Multicomputer Programming with Modula-3D

K. Rustan M. Leino

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125, U.S.A.
rustan@cs.caltech.edu

4 June 1993

Abstract

In this note, we extend an object-oriented language to support programming fine-grain multicomputers. The new constructs have a simple semantics and provide a nice way to write distributed programs. The semantics of the constructs are independent of how a program is distributed. We also show a set of simple conditions under which even the outcome of a program is independent of how its control and data are distributed. We show some strengths and weaknesses of the programming model, and describe and critique our implementation of the language.

Acknowledgements

I foremost want to thank my research advisor, Jan van de Snepscheut, for the support and guidance he has provided and the many constructive comments and suggestions he has bestowed on this report.

I also want to thank Greg Nelson and Bill Kalsow at DEC SRC for the helpful insights they have shed and the many questions they have answered.

The comments resulting from discussions with Peter Hofstee and Robert Harley in our research group meetings have been most helpful and are much appreciated.

Last, but not least, I want to thank India for her constant loving support.

Master's thesis provided as partial fulfillment of the degree of Master of Science.

The research described herein was supported in part by the DEC System Research Center.

Parts of this report appear in [19].

Technical Report Caltech-CS-TR-93-15.

Contents

0 Introduction **0**

0.0 Motivation 0

0.1 Why Modula-3? 1

1 Language Extensions **2**

1.0 Major Ingredients 2

1.1 Modula-3D Definition 4

1.2 Design Decisions and Alternatives 7

2 Programming Model **14**

2.0 Simulating Channels 14

2.1 Distributing a Program 16

2.2 Abstraction and Reusability 21

2.3 Proof Model 27

2.4 Selection 30

2.5 Dynamic Computations 31

2.6 Summary 32

3 Implementation **34**

3.0 Overview 34

3.1 Mosaic Modula-3D 36

3.2 Message Types 36

3.3 Interrupt Handler 37

3.4 RPC Protocol 39

3.5 Call and Return Descriptions 40

3.6 Marshalling and Unmarshalling 41

3.7 Reference Counts and Garbage Collection 43

3.8 Remote NEW Calls 44
3.9 Reference Counting Correctness 46
3.10 Enhancements 46
Remote Procedure Calls 46
Interprocessor Communication 47
Method Suites 48
3.11 Performance 48
Size 48
Speed 51
3.12 Possible Improvements 52

4 Conclusion **56**

Chapter 0

Introduction

0.0 Motivation

Due to their "modest cost and open-ended expandability", fine-grain multicomputers are an increasingly more attractive means for performing large computations ([27, 26]). Typical interprocessor communication operations provided by languages used to program such machines are send and receive. Forcing the programmer to deal with the details of each communication, these operations hinder the programmer from thinking about good and nice solutions for the problem at hand. As [27] states it, "because we have tacked these primitives onto programming languages simply as external functions, the process code is unnecessarily baroque." Instead, by providing in a programming language a different means of doing interprocessor communication, one can return the programmer's attention to the problem at hand.

Object-oriented languages generally facilitate a higher level of abstraction and expressiveness from which the multicomputer programmer can benefit. However, current object-oriented languages do not provide the right set of constructs to make an efficient distributed implementation possible, because there is no way to express the locality of data. Hence, there is a need to find a good set of object-oriented language constructs to rectify the situation. Such constructs must

- have clear semantics that is simple enough for the programmer to understand and use,
- admit an efficient distributed implementation,
- provide a nice way to write distributed programs, and
- admit easy performance tuning.

In this note, we introduce some extensions to Modula-3 that not only fulfill the above requirements, but also posses another very important property: we can give a few simple conditions under which the outcomes of our programs are independent of how their computations and data are distributed. We call the result, a distributed Modula-3, Modula-3D. The extensions, or variations thereof, may be applied to other languages.

We proceed by first justifying our choice of starting point, Modula-3. In Chapter 1, we describe the extensions and discuss some design alternatives thereof. In Chapter 2, we explore the Modula-3D programming model, and point out some of its strengths and weaknesses. In Chapter 3, we describe some interesting parts of our Modula-3D implementation found on the Mosaic multicomputer at Caltech, and show some performance figures. Critiquing our implementation, we mention some improvements and experimental changes that can be made. Finally, we offer some concluding remarks.

0.1 Why Modula-3?

We have chosen Modula-3 as starting point because it is a language already equipped to support object-oriented as well as concurrent programming. Thus, our efforts can be focussed on the distributed constructs, rather than on defining a whole new language from scratch.

When writing programs for a fine-grain multicomputer, memory management is of great concern. For example, in the Caltech Mosaic multicomputer with its designed 16 K nodes (256 of which are currently present), each node has only 64 KB RAM (see [28, 30, 29]). Therefore, it is convenient to have the language, rather than the programmer, assume responsibility of heap management. Modula-3 is geared to allow language implementations to feature a garbage collector.

Our language extensions involve types. Here, Modula-3 provides type safety.

Last, but not least, for a language that provides object types and concurrency (and much more), Modula-3 is quite simple. The semantics of Modula-3 is concise and well defined. (See [23] or [6], an exceptionally well-written report.) Since our goal is to provide something the programmer can understand, Modula-3 fits right in.

An alternative starting point would have been C++. This route is chosen in, for example, Compositional C++ (see [8, 7]). However, as C++ is, for one, not a simple language (see [10]), our results are not going to be simple either.

A different version of network objects has been implemented at the DEC Systems Research Center (see [32]). These are implemented in Modula-3, and are more general than those presented here; for example, objects may be shared between different programs. However, the design goals and assumptions of the DEC SRC network objects differ from ours. Moreover, as the DEC SRC network objects are implemented using Modula-3, rather than as extensions to Modula-3, they are not as convenient to use as those presented here. For example, more types are needed, and network objects are created differently from other objects.

Another attempt at combining object-oriented and concurrent programming is the Parallel Object-Oriented Language, POOL (see [25]). This language does not provide inheritance, so one may argue that, in spite of its name, POOL is not really an object-oriented language. In POOL, every object corresponds to a process; here, we treat these two separately.

Orca (see [0]) is another language that uses “shared data-objects”. As with POOL, Orca does not feature inheritance, which we will demonstrate to be useful. Using a variation of conditional critical regions (see [12, 2, 3, 4]) instead of Modula-3D’s monitor-like constructs (see [13, 4, 14, 5, 23]), Orca offers a simpler means of guarding atomic statements. However, the language does not allow interaction between objects to be done atomically, a feature we deem essential.

Chapter 1

Language Extensions

In this chapter, we present the language extensions. First, we state our requirements and introduce the major ingredients of our extensions. Then, we give the precise Modula-3D definition, in terms of differences from Modula-3 (see [6] or Chapter 2 of [23]). Finally, we discuss some design decisions that led to our definition of Modula-3D, and some alternatives that we considered.

1.0 Major Ingredients

We were careful to start with a relatively simple language and vowed to make our extensions as simple and few as possible while still providing the programmer with some easily understood constructs with which to write distributed programs conveniently. Rather than introducing explicit interprocessor communication, we introduce objects that may be shared among processors. Having decided that, we need to address three issues: we need

- a way to specify that objects of a type may be shared between processors,
- a way to distribute data, and
- a way to distribute control.

As for the first of these, we introduce *network object types*, whose values we call *network objects*. An object type is a network object type exactly when it is a subtype of a new built-in type called `NETWORK`. Like any object, a network object is a reference to a pair consisting of a data record and a method suite, and as for any object, a network object has one copy of its data record. The latter of these is suggested by the small amount of memory that the multicomputers we have in mind may have. Furthermore, it avoids cache consistency problems in language implementations.

As for distributing data, we introduce a new flavor of the familiar `NEW` call. When the type provided to `NEW` is a network object type, an extra parameter may be specified. This parameter specifies on which processor the data record of the new object is to be allocated. If omitted, the parameter defaults to the local processor, which reduces the difference between network objects and other references. Once allocated on a processor, the data record of an object will not be moved to another processor.

Finally, as for distributing control, for network objects whose data records reside on a remote processor, we let method invocations translate into remote procedure calls. We choose for these calls to be synchronous, meaning that they will not terminate until the method returns, so that the semantics of a method invocation does not depend on where the object whose method is being invoked resides.

The described extensions are not specific to Modula-3 and can be applied to other languages. Associated with the extensions are some restrictions. The restrictions tend to be more language peculiar, and may not be as simple if trying to extend languages that are not as simple as Modula-3.

To describe the most vital of the restrictions, we first introduce some terminology. The processor on which the data record of an object resides is called the object's *host*. A distinction is made between *local* and *remote* network objects. A network object is said to be local to its host, and remote to all other processors.

Methods may be invoked on any object, and their semantics is independent of where the object's data record resides. However, to make implementations of network objects feasible on machines where processors have separate memories, data fields of remote objects can be accessed via methods only. If network objects are used as abstract data types, this is what is done anyway.

Moreover, since only network objects are shared among processors, and since method invocation on these objects is the only way to do interprocessor communication, method invocation on and creation of network objects may not involve procedure values or references other than network objects. In particular, no method parameter, return value, or exception argument may be of a type that includes procedure values or references other than network objects. Similarly, since `VAR` and `READONLY` parameters involve aliasing of variables, only `VALUE` parameters are allowed. A network object type may certainly include fields of procedure types or reference types other than network object types. However, when a network object is created on a specified processor, the given bindings may not include those fields.

We now describe how programs are executed. The programmer writes one program, which is executed on every processor. Thus, variables declared at the outermost scope of the program text result in one copy per processor. These copies are not kept in synch—they are treated as different variables. All processors execute the initialization, which consists of the body of all modules except the main module. The initialization may not include the creation of any remote network object. Furthermore, a processor delays any remote request to create a network object until it has completed initialization. The language implementation designates one processor as the *main* processor. After the main processor completes its initialization, it executes the body of the main module. When other processors complete their initialization, they proceed by servicing requests to create network objects and to execute methods on these. Program execution terminates when the body of the main module terminates, regardless of execution of other threads on any processor. Implementations need also provide the threads on each processor with the ability to get the local processor ID. For example, Mosaic Modula-3D provides an interface `Processor` with a procedure `ID` which returns just that (see Section 3.1).

The semantics of a local and a remote method invocation are the same; however, their outcomes might in general differ, since they have access to different instances of global variables. We now describe the conditions under which the outcome of a program is independent of how it is distributed. In other words, under these conditions, the value of the processor parameter of `NEW` calls does not affect a program's outcome. The conditions, called the *distribution conditions*, are:

- data fields of network objects are accessed via methods only,

- no global variable is used in a program,
- the processor ID is used only to compute values for the processor parameter of `NEW`, and only valid processor IDs are used for this parameter, and
- the value of `Thread.Self()` is not used.

These conditions are sufficient, but not necessary. By following the conditions, a programmer can tweak the performance of a program, without changing the program's outcome, by changing the way network objects are distributed.

1.1 Modula-3D Definition

In this section, we present the Modula-3D language definition in terms of the Modula-3 definition found in [6] and [23]. The section numbers shown refer to those in [6]; those in [23] are similar.

2.9 Objects

After the second paragraph, add the paragraph:

> If the type is a subtype of `NETWORK`, it is said to be a *network object type*. Variables of such types are called *network objects*. A network object may be local or remote, as described in Section 6.9 New.

Append to third paragraph:

> If `o` is a network object, then `o.m(...)` is executed on the processor where `o`'s data record resides.

Append to fourth paragraph:

> If `T` is a network object type, then `T.m` refers to a value that, if invoked, has the effect of calling `T`'s `m` method locally.

As a new paragraph before the paragraph starting with "A method override", add:

> If `T` is a network object type, the parameters and exception arguments in `sig` may not include procedure types or reference types other than network object types. Consequently, the raises set may not by `ANY`. Furthermore, the parameters in `sig` are restricted to mode `VALUE`.

2.11 Predeclared opaque types

Replace with:

> The language predeclares the three types:

```
TEXT <: REFANY
MUTEX <: ROOT
NETWORK <: ROOT
```

which represent text strings, mutual exclusion semaphores, and network objects, respectively. These are opaque types as defined in Section 4.6 Opaque Types. The properties of `TEXT` and `MUTEX` are specified in the required interfaces `Text` (Section 9.1) and `Thread` (Section 9.2).

3.2 Procedure call

Replace the last paragraph with:

A procedure call can also have the form:

```
o.m(Bindings)
```

where `o` is an object and `m` names one of `o`'s methods. For local network objects, this is equivalent to:

(`o`'s `m` method) `(o, Bindings)`

For remote network objects, the former invokes `o`'s `m` method remotely, whereas the latter invokes a procedure locally passing `o` as a parameter, which is quite likely to result in a run-time error in the called procedure.

4.6 Opaque types

Append to the last paragraph, regarding the type declaration `TYPE T <: U` :

It is a static error if `U` is `ROOT` and `T` is a network object type that, in the current scope, is not revealed to be a subtype of `NETWORK`.

4.7 Revelations

Append to the third paragraph, regarding the partial revelation `REVEAL T <: V` :

It is a static error if `V` is `ROOT` and `T` is a network object type that, in the current scope, is not revealed to be a subtype of `NETWORK`.

5 Modules and interfaces

In the fourth paragraph, change the sentence "The effect of executing a program [...]" to:

The effect of executing a program is to execute the bodies of each of its modules, as described below.

Replace the sixth paragraph with:

A program is executed on every processor. Thus, variables declared at the outermost scope of the program text result in one copy per processor. These copies are not kept in synch—they are treated as different variables. All processors execute the initialization, which consists of the body of all modules except the main module. The initialization may not include the creation of any remote network object. Furthermore, a processor delays any remote request to create a network object until it has completed initialization. The language implementation designates one processor as the *main* processor. After the main processor completes its initialization, it executes the body of the main module. When other processors complete their initialization, they proceed by servicing requests to create network objects and to execute methods on these. Program execution terminates when the body of the main module terminates, regardless of execution of other threads on any processor.

Implementations need also provide the threads on each processor with the ability to get the local processor ID.

6.3 Designators

Replace paragraph explaining `o.f` with:

If `o` denotes an object and `f` names a data field specified in the type of `o`, then `o.f` denotes that data field of `o`. It is a checked run-time error if `o` is a remote network object. Otherwise, `o.f` is a writable designator whose type is the declared type of the field.

