictions based on scheduling "priority."

(3) There are other subtle interactions between monitors and signals. Programming all but the simplest applications with the primitives in the "Processes" module is quite tricky and error-prone. Fortunately, only a few programming idioms cover most of the tricky cases. These include passing a parameter to the root procedure of a new thread; synchronizing one thread with the termination (maybe with a result) of another; invoking WAIT from inside a monitor; and identifying critical sections. It is well worthwhile to provide explicit support for these cases.

The Threads Module and the LOCK Statement

Most of the facilities in Modula-2+ for programming with concurrent threads of control (sometimes called "lightweight processes") are provided by the "Threads" module. Three notions are important: Thread, Mutex, and Condition.

A thread of control (Threads.Thread) is created by Fork:

```
TYPE Forkee = PROCEDURE(REFANY): REFANY;

PROCEDURE Fork(Forkee, REFANY): Thread;
```

Fork creates a new thread of control within the caller's address space and causes it to call the indicated Forkee with the indicated REFANY. When the Forkee returns, its result may be acquired by Join:

```
PROCEDURE Join(Thread): REFANY;
```

Join synchronizes with the specified thread, waiting, if necessary, until the Forkee returns. A thread that has returned from the Forkee and is not referenced (e.g., by an outstanding Join) will be reclaimed by the garbage collector.

Threads can be synchronized explicitly by means of Mutexes:

```
TYPE Mutex;

PROCEDURE Acquire(VAR mutex: Mutex);
        (* If mutex is "available," mark it "unavailable"
           and continue. Otherwise, wait for it to become
           "available" and try again then. Do all this
           atomically with respect to other threads *)

PROCEDURE Release(VAR mutex: Mutex);
        (* Mark the mutex "available" and continue. This
           may cause other threads waiting in Acquire to
           be awakened *)
```

The language provides the following syntax for mutual exclusion:

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

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

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

Threads use Conditions to notify one another of potentially interesting state changes.

```
TYPE Condition;

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

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

Wait is called from a thread that holds the mutex. It causes the thread to block and the mutex to be released. When the thread is subsequently awakened, it re-acquires the mutex (this may require first blocking on the mutex).

Broadcast and Signal are generally 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. Thus, Signal is more efficient if your algorithm does not require waking multiple threads. In case of doubt, use Broadcast instead.

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 released 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 imply that a return from a Wait should be viewed as merely a hint that some action may need to be taken. It suggests that the matter should be re-considered, and does not guarantee (as in some schemes) that action is required. Therefore, Waits should occur in WHILE loops of the form shown below. As long as programs treat return from a Wait as a hint, extra Signals can affect performance, but not correctness.

Finally, mutexes and conditions must be initialized using the InitMutex and InitCondition procedures from the Threads module. If they are not properly initialized in this way, chaos will likely ensue. Mutex and condition values should never be copied, assigned, passed as value parameters, or otherwise used explicitly. All operations on mutexes or conditions take them as VAR parameters.

The following small program illustrates the use of these constructs:

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

CONST BufferMax = 10;

TYPE
  BufferIndex = [0..BufferMax-1];
  BufferObject = RECORD
    mutex: Threads.Mutex;
    in, out: BufferIndex;
    contents: ARRAY BufferIndex OF INTEGER;
    nonFull, nonEmpty: Threads.Condition
  END;
  Buffer = REF BufferObject;
```

```
PROCEDURE Produce(buffer: Buffer; i: INTEGER);
  BEGIN
    LOCK buffer^.mutex DO
      WHILE (buffer^.in+1) MOD BufferMax = buffer^.out DO
        Threads.Wait(buffer^.mutex, buffer^.nonFull);
      END;
      buffer^.contents[buffer^.in] := i;
      buffer^.in := (buffer^.in+1) MOD BufferMax;
      Threads.Broadcast(buffer^.nonEmpty);
    END;
  END Produce;

PROCEDURE Consume(buffer: Buffer): INTEGER;
  VAR i: INTEGER;
  BEGIN
    LOCK buffer^.mutex DO
      WHILE buffer^.in = buffer^.out DO
        Threads.Wait(buffer^.mutex, buffer^.nonEmpty);
      END;
      i := buffer^.contents[buffer^.out];
      buffer^.out := (buffer^.out+1) MOD BufferMax;
      Threads.Broadcast(buffer^.nonFull);
    END;
    RETURN i;
  END Consume;

PROCEDURE Producer(x: REFANY): REFANY;
  VAR
    buffer: Buffer;
    i: INTEGER;
  BEGIN
    buffer := NARROW(x, Buffer);
    FOR i := 1 TO 10000 DO Produce(buffer, i) END;
    Produce(buffer, 0);
    RETURN NIL;
  END Producer;

PROCEDURE Consumer(x: REFANY): REFANY;
  VAR
    buffer: Buffer;
    i: INTEGER;
  BEGIN
    buffer := NARROW(x, Buffer);
    REPEAT i := Consume(buffer) UNTIL i = 0;
    RETURN NIL;
  END Consumer;
```

```
(* sample main program *)

VAR
  b: Buffer;
  p, c: Threads.Thread;
  dummy: REFANY;
BEGIN
  NEW(b);
  WITH b^ DO
    in := 0; out := 0;
    Threads.InitCondition(nonEmpty);
    Threads.InitCondition(nonFull);
    Threads.InitMutex(mutex);
  END;
  p := Threads.Fork(Producer, b);
  c := Threads.Fork(Consumer, b);
  dummy := Threads.Join(p);
  dummy := Threads.Join(c);
END ProCon.
```

