Extensions to an object-oriented programming language for programming fine-grain multicomputers

K. Rustan M. Leino
Computer Science
California Institute of Technology
Pasadena, CA 91125
4 December 1992

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.

0 Introduction

Due to their "modest cost and open-ended expandability", fine-grain multicomputers are an increasingly more attractive means for performing large computations ([9, 8]). 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 [9] 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 justify our choice of Modula-3, present the language extensions, discuss their implementation, and show their use in a simple example. The proposed language extensions have been implemented on the Mosaic multicomputer at Caltech.

1 Why Modula-3?

We have chosen Modula-3 as a 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 [10]). 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 [6] or [0], 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 [1, 2]). However, as C++ is, for one, not a simple language (see [3]), 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 [11]). 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.

Yet another attempt at combining object-oriented and concurrent programming is the Parallel Object-Oriented Language, POOL (see [7]). 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.

2 Language Extensions

Rather than introducing explicit interprocessor communication, we introduce *network objects*. These are values of special *network object types*. The data fields and method suites of network objects are stored on one processor, but their methods can be invoked from any processor with a reference to them.

To add network objects to a language, we need essentially two mechanisms: one to identify network object types and one to create values of these types. In Modula-3, (traced) object types form a hierarchy descending from the built-in object type `ROOT`. We extend Modula-3 with a new built-in object type called `NETWORK`, a subtype of `ROOT`. Subtypes of `NETWORK` are called network object types. When `NEW` is used to create a network object, it can take an extra parameter. This parameter indicates on which processor the object is to be allocated.

Associated with these extensions are some restrictions. To describe the most vital of these, we first introduce some terminology. The processor on which 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 resides. However, to make implementations of network objects feasible on machines where processors have separate memories, 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 or return value 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 duplicated 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 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 their methods. Program execution terminates when the body of the main module terminates, regardless of execution of other threads on any processor. Implementations need also provide each processor with the ability to get its processor ID. One way of doing this is to add a variable Main.pid, initialized by the runtime system to the current processor ID.

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:

- fields of network objects are accessed via methods only,
- the only global variable used in a user program is the processor ID, and
- 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.

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.

3 Implementation

In order to implement the above language extensions, the runtime system needs to be able 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 [11].) 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 [11] for a variation of these surrogates.)

Creating a remote network object involves interprocessor 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 an exception) 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. 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. Note that the language definition could easily restrict the processors on which a client can create objects. However, restrictions on which objects could be referenced from where would be unacceptable, because then programmers need to distinguish between different objects of the same type. For this reason, the language extensions presented here do not restrict the processor parameter of `NEW`.

The runtime 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 network object on some other processor; that is, whether any surrogate of the object exists. To this end, 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 Greg Nelson,[5].) Finally, when a local garbage collector reclaims the storage of a surrogate object, it notifies the host which then decrements the reference count.

4 Sample Program

We give a solution, written in our extended Modula-3, to a standard parallel programming problem, namely that of generating prime numbers by means of the sieve of Eratosthenes (see, for example, [4]). Before we start, we introduce a type, `Buffer`, that we will need. This type features `put` and `get` methods to access the buffer. Elements are read in first-in first-out order, and `put` and `get` operations are suspended if the buffer is full or empty, respectively.

```
TYPE
  Buffer = 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;

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;

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;
```

Now, 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 follows:

```
TYPE
  PrimeConsumer = NETWORK OBJECT
      (* ...  *)
    METHODS
      consume( x: INTEGER ) := Consume;
      end() := EndConsume
    END;
```

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

We now 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 `G.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;
```

Now, we can declare procedure `Generate`:

```
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;
```

and also the main body of our program:

```
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:

```
TYPE
  Filter = OBJECT
      cl: FilterClosure
    METHODS
      (* as before *)
    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 need to make sure the main body does not terminate until its end method 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 running on one processor, we face the issue of distributing the data and control of our program. With our primitives, this is the easy part; 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;
```

This change, 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, pid+1 )
```

As our program adheres to the distribution conditions given in a previous section, 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 could, for example, write a procedure, `MapFilter` say, that maps more `Filter` objects to each higher numbered processor. Then the call to `NEW` would look something like:

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

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

Another curious way to tune the performance of this program, enabled in its simplicity by our use of an object-oriented language, is to call the `Buffer.init` method with a parameter other than the default 1. This introduces some amount of slack. It is interesting to note that the slack is just another parameter computed at runtime, so instead of being restricted to some fixed slack, as in many multicomputer languages today, using different size buffers is here just a matter of programming.

5 Conclusions

We have presented some extensions to an object-oriented language that support programming fine-grain multicomputers. These extensions are easy for programmers to understand. We have shown a sample program in which distributing the computation and its data was entirely trivial. Since the distribution conditions are met, the distribution of the computation and its data is easily reconfigured to enhance performance without rethinking the program design. We also showed how the runtime system can be changed to support our language changes. These changes can be implemented efficiently. Also, as only network objects are shared among the processors, other types do not require any overhead.

In short, satisfying the requirements stated in the introduction, our language extensions help bring back the programmer's attention from issues dealing with interprocessor communication to solving algorithmic problems. And due to the existence of the simple distribution conditions, we have shown that distributed programming need not be any harder or more involved than parallel programming in general.

6 Acknowledgements

I am very grateful to my advisor, Jan L. A. van de Snepscheut, for many discussions, encouragement, and ongoing support. I am also grateful to Greg Nelson for his enthusiasm about my starting this work, and to Robert J. Harley and H. Peter Hofstee for the many comments and good suggestions they provided on the writeup of this paper.

References

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

[1] 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.

[2] K. Mani Chandy and Carl Kesselman. The CC++ language definition. Technical Report Caltech-CS-TR-92-02, California Institute of Technology, 1992.

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

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

[5] Greg Nelson. Private communications.

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

[7] 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.

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

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

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

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