6.9 New

The first paragraph of this section states that the allocated type of the reference returned by `NEW(T, ...)` is `T`. There is no change to this paragraph.

At the end of the section, add:

If `T` is a network object type, the `NEW` operation has the form:

```
NEW(T, Processor, Bindings)
```

where `Processor` indicates on which processor the new object is to be allocated. The type of `Processor` is implementation-dependent, but may typically be `INTEGER`. It is a checked run-time error if `Processor` specifies a non-existing processor.

The allocation is said to be local if `Processor` indicates the current processor, and remote otherwise. If "`, Processor`" is omitted, the local processor is assumed.

If "`, Processor`" is present, `Bindings` cannot be provided for data fields that contain procedure values or references other than network objects.

8.2 Reserved identifiers

Add:

```
NETWORK
```

1.2 Design Decisions and Alternatives

In defining Modula-3D, several design decisions were made. In this section, we describe some of these decisions and their alternatives. We remind the reader that we are aiming for as few changes as possible, and for a language that is easy to use for the programmer.

In some programs, a thread may synchronize or communicate with several other threads near the same time. When these communications can be performed, possibly in parallel, without any computation in between, we call them *clustered communications*. (See Section 2.2 for an example.) Rather than performing one context switch per communication, one can gain efficiency if only one context switch is performed. In other words, there is no reason for the run-time system to reschedule the thread until all communications have been performed. As clustered communications cannot, in general, be detected by a compiler, it is nice if the programming language provides some construct with which to identify these. An example of a construct for this is parallel communication composition, written `//`, in mcc (see [11]). Modula-3D does not attempt to address this issue. To execute clustered communications in parallel, one needs to explicitly create one thread for each communication, and then join these threads, a clumsy solution indeed, and one in which the number of context switches has not decreased. It is not clear what the right way to define such a construct in a channel-less language is, and thus it is not part of Modula-3D, nor of Modula-3 for that matter.

The scarce memory of a fine-grain multicomputer suggests keeping only one copy of the data record of each network object. This differs from, for example, [17]. As described earlier, this also avoids cache consistency related problems. Consequently, Modula-3D defines the notion of local and remote objects. To facilitate the creation of remote network objects, Modula-3D features a remote `NEW` call,

```
NEW(T, Processor, Bindings)      .
```

Several choices were made in defining this call. We discuss two of them here.

The first choice regards the `Processor` parameter. This parameter determines where to allocate the data record of a network object. If omitted, the local processor is assumed. One alternative is not to introduce this parameter, and instead let the run-time system allocate the object on an appropriate processor. This idea was rejected because some processors may have special capabilities, like access to display memory, hardware floating point support, or a disk drive. A programmer then needs to be able to specify one of these. Moreover, the parameter allows the programmer to take advantage of locality in order to write more efficient programs.

Another alternative is to allow the `Processor` parameter, but let the placement of the allocation be up to the run-time system whenever the parameter is omitted. This may in fact be a better choice, but before we had tried running some actual programs, we thought it better to choose the simpler alternative. The programmer can, of course, specify `Processor` as a call to a procedure that returns the ID of a random processor. To efficiently implement some algorithm that distributes objects so as to balance memory consumption among the processors, altering the run-time system may be preferable because the run-time system can take advantage of compiler-generated information and machine-specific optimizations.

The second choice regards the type of the reference returned from remote `NEW` calls, and how a different definition can avoid introducing a new checked run-time error for when fields on a remote object are dereferenced. The data record of a remote object resides on a remote processor. Generally, the language provides two ways to access these fields: invoking a method or dereferencing the fields directly. We decided early on that method invocations are to be executed remotely. Doing the same thing for field dereferences would make implementations more complicated. Abstraction being a central part of object-oriented programming, these complications are unnecessary because fields tend to be dereferenced only within the object's methods. As the methods execute where the data record resides, only allowing fields to be dereferenced for local objects seems rational.

There is a choice in how to enforce this restriction. We may either introduce a new checked run-time error for this purpose or define the language so that the restriction is enforced by static errors and existing checked run-time errors. It turns out that Modula-3 allows for either extension, but we felt the former resulted in a language easier to understand for the programmer, so that is what we chose for Modula-3D. The rest of this section describes and discusses the latter of the two alternatives—a suggestion due to [24] and a solution similar to the one chosen in [32]. As the details of this alternative are quite intricate, a non-expert reader may want to skip the rest of this section on a first reading.

When a remote object of type `T` is created, Modula-3D specifies that the allocated type of the object returned from `NEW` is `T`. An alternative is to define `NEW` to allocate a data record and method suite of type `T` on the specified processor, but then return to that pair a reference whose allocated type is the *pure* type of `T`, where the pure type of an object type `T` is the supertype of `T`, closest to `T`, that does not have any data fields. Invoking methods on the returned object have the same effect as having invoked those methods on the remote object. We give an example to demonstrate how one writes programs using this. Consider the following program.

```
TYPE
  B = NETWORK OBJECT f: INTEGER METHODS m(): INTEGER := F END;
  D = B OBJECT g: INTEGER METHODS n(): INTEGER := G END;

PROCEDURE F( b: B ): INTEGER =
  BEGIN
    RETURN b.f
  END F;

PROCEDURE G( d: D ): INTEGER =
  BEGIN
    RETURN d.g
  END G;
```

The intention is that method `m` returns the value of field `f`, and `n` the value of `g`. Neither `B` nor `D` is a pure type; in fact, the pure type of both `B` and `D` is `NETWORK`. Hence, using the idea of pure types,

```
NEW( B, Processor, f := 5 )
```

returns a reference whose allocated type is `NETWORK`. As `NETWORK` does not have any methods, the value of `f` cannot be retrieved. Instead, the type hierarchy needs to be changed so that the pure types of `B` and `D` differ from `NETWORK`. For example,

```
TYPE
```

```
  A = NETWORK OBJECT METHODS m(): INTEGER END;
  B = A OBJECT f: INTEGER OVERRIDES m := F END;
```

solves the problem for B. Here, A emerges as an "abstract" type, meaning that the programmer does not intend to create new objects of that type; rather, A serves more as a placeholder in the type hierarchy. So then, what about D and its field g? One solution is the following.

```
TYPE
  A = NETWORK OBJECT METHODS m(): INTEGER; n(): INTEGER END;
  B = A OBJECT f: INTEGER OVERRIDES m := F END;
  D = B OBJECT g: INTEGER OVERRIDES n := G END;      .
```

However, this equips B with a method n that is never to be invoked. The problem is that we have only defined one pure type, and we want to distinguish between two types. Hence, we need to introduce another pure type, call it C. The declarations of the pure types will then be something like:

```
TYPE
  A = NETWORK OBJECT METHODS m(): INTEGER END;
  C = x OBJECT METHODS n(): INTEGER END;      ,
```

where x is some subtype of A. Clearly, B should be a subtype of A, and D of C. If we let B be a supertype of C, then C is no longer a pure type; hence, we choose A as the supertype of C:

```
TYPE
  A = NETWORK OBJECT METHODS m(): INTEGER END;
  B = A OBJECT f: INTEGER OVERRIDES m := F END;
  C = A OBJECT METHODS n(): INTEGER END;
  D = C OBJECT g: INTEGER OVERRIDES n := G END;      .
```

But now D no longer has a default for its m method, and nowhere in the data record of D is field f. This requires that f be added as a field to D as well, and that D also override method m:

```
TYPE
  D = C OBJECT f, g: INTEGER OVERRIDES m := DF; n := G END;

PROCEDURE DF( d: D ): INTEGER =
  BEGIN
    RETURN d.f
  END DF;      .
```

In general, every non-pure subtype of A needs to add f and override m. Alternatively, only f may be added and a default of m can be provided in A:

```
TYPE
  A = NETWORK OBJECT METHODS m(): INTEGER := AF END;
```

```
  B = A OBJECT f: INTEGER END;
  C = A OBJECT METHODS n(): INTEGER := CG END;
  D = C OBJECT f, g: INTEGER END;

PROCEDURE AF( a: A ): INTEGER =
  BEGIN
    TYPECASE a OF
      B( b ) => RETURN b.f
    | D( d ) => RETURN d.f
    END
  END AF;

PROCEDURE CG( c: C ): INTEGER =
  BEGIN
    TYPECASE c OF
      D( d ) => RETURN d.g
    END
  END CG;      .
```

This has the disadvantage of using the `TYPECASE` statement, which needs to be altered every time a new non-pure subtype of `A` is added.

In either case, new subtypes are added by adding a new pure type as a subtype of an existing pure type, and then adding a non-pure subtype of the new pure type. Then, the fields missing from all pure supertypes are added, and either methods are duplicated or a new case is added to the appropriate `TYPECASE` statements. As a consequence, an interface that contains the type declaration of `B` must also contain type `A` if it is expected that a client may want to extend `B`.

Suppose that the types `B` and `D` were initially written without being subtypes of `NETWORK`. One day it is decided that distributing these objects is a good idea. Then, in addition to constraints on parameters and the like, the type hierarchy now needs to be changed.

Finally, this scheme seems to rub the distinctions between local and remote objects in the face of the programmer, because the programmer needs to be aware of what the pure types of the available types are. As currently defined, Modula-3D does not present the programmer with such worries.

We now address the advantages of the pure type scheme. If `rp` is the ID of a remote processor, `NEW( B, rp )` allocates an object of type `B` on processor `rp`, and then returns an object of type `A`. The returned object may be some kind of local surrogate of the object allocated on `rp`. Thus, something like:

```
VAR
  local: B := NEW( B );
  remote: A := NEW( B, rp );
```

is okay, whereas:

```
VAR
  remote: B := NEW( B, rp );
```

is a `NARROW` checked run-time error.

If a programmer tries to avoid having to call a method to access the field `f`, and thus tries to dereference it directly, there is no problem provided that the variable has static type `B` or `D`. If the variable, say `a`, has static type `A`, say, an explicit `NARROW` is needed:

```
x := NARROW( a, B ).f       .
```

If the concrete type of `a` is not a subtype of `B`, then execution of this statement will result in a checked run-time error, as is defined in Modula-3.

In summary, the idea of pure types is nice in that it requires no introduction of a new checked run-time error. However, as is demonstrated by the examples above, it affects how a programmer may define and extend types. (Not shown here are some minor details regarding partially opaque types and pure types that the language definition needs to address.)

To take advantage of Modula-3's partially opaque types, trying to go around the drawbacks of the previously presented scheme, we may be tempted to define the pure type of `T` as the supertype of `T`, closest to `T`, that *in the current scope* is not revealed to have any data fields. We can then solve the dilemmas of the above example by hiding `f`, as in:

```
INTERFACE example;
TYPE
  B <: A;
  A = NETWORK OBJECT METHODS m(): INTEGER END;
  D <: C;
  C = B OBJECT n(): INTEGER END;
END example.


MODULE example;
REVEAL
  B = A BRANDED OBJECT f: INTEGER OVERRIDES m := F END;
  D = C BRANDED OBJECT g: INTEGER OVERRIDES n := G END;

PROCEDURE F( b: B ): INTEGER =
  BEGIN
    RETURN b.f
  END F;

PROCEDURE G( d: D ): INTEGER =
  BEGIN
    RETURN d.g
  END G;

BEGIN
END example.      .
```

Any module that imports `example` now gets `B` and `D` as the pure types of themselves. Moreover, the implementation of the types does not duplicate field `f` or method `m`. Thus, `B` can be subtyped without the unnatural complications shown in the previously discussed scheme. However, we no longer receive the benefit of not needing an extra checked run-time error. Suppose the previous interface also contained a procedure `Z`, implemented as follows.

```
PROCEDURE Z( b: B ) =
  BEGIN
    b.f := 0
  END Z;
```

Then consider the program:

```
MODULE Main;
IMPORT example;
VAR b: example.B;
BEGIN
  b := NEW( example.B, rp );
  Z( b )
END Main.
```

,

where `rp` indicates some remote processor. In `Main`, `example.B` is the pure type of itself. Thus, the allocated type of the reference returned by the `NEW` call is `example.B`, and the assignment to `b` and the call to `Z` with `b` execute without causing `NARROW` faults. However, the dereference `b.f` in `Z` cannot be expected to succeed as the data record of `b` was allocated on processor `rp`. So much for that attempt.

Chapter 2

Programming Model

In this chapter, we illustrate some strengths and weaknesses of the Modula-3D programming model. To at the same time provide a flavor of the language, we proceed by showing some sample problems and their solutions written in Modula-3D.

2.0 Simulating Channels

A goal of Modula-3D was to get away from programming with send and receive primitives across channels. To show that we can still make use of first-in-first-out buffers in Modula-3D, we show the implementation of a type `Buffer`. We start with its type declaration.

```
TYPE
  Buffer = NETWORK OBJECT
      mu: MUTEX;
      notEmpty: Thread.Condition;
      notFull: Thread.Condition;
      a: REF ARRAY OF INTEGER;
      n: CARDINAL := 0;
      nextGet: CARDINAL := 0;
      nextPut: CARDINAL := 0
    METHODS
      init( size: [1..LAST(INTEGER)] := 1 ): Buffer := BufInit;
      put( x: INTEGER ) := Put;
      get(): INTEGER := Get
    END;
```

The data fields `mu` through `nextPut` are to be used only by the implementation. Modula-3D provides the ability to hide these by in an interface only mentioning the type's methods. However, as that is not the focus of this example, we have shown the entire type declaration.

Mentioning the type `NETWORK` in front of the keyword `OBJECT` means that `Buffer` is an immediate subtype of `NETWORK`. In other words, we have declared `Buffer` to be a network object type whose instances can be shared among the processors.

The `init` method initializes a new `Buffer` object. Its parameter specifies the maximum number of elements that the buffer may contain, called the *slack*. A default value of 1 is specified for this parameter; in other words, if the actual parameter for `size` in an invocation of `init` is omitted, it defaults to 1.