1.6 Interface Extensions

Standard Modula-2 presents a particular model of the ways in which parts of a system should fit together. Most notably, the language requires that an implementation module export exactly one "interface" (definition module) and that an interface be exported by exactly one module; indeed, the Modula-2 book says (p. 81) that an implementation module and corresponding definition module constitute a unit, and that the definition module should be thought of as a "prefix to the implementation part" of the unit.

These limitations severely restrict the designer of non-trivial packages, which often are best organized with multiple implementation modules collectively exporting multiple interfaces. Accordingly, the declaration syntax and semantics in interfaces have been extended to permit this kind of package structure.

We extend a "definition" (see syntax lines 90-93 in WIR82) as follows:

```
definition =
  . . .    (existing forms are unchanged |
  VAR {ident ":" type "=" qualident ";"} |
  ProcedureHeading "=" qualident ";" |
  EXCEPTION {ident ["(" qualident ")"] "=" qualident ";"}
```

The intent of these new declaration forms is to equate the identifier being defined with the one to the right of the equal sign. For the VAR declaration, the type on the right-hand side must "equal" the type on the left-hand side in the sense defined in section 2, below. For the PROCEDURE declaration, the types of the two sides must be procedure types that are "redefinable" in the sense defined in section 2. For the EXCEPTION declaration, the right-hand side must name an EXCEPTION that has a parameter (the parenthesized qualident) if and only if the left-hand side does and, if a parameter is present, the types must be equal. Also note that the existing syntax for TYPE and CONST declarations already permits a qualident to appear on the right-hand side.

On the surface, these new declaration forms would seem to have little to do with the need for multiple export, multi-module implementations mentioned earlier. However, we can achieve the effect we want by having each module of a package export a single interface, and then "layering over" this interface with one or more others that effectively regroup the declarations. Such "umbrella" interfaces consist entirely of definitions of the form

```
n = mod.n
```

1.7 Open Arrays

Standard Modula-2 provides a mechanism for the manipulation of arrays when the number of elements is not known until runtime. But it restricts such "open arrays" to be formal parameters of procedures; the programmer cannot create new open array variables or allocate open array objects.

In unsafe programs, using loopholes, the programmer can use System.Allocate and address arithmetic in place of subscripting to circumvent this restriction, but not in safe programs. Accordingly, we extended the use of open arrays as follows:

Modula-2+ allows open arrays as types. For example,

```
TYPE Array: ARRAY OF ElementType;
```

In addition to their use in standard Modula-2, open array types can be used in the declaration of REF types. NARROW and TYPECASE can be used with such REF types.

```
TYPE ArrayRef: REF Array;
```

The form of NEW used with variables of type ArrayRef takes the number of elements as its second parameter. NEW returns NIL if the number of elements specified is zero.

```
VAR a1, a2: ArrayRef;
NEW(a1, 613);  a2 := NIL;
IF a2 = NIL THEN NEW(a2, NUMBER(a1^));
```

The dereferencing operator "^" following a REF to an open array can appear in three contexts: as an actual array parameter corresponding to an open array formal parameter with a matching element type, as a "designator" preceding a bracketed subscript expression, or as an actual parameter to the standard procedures HIGH and NUMBER.

Example, continuing the code above:

```
VAR et: ElementType;
PROCEDURE Sum(a: ARRAY of ElementType): ElementType;
  .
  .
  .
et := Sum(a1^);
```

Other Example:

```
PROCEDURE Equal(a1, a2: ArrayRef): BOOLEAN;
  VAR i: CARDINAL;
  BEGIN
    IF a1 = a2 THEN RETURN TRUE END;
    IF NUMBER(a1^) # NUMBER(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;
```

1.8 Low-Level Control of Data Size and Layout

Generally, the elements of records and arrays are laid out sequentially in an imaginary bit-addressed memory. To improve performance, the compiler attempts to align elements on byte or word boundaries.