The `init` method is implemented by procedure `BufInit`, as shown in the type declaration by specifying `BufInit` as a default value for `init`. We write procedure `BufInit` as follows.

```
PROCEDURE BufInit( buf: Buffer; size: [1..LAST(INTEGER)] ): Buffer =
  BEGIN
    buf.mu := NEW( MUTEX );
    buf.notEmpty := NEW( Thread.Condition );
    buf.notFull := NEW( Thread.Condition );
    buf.a := NEW( REF ARRAY OF INTEGER, size );
    RETURN buf
  END BufInit;
```

By convention, the `init` method returns the object to which it is applied. This allows a variable declaration in a program to be written as:

```
VAR buffer := NEW( Buffer ).init();      .
```

`Buffer` features two methods `put` and `get`. These put elements into and remove elements from the buffer, respectively, and may be used as:

```
buffer.put( 12 ); x := buffer.get()      .
```

A thread invoking these methods will be suspended while the buffer is full or empty, respectively. The methods are implemented as follows.

```
PROCEDURE Put( buf: Buffer; x: INTEGER ) =
  BEGIN
    LOCK buf.mu DO
      WITH size = NUMBER( buf.a^ ) DO
        WHILE buf.n = size DO Thread.Wait( buf.mu, buf.notFull ) END;
        buf.a[ buf.nextPut ] := x;
        buf.nextPut := ( buf.nextPut + 1 ) MOD size;
        INC( buf.n );
        Thread.Signal( buf.notEmpty )
      END
    END
  END Put;
```

```
PROCEDURE Get( buf: Buffer ): INTEGER =
  VAR x: INTEGER;
  BEGIN
    LOCK buf.mu DO
      WHILE buf.n = 0 DO Thread.Wait( buf.mu, buf.notEmpty ) END;
      x := buf.a[ buf.nextGet ];
      buf.nextGet := ( buf.nextGet + 1 ) MOD NUMBER( buf.a^ );
      DEC( buf.n );
      Thread.Signal( buf.notFull )
    END;
    RETURN x
  END Get;
```

The `Put` and `Get` procedures use monitor mutex and condition variables (see [13, 4, 14, 5, 23]) to do the synchronization.

Objects of type `Buffer` are used the same way, be they on the local or on a remote processor. The element type of this buffer implementation is `INTEGER`. For writing a general implementation of `Buffer`, Modula-3D features generics. Alternatively, the element type can be specified as `NETWORK`, so that the buffer may be used for any network object type. The differences between these two approaches is that the former requires one instantiation of the generic buffer for each type, whereas the latter allows for just one copy of the code and allows one buffer to contain elements of different types.

Note that `Buffer` allows any number of threads to invoke the `put` and `get` methods of a `Buffer` object. Hence, this aspect of `Buffer` is more versatile than point-to-point channels provided by some other languages (see *e.g.* [15, 22, 16, 20]).

The efficiency of a program that uses buffers may depend on the slack. Thus, using a different value for the `size` parameter to `init` provides a way to tune the performance of a program using `Buffer`. Moreover, it is interesting to note that the slack is, in general, determined at run-time. Differing from some multicomputer languages today, this equips the programmer with more flexibility.

2.1 Distributing a Program

In this section, we show how easy it is to distribute a program. The example we use is a standard parallel programming problem, namely that of generating prime numbers by means of the sieve of Eratosthenes (see, for example, [15]). We will make use of type `Buffer` as defined in Section 2.0.

Specifically, the problem is to create a `PrimeConsumer` object, to invoke its `consume` method for each prime in the range `2..Max`, where `Max` is a constant at least 2, and then to invoke its `end` method. The type `PrimeConsumer` is given as:

```
TYPE
  PrimeConsumer = NETWORK OBJECT
      (* ... *)
    METHODS
```

```
    consume( x: INTEGER ) := Consume;
    end() := EndConsume
  END;
```

where `Consume` and `EndConsume` are appropriately declared procedures.

We proceed in three steps. First, we write a sequential program that solves the problem. Then, we introduce concurrency, and finally, we distribute the program.

We declare a type `G` whose `generate` method solves the specified problem for a given `PrimeConsumer` object:

```
TYPE
  G = OBJECT
    METHODS
      generate( p: PrimeConsumer ) := Generate
    END;
```

To implement `Generate`, we declare, keeping in mind the sieve of Eratosthenes, a type `Filter`:

```
TYPE
  Filter = OBJECT
      n: INTEGER;
      next: Filter := NIL;
      p: PrimeConsumer
    METHODS
      init( n: INTEGER; p: PrimeConsumer ): Filter := Init;
      try( x: INTEGER ) := Try;
      end() := End
    END;

PROCEDURE Init( f: Filter; n: INTEGER; p: PrimeConsumer ): Filter =
  BEGIN
    f.n := n; f.p := p;
    p.consume( n );
    RETURN f
  END Init;

PROCEDURE Try( f: Filter; x: INTEGER ) =
  BEGIN
    IF x MOD f.n = 0 THEN
      (* skip *)
    ELSIF f.next # NIL THEN
      f.next.try( x )
    ELSIF f.n * f.n >= Max THEN
      f.p.consume( x )
```

```
    ELSE
      f.next := NEW( Filter ).init( x, f.p )
    END
  END Try;

PROCEDURE End( f: Filter ) =
  BEGIN
    IF f.next # NIL THEN f.next.end() ELSE f.p.end() END
  END End;
```

This allows us to write procedure `Generate` as:

```
PROCEDURE Generate( <* UNUSED *> g: G; p: PrimeConsumer ) =
  VAR f := NEW( Filter ).init( 2, p );
  BEGIN
    FOR x := 3 TO Max DO
      f.try( x )
    END;
    f.end()
  END Generate;
```

The main body of our program is then just simply:

```
BEGIN
  NEW( G ).generate( NEW( PrimeConsumer ))
END Main.
```

This concludes the development of our sequential program. The next step is to introduce concurrency, which we do by changing type `Filter` and its default methods to the following.

```
TYPE
  Filter = OBJECT
      cl: FilterClosure
    METHODS
      init( n: INTEGER; p: PrimeConsumer ): Filter := Init;
      try( x: INTEGER ) := Try;
      end() := End
    END;

  FilterClosure = Thread.Closure OBJECT
      buf: Buffer;
      n: INTEGER;
      next: Filter := NIL;
      p: PrimeConsumer
    OVERRIDES
```

```
    apply := Apply
  END;

PROCEDURE Init( f: Filter; n: INTEGER; p: PrimeConsumer ): Filter =
  BEGIN
    f.cl := NEW( FilterClosure, buf := NEW( Buffer ).init(), n := n, p := p );
    p.consume( n );
    EVAL Thread.Fork( f.cl ); (* this creates a thread executing f.cl.apply() *)
    RETURN f
  END Init;

PROCEDURE Try( f: Filter; x: INTEGER ) =
  BEGIN
    f.cl.buf.put( x )
  END Try;

PROCEDURE End( f: Filter ) =
  BEGIN
    f.cl.buf.put( 0 )
  END End;

PROCEDURE Apply( cl: FilterClosure ): REFANY =
  VAR x: INTEGER;
  BEGIN
    LOOP
      x := cl.buf.get();
      IF x = 0 THEN EXIT END;
      IF x MOD cl.n = 0 THEN
        (* skip *)
      ELSIF cl.next # NIL THEN
        cl.next.try( x )
      ELSIF cl.n * cl.n >= Max THEN
        cl.p.consume( x )
      ELSE
        cl.next := NEW( Filter ).init( x, cl.p )
      END
    END;
    IF cl.next # NIL THEN cl.next.end() ELSE cl.p.end() END;
    RETURN NIL
  END Apply;
```

We want the main body to terminate only after the `end` method of the `PrimeConsumer` object has been invoked. To ensure this, we declare a subtype of `PrimeConsumer`:

```
TYPE
```

```
    PrmCons = PrimeConsumer OBJECT
        done: BOOLEAN := FALSE;
        mu: MUTEX;
        c: Thread.Condition
      METHODS
        wait() := Wait
      OVERRIDES
        end := EndPrmCons
      END;

  PROCEDURE EndPrmCons( p: PrmCons ) =
    BEGIN
      PrimeConsumer.end( p ); (* invokes end method of superclass *)
      LOCK p.mu DO
        p.done := TRUE;
        Thread.Signal( p.c )
      END
    END EndPrmCons;

  PROCEDURE Wait( p: PrmCons ) =
    BEGIN
      LOCK p.mu DO
        WHILE NOT p.done DO Thread.Wait( p.mu, p.c ) END
      END
    END Wait;      ,
```

which lets us modify the main body to:

```
  BEGIN
    WITH p = NEW( PrmCons, mu := NEW( MUTEX ), c := NEW( Thread.Condition )) DO
      NEW( G ).generate( p );
      p.wait()
    END
  END Main.       .
```

Now that we have a parallel solution for one processor, we face the issue of distributing the data and control of our program. This is simple with Modula-3D; it requires only two modifications. The first is to change the appropriate types into network object types. Since we have many `Filter` objects, we choose to make `Filter` a network object type. Consequently, `PrimeConsumer` needs to be a network object type, too. A careful reading of the problem statement shows that it already is. Our new `Filter` declaration reads:

```
  Filter = NETWORK OBJECT
    (* as before *)
  END;      .
```

(We are not exploiting that Section 2.0 declared `Buffer` to be a network object type.)

This change of type `Filter`, by itself, does not change the program at all. However, it does allow us to modify the `NEW( Filter )` calls, which is the only other thing we need to do to distribute our program. We assume for this example that the processors are numbered consecutively from the main processor, and that there are sufficiently many processors. We choose a simple distribution of the `Filter` objects, and change `NEW( Filter )` in `Apply` to:

```
NEW( Filter, Processor.ID()+1 )      ,
```

where `Processor.ID()` returns the ID of the local processor.

As our program adheres to the distribution conditions given in Section 1.0, these trivial changes transmute our parallel program into a distributed one without affecting its outcome.

Note that one may expect `Filter` objects on higher numbered processors to do less work. Hence, one can, for example, write a procedure, `MapFilter` say, that maps more `Filter` objects to each higher numbered processor. Then the call to `NEW` looks something like:

```
NEW( Filter, MapFilter( ... ))      ,
```

where `MapFilter` is called with parameters needed to determine the processor location of a new `Filter` object.

2.2 Abstraction and Reusability

In this section, we show how the Modula-3D abstraction features allow the solution to a problem to be divided into logical parts, and how the solutions of these may be reused for solving similar problems.

Many numerical problems are so-called *nearest-neighbor computations*, also known as *cellular automata*. The scenario is an arbitrary graph and a discrete time line. Every vertex has a value. At every time step, the new value of a vertex depends on the previous values at the vertex itself and its direct neighbors. In general, different vertices may have different numbers of neighbors and may employ different functions for computing their next values.

Of course, the computations done at each vertex depend on the specific problem at hand. Nevertheless, the structure of every cellular automaton program is the same. For example, the way the synchronization between time steps and the communication of values to neighbors are done do not depend on the problem. Hence, our goal is to write an interface that can support any problem of the described nature. This effects a nice separation of concerns: those involved in solving the general problem and those in solving a specific problem. Consequently, our solution will lend itself to reuse. That is, someone (whom in the sequel we will refer to as a *client*) interested in a particular problem can simply override a `compute` method to do the custom computation, rather than having to worry about rewriting the entire algorithm.

We start out by defining a type `Cell`, of which we show the part in which a client is interested.

```
CONST
  LastTime: Time = (* some number *);
```

```
TYPE
  Time = CARDINAL;
  Cell <: Public; (* <: may be pronounced "is a subtype of" *)
  Public = NETWORK OBJECT
      values: REF ARRAY OF Value
    METHODS
      init( neighbors: ARRAY OF Cell ): Cell;
      (* neighbors specifies the cell's neighbors *)

      run(): Value;
      (* Computes and returns the value at time LastTime *)

      compute( prev: Value; time: Time ): Value;
      (* To be replaced by subtypes. Computes the value at the given time.
        If time # 0, then prev is the value at time time-1 , and values
        contains the values of the neighbors at that time. If time = 0 , prev
        and values are undefined.  *)
    END;
```

The `init` method takes as a parameter an open array of neighboring `Cell` objects. Hence, the interface does not restrict the number of neighbors that a cell may have.

Before showing the implementation of `Cell`, we show an example of how a client may use this type. Consider a simulation of a 2-dimensional sheet of material with different heat sources along its sides. The temperatures along the sides are constant and the temperature at any one interior point is computed as the average temperature during the previous time step of the points surrounding it. We divide the 2-dimensional sheet into cells, each of which has four neighbors. Similarly, we divide the sides up into cells, each of which has one neighbor. We may then implement these interior and exterior cells as follows.

```
TYPE
  Interior = Cell OBJECT
    OVERRIDES
      compute := ComputeInterior
    END;
  Exterior = Cell OBJECT
      v: Value
    OVERRIDES
      compute := ComputeExterior
    END;
```

This declares two new types, `Interior` and `Exterior`, each a subtype of `Cell`. We write their `compute` methods as:

```
PROCEDURE ComputeInterior( cell: Interior; prev: Value; time: Time ): Value =
  VAR v: Value := 0;
  BEGIN
    IF time # 0 THEN
      FOR i := FIRST( cell.values^ ) TO LAST( cell.values^ ) DO
        INC( v, cell.values[i] )
      END;
      v := v DIV NUMBER( cell.values^ )
    END;
    RETURN v
  END ComputeInterior;

PROCEDURE ComputeExterior( cell: Exterior; prev: Value; time: Time ): Value =
  BEGIN
    RETURN cell.v
  END ComputeExterior;
```

We expect the reader immediately be delighted with how little code is required to write these types.

To start this computation, the appropriate cells are created using `NEW`. Then, the `init` method is invoked on the cells to initialize them, which includes setting up their neighbors. Notice how the program text does not depend on the structure that we described (except that interior nodes are assumed to have a positive number of neighbors, so as to avoid division by 0). Instead, the implementation allows any number of neighbors for the two types of cells, and the number of neighbors may even vary from one object to another of the same type. After the objects have been initialized, the `run` method is invoked in parallel for all the cells. This involves creating a thread per each cell. (An alternate implementation of `run` may create the threads there.)

After the computation completes, the final values need to be reported to the user. This can be done, for example, by passing an extra object, equipped with, say, a `reportFinalValue` method, to each cell during initialization.