For most programs, this works well. But for programs that control devices, or ones with critical performance requirements, or ones that deal with binary data from files or communication networks, it is sometimes necessary for the programmer to specify the precise layout of data in memory.

The Modula-2+ declarations

```
TYPE
     T = BITS 4 FOR [0..15];
     B = BITS 1 FOR BOOLEAN;
```

specify that each datum of type T occupies four bits; of type B, one bit. Operations that apply to the subrange [0..15] also apply to type T; ones that apply to BOOLEAN also apply to type B.

The general form of this new type constructor is:

```
BITS ConstExpression FOR type
```

It creates a new type with the specified size.

When allocating records and arrays, the compiler lays out elements of BITS types as they appear, without attempting to adjust their alignment. For example, "packed" types would be declared thus:

```
TYPE AT = ARRAY IndexType OF T;
     BT = ARRAY IndexType OF B;
     R = RECORD a,b,c,d: B; t: T END; (* size = 8 bits *)
```

1.9 Miscellany

To make type system breaches more evident, the added construct

```
LOOPHOLE(<expression>, <typename>)
```

is semantically equivalent to the standard Modula-2 "type transfer function" <typename>(<expression>).

The relational operators equal and unequal have been extended to apply to procedure types.

The syntax has been extended to allow empty cases in CASE, TYPECASE, and TRY ... EXCEPT statements. This allows a vertical bar to appear before the first case and before the optional ELSE clause.

Constant expressions have been extended to include applications of the following standard procedures:

```
FIRST, LAST,
HIGH, NUMBER (with fixed size arrays),
```

and the following procedures from the standard System interface:

Size, TSize, ByteSize, TByteSize

Efficient runtime libraries for dealing with garbage-collectible strings of characters (called "Texts") and associated streaming functions have been provided. The compiler has been extended to recognize and open-code selected performance-critical operations from these and from the Threads interface.

2. Type-Checking

Modula-2+'s rules for type-checking are straightforward but occasionally restrictive. Unfortunately, these rules are not stated precisely in the Modula-2 book, but a precise specification is required for effective and efficient use of the language.

This section introduces the simple ideas underlying the Modula-2+ type system; Appendix B contains a precise description of the Modula-2+ type-checking rules and applicability charts for the binary infix operators of the language.

2.1 Names and Uniqueness

Modula-2+ comes with a pre-defined collection of unique types, called "standard" types. Other unique types are created via the type contsructors of the language, which are:

```
ARRAY ...
RECORD ...
SET ...
POINTER ...
REF ...
PROCEDURE ...
(identList)                    (* enumeration *)
[constExpr..constExpr]         (* subrange *)
BITS ...                       (* type with bit size *)
```

A program identifies a type either by naming a previously defined type, by naming a new "opaque" type, or by writing a type constructor. A name is required for any type that is mentioned more than once, because, in general, two applications of the same type constructor to the same arguments return distinct types.

Names of the standard types are:

INTEGER, CARDINAL, UNSIGNED, CHAR, REAL, LONGREAL, BITSET, BOOLEAN, PROC, REFANY

Names of types that are defined in the standard interfaces are:

System interface: Address, Word, Byte, Process, FailArg;
IO interface: File.

Each type named above is unique.

Each syntactic occurrence of a type constructor produces a new unique type.

Normally, one defines a name for a type with a declaration of the form

```
TYPE name = type;
```

but in a DEFINITION module one can also define a name for a new "opaque" type via a declaration of the form

```
TYPE name;                (* standard Modula-2 *)
```

or of the form

```
TYPE name = OPAQUE;       (* Modula-2+ *)
```

The OPAQUE type specification is completed in the corresponding implementation module by repeating the name in a normal type declaration; such a declaration specifies what is called the "concrete" type. See section 1.4 for more information about opaque types.

2.2 Predicates Used for Type-Checking

The following predicates are used for type-checking: EQUAL, COMPATIBLE, ASSIGNABLE, PASSABLE, and REDEFINABLE.

EQUAL is used to distinguish unique types.

COMPATIBLE is used to check compatibility between the operands of binary infix operators (e.g., +, =, IN) and between the selector expression and the arm labels in a case statement. COMPATIBLE is also used by the ASSIGNABLE predicate.

ASSIGNABLE is used to check the compatibility of:
- the left and right sides of an assignment statement;
- a subscript expression with an array index type;
- a return expression with a procedure result type;
- an argument to Raise with an exception parameter type;
- IF, WHILE and REPEAT control expressions with BOOLEAN;
- and the TO and FROM expressions of a FOR statement with the type of its index variable.

ASSIGNABLE is different from COMPATIBLE: for example, it disallows assignment to a constant. COMPATIBLE treats constants and non-constants symmetrically.

ASSIGNABLE is also used by the PASSABLE predicate, which determines whether a given actual parameter may be passed as a

given formal parameter.

REDEFINABLE is used to check the compatibility of:
- a ProcedureHeading in a definition module with the corresponding ProcedureDeclaration in its implementation module;
- and a ProcedureHeading in a definition module on the left-hand side of an equated definition with the ProcedureHeading named by the right-hand side.

Appendix B contains precise definitions of these predicates.

3. Programming Conventions

In this section, we offer some hints (with rationale) about how to avoid some common pitfalls in the use of Modula-2+. This is not intended to be a complete style manual for Modula-2+ programming; rather, it includes the few most striking items that we encountered when first learning Modula-2.

Thoughtful choice of conventions is important. A well-designed programming language will make it easier to write good programs if its facilities are used with taste and understanding. By adopting a style that exploits the language's strengths and circumvents its weaknesses, one can do a better and less frustrating job of applying it to the task at hand. And it is better to apply a consistent style uniformly from the outset than to evolve an inconsistent one in response to bad experiences. The notes on programming conventions in this section and on formatting conventions in the next derive from the application of our experience programming in similar languages to the specific features and restrictions of this language.

3.1 Interfaces

(a) Modula-2 provides no way to specify that a variable in a DEFINITION module is meant to be read only by clients. Furthermore, client access to such a variable is unsynchronized with actions of the implementation. For these reasons, DEFINITION modules should contain variables (VARs) only when synchronization is not an issue; otherwise the variables should be declared in the corresponding implementation, and accessed by clients through procedures that read, change or return their values.

(b) Modula-2 limits procedures to at most a single return value. VAR parameters are specifically intended to enable procedures to return multiple values. They also can be used to reduce the overhead of passing large "value" parameters. However, since the language provides no syntax to distinguish these conceptually different applications of VAR parameters, a clarifying comment will help the reader to understand the procedure's semantics.

Example:

```
PROCEDURE Sum(VAR a (*inout*), b (*in*): Matrix);
```

3.2 Implementations

(a) In an implementation module, imported facilities should generally be referenced with qualified names. This makes code explicit about its non-local dependencies, hence easier to read. For example, RealFns.Sqrt and RealFns.ArcTan should be used as follows:

```
IMPORT RealFns;
...
x := RealFns.ArcTan(RealFns.Sqrt(3.0));
```

rather than

```
FROM RealFns IMPORT Sqrt, ArcTan;
...
x := ArcTan(Sqrt(3.0));
```

However, there is room for individual preference here. In a program dominated by REAL numbers, the latter form might be easier on the reader, since the function names are unlikely to be confused with functions from other interfaces. Similarly, heavily used procedures from standard I/O interfaces might be referenced in the second style. If the program text would become greatly cluttered with a repeated interface name, use the second form to eliminate the repeated name. If you have any doubt, however, use the first form.

(b) Use explicit subranges of INTEGER wherever possible to make the intended semantics clear. In Modula-2+, the type UNSIGNED is defined as [0..2^32-1], the same as standard Modula-2's CARDINAL, and CARDINAL is defined as the non-negative subrange of INTEGER [0..2^31-1].

(c) There are times when the Modula-2 type system is inadequate for expressing a programmer's intentions. A mechanism for circumventing the restrictions imposed by the compiler in such situations is required. But breaches of the type system often lead to subtle bugs that are difficult to find; such breaches should be used with care, only when necessary, and should be localized. To make the type breach easy to notice, use LOOPHOLE rather than a "type transfer function."

Similarly, implementations should be SAFE whenever possible. It is best to consider unsafe constructs as akin to type system breaches.

(d) The WITH statement makes record field names accessible without qualification and can therefore mask variables in

enclosing scopes. Since the masking is implicit (the field names do not appear locally to remind the reader what is happening), subtle errors can occur if a significant amount of code appears inside a WITH statement.

We find it best to use the WITH statement only for replacing all or most of the fields of a record. In essence, this facility should be treated as a record constructor combined with an assignment statement.

(e) Data that needs to be protected by one or more mutexes should generally be organized in an appropriate data structure using the type system of the language. This is sometimes called a "monitored object" style and is worthwhile even when only a single instance of the "object" is necessary. The alternative approach (simply listing the variables in the outer scope of a module) is inferior because it fails to indicate clearly which variables are "monitored" and which are not.