Now for the implementation of type `Cell`. Type `Cell` is revealed as follows.

```
REVEAL
  Cell = Public BRANDED OBJECT
      mu: MUTEX;
      computing: Thread.Condition;
      writing: Thread.Condition;
      value: Value;
      neighbors: REF ARRAY OF Cell;
      written: REF ARRAY OF BOOLEAN;
      nWritten: CARDINAL := 0
    METHODS
      put( source: Cell; value: Value ) := Put
    OVERRIDES
      init := Init;
```

```
    run := Run
  END;
```

Note that this adds a `put` method, not visible from the previously presented interface. Completing the implementation, we write the three methods. Firstly, `init`, which is to be invoked as the first method on a `Cell` object, simply initializes the cell.

```
PROCEDURE Init( cell: Cell; neighbors: ARRAY OF Cell ): Cell =
  BEGIN
    cell.mu := NEW( MUTEX );
    cell.computing := NEW( Thread.Condition );
    cell.writing := NEW( Thread.Condition );

    WITH n = NUMBER( neighbors ) DO
      cell.neighbors := NEW( REF ARRAY OF Cell, n );
      cell.values := NEW( REF ARRAY OF Value, n );
      cell.written := NEW( REF ARRAY OF BOOLEAN, n )
    END;

    cell.neighbors^ := neighbors;
    FOR i := FIRST( cell.written^ ) TO LAST( cell.written^ ) DO
      cell.written[i] := FALSE
    END;

    RETURN cell
  END Init;
```

A cell communicates a value to a neighbor by invoking the `put` method on the neighbor. We assume that there are no parallel edges, that is, the neighbors of a cell are distinct.

```
PROCEDURE Put( cell: Cell; source: Cell; value: Value ) =
  VAR i: CARDINAL := 0;
  BEGIN
    WHILE cell.neighbors[i] # source DO INC( i ) END;
    LOCK cell.mu DO
      WHILE cell.written[i] DO Thread.Wait( cell.mu, cell.writing ) END;
      cell.values[i] := value;
      cell.written[i] := TRUE; INC( cell.nWritten );
      IF cell.nWritten = NUMBER( cell.written^ ) THEN
        Thread.Signal( cell.computing )
      END
    END
  END Put;
```

Lastly, we write the method containing the loop over time. Each iteration consists of three phases: communicate value to neighbors, wait for values from neighbors, compute next value.

```
PROCEDURE Run( cell: Cell ): Value =
  BEGIN
    cell.value := cell.compute( 0, 0 );
    FOR time := 1 TO LastTime DO
      (* communicate value to neighbors *)
      FOR i := FIRST( cell.neighbors^ ) TO LAST( cell.neighbors^ ) DO
        cell.neighbors[i].put( cell, cell.value )
      END;
      LOCK cell.mu DO
        (* wait for values from neighbors *)
        WHILE cell.nWritten # NUMBER( cell.written^ ) DO
          Thread.Wait( cell.mu, cell.computing )
        END;
        (* compute next value *)
        cell.value := cell.compute( cell.value, time );
        FOR i := FIRST( cell.written^ ) TO LAST( cell.written^ ) DO
          cell.written[i] := FALSE
        END;
        cell.nWritten := 0; Thread.Broadcast( cell.writing )
      END
    END;
    RETURN cell.value
  END Run;
```

Before going on, we give a proof of absence of deadlock by induction on the number of iterations of the outer loop in `Run`. We assume that every invocation of `compute` terminates. Then, our base case of 0 iterations holds. After n deadlock-free iterations, we show, phase by phase, that the following iteration does not have any deadlock either.

Consider a particular cell `cell`, and let `N` denote `NUMBER( cell.written^ )`. By inspection of both `Put` and `Run`, `cell.mu` is never locked indefinitely. Between the times condition `cell.nWritten # N` becomes `FALSE`, `N` invocations of the cell's `put` method have completed, and between the times condition `cell.written[i]` becomes `FALSE` for a particular `i`, an iteration of the outer loop of the cell's `run` method has completed. Thus, after it has completed n iterations of its outer loop in `run`, the cell will not pass its second phase again until `N` invocations of `cell.put` have completed. As each completion of the `put` method changes the value of `cell.written[i]` from `FALSE` to `TRUE`, no two of the next `N` completed invocations of `put` have the same `i`. As a cell's neighbors are assumed to be distinct, each of the `N` neighbors is thus be able to complete their `put` invocations on this cell. Hence, all cells are able to complete the first phase of their iteration n.

The last neighbor to complete its `put` invocation on `cell` finds `cell.nWritten = N`, and thus signals `cell.computing`. This means `cell` will be able to complete its next second phase. Moreover, as we assumed invocations of `compute` terminate, the cell is able to complete the third phase. Hence, every cell is able to complete iteration n, proving the absence of deadlock.

Now for the code. Note that the phase in which a cell communicates its value to its neighbors, where one expects that concurrency can be used, is actually done sequentially. (See the remark on clustered communications in Section 1.2.) In order to perform these calls in parallel, one thread per neighbor needs to be created. However, this whole business requires much more code to be written. A different approach to improve this aspect of the current program is to double the size of the buffer used to store the neighbors' values.

The strong point of the above cellular automaton is that it is easy to specialize for a particular kind of problem. Moreover, the solution is nice in that it allows for graphs in which the connections between the vertices are irregular. In fact, even self-loops are allowed (but not parallel edges because of the way `Put` is implemented). Moreover, neighboring vertices may even have different types.

Finally, a note about the type `Value`. One could choose `Value` to be `INTEGER` or `REAL` for many computations. However, our solution uses one thread per cell, so, in most cases, that choice for `Value` seems on the expensive side. Rather, to allow cells to contain more complex values, `Value` may be chosen as `NETWORK`. Then, in addition to allowing neighboring cells to be of different types, the types of values carried by the cells may be of different types.

As an example, a `Value` may contain a 2-dimensional array of integers, and the `compute` method may compute the next value for the entire array. If such cells are arranged in a 2-dimensional grid, then the neighbors only need the elements around the sides of the matrix. Thus, a nice improvement on the above program would be to change the body of the loop:

```
(* communicate value to neighbors *)
FOR i := FIRST( cell.neighbors^ ) TO LAST( cell.neighbors^ ) DO
  cell.neighbors[i].put( cell, cell.value )
END
```

to read:

```
  cell.neighbors[i].put( cell, cell.ni( i ))      ,
```

where method `ni` (for *neighbor's interest*) returns the part of the cell's value to be communicated to the cell's neighbor `i`. (One may also consider using `cell.neighbors[i]` as a parameter in place of `i`, or maybe use both.) The return type of `ni` is, of course, `Value`, and this method may be replaced by subtypes.

In this 2-dimensional grid example, a cell then creates a network object, of type `Side`, say, containing a 1-dimensional array into which it copies the elements along the appropriate one of its sides. To adhere to the distribution conditions, the array elements contained in a `Side` object need to be accessed via methods. Although neighboring cells may be located on different processors, we would like these methods invocations to be local since there may be many of them. If the array is of a fixed size, it can be assigned an initial value through use of the `Bindings` to the `NEW` call. Then, rather than allocating the `Side` object on the local processor, allocating it on the processor on which the neighboring cell resides proves more economical. For this reason, Mosaic Modula-3D features a procedure:

```
PROCEDURE Of( net: NETWORK ): INTEGER;
```

in the `Processor` interface (see Section 3.1). `Processor.Of( net )` returns the host of `net`. This shows how a large value inexpensively can be communicated to a neighbor.

Note that a cell cannot assume a neighbor to be through with the previous value that it received. Thus, a cell needs to allocate a new `Value` object for each of its neighbors during every iteration. Alternatively, two objects may be used alternatingly. The former of these is easier to implement, and works nicely because of Modula-3D's distributed garbage collector.

2.3 Proof Model

In this section, we say something about the proof model that one uses to reason about synchronization in Modula-3D. We illustrate by discussing a mutual exclusion problem. We will devote less attention to the details of this program than we have to the programs presented in previous sections.

Modula-3D, being an extension of Modula-3, provides constructs, *viz.*, `MUTEX` variables, that implement mutual exclusion. However, when using a distributed memory machine, one may still want to implement mutual exclusion in a distributed way to avoid the bottleneck of having a `MUTEX` on one processor. One way to do this is in a token ring. We describe a solution, referred to as *perpetuum mobile* [21, 18].

The scenario is that of having some number of *servers* and an equal number of *clients*. The servers are connected in a ring, and each client is connected with a unique server. Clients submit to their designated server requests for entering the critical section, and the servers together implement the mutual exclusion of requests being granted. The idea is to have a *token* that travels in one direction around the ring. There is always exactly one token. When a server receives the token, it will pass it to its client if the client has a pending request for entering its critical section; otherwise, the server passes the token on to the next server. A client may enter and remain in its critical section only while it holds the token, and is to return the token to its server within a finite amount of time. There being exactly one token thus guarantees mutual exclusion.

We write our program as follows.

```
TYPE
  Server = NETWORK OBJECT
      token, request: BOOLEAN := FALSE;
      mu: MUTEX;
      hasToken, requestGranted: Thread.Condition
    METHODS
      init( n: CARDINAL; root: Server ): Server := Init;
      transmitToken() := Transmit;
      requestToken() := Request
    END;

  ServerClosure = Thread.Closure OBJECT
      s, next: Server
    OVERRIDES
      apply := Apply
    END;

  Client = Thread.Closure OBJECT
```

```
      s: Server
      (* ... *)
    OVERRIDES
      apply := Run
    END;

PROCEDURE Init( s: Server; n: CARDINAL; root: Server ): Server =
  VAR next: Server;
  BEGIN
    s.mu := NEW( MUTEX );
    s.hasToken := NEW( Thread.Condition );
    s.requestGranted := NEW( Thread.Condition );
    IF n = 0 THEN
      next := root
    ELSE
      next := NEW( Server ).init( n-1, root )
    END;
    EVAL Thread.Fork( NEW( ServerClosure, s := s, next := next ));
    EVAL Thread.Fork( NEW( Client, s := s ));
    RETURN s
  END Init;

PROCEDURE Apply( scl: ServerClosure ): REFANY =
  BEGIN
    WITH s = scl.s DO
      LOOP
        LOCK s.mu DO
          WHILE NOT s.token DO Thread.Wait( s.mu, s.hasToken ) END;
          IF s.request THEN
            s.request := FALSE; Thread.Signal( s.requestGranted );
            Thread.Wait( s.mu, s.hasToken )
          END;
          s.token := FALSE
        END;
        scl.next.transmitToken()
      END
    END
  END Apply;

PROCEDURE Transmit( s: Server ) =
  BEGIN
    LOCK s.mu DO
      s.token := TRUE; Thread.Signal( s.hasToken )
    END
```

```
    END Transmit;

  PROCEDURE Request( s: Server ) =
    BEGIN
      LOCK s.mu DO
        s.request := TRUE;
        Thread.Wait( s.mu, s.requestGranted )
      END
    END Request;

  PROCEDURE Run( c: Client ): REFANY =
    BEGIN
      (* init *)
      LOOP
        (* non-critical section *)
        c.s.requestToken();
        (* critical section *)
        c.s.transmitToken()
      END
    END Run;

  PROCEDURE Start( n: [2..LAST(INTEGER)] ) =
    BEGIN
      WITH s = NEW( Server ) DO
        s.init( n-1, s ).transmitToken()
      END
    END Start;
```

Procedure `Start` initiates the creation of a ring of `n` servers and their clients. Type `Client` and its `apply` method (procedure `Run`) can be modified to fit a specific application.

The interesting thing to note here is that our program is written for a distributed-memory machine. Yet, judging from the program text, it appears to be a program written for a shared-memory machine. The observation is indeed correct; the proof model that one uses to prove a Modula-3D program is that of a shared-memory model.

On a different note, the `Server.init` method above not only initializes one `Server` object, but also initiates the creation of the rest of the ring. One may argue that a logical separation between these two initializations is nice. Furthermore, one may be interested in initializing a ring of *other* objects. This leads one to consider writing an object type `Ring` that deals with the creation of the ring only. Since different applications will require different actions to be done for each ring element, type `Ring` may feature a method that does the work. That method can then be overridden to fit the needs of a particular application.

2.4 Selection

It is often desirable to specify the conditions under which a thread may execute certain code fragments. We call this *selection*. For example, in Section 2.0, `Buffer.put` is implemented to suspend any thread attempting to add an element to a `Buffer` object while the buffer is full. We discuss different language features for doing selection.

CSP-like languages provide a `select` statement, which is like an `IF` statement but which suspends if none of its guards is true. The guards of a `select` statement are conditions that may include *probes* of channels. A probe on a channel is true if a communication on that channel will not suspend. Thus, to implement something like `Buffer.put`, the channel for `put` requests is probed in conjunction with something like `n < N`, where `n` and `N` are the current, respectively maximum, number of elements that can be stored in the buffer. Writing code for just this is easy.

The problem with using a `select` statement is that the contents of a message cannot be part of a guard. For example, it is not possible to implement our `Buffer` using only one channel, because then it is not know whether to use the probe in conjunction with `0 < n` or `n < N`. Instead, one channel per request is required.

Our `Buffer` implements one first-in-first-out *stream* of elements. We may also consider implementing a buffer that can handle multiple streams, all sharing the same fixed buffer space. For fairness among the streams, we may require that a particular stream occupy at most a certain proportion of the total buffer space. The "full" and "empty" conditions are then functions of the individual streams. Using a `select` statement to implement such a buffer then requires a `put` and a `get` channel per stream. (If the channels are point-to-point, then every stream needs as many `get` and `put` channels as there are readers and writers.)

Other languages like Compositional C++ [8, 7] feature *atomic* methods as a means of providing exclusive access to objects. This may reduce the complexity of some tasks as it provides an implicit `MUTEX` in every object and an implicit `LOCK` statement in each atomic method. However, such mechanisms do not lend themselves to writing something like `Buffer.put` where a condition is involved. Adding to this some flexibility, the language C+– [31] features atomic methods that have one of two states, *active* and *passive*, changeable by programs. Invocations of a method are suspended while the method is in the passive state. This mechanism allows conditions to be used, but the conditions cannot be functions of the parameters of a particular method invocation.

A more flexible approach is taken in Orca (see [0]), where a method (called an *operation* may contain a `guard` statement. The `guard` statement allows a programmer to specify the conditions under which the operation may be executed. When executed, the operation is executed atomically. This, a form of conditional critical regions (see [12, 2, 3, 4]), is a nice feature. However, all operations in Orca are atomic, so performing complex tasks involving several objects is not possible.

Modula-3D provides synchronization in a more exposed way through the use of `MUTEX` and condition variables, a form of monitors (see [13, 4, 14, 5, 23]). Sometimes these constructs seem to involve writing more code than some of their more high-level counterparts described above. However, the step of going from a simple specification to a more involved one need not be as large as when using the counterpart features. In particular, such steps *are* possible.

2.5 Dynamic Computations

The programs used to demonstrate Modula-3D features in the previous sections all have some predictable structure—a structure that may give a hint as to how to go about distributing the solutions. With its nice type system and distributed garbage collector, one may expect Modula-3D to be suited for handling problems that include irregular computations and dynamic data structures. In this section, we discuss such a problem.

The problem is to write a parallel LISP interpreter. The LISP data structure can be described as being composed of *nodes*, where a node may be an *atom* or a pair of nodes. This maps nicely to three types, `Node`, `Atom`, and `Pair`, where the two latter are subtypes of the former. The common features of `Atom` and `Pair` are then implemented once, in type `Node`. By making `Node` a subtype of `NETWORK`, objects of these types may be shared between processors. As the straightforward interpretation of LISP creates a large number of nodes, many of which are used only for short periods of time, LISP interpreters usually make use of a garbage collector. By implementing them as objects, these nodes become subject to garbage collection, courtesy of the Modula-3D run-time system. Hence, writing a distributed LISP interpreter in Modula-3D need not involve writing a garbage collector.

Our LISP interpreter implements the five built-in functions `car`, `cdr`, `cons`, `atom`, and `eq`. In addition to the built-in LISP constants `nil` and `t`, our LISP interpreter also features `cond`, `define`, and `quote`, but that concludes the list of built-ins. Therefore, the task of writing the interpreter itself is straightforward. In order to allow for parallel execution, the LISP computations are broken into small pieces, called *work items*. To distribute the work, these work items are then added to a *work pool* (or *task pool*). That is all there is to the LISP interpreter (except the parser).

Notice that the above description of the interpreter only mentions LISP factors; it leaves a clear interface between them and a work pool. The work pool interface is specified in general terms, rather than in terms of the LISP interpreter. We show the interface of our `Work` module here.

```
INTERFACE Work;

CONST
  DefaultWorkers = (* some positive number *);

TYPE
  Pool <: PublicPool;
  PublicPool = NETWORK OBJECT
    METHODS
      init( nWorkers: CARDINAL := DefaultWorkers ): Pool;
      add( item: Item )
    END;

  Item <: PublicItem;
  PublicItem = NETWORK OBJECT
    METHODS
      work( pool: Pool )
```

```
        END;

    END Work.
```

To use this interface, one subtypes `Work.Item` and replaces its `work` method. Then, `Work.Item` objects are added to a `Work.Pool` object. This will result in the `work` method being invoked on the work items, in some order determined by the work pool implementation. The interface admits both simple and fancy implementations. The point is that the work pool only needs to be written once, and that it is available through the `Work` interface via subtyping.

Finally, we remark on the load balancing that is to be implemented among processors in the work pool. Modula-3D does not specify any way for a programmer to get information about how much memory is free on a particular processor, or how busy a processor is. In the absence of such information, if each work item in the pool requires roughly the same amount of work, the load can be balanced based on the number of work items at each processor. Otherwise, this may suggest that a Modula-3D run-time system implementation provide the needed information, perhaps through an interface like the following.

```
INTERFACE Processor;
PROCEDURE FreeMemory( pid: INTEGER ): INTEGER;
PROCEDURE Utilization( pid: INTEGER ): [0..100];
END Processor.
```

There are difficulties involved, however. For example, since unreferenced memory is garbage collected, calculating exactly how much memory is available is costly. A low-cost alternative is to calculate only a lower bound on the amount of available memory. (See also Section 1.2 and Section 3.12.) On the Mosaic, it is also very difficult to calculate processor utilization, because no clock is available.

2.6 Summary

We have described several strengths and weaknesses of Modula-3D and shown programs in which these are visible. In summary, the Modula-3D programming model, intended to steer away from details of primitive send and receive operations, arms the programmer with network objects and familiar method invocations. Programs are easy to distribute, as adhering to the distribution conditions implies the particular distribution of a program's data and control does not affect the outcome of a program. Modula-3D presents a model in which there appears to be one address space, and where objects can be shared between processors.

The type system and modules of Modula-3D provide an arena well suited for abstraction. With abstraction comes reusability, thus letting the programmer make use of libraries that already solve standard problems. We have shown subtyping a useful vehicle in writing and reusing general solutions. Moreover, having a distributed garbage collector is instrumental in the reduction of the complexity of programs and the specification of interfaces.

Modula-3D uses threads, that is, light-weight processes, explicitly. The use of threads sometimes does not appear to be a light-weight experience for programmers, especially if trying to take advantage of clustered communications. Monitors serve as the underlying means for synchronization. While they provide flexibility, for some applications they seem inordinately rudimentary.

Overall, we are pleased with the ease in which we can write a distributed program for a multicomputer using Modula-3D. The presence of high-level features like object types makes the task of writing and changing a program's data structures pleasant.

Chapter 3

Implementation

In this chapter, we describe some highlights of implementing Modula-3D. We first give an overview of the issues involved in implementing network objects and some ideas on how this may be done. Then, we spend several sections presenting the associated details as found in our implementation of Modula-3D on the Mosaic multicomputer at Caltech. We give and discuss some Mosaic Modula-3D performance figures, and then conclude with some remarks on how the implementation may be improved.

3.0 Overview

In order to implement the "D" in Modula-3D, the run-time system needs to support the creation of, referencing of, method invocation on, and garbage collection of network objects. We mention some of the issues involved in each of these areas.

We distinguish between local references and global identifiers of network objects. A local reference discriminates between the local network objects on one processor, whereas a global identifier discriminates between the network objects on all processors. Thus, a local reference is usually implemented as the address of the data record of an object, and a global identifier as the processor ID of a host coupled with a local reference. If the address of an object's data record may change during execution, as in the case of a copying garbage collector, using a local reference as part of a global identifier is likely to have a large impact on efficiency. Instead, a level of indirection is called for in these cases. Then, a global identifier contains an index into a table that contains the local references of all network objects on the host.

A reference to a remote network object (called the *concrete* object) can be implemented via a local *surrogate* object. The typecode of the surrogate object is the same as for the concrete object, since the type of an object is not dependent on where it resides. (This deviates from the network objects in [32].) However, the data record and method suite of the surrogate object differ from those of the concrete object. In particular, the data record needs only contain the global identifier of the concrete object, and the entries of the method suite are procedures that will cause the methods to be executed remotely. (See [32] for a variation of these surrogates.)

Creating a remote network object involves communication between the requesting processor (called the *client*) and the host. The client sends a message to the host, passing an encoding of the parameters of the `NEW` call. The host creates the object's data record, pairs it with a method suite, and returns a global identifier of the object. The client then creates a surrogate object for the concrete one.

Each procedure in a surrogate's method suite performs a remote procedure call to the host, where the concrete object's method is executed. Remote procedure calls are implemented by specifying a protocol between a client and a host. The messages sent to the host include an encoding of the procedure to be called and its actual parameters. Those sent back include the procedure's outcome (normal or exceptional) and its return value or resulting exception. The calling thread on the client is suspended for the duration of the call, and a thread is created on the host to handle the work performed there.

Since all parameters are passed by value and all references are network objects, the only non-trivial issue in encoding and decoding is handling network objects. The encoding of a network object is simply its global identifier. To decode an object on its host, the local reference of the object is extracted. If the object is remote and a surrogate for the object already exists on the decoding processor, the same surrogate is used; otherwise, a surrogate for the object is created. Hence, a processor contains at most one surrogate for each concrete object. This simplifies comparing references for equality. So that it can quickly be determined whether a local surrogate for a particular object exists, every processor keeps a table of all its local surrogates.

Remote procedure calls are somewhat more complicated if the hardware does not provide a programming interface in which all processors are fully connected. As the language allows an object to be referenced from any processor, remote procedure calls may then need to involve processors on links between the client and host.

The run-time system is allowed to reclaim the storage of any allocated (traced) piece of storage to which no traced reference exists. To provide this for network objects, the garbage collector running on each processor needs to be able to determine whether or not there exists a reference to a local network object on some other processor; that is, whether any surrogate of the object exists. Ignoring cyclic structures spanning multiple processors, a reference count for each network object, counting the number of surrogates of it, can be used. A local garbage collector then treats a positive reference count as a reference to the object.

A processor (called the *client*) creates a surrogate when it receives a reference to a new network object. This reference is sent to the client from a processor, possibly the host, called the *mediator*. As the client creates the new surrogate, it reports this to the host by sending a new-surrogate message. It is important that the host does not collect the network object before this new-surrogate message arrives. As a network object is only collected if the information available to the host shows no references to it, the mediator keeps its reference to the object until the client has had a chance to report its new surrogate to the host. In particular, the sending thread on the mediator is suspended until the client sends back an acknowledgement. The client does not send this acknowledgement until the host has updated the reference count to include the new surrogate. This, in turn, is ensured by suspending the thread on the client until the host sends back an acknowledgement of the new-surrogate message. (This scheme is due to [24].) Finally, when a local garbage collector reclaims the storage of a surrogate object, it notifies the host which then decrements the reference count.

3.1 Mosaic Modula-3D

Mosaic Modula-3D consists of a compiler, a (pre-)linker, and a run-time system. The compiler translates Modula-3D into C, which is then compiled using the GNU C compiler, version 1.4x, targeted for the Mosaic. The Mosaic Modula-3D compiler and linker are based on the DEC SRC Modula-3 compiler and linker, whereas most all of its run-time system has been written from scratch.

The compiler and linker are completely written in Modula-3. The run-time system contains four functions written in C (*e.g.*, `memcpy` and `memmove`) and 30 routines written in Mosaic assembly. Roughly half of the assembly routines are used for reporting run-time errors and each about 3 instructions long. The other assembly routines perform Mosaic specific tasks such as turning interrupts on and off, and saving and restoring registers. The rest of the run-time system is written in Modula-3D, and consists of about 5280 lines of code.

Other than implementing the "D" features in the compiler and linker, Mosaic Modula-3D differs from SRC Modula-3 in that it computes run-time type information at link-time rather than at the beginning of run-time. This reduces the amount of code and data needed on every processor during run-time.

Mosaic Modula-3D features the required interfaces, like `Text` and `Thread`, except those relating to floating point numbers, which are not supported. Floating point support can be added by providing the software routines that perform these operations, as they are not directly supported in the hardware.

To comply with the Modula-3D definition to provide threads with the ability of getting the local processor ID, Mosaic Modula-3D includes an interface called `Processor`. This interface contains several procedures related to processor IDs. For example, `Processor.ID()` returns the ID of the local processor and, processor IDs in Mosaic Modula-3D having type `INTEGER`, `Processor.Min()` and `Processor.Max()` the lowest and highest ID, respectively, of any processor.

There are also some other restrictions in the implementation. For example, return values of, and arguments of exceptions raised by, the methods of network object types must fit in one word and may not be of an `ARRAY`, `RECORD`, or `SET` type. Moreover, network objects may not be part of the `Bindings` specified in a remote `NEW` call, and open arrays are disallowed as parameters of the methods of network object types. These restrictions have been imposed to make this initial Mosaic Modula-3D implementation simpler, but can all be removed in future implementations. Another shortcoming is that the underlying C compiler does not handle all bitfield operations correctly. Therefore, the use of packed types in Mosaic Modula-3D is discouraged.

We continue by describing the implementation in more detail. Section 3.2 through Section 3.9 describe the details of how network objects are implemented in Mosaic Modula-3D. Section 3.10 describes some of the implementation's enhancements. Section 3.11 gives an account of the performance, and Section 3.12 mentions some ways in which the implementation can be improved.

3.2 Message Types

A processor may send or receive messages of different types. All messages consist of a fixed size header followed by a body, whose size depends on the particular message. The header of each message takes the following form.

```
TYPE
  MsgType = { NewCall, Reply, IncRef, DecRef, New };
  GlobalID = RECORD pid: INTEGER; id: INTEGER END;
  Header = RECORD
      type: MsgType;
      src: GlobalID;
      param: INTEGER
    END;
```

The header is interpreted by the interrupt handler, as described below. The body portion of the messages is an array of words with no interpretation to the interrupt handler.

The different types of messages are identified by the value of the `type` field, and are described as follows:

`MsgType.NewCall` Used to initiate a remote procedure call. `src` specifies the calling thread. In particular, `src.pid` is the calling processor, and `src.id` is the ID of the calling thread on that processor. `param` gives the size of the body of the message and is positive. The body contains information about the call to the performed. (See Section 3.4.)

`MsgType.Reply` Message to the thread whose ID is `param`. The destination thread is assumed to be suspended, awaiting this message, at the time of its arrival. The body of this message is of variable size. `src` is not used.

`MsgType.IncRef` Used to increment the reference count of the object at address `param` in the receiver's address space. (See Section 3.7.) `src` specifies the calling thread, so that the concrete type of the concrete object can be returned. The body is empty.

`MsgType.DecRef` Used to decrement the reference count of the object at address `param` in the receiver's address space. (See Section 3.7.) `src` is not used, and the body is empty.

`MsgType.New` Used in remote `NEW` calls. (See Section 3.8.) `src` specifies the calling thread, and `param` is a pointer to the typecell of the object's type. An empty body indicates all default `Bindings`, whereas a non-empty body contains the object's initial value as specified on the caller side by custom `Bindings`.

3.3 Interrupt Handler

As described in the previous section, every message begins with a fixed header of type `Header`. As the interrupt handler is initialized, it sets up to receive such a header. Once a header is received, the interrupt handler, which is then invoked, inspects the header's `type` field, and takes action accordingly, described below. The action taken may include receiving the body of the message. After receiving a message, the interrupt handler sets up to receive a new message.