Appendix A

Collected Syntax

The syntax below follows the conventions used in Appendix 1 of "Programming in Modula-2, Second Edition" by Niklaus Wirth. Numbers to the left of each rule refer to production numbers in that Appendix (rules without numbers are extensions for Modula-2+). Additions to a rule are indicated by an ellipsis.

A.1 Exceptions and Finalization

```
 41  ProcedureType = PROCEDURE [FormalTypeList] [RAISES raisees].

 53  statement = [ ... | TryStatement | ... ].
     TryStatement = TRY StatementSequence TryTail END.

     TryTail = FINALLY StatementSequence |
       PASSING raisees | PASSING raisees ";" |
       EXCEPT [HandlerArm {"|" HandlerArm}]
         [ELSE ["(" ident ")"] StatementSequence].

     HandlerArm = [QualidList ["(" ident ")"] ":" StatementSequence].

     raisees = "{" [QualidList] "}".

     QualidList = qualident {"," qualident}.

 73  ProcedureHeading = PROCEDURE ident
       [FormalParameters] [RAISES raisees].

 75  declaration = ... |
       EXCEPTION {IdentList ["(" qualident ")"] ";"}.
 90  definition  = ... |
       EXCEPTION {IdentList ["(" qualident ")"] ["=" qualident] ";"}.
```

A.2 Safety

```
 24  type = ... | REF ident | REF type.

 96  CompilationUnit = [SAFE] DefinitionModule |
 97    [SAFE] [IMPLEMENTATION] ProgramModule .
```

A.3 Runtime Types

```
53  statement = [ ... | TypecaseStatement | ... ].
    TypecaseStatement = TYPECASE expression OF tcase
      {"|" tcase} [ELSE StatementSequence] END.

    tcase = [QualidList ":" StatementSequence] |
      qualident ["(" ident ")"] ":" StatementSequence.
```

A.4 Opaque Types

```
90  definition = ... |
91    TYPE {ident ["=" type | "=" OPAQUE] ";"}.
```

A.5 Concurrency

```
53  statement = [ ... | LockStatement | ... ].
    LockStatement = LOCK designator DO StatementSequence END.
```

A.6 Interface Extensions

```
90  definition = ... |
92    VAR {ident ":" type ["=" qualident] ";"} |
93    ProcedureHeading ["=" qualident] ";" |
      EXCEPTION {IdentList ["(" qualident ")"] ["=" qualident] ";"}.
```

A.7 Open Arrays

```
30  ArrayType = ARRAY [SimpleType {"," SimpleType}] OF type.
```

A.8 Miscellaneous Extensions

```
24  type = ... | BITS ConstExpression FOR type.

36  variant = [CaseLabelList ":" FieldListSequence].
65  case = [CaseLabelList ":" StatementSequence].
```

Appendix B

Type Checking Rules and Binary Operators

B.1 Predicates Used for Type-Checking

This section defines the type-checking predicates that are used by the compiler: EQUAL, COMPATIBLE, ASSIGNABLE, PASSABLE, and REDEFINABLE.

To facilitate the definitions, it is useful to introduce the following functions and names:

> BaseType: a function on types. BaseType(t) peels off layers of subrange and BITS specifications until it finds a type without these.
>
> ElementType: a function on ARRAY types. If X is an ARRAY type, then ElementType(X) is the type of the elements.
>
> t1, t2, src, srcB, dst and dstB: names for types. By convention,
>
> t1, t2, src and dst are arbitrary types,
> srcB is a name for BaseType(src),
> and dstB is a name for BaseType(dst).

EQUAL (See section 2.1 for a discussion of unique types)

Two types t1 and t2 are EQUAL if
- they are the same (i.e., unique) standard type;
- or they are the same type from a standard interface;
- or they are the same syntactic occurrence of a type constructor;
- or they are the same opaque type;
- or one is an opaque type
  - and the compiler is analyzing its implementation module;
  - and the other is the corresponding concrete type.

EQUAL is used to distinguish unique types.

COMPATIBLE

```
The types src and dst are COMPATIBLE if:
  srcB is EQUAL to dstB;
  or the (unordered) pair {srcB, dstB} is one of the following:
    {CARDINAL, INTEGER}
    {CARDINAL, UNSIGNED}
    {<a REAL constant>, REAL}            (* literals and constants
    {<a REAL constant>, LONGREAL}           are treated specially *)
    {<a CHAR constant>, CHAR}
    {<a CHAR constant>, ARRAY [...] OF CHAR}
    {<a string constant>, ARRAY [...] OF CHAR}
    {ADDRESS, CARDINAL}
    {ADDRESS, UNSIGNED}
    {ADDRESS, INTEGER}
    {ADDRESS, <a POINTER type>}
  or both srcB and dstB are PROCEDURE types;
    and both types have the same number of parameters;
    and the result types of srcB and dstB are EQUAL;
    and the set of exceptions raised by srcB is a subset of dstB's;
    and for each pair of corresponding parameters p1 and p2
      p1 and p2 are both VAR parameters or neither is;
        and either
          both p1 and p2 have EQUAL types;
          or both are open arrays with EQUAL element types.
```

COMPATIBLE is used to check compatibility between the operands of binary infix operators (e.g., +, =, IN) and (in a case statement) between the case selector expression and the case arm labels. COMPATIBLE is also used by the ASSIGNABLE predicate.

ASSIGNABLE

```
A value of type src is ASSIGNABLE to a variable of
type dst if
  src and dst are COMPATIBLE and dst is not a constant;
  or srcB is a REF type and dstB EQUALs REFANY;
  or both srcB and dstB are in {INTEGER, CARDINAL, UNSIGNED}.
```

ASSIGNABLE is used to check the compatibility of:
- the left and right sides of an assignment statement;
- a subscript expression with an array index type;
- a return expression with a procedure result type;
- an argument to Raise with an exception parameter type;
- IF, WHILE and REPEAT control expressions with BOOLEAN;
- and the TO and FROM expressions of a FOR statement with the type of its index variable.

ASSIGNABLE is also used by the PASSABLE predicate.

PASSABLE

An expression E of type src is PASSABLE as a
parameter of type dst if:
  the parameter is passed by value and src is ASSIGNABLE to dst;
  or the parameter is passed by var and dst EQUALs src;
  or dstB EQUALs System.Word and TSize(srcB) <= System.WordSize;
  or dstB EQUALs System.Byte and TSize(srcB) <= System.ByteSize;
  or either srcB or dstB EQUALs ADDRESS and the other is a POINTER;
  or dstB is an open ARRAY
    and srcB is an ARRAY;
    and ElementType(dstB) EQUALs ElementType(srcB);
  or dstB is an open ARRAY
    and BaseType(ElementType(dstB)) EQUALs CHAR;
    and E is a STRING or CHAR constant;
  or dstB is an open ARRAY
    and ElementType(dstB) is WORD or BYTE.

REDEFINABLE

Two procedure types t1 and t2 are REDEFINABLE if:
  t1 and t2 are COMPATIBLE;
  and the exceptions raised by t1 and t2 are the same;
  and corresponding parameters have the same name.

  REDEFINABLE is used to check the compatibility of:
    a ProcedureHeading in a definition module with the corresponding
      ProcedureDeclaration in its implementation module;
    and a ProcedureHeading in a definition module on the left-hand side
      of an equated definition with the ProcedureHeading named by
      the right-hand side.

B.2 Binary Infix Operators

The following charts present applicability rules for the binary infix operators, which label their rows. In all cases except IN, operands must be COMPATIBLE. Column headings, described below, characterize the pair of parameters to the COMPATIBLE predicate:

N	whole number: INTEGER, CARDINAL, UNSIGNED
A	System.Address
R	REAL, LONGREAL
S	SET
B	BOOLEAN
Ch	CHAR
E	enumeration
P	POINTER, REF, REFANY
St	string constant
ACh	ARRAY <bounds> OF CHAR

For the IN operator, e IN s (result: BOOLEAN), e must be COMPATIBLE with the range of set s. Charts for other operators:

relations (result: BOOLEAN)

	N	A	R	S	B	Ch	E	P	St	ACh
=	+	+	+	+	+	+	+	+	+	+
#	+	+	+	+	+	+	+	+	+	+
<>	+	+	+	+	+	+	+	+	+	+
>=	+	+	+	+	+	+	+			
>	+	+	+		+	+	+			
<=	+	+	+	+	+	+	+			
<	+	+	+		+	+	+			

arithmetic and set operators (the result has the operand type)

	N	A	R	S
+	+	+	+	+
-	+	+	+	+
*	+		+	+
/			+	+
DIV	+			
MOD	+			

Boolean operators (result: BOOLEAN)

	B
AND	+
&	+
OR	+

Appendix C

Formatting Conventions

It is a curious fact of programming life that practitioners consistently underestimate both the utility and the effort required to produce a complete, integrated, operational set of conventions for formatting programs. Such conventions address indentation, capitalization, spacing, etc.

Common formatting conventions are worth a lot, especially when several people work together on large projects. In addition to the communication inefficiencies caused by differing conventions, newcomers to a programming language often spend a significant amount of time incrementally developing and retrofitting their own style, usually re-learning what turn out to be simple lessons that others have already learned. While this is not always wasteful, it is clearly worthwhile to have a good set of guidelines at hand, if only for reference. Also, adherence to common conventions makes automatic formatting tools easier to provide and more useful.