`MsgType.NewCall` messages are handled as described in Section 3.4. `MsgType.Reply` messages have a particular thread as destination. It is assumed that the destination thread is suspended at the time the message arrives, awaiting the message with an already allocated buffer. The interrupt handler receives the body of the message into this buffer, and then wakes up the receiving thread. (See below.) In response to

`MsgType.IncRef` and `MsgType.DecRef` messages, the interrupt handler increases or decreases, respectively, the reference count of the specified object, as described in Section 3.7. Finally, `MsgType.New` messages are handled as described in Section 3.8.

The interrupt handler replies, with a `MsgType.Reply` message, to incoming `MsgType.IncRef` and `MsgType.New` messages, and to no other messages.

The interrupt handler also handles the sending of messages. For this, it keeps two queues, described below. Sending of a message is done by enqueuing the message into one of the queues; except, if both queues are empty, the message is sent right away. As a send completes, the interrupt handler initiates the next send, if there is one.

The two queues, and their types, are:

```
TYPE
  Surrogate = UNTRACED ROOT OBJECT
      gid: GlobalID;
      next: Surrogate
    END;
  SendNode = UNTRACED ROOT OBJECT
      next: SendNode := NIL;
      dest: INTEGER;
      ack: Thread.T := NIL;
      msgFirst: ADDRESS;
      msgLast: ADDRESS
    END;
VAR
  DecRefQueue: Surrogate := NIL;
  SendQueue: SendNode := NIL;
```

Nodes on `DecRefQueue` are given priority over those on `SendQueue`, as each `MsgType.DecRef` message enables some memory to be freed up, locally and possibly also at the message destination. The interrupt handler interprets the nodes on the two queues as follows. A `Surrogate` translates into a `MsgType.DecRef` message to processor `gid.pid`, in which `param` has value `gid.id`. After a `Surrogate` has been used, it is deallocated using `DISPOSE`.

A `SendNode` translates into a message to processor `dest`. The (possibly empty) body immediately follows the header in a buffer that begins at address `msgFirst` and extends to address `msgLast`. All sends, except those done directly by the interrupt handler, result in the sending thread being suspended until the send has completed. The sending thread is then recorded as `ack`. When a send-and-receive operation is performed (see next paragraph), the completion of the send does not result in waking up a thread. Then, `ack` is left as `NIL` instead. Once the send corresponding to a `SendNode` completes, the interrupt handler wakes up the `ack` thread if `ack` is not `NIL`, and then deallocates the `SendNode`.

A thread may prepare to receive a message. This is only done in conjunction with sending a message, to which the message to be received is a reply. These reply messages are the messages of type `MsgType.Reply`. Before calling the procedure to perform this send-and-receive operation, the thread allocates a buffer into which to receive the reply. The send-and-receive procedure then proceeds as follows. First, a variable of type:

```
TYPE
  ReceiveNode = UNTRACED REF RECORD
      next: ReceiveNode := NIL;
      t: Thread.T := NIL;
      bufFirst: ADDRESS;
      bufLast: ADDRESS
    END;
```

is allocated. In it, `t` is set to the requesting thread, and `bufFirst` and `bufLast` are set to the boundaries of the receiving buffer. Then, the node is added to the linked list:

```
VAR
  Receivers: ReceiveNode := NIL;      .
```

Then, a `SendNode` is prepared for the requested send. Since the thread is to be woken up once the receive (rather than the send) completes, the `ack` field is left as `NIL`. Note that the receive completes no sooner than the send does. Finally, as the send is initiated or the `SendNode` is enqueued, the requesting thread is suspended.

When the interrupt handler receives a `MsgType.Reply` message, it finds and removes the node on the `Receivers` list that has `t` equal to the destination thread. From the `bufFirst` and `bufLast` fields of this node, the interrupt handler can set up to receive the body of the message. Once the body has been received, the receiving thread is woken up, and the `ReceiveNode` is deallocated.

3.4 RPC Protocol

A remote procedure call is carried out between a *client* and a *server*. The former is the caller and the latter the callee. This section describes the protocol used between client and server. After each send in this protocol (except, of course, the last one of the client and the last of the server), a thread waits for a reply.

0. Client: The client calculates the size of the call description. (See Section 3.5.) Then it allocates this buffer, fills it appropriately, and sends it as a `MsgType.NewCall` message to the host.

1. Server: On receipt of the `MsgType.NewCall` message, the interrupt handler allocates an array of `param` words, into which it receives the rest of the message. Then, it creates an object of type:

```
TYPE
  NewClosure = Thread.Closure OBJECT
      client: GlobalID;
      callDescr: REF ARRAY OF INTEGER
    OVERRIDES
      apply := (* ...  *)
    END;      ,
```

in which `client` specifies the client thread and `callDescr` the newly allocated array. Then, a thread is forked using this closure. This concludes the work performed by the interrupt handler. The rest of the work is done by the new thread, which runs under the same conditions as any other thread.

The new thread unmarshals the call description `callDescr`. (See Section 3.6.) Then, it pushes the parameters on the stack and calls the specified procedure.

As the procedure returns, the return description is placed in a buffer. (See Section 3.5.) Then, the return buffer is sent back to the client.

2. Client: The client unmarshals the return buffer. (See Section 3.6.) Then, it sends an acknowledge message back to the server.

Finally, if the outcome of the procedure was normal, the client returns the return value to the calling procedure; otherwise, it raises the exception.

3. Server: After receiving the acknowledge message, the thread terminates.

In some cases (see Section 3.6), the server can terminate as soon as it has completed step 1. Then, the client does not send any acknowledge message in step 2, step 3 is omitted altogether, and the server terminates after sending the return buffer in step 1.

3.5 Call and Return Descriptions

This section describes the format of the buffers sent in steps 1 and 2 of the RPC protocol.

The call description buffer begins with a record of type:

```
TYPE
  CallDescrHeader = RECORD
    proc: ADDRESS;
    pt: ProcType
  END;      ,
```

where `proc` is the address of the procedure to be called, and `descr` is a pointer to a description of the procedure's type. Both of these are pointers in the server's address space. The rest of the buffer consists of the parameters, described below.

The procedure's type is encoded in the form:

```
TYPE
  ProcType = UNTRACED REF RECORD
      paramEncodingSize: CARDINAL;
      paramFormalSize: CARDINAL;
      returnType: ReturnValue;
      params: REF ARRAY [ 0..n-1 ] OF BITS 1 FOR Value
    END;      ,
```

where `paramEncodingSize` is the number of words in the encoding of the parameters, `paramFormalSize` is the number of words that the parameters take up on the call stack, `returnType` describes the procedure's return type, and `params` describes the words pushed onto the stack as parameters to the procedure. The number `n` is treated as equalling `paramFormalSize`. The types `Value` and `ReturnValue` are defined as:

```
TYPE
  Value = { Word, NetworkObject };
  ReturnValue = { None, Word, NetworkObject };      .
```

The parameters in the call description buffer are placed one after the other. The encoding of the parameters has the following interpretation:

`Value.Word`	any word ,
`Value.NetworkObject`	a `GlobalID` value .

When the stack is prepared for the call to the procedure, a `Value.Word` is simply pushed on the stack. A `Value.NetworkObject` receives special treatment as described in Section 3.6. This treatment results in one word, *viz.*, an object, which is then pushed on the stack.

The return description buffer has type:

```
RECORD
  server: INTEGER;
  exception: ExceptionName;
  word0: Word.T;
  word1: Word.T
END     ,
```

where `server` is the thread ID of the server thread, or undefined if no acknowledgement is needed (see Section 3.6). The field `exception` is `NIL` if the procedure's outcome was normal; if the outcome was an exception, `exception` identifies the exception. (The type `ExceptionName` is defined in an exception module, not described in this report.)

For normal outcomes, `word0` and `word1` describe the return value; for exceptional outcomes, they describe the exception argument. If the return value or exception argument has type `ReturnValue.None`, both `word0` and `word1` are undefined. If it is `ReturnValue.Word`, `word0` is the word returned, and `word1` is undefined. If it is `ReturnValue.NetworkObject`, then `word0` and `word1` correspond to `pid` and `id` in a `GlobalID` record. A network object is unmarshalled as described in Section 3.6. The implementation restricts the return value of, and the arguments of exceptions that can be raised by, network object type methods to fit in one word.

3.6 Marshalling and Unmarshalling

The operation of converting the representation of a value to one that is sent in a message (called the *wire representation*) is called *marshalling*. The dual operation is called *unmarshalling*. These operations are straightforward, except when they involve network objects. This section describes the marshalling and unmarshalling operations of network objects.

The wire representation of a network object, as mentioned in Section 3.5, is a `GlobalID` value. In it, `pid` is the processor ID of the object's host, and `id` is a reference to the object in the host's address space. The wire representation of `NIL` has `id` as 0. A surrogate similarly contains a `GlobalID` value, as described in Section 3.3. Thus, marshalling is done as follows: if the network object is `NIL`, the wire representation of `NIL` is used; if the network object is concrete, the local processor ID and the reference to the object are used; if the network object is a surrogate, the surrogate's `GlobalID` value is used.

Unmarshalling takes more work. As described in Section 3.0, there is at most one surrogate per concrete object per processor, and the use of a (hash) table of existing surrogates enables rapid access to a surrogate given a global ID. (See also Section 3.7.) We recall that the concrete type of a network object is not part of its wire representation. Rather, this information is found on the object's host. As unmarshalling a new remote network object involves sending an acknowledged `MsgType.IncRef` message to the host (see Section 3.7), we choose to include the concrete type information in the acknowledge message.

Unmarshalling a remote network object from a wire representation `gid` is thus done by:

If `gid.id = 0`, return `NIL`.
If `gid.pid` is the local processor ID, return `gid.id`.
Search the table for the key `gid`. If present, return the associated reference.
Send a `MsgType.IncRef` to processor `gid.pid`.
On receipt of its acknowledgement (which, recall, contains the concrete type of the object), allocate and initialize a new surrogate object, and enter a reference of the surrogate into the table under the key `gid`.
Finally, return a reference to the surrogate.

Two details are left out of the stated algorithm. Firstly, and most importantly, accesses to the table need to be performed exclusively. This is done by entering the system critical section prior to the search, and exiting it after the described procedure. However, if a `MsgType.IncRef` message is to be sent, the critical section is exited until the acknowledge message is returned. This allows other threads to execute in the meantime. It also gives another thread the opportunity to enter the global ID into the table while the `MsgType.IncRef` message is being sent. Therefore, as the critical section is entered the second time, the table is searched again. If `gid` is not present, it is entered as described. If it is present, the associated surrogate reference is retrieved and the critical section is exited. At this time, the reference count of the concrete object has been incremented in excess of its number of surrogates. Thus, a `MsgType.DecRef` message is sent to the host. Finally, the surrogate reference is returned.

Having said that, we note that the long path of the above is not likely to be executed very often. At this time, we also briefly mention that an alternative solution is to enter some kind of "surrogate-to-be" into the table before exiting the critical section. If a surrogate for `gid` is not present when a thread searches the table, but a surrogate-to-be for `gid` is, the thread is suspended until that surrogate-to-be is converted into a proper surrogate.

The other detail left out of the above algorithm is that allocating memory in the usual way cannot, in this implementation, be done from within the system critical section. Thus, the surrogate is allocated before the critical section is entered the second time. If the long path is taken, then this piece of memory is deallocated outside the critical section the second time.

Allocating a new surrogate object is done as follows. First, an equivalent of:

```
NEW( Surrogate, gid := GlobalID{ pid := pid, id := id } )
```

is done. Subsequently, the typecode and storage class of the allocated object is changed to those of the concrete object. Finally, the object's method suite is changed to one in which all methods are stubs that perform the remote procedure calls. More precisely, a stub calls a general procedure that does remote procedure calls, passing it a procedure type description, as described in Section 3.5.

The aforementioned hash table of surrogates is organized as follows. The table maps global IDs to surrogates that exist locally. A global ID is mapped into one of a fixed number of buckets. The buckets are represented by an array whose elements have type `Surrogate`. `Surrogate` objects whose global ID map to the same bucket are linked using the `Surrogate next` field (see Section 3.3).

The reason for the last acknowledgement received in step 3 of the RPC protocol is so that the client gets to unmarshal the return value or exception argument before the thread on the server terminates. This ensures that the network object reference counts are maintained correctly. (See Section 3.9.) Since this is only important for returned network objects, and only for network objects that reside on processors other than the client processor, the acknowledgement can be avoided in other cases. This condition can be checked by both client and server for each remote procedure call, so the choice of using the acknowledgement can be tailored on the fly.

The Mosaic Modula-3D implementation is a bit restrictive in the above. For example, return values and exception arguments are restricted to fit in one word. (See Section 3.1 for details.)

3.7 Reference Counts and Garbage Collection

As described in Section 3.0, every concrete network object has a reference count which counts the number of surrogates of the object. When a processor receives a `MsgType.IncRef` message, the interrupt handler increments the reference count of the specified object. Then, a variable of type:

```
TYPE
  AddressReply = SendNode OBJECT
      header: Header;
      addr: ADDRESS
    END;
```

is allocated. (Section 3.3 contains the `SendNode` type declaration.) In it, `header.type` is set to `MsgType.Reply`, `header.param` to the value of `src.id` in the received `MsgType.IncRef` message, and `addr` to the address of the typecell of the type of the given object, `param`. The inherited field `dest` is set to the given `src.pid`, `msgFirst` to `ADR(header)`, `msgLast` to `ADR(addr)`, and `ack` is left as `NIL`. Then, this node is appended to `SendQueue` (or sent directly).

On receipt of a `MsgType.DecRef` message, the count is decremented, but no acknowledge message is prepared.

No other action is taken by the interrupt handler. So, for example, the interrupt handler does not take any special action if the reference count of an object reaches 0.

Instead, the garbage collector is the one to read and interpret the reference counts. In addition to marking heap nodes from stacks and globals, the garbage collector iterates through all the nodes in the heap to find local network objects. If the reference count is positive for such a local network object, it is used as a root. Using this scheme, externally unreferenced cyclic structures spanning multiple processors are not collected, but all other garbage is collected eventually.