This section offers a complete set of conventions for formatting Modula-2+ programs and interfaces: indentation, spelling and punctuation rules are presented, as well as guidelines for placing comments.

The following points of style produce a visually pleasing program. Consistently applied, they also provide syntactic cues to semantics that make a program easier to read.

C.1 Spelling

Identifiers are written entirely in lower case except as follows:

a. The initial letter of each embedded word except the first is capitalized. For the purposes of this rule, an embedded acronym is a sequence of one-letter words. To avoid possible conflicts with keywords (see below), do not use identifiers of length greater than one that contain only capital letters.

b. Identifiers that name modules, procedures, exceptions, types, and constants start with an upper-case letter. All of these are

compile-time constants. All variables, including procedure variables, start with a lower-case letter.

Thus, for example,

variableName	thisCPU
fieldName	status
ModuleName	Parser
ProcedureName	Insert
TypeName	VMCache
AnException	Overflow
ConstantName	WordSize
EnumerationElement	Offline

Note that the elements of an enumeration type are semantically similar to constants and are therefore capitalized.

c. Reserved words and standard identifiers (except interface names) are written entirely in upper case. Other capitalizations of the same word are available to the programmer for other purposes (for example, "set" and "lock" are perfectly good variable names).

d. Identifiers in standard interfaces (and the interface names themselves) follow the above rules (e.g., System.Word, IO.input). Thus, a programmer need not be concerned with whether the interface is "built-in" to the compiler or not.

e. Use "#" rather than "<>". Use "AND" rather than "&".

C.2 Punctuation

a. A space appears before and after vertical bar, and before and after equal signs in definitions and declarations. Spaces usually appear before and after binary operators (including assignment) when the statement containing them is long, but they may be omitted when the statement is short (e.g., i:=i+1). A newline sometimes appears in place of the space after these characters.

b. A space appears after colon, comma, and semicolon, but none before. A newline often follows semicolon (and sometimes comma) instead.

c. Except as required by adjacent tokens, no spaces appear before or after left and right parentheses, left and right square brackets, left and right curly brackets, caret (up-arrow), dot-dot, and period. (A newline follows the period at the end of a module.) If the token on the left (or right) of

these characters requires a space after (or before) it, one should appear. For example, consider the left parentheses in

```
PROCEDURE Positive(x: INTEGER): BOOLEAN;
```

and

```
TYPE Color = (Red, Green, Blue);
```

d. A space appears after left-comment and before right-comment.

e. A semicolon follows the last statement in a statement sequence and the last field in a field list; this makes insertions and deletions somewhat easier.

C.3 Indentation

Indenting is used to emphasize program structure. Each nesting level is two spaces wide. (If a formatter is available that parameterizes the width, four spaces can also be used.) The following illustrates the recommended form of each Modula-2+ construct ('ss' represents a statement sequence):

```
CONST
  A = 613;
  B = (1, 3, 7);

VAR
  var: Type1;
  x, y, z: Type2;
```

```
TYPE
  Index = [0..15];
  Object = RECORD
    f1: Type1;
    f2, f3: Type2;
  END;
  Handle = REF Object;
```

```
PROCEDURE P;
  VAR
    b: BOOLEAN;
    i: INTEGER;
  BEGIN
    ss
  END P
```

```
IF bool THEN
  ss
ELSIF bool THEN
  ss
ELSE
  ss
END
```

```
WHILE bool DO
  ss
END

FOR i := 1 TO 10 DO
  ss
END
```

```
REPEAT
  ss
UNTIL bool

LOOP
  ss
END

WITH d DO
  ss
END

LOCK d DO
  ss
END
```

```
CASE expr OF
| c1:
    ss
| c2:
    ss
| ELSE
    ss
END
```

```
TYPECASE ra OF
| t1(v1):
    ss1
| t2(v2):
    ss2
| ELSE
    ssn
END
```

```
TRY
  ss
EXCEPT
| e1(v1):
    ss1
| e2(v2):
    ss2
| ELSE
    ssn
END
```

```
TRY
  ss
PASSING {e1,e2}
END

TRY
  ss
FINALLY
  ss
END
```

```
MODULE Impl;
IMPORT I1, I2;
CONST C = 47;
VAR v: INTEGER;
PROCEDURE P;
  BEGIN
  END P;
BEGIN
  ss
END Impl.
```

A statement sequence is indented under the construct that introduces it, which lines up vertically with its corresponding END. Note also the similarity of the case discriminations in CASE, TYPECASE, and TRY ... EXCEPT. Declarations are indented

one level, including declarations of nested procedures. (Exception: the declarations in the outermost module are not indented; in a nested module, they would be.) If the declaration requires more than one line, its components are indented another level, with at most one type per line.