When everything of interest has been marked and the garbage collector enters its storage reclamation phase, any network object surrogate whose storage is to be reclaimed undergoes some special treatment. First, the table that keeps track of surrogate objects is updated. More precisely, the entry for the reclaimed surrogate is removed. Old surrogate objects are kept in the heap until the corresponding `MsgType.DecRef` message has been sent out. To achieve this, the garbage collector changes the typecode and storage type of the heap node containing the surrogate from those of the concrete object back to those of `Surrogate`. As the storage type of `Surrogate` is untraced, this prevents the node from being garbage collected. The old surrogate is then placed on the linked list `DecRefQueue` (or the corresponding `MsgType.DecRef` message is sent directly). (See Section 3.3.) Once the `MsgType.DecRef` message has been sent, the `Surrogate` object is deallocated using `DISPOSE`.

The run-time system needs to be able to distinguish between local and remote network objects. To this end, concrete network objects include a `pid` field, which specifies the ID of the object's host processor. As surrogate objects also include such a field, namely `gid.pid`, this processor ID field will distinguish between concrete and surrogate objects on a processor, provided that the field occurs at the same offset into both a concrete and a surrogate object. To meet this requirement, the type `NETWORK` is revealed as:

```
REVEAL NETWORK = ROOT OBJECT pid: INTEGER; refcount: CARDINAL := 0 END;
```

Incidentally, the compiler emits checks that verify that fields are not dereferenced on remote objects. These checks also make use of the `pid` field. If a test fails, a checked run-time error is reported.

The garbage collector also first establishes whether a heap node contains a network object at all. Since the typecode of a surrogate is changed to be that of the concrete type, testing the node's typecode to be a `NETWORK` subtype does the trick.

3.8 Remote NEW Calls

As a mechanism to create remote network objects, Modula-3D features remote `NEW` calls. Such a call creates a (concrete) network object remotely, while locally creating and returning a surrogate for the concrete object. In this section, we describe the implementation of a `NEW` call in which a `Processor` is specified. Note that network objects may also be allocated locally by not specifying a `Processor` in the `NEW` call. This works in the same way as any local `NEW` call, except that the allocator will initialize the `pid` field to be the ID of the local processor.

Before explaining the algorithm, we first make two observations. The value of `Processor` is not known at compile time, yet different actions are taken depending on whether the object is to be allocated locally or remotely. Also, remote calls are done differently when the `NEW` call includes `Bindings` and when it does not. It is nice not to have to change the code produced by the compiler to assign the `Bindings` to the fields of the new object.

We now describe the most general algorithm, beginning with the code generated by the compiler. The compiler generates code to do the following.

> Execute r := `PrepareAlloc(` type description, `Processor );`
> Assign the fields of `r` using `Bindings`;
> Return as the new reference `RemoteNew(` type description, `Processor, r )`

First, we consider the case where `Bindings` are not specified. Then, the first two lines in the above algorithm are skipped, and `r` is passed in as `NIL` to `RemoteNew`. If `Processor` indicates the local processor, then `RemoteNew` proceeds as in a usual `NEW` call, returning a new object. If `Processor` indicates a remote processor, `RemoteNew` follows the protocol given below using no `Bindings`. Finally, `RemoteNew` returns the new surrogate (see below).

If, on the other hand, `Bindings` are specified, all three lines given above are used. When `Processor` indicates the local processor, `PrepareAlloc` proceeds as in a usual `NEW` call, returning a new object. Then, the `Bindings` are assigned by the compiler generated code. Finally, `RemoteNew`, which now gets the local processor ID as `Processor` and a non-`NIL` value as `r`, simply returns `r`.

When `Bindings` are specified and `Processor` indicates a remote processor, `PrepareAlloc` allocates an untraced array of words. This array is large enough to hold a `Header` record (see Section 3.2) followed by a data record of an object of the given type. The "data record portion" of this array is initialized as if it really were an object of the given type, and the address of the beginning of this portion is returned. The `RemoteNew` invocation then continues as described below when `Bindings` are specified, and results in a reference to a surrogate for the new object.

The protocol for `NEW` calls is:

0. Client: The client sends a `MsgType.New` message to the `Processor` indicated in the remote `NEW` call. If `Bindings` are specified, this message consists of the word array allocated as described above; otherwise, it is simply a `Header` with no body. To simplify this algorithm and avoid marshalling and unmarshalling of this buffer, the implementation disallows network objects to be part of `Bindings`.

1. Server: The interrupt handler calls `NEW` to allocate a new object of the given type. If the body of the message is non-empty, it contains the initial value of the object. If so, the body is received into the newly allocated space.

Then, the `pid` field in the newly allocated object is set to the processor ID of the server. Also, the `refcount` field is set to 1, so as to account for the surrogate the client already created.

Next, a variable of type `AddressReply` (see Section 3.7) is allocated. In it, `header.type` is set to `MsgType.Reply`, `header.param` to the thread ID of the requester, and `addr` to the reference of the newly allocated network object. The inherited field `dest` is set to the processor ID of the requester, `msgFirst` to `ADR(header)`, `msgLast` to `ADR(addr)`, and `ack` is left as `NIL`. Then, this node is appended to `SendQueue` (or sent directly).

2. Client: If `Bindings` were specified, the aforementioned word array is deallocated. The client allocates a `Surrogate` object, in which it sets `gid.pid` to the indicated `Processor` and `gid.id` to the returned concrete reference. Once that is done, the surrogate's typecode and method suite pointer are changed as described in Section 3.6. Finally, the surrogate is entered into the local table of surrogates.

3.9 Reference Counting Correctness

We define correctness of the reference counting scheme to be: except possibly at times when new surrogates are being created and reference count update messages are in transit, a concrete object's reference count equals the number of surrogates for it, and there are no surrogates other than those for existing concrete objects. We assume the local garbage collectors to work; that is, when a surrogate or concrete object is not reachable from any local root, and the concrete object's reference count is 0, the surrogate or concrete object is collected during the next garbage collector invocation. Notice that our definition of correctness does not require cyclic structures spanning multiple processors to ever be collected. We now argue that our implementation works with respect to our correctness criteria.

A concrete object is not collected if it has a positive reference count. When a new remote network object is unmarshalled on a processor (call it the *client* processor), a `MsgType.IncRef` message is sent to the host processor. Correctness requires that the concrete object has not been collected by the time the `MsgType.IncRef` message arrives at the host. Thus, there needs to be some reference to the network object at some processor. The processor from which the client received the network object (call it the *mediator* processor, which may be the same as the host) has a reference to the object as it was sent. So, provided the mediator does not get rid of this reference before the `MsgType.IncRef` message arrives at the host, correctness follows.

There are three ways that network objects are sent in messages: steps 0 and 1 of the RPC protocol, and step 1 of the remote `NEW` call protocol. In the case of the former two, the sender keeps a reference to the object on its stack. This reference will remain until the sender receives a message back from the receiving side (steps 2 and 3, respectively). After the receiving side unmarshals a new object and sends a `MsgType.IncRef` message, it waits for an acknowledgement from the host before it continues to do anything else. Thus, at the time the `MsgType.IncRef` message arrives at the host, it is known that there is a reference to the object at the mediator.

In the case of the remote `NEW` call, the server sets the reference count to 1 before the reference is given out to the client. Thus, the object will not be collected before the client sends its `MsgType.DecRef` message. Moreover, the client will not count this object twice, since the client is the only processor with a reference to the newly allocated object. This concludes the correctness argument.

Note that the correctness argument needs for the `MsgType.IncRef` calls to be synchronous. That is, a `MsgType.IncRef` message has to be acknowledged. The same is not necessary for `MsgType.DecRef` messages.

3.10 Enhancements

In this section, we describe some of the performance enhancements in the implementation. We first describe a couple of enhancements of what has been described in the last several sections and then one of method suites.

Remote Procedure Calls

The initialization performed by the processors other than the main processor terminates before any message has been received. At this point, the main thread does nothing else. Rather than getting rid of this thread, this thread is reused to service some of the incoming remote procedure calls. That is, when the interrupt handler on one of these processors receives a `MsgType.NewCall` message, it first checks to see if the main thread is available. If it is, it is used, and there is also a `NewClosure` object that is reused. If the main thread is not available, the interrupt handler forks a new thread as stated in Section 3.4.

One may consider having more than one thread per processor available to be reused to service remote procedure calls, but only one is being used in the current implementation.

This enhancement does not come for free. A problem may be that the thread's ID is stored somewhere. This ID can be used in calls to the `Thread` interface, and may then not appear to work as expected, since the main thread never terminates. There are only two ways to get a hold of a thread ID, namely the return values of `Thread.Fork` and `Thread.Self`. The former is only called on a thread once, and for the main thread, this is done by the run-time system. However, a user program may get a copy of the thread ID of the main thread by calling `Thread.Self`. Thus, this implementation warns the programmer about calling `Thread.Self` in module initialization routines and in threads that execute remote procedure calls. This restriction does not seem severe because it is hard to imagine for what useful purpose the ID of an RPC thread would be to the program.

Interprocessor Communication

Every regular send and receive includes allocating a new `SendNode`, `AddressReply`, or `ReceiveNode` object. To increase the performance of the send and receive procedures, these nodes, rather than being disposed of, are placed on available lists. The implementation contains two such lists, one for `SendNode` and `AddressReply` objects, and one for `ReceiveNode` objects. Since `AddressReply` is a subtype of `SendNode`, no proper `SendNode` objects are actually allocated; instead, only `AddressReply` objects are.

When a new object of one of the types described above is needed, one is taken off the corresponding list if the list is non-empty. If the list is empty, a `NEW` call is used. In order for this enhancement to not spend too much of the scarce memory resources on unneeded objects of the types described, the two lists are purged immediately before any garbage collection.

As for send operations, we realize that although the `SendQueue` is needed in general because we are not guaranteed that a message is sent immediately, it is usually the case that every message sent completes after one machine language instruction. This is because sending a message usually does not block, and the sending DMA has higher priority to the memory bus than does the processor. Thus, rather than always performing a context switch after initiating the hardware to start a new send, the interrupt status word is examined once to see if the message was actually sent. If it is, the send flag in the interrupt status word is immediately acknowledged. This prevents not only the context switch to suspend the sending thread, but also the context switch to the interrupt handler thread.

In lines with this enhancement, the interrupt handler is also set up to not return for as long as there is still a pending interrupt. This prevents switching from the interrupt handler thread and then immediately back to it.

Method Suites

The Mosaic Modula-3D linker reduces the space used by method suites in the following way. Consider two object types, `A` and `B`, where `B` is an immediate subtype of `A`. If the method suite of `A` is a prefix of that of `B`, `A` will use the same method suite as `B`.

3.11 Performance

In this section, we discuss the performance of Mosaic Modula-3D. We first discuss size and then speed.

Size

The DEC SRC implementation of Modula-3 links in an entire library if anything at all is used from that library. Consequently, the entire standard library, including interfaces like `Thread`, `Text`, and `Word`, is always included in every program. Stemming from this implementation, Mosaic Modula-3D uses the same scheme. Thus, every Mosaic Modula-3D program contains the entire Mosaic Modula-3D run-time system. Changing this would be a nice improvement for a machine like the Mosaic, which only has 64 KB of RAM on every node. (See Section 3.12.)

The situation with the libraries being what it is, we can measure the size of the run-time system by measuring the size of an empty program:

```
MODULE Main;
BEGIN
END Main.
```

A breakdown of the sizes of all components of an empty program is found in Table 3.0. The three rightmost columns in this table show the size of an empty program when compiled with all run-time checks (`NIL`-dereference, stack overflow, range, assert, and other checks), with asserts but no other run-time checks, and with no run-time checks at all, respectively. (Special versions of the compiler were furnished to obtain these measurements.) The discrepancies between the sum of the component sizes and the "total" stem from that each actual number has been rounded up to a multiple of 8.

The figures shown in the table sum the size of both code and (initialized and uninitialized) data. The code accounts for the largest part of the space. More precisely, the data takes up 2444 bytes, a number that stays the same regardless of the level of run-time checks included. Of this, 628 bytes are used by the static part of the surrogate hash table, and 1436 bytes are used by the type and module run-time information. We now discuss some of the components.

The `Text` interface, which is required by Modula-3, is rather large. As it performs many operations in which array indices may be out of range and references may be `NIL`, the run-time check overhead for the module is big. However, many Modula-3D programs do not use any strings, or if they do, only few of the procedures in the `Text` interface are used often.

Another interesting interface is the `Word` interface. It is also required by Modula-3, but the compiler inlines these calls when used in a usual fashion. However, the value of the procedures in the interface (*e.g.*,

Component	*Bytes*	*Bytes*	*Bytes*
Heap manager/garbage collector	7756	4976	4492
Threads and their scheduling	5216	3560	3000
Synchronization primitives in `Thread`	1364	1364	1364
IPC (send/receive/interrupt handler)	3444	2340	1996
RPC	3932	2848	2804
Surrogate hash table functions	1640	1388	1288
`Text` interface	1700	988	988
`Word` interface	1356	1132	1132
Type and module run-time information	1484	1484	1484
Math run-time (`DIV`, `MOD`, `IN`)	960	740	740
Exception handling	820	448	448
Processor ID procedures	236	148	148
Code to call module initializations	356	252	248
Run-time error reporting	365	352	352
Mosaic specific register operations	396	340	296
C run-time functions (*e.g.*, `memcpy`)	340	340	340
Output to terminal	476	336	336
Floating point support stubs	64	40	40
Miscellaneous run-time support	64	48	48
Empty program module	120	104	104
Total	32088	23228	21360

Table 3.0: Breakdown of size of run-time system

`Word.And`) may still be used; for example, they may be assigned to variables. For that seemingly silly reason, the interface needs to be there, unless a scheme is devised to only include the procedures or modules that are actually used by an application program.

The implementation of the `Thread` interface (listed as two components in Table 3.0) takes up a relatively large space. A fair amount of it is taken up by run-time checks and asserts. Also, the interface provides several procedures that not all applications use. As an example, those related to alerts account for an amount of space disproportionate to the frequency of their use; in fact, none of the programs presented in Chapter 2 use alerts.

Finally, the heap manager and garbage collector implementation is an important part of the run-time system. Here we find the space that it occupies rather high, but also realize that almost half of the emitted code consists of run-time checks.