The above forms only apply to constructs that do not fit on a single line. For example, if the statement sequence following a THEN, ELSE, or case label is short (e.g., a single statement or a few short statements), it can appear on the same line with the tokens that introduce and terminate it. Similarly, if the statement sequence of a loop body is short (even non-existent), it can be moved up to the line that introduces the loop, along with the trailing END. Thus,

```
IF bool THEN x := y; y := z END;
FOR i := 1 TO 10 DO a[i] := 0 END;
CASE x OF
| 1..3: y := 0;
| 4: y := -1;
| ELSE y := z;
END
```

are all acceptable forms. The generalization here is: put the whole construct on one line, if it fits.

Long statements which require more than one line are broken where white space would normally appear, with the continuation lines indented two levels.

C.4 Comments

The text of a multiline comment begins on the same line as the opening left-comment. Subsequent lines are indented the same as the opening left-comment. The terminating right-comment appears on the last line of the comment.

```
(* A comment that fits entirely on one line by itself. *)

(* A long comment that does not fit on one line, and
the filler necessary to make it do so, and the filler
necessary to make it do so. *)
```

By convention, comments which refer to a group of items appear before the group. "Comment boxes" are often used to set off what is logically a section heading; they are constructed as follows:

```
(******************)
(*  Section Name  *)
(******************)
```

Comments associated with a single definition or declaration appear immediately after the definition or declaration (not before) and at the same level of indentation. In modules and in procedure implementations, they immediately follow the module (procedure) heading, which includes the import and export clauses. Comments in running code follow the normal flow of control.

When comment brackets are used to "comment out" a section of code, the terminating right-comment appears on a line by itself, lined up vertically with the opening left-comment.

C.5 Interfaces

A few additional guidelines make interfaces, which represent the most heavily read code, a bit more uniform from one to the next.

Procedures with more than two parameters of different types should have their parameters listed on separate lines; items should be grouped logically on each line. Each procedure should be immediately followed by a comment describing its operation (if it is not evident from the procedure's name). A blank line separates procedure declarations.

```
PROCEDURE ProcName(
  parm1: Type1;
  parm2, parm3: Type3;
  VAR parm4: Type4)
  : ReturnType
  RAISES {E1, E2};
  (* Short, one-line description of the procedure.
  More information, if necessary, appears here, in
  multiline comment format. *)

PROCEDURE NextProc ...
```

A general comment describing the interface as a whole should appear immediately following the module header, before any definitions.

Examples of these conventions appear throughout this report.

REFERENCES

ADA82

US Department of Defense, "Reference Manual for the Ada Programming Language," AdaTEC, July 1982

BIR73

Birtwistle, G., et al., "Simula Begin," Auerbach, Philadelphia PA, 1973

DEU80

Deutsch, L. P., and Taft, E. A., "Requirements for an Experimental Programming Environment," Xerox PARC Technical Report CSL-80-10, June 1980

LAM80

Lampson, B. W., and Redell, D. D., "Experiences with Processes and Monitors in Mesa," CACM 23,2, Feb 1980

LEV77

Levin, R., "Program Structures for Exceptional Condition Handling," Department of Computer Science Technical Report, Carnegie-Mellon University, June 1977

LIS81

Liskov, B., et al., "CLU Reference Manual," Lecture Notes in Computer Science 114, Springer-Verlag, 1981

GOL83

Goldberg, A., and Robson, D., "Smalltalk-80: The Language and Its Implementation," Addison-Wesley, 1983

MIT79

Mitchell, J. G., et al., "Mesa Language Manual," Report CSL-79-3, Xerox PARC, Palo Alto CA, 1979

POW84-1

Powell, M., "Modula-2: Good News and Bad News," Proc. CompCon, Spring 1984

POW84-2

Powell, M., "A Portable Optimizing Compiler for Modula-2," Proc. SIGPLAN Compiler Construction Conference, June 1984

TEI78

Teitelman, W., et al., InterLISP Reference Manual, Xerox PARC, Palo Alto CA, 1978, third revision

TEI84

Teitelman, W., "A Tour Through Cedar,"
IEEE Software, April 1984, Volume 1, Number 2

WIR82

Wirth, N., "Programming in Modula-2," New York, NY,
Springer-Verlag, 1982

Top half of the page is marked with an "a," bottom half with a "b," whole page with neither.

digital Systems Research Center Reports

1: "A Kernel Language for Modules and Abstract Data Types"
R. Burstall and B. Lampson

2: "Optimal Point Location in a Monotone Subdivision"
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301