We remark on the current compilation process. As mentioned in Section 3.1, Mosaic Modula-3D emits C code, which, independently, is translated into assembly. The Modula-3D-to-C phase does not do any optimization; rather, it leaves all optimization to the C-to-assembly phase. As it turns out, the C compiler does so poorly. We give a small example of this translation process. Consider the following Modula-3D program segment.

```
FOR i := 1 TO 1000 DO
  FOR j := 1 TO 1000 DO
    (* skip *)
  END
END
```

It translates into the following C program segment:

```
_z1 = 1 ; _z3 = 1000 ;
{ _LOCAL _t223565b7 i;
  for (; _z1 <= _z3; _z1 += 1)
  { i = (_t223565b7) _z1;
    _z4 = 1 ; _z5 = 1000 ;
    { _LOCAL _t223565b7 j;
      for (; _z4 <= _z5; _z4 += 1)
      { j = (_t223565b7) _z4;
} } } }      ,
```

which in turn translates into the following Mosaic assembly code:

```
  mov #1,-1[bp]
  mov #1000,-2[bp]
  mov -1[bp],r3
  mov -2[bp],r0
  cmp r0,r3
  jgt L115

  mov #1000,r5
L72:
  mov #1,r4
L71:
  inc r4,r4
  cmp r5,r4
  jle L71

  mov -1[bp],r0
  inc r0,r0
  mov r0,-1[bp]

  mov -1[bp],r3
  mov -2[bp],r0
  cmp r0,r3
  jle L72
L115:      .
```

As is seen from this example, the assembly code generated from the Modula-3D code is appalling.

Speed

Now we turn our attention away from size and focus on speed. Table 3.1 shows the execution times of several tests, each performed with and without run-time checks, respectively (as in the leftmost and rightmost *Bytes* columns in Table 3.0). The columns of Table 3.1 indicate the task, the number of microseconds used to

Task	μs	μs	allocations	garbage collections	
1 of 1,000,000 iterations of skip	0.240	0.240	[0,0]	[0,0]	[0,0]
local method invocation	2.70	1.70	[0,0]	[0,0]	[0,0]
remote method invocation, 1 hop away	476.0	341.0	[1,1]	[9,6]	[6,4]
remote method invocation, 10 hops away	477.0	344.0	[1,1]	[9,6]	[7,4]
unacknowledged send operation	51.7	33.7	[0,0]	[0,0]	[0,0]
acknowledged `MsgType.IncRef` message	181.6	123.5	[0,0]	[0,0]	[0,0]
untraced object allocation and disposal	72.0	53.0	[1,0]	[0,0]	[0,0]
traced object allocation	60.3	50.3	[1,0]	[2,0]	[2,0]
remote `NEW` call, 2 hops away	497.0	380.8	[1,1]	[6,4]	[3,3]
1 of 100 successive garbage collector invocations	2000	1400	[0,0]	[100,0]	[100,0]
thread creation and 2 context switches	418.0	313.0	[3,0]	[29,0]	[20,0]
pair of context switches	59.0	46.0	[0,0]	[0,0]	[0,0]

Table 3.1: Execution times of some operations

perform the task with and without run-time checks, the number of [local,remote] allocations performed during the task, and the number of [local,remote] garbage collections that were invoked during 10,000 executions of the task with and without run-time checks. The difference between the numbers in the last two columns is due to that the program is smaller when there are no run-time checks (here, 23 KB *vs.* 34 KB), and thus fewer garbage collections are needed.

The "1,000,000 iterations of skip" refers to the nested loop shown earlier in this section. Execution of that code segment involves the execution of 3,008,007 instructions. Thus, we find an average number 12.53 instructions per microsecond, which lets us view the above figures in terms of instructions, shown in Table 3.2.

From Tables 3.1 and 3.2, we notice that a remote method invocation is by far more expensive than a local one. The reasons for the problems discussed above regarding the size of the code (run-time checks and poor compilation) affect the number of instructions executed, and thus also the execution speed of a task. Furthermore, the creation of threads, the sending and receiving of messages, and the packing and unpacking of parameters all require memory allocations. Seemingly, the heap allocator is not very fast, and that slows down these other operations.

On another note, it appears from Table 3.2 that remote method invocations to processors further away take more instructions than those to nearer processors. That is of course not true. Instead, this arises only from that the numbers in Table 3.2 are calculated directly from the times shown in Table 3.1. Nevertheless, it is curious that there is rather large difference in time between the two, as such a difference is supposed to be virtually zero for the Mosaic machine, especially when the network is not loaded.

Task	Instructions	
1 of 1,000,000 iterations of skip	3	3
local method invocation	34	21
remote method invocation, 1 hop away	5964	4273
remote method invocation, 10 hops away	5977	4310
unacknowledged send operation	648	422
acknowledged `MsgType.IncRef` message	2275	1547
untraced object allocation and disposal	902	664
traced object allocation	756	630
remote `NEW` call, 2 hops away	6227	4771
1 of 100 successive garbage collector invocations	25060	17542
thread creation and 2 context switches	5238	3922
pair of context switches	739	576

Table 3.2: Instruction counts of some operations

This concludes our study or performance numbers. We now go on to discuss some improvements that can be made.

3.12 Possible Improvements

In our experience with Mosaic Modula-3D, we have found that although the programming model provides a nice setting in which to write distributed programs, the implementation has limitations. The most frequently encountered of these limitations are memory related.

As shown in Section 3.11, the compiled machine code is not terrific and thus lends itself to improvements. We have also seen that run-time checks account for a good portion of the code. Thus, we desire to reduce the number of run-time checks, without compromising safety, and the space used by each check. We mention some techniques by which this can be achieved.

Having the hardware detect `NIL`-dereferences and stack overflows would help, because the checks need then not be explicit. Being the small and simple processor it is, the Mosaic does not provide any such support.

Aid can also be provided by an instruction set and a compiler that are geared towards having explicit run-time checks in the code. Such a combination can allow the run-time checks to occupy very little space. For example, consider a `NIL` check. A compiler may load a reference into a register, perhaps using a `mov` instruction. If this reference is to be dereferenced, it is convenient to have the `mov` instruction alter the zero flag according to the value of the reference, so the `mov` does not need to be immediately followed by a compare instruction. (In the Mosaic, the `mov` instruction does not modify any of the status flags.)

Then, it would be nice to have a branch instruction that jumps to an address specified in a register or at a given offset from the address stored in some register. This allows the run-time system to keep one register pointing to a table of run-time error reporting routines. The branch instruction can include a small (*e.g.*, 4-bit) constant and the register number, so that the entire branch instruction does not occupy more than one word.

As for the compiler, one would like it not to `NIL` check the same reference again for as long as it remains unchanged in the register. This is something the current C compiler does not do, and perhaps not surprisingly so. However, by letting the Modula-3D front end provide pertinent information about these references, it is possible for a back end to avoid unnecessary duplication of these `NIL` checks. Moreover, a back end can make use of the fact that a call, or jump rather, to a run-time error reporting routine never returns. Hence, no instructions for writing back values stored in registers prior to the call are necessary.

A more advanced feature of the compiler, or auxiliary tool, is to establish a proof of that some particular run-time checks are not needed. This is attempted in [9], and would be a nice addition to the compiler.

We have discussed that the entire run-time system is always present in memory, and that several parts of the run-time system are never used by some programs. Hence, linking only with the pieces that will actually be used would help reduce the amount of code present on each processor.

A more general problem is that some parts of the code are only used during parts of the computation or only by some of the processors. For example, the parser that is part of the LISP interpreter discussed in Section 2.5 is usually only executed during a small part of the computation, and then only on the main processor. Also, some programs may be so large that they do not fit in the memory of a processor at all, even though the parts of the code that are used at any one time do. This problem can be tackled using some form of demand paging, described next.

The code of a program is partitioned into segments. For this, the locality of code determined by the programmer may be exploited by letting one segment correspond to the code generated for one module. Some segments, like the thread scheduler, are always needed on every processor, whereas the others can be loaded on demand. When a thread on a processor calls the code in a segment that is not present in the local memory, the run-time system sends a request to a near processor that has a copy of the that segment. Once the segment is received, the run-time system loads it appropriately and schedules the thread that needed it.

In small processors like the Mosaic, one may not expect to see hardware support for demand paging. Thus, the compiler needs to generate special code for potential inter-segment procedure calls and returns. Identifying segments with modules then seems like an even better idea, as the interfaces between modules are explicit in Modula-3D.

To guarantee that there is always a close by processor that contains a desired segment, and to easily be able to compute the ID of that processor, segment s of N , say, can be stored permanently on all processors whose integer ID modulo N is s .

Finally, we discuss the event of running out of memory at run-time. In a machine like the Mosaic, which is designed to have 16 K nodes, each of which is equipped with 64 KB of RAM, the total amount of memory is 1 GB. With that much memory, one would not expect to run out of memory for a large proportion of programs run today. However, in the current Mosaic Modula-3D implementation, if one nodes runs out of memory, the entire computation stops with a run-time error.

Anticipating the need for techniques that attempt to reduce the chances of getting such out-of-memory faults, the Mosaic Modula-3D run-time system features a queue, onto which threads with unsatisfiable memory requests may be placed. In particular, if immediately following a run of the garbage collector, there still is not enough memory to satisfy a given allocation request, the requesting thread may be placed on the said queue. After each garbage collection, the threads on this queue are woken up so that they may once again attempt their requests. We have not experimented with actually adding threads to this queue, so we cannot report on the difference this may make.

The current Mosaic Modula-3D run-time system responds to each incoming request as the interrupt for it is received. For example, as soon as the interrupt handler is invoked to handle an incoming remote procedure call, the interrupt handler allocates some memory for the incoming message and then creates a thread to execute the call. An alternate scheme is to queue up some of these incoming requests when free memory is particularly low. That may solve some of the problems of running out of memory on a processor, but not all, since the existing threads may be suspended on condition variables that are to be signaled in other incoming remote procedure calls that are queued.

A more general solution is to, by changing the language definition, give the run-time system some freedom in distributing data and control. For example, if a `NEW` request is received by a processor with only a small amount of free memory, the request can be forwarded to some other processor with more memory. A related scheme is explored in Madre (see [1]).

Chapter 4

Conclusion

We have presented for an object-oriented language some extensions that support programming fine-grain multicomputers. We have applied these to Modula-3, creating a new language, Modula-3D. We presented the language definition, and remarked on some of the language definition decisions we made and on what we rejected. Using several sample programs, we discussed strengths and weaknesses exhibited by Modula-3D. We also described some interesting aspects of our implementation of Modula-3D on the Mosaic multicomputer at Caltech.

Meeting our goal of furnishing an easy-to-understand and easy-to-use language for multicomputers, we aim to help bring back the programmer's attention from issues dealing with interprocessor communication to finding algorithmic solutions for the actual problems at hand. For the ease of distributing parallel programs, the comfort of using a safe multicomputer language, the ability to write distributed solutions to general problems, and the convenience of reusing existing solutions, we think Modula-3D the right choice.

Bibliography

[0] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: A language for distributed programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[1] Nanette Boden. Run-time systems for fine-grain multi-computers. Technical Report Caltech-CS-TR-92-10, California Institute of Technology, 1993. PhD thesis.

[2] Per Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[3] Per Brinch Hansen. Concurrent programming concepts. *ACM Computing Surveys*, 5(4):223–245, December 1973.

[4] Per Brinch Hansen. *Operating systems principles.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[5] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–206, June 1975.

[6] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[7] Peter Carlin, K. Mani Chandy, and Carl Kesselman. The CC++ language definition. Technical Report Caltech-CS-TR-92-02, California Institute of Technology, 1993.

[8] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. Technical Report Caltech-CS-TR-92-01, California Institute of Technology, 1992.

[9] Dave Detlefs and Greg Nelson. Private communications on extended static checking, 1992, 1993.

[10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

[11] Marcel R. van der Goot. Multicomputer C user manual. Technical Report Caltech-CS-TR-93-17, California Institute of Technology, 1993.

[12] C.A.R. Hoare. Towards a theory of parallel programming. In C.A.R. Hoare and R.H. Perrott, editors, *Operating Systems Techniques.* Academic Press, New York, 1972.

[13] C.A.R. Hoare. A structured paging system. *Computer Journal*, 16(3):209–215, 1973.

[14] C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[15] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[16] INMOS Limited, Prentice-Hall Int., Englewood Cliffs, NJ. *Occam Programming Manual*, 1984.

[17] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, 1992.

[18] G. Le Lann. Distributed systems, towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

[19] K. Rustan M. Leino. Extensions to an object-oriented programming language for programming fine-grain multicomputers. Technical Report Caltech-CS-TR-92-26, California Institute of Technology, 1992.

[20] Johan J. Lukkien and Jan L.A. van de Snepscheut. A tutorial introduction to Mosaic Pascal. Technical Report Caltech-CS-TR-91-02, California Institute of Technology, 1991.

[21] Alain J. Martin. Distributed mutual exclusion on a ring of processes. *Science of Computer Programming*, 5:265–276, 1985.

[22] David May. OCCAM. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983.

[23] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[24] Greg Nelson. Private communications, 1992.

[25] E.A.M. Odijk. The DOOM system and its applications: a survey of ESPRIT 415 subproject A. In *PARLE 87*, volume 258 of *Lecture Notes in Computer Science*, pages 461–479. Springer-Verlag, 1987.

[26] Charles L. Seitz. Concurrent VLSI architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, December 1984.

[27] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[28] Charles L. Seitz. Mosaic C: an experimental fine-grain multicomputer. In Alain Bensoussan and Jean-Pierre Verjus, editors, *Future Tendencies in Computer Science, Control, and Applied Mathematics*, volume 653 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992. Proceedings of the International Conference on the Occasion of the 25th Anniversary of INRIA.

[29] Charles L. Seitz. Submicron systems architecture, semiannual technical report. Technical Report Caltech-CS-TR-92-17, California Institute of Technology, 1992.

[30] Charles L. Seitz, Nanette J. Boden, Jakov Seizovic, and Wen-King Su. The design of the Caltech Mosaic C multicomputer. In *University of Washington Symposium on Integrated Systems*, March 1993.

[31] Jakov N. Seizovic. Architecture and programming of a fine-grain multicomputer. Technical Report Caltech-CS-TR-93-18, California Institute of Technology, 1993. PhD thesis.

[32] DEC SRC. DEC SRC network objects. To be published, 1993.