Adding Concurrency to $\mathcal{LOOM}$

by

Hilary K. Browne

A Thesis
Submitted in partial fulfillment of
of the requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts
May 21, 1997

Abstract

Over the past decade, both object-oriented and concurrent programming languages have become popular tools for solving complex problems, and language designers have recognized that combining these two paradigms can be useful. Concurrency allows object-oriented languages to address naturally concurrent problems and potentially speed up execution. Object-oriented features can improve the organization of concurrent programs and potentially remove some of the low-level process control by encapsulating it in object structure. In this thesis, we explore the addition of concurrency to the object-oriented language $\mathcal{LOOM}$, focusing on providing a safe, easy to use language for programmers familiar with object-oriented programming but not necessarily with concurrency. We introduce a large grain of concurrency by identifying processes with objects, and allow these objects to communicate both synchronously and asynchronously using the existing message structure. Special semantics for self-inflicted messages prevent deadlock when processes send messages to themselves, but mutually referential objects can cause deadlock. Conditional synchronization of message reception is handled with separately inheritable pre- and post-condition sections in classes, avoiding extensive redefinitions of synchronization constraints in subclasses and most forms of inheritance anomalies.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Once upon a time, the designers of object-oriented languages and the designers of concurrent languages had very different goals. Quite simply, while the former focused on software engineering problems, the latter were concerned with speed. Oddly, these opposing software goals both stemmed from the same hardware developments. Over the past decade, increasingly powerful computer technology has allowed programmers to tackle larger and more complicated problems. Faster processors have led some language designers to focus on software engineering goals to support the coding and organization of large programs. Architectures consisting of parallel and distributed processors have led other designers to create languages that can take advantage of multiple processing units to solve computationally intensive and naturally concurrent problems more rapidly.

The designers of object-oriented languages noted that solutions to certain large problems often required programs with different components responsible for different functions. They focused on designing languages to support high-level software engineering goals like code modularity, reuse, and maintainability, using abstract object modules to isolate the data and functionality of different program components from each other. With these languages, a programmer could reuse or extend an object simply by knowing about its publicly available interface of data and operation types. Changes made to the actual implementation code of the object would thus be transparent to clients using the object unless the interface changed as well. By raising the level of language constructs above any specific architecture, designers also sought to improve program reliability and type-safety. Designing a strict but still expressive set of constructs removed from low-level architecture considerations minimized the chance that programmers could exploit the architecture in unreliable ways, and allowed for the easy integration of compile-time type-checking algorithms. The run-time overhead incurred by raising the level of code (e.g. garbage collection, dynamic method invocation) was unfortunate, but considered acceptable given increasing processor speeds.

Conversely, designers of concurrent languages saw increased processor speed and parallel and distributed architectures as an opportunity to address naturally concurrent applications and time or data intensive problems that previously had been, for practical purposes, unsolvable. Independent tasks within a program could be dispatched to different processors to be executed in parallel, or the same repetitive task could be performed by many processors on large chunks of data. Any overhead required to allow separate processors to communicate or cooperate was outweighed by the overall gain of using multiple processors. Thus, designers created concurrent languages with low-level constructs to spawn multiple abstract processes, to protect shared data from multiple process contention, and to control process communication and synchronization. The more direct control a programmer had over such processes, the faster his programs would run if sufficient physical processors or operating system threads were available. Such programs were more difficult to reason about because of the non-deterministic execution order of many potentially interfering processes, but the speed payoff could be enormous.

Designers of both types of languages have met with a great deal of success in recent years. Object-oriented languages such as Eiffel [Wei95] and C++ [ES90] have proven their merit for large programming projects. Languages with concurrent constructs such as Ada and Linda [BA90] can take advantage of powerful and complex architectures to solve more difficult problems, while languages with support for distributed concurrent computations such as Obliq [Car95] are becoming extremely important for network programming. Until recently, neither group paid much attention to the design goals of the other, because in many respects they were contradictory. While both sought to create languages that abstracted from the bare architecture, their approaches were quite different. Object-oriented designers sought to help programmers write reliable programs by insulating them entirely from architecture considerations with a strict set of high-level constructs. However, many concurrent designers felt that programmers could write faster programs if they had few restrictions placed on their ability to manipulate processes on an abstract architecture of processors.

Thus, safety and speed concerns were opposed in these two language design choices. The overhead of high-level, type-safe, object-oriented languages slowed execution speed. Low-level concurrent languages improved speed but the non-deterministic programs they supported jeopardized the data integrity that object-oriented languages sought to ensure, and they generally provided little support for code reusability and scalability. However, as technology for machine architectures is rapidly approaching the limit imposed by the speed of light, and theoreticians have formalized elegant mathematical proofs for type-safety, the gap between these two conflicting language designs is narrowing. It is becoming more difficult to increase the speed of processors and easier to implement safety features such as type-checking. Further, researchers including Robert Harper at Carnegie Mellon University have shown that translating high-level languages into typed target languages for compilation not only provides the expected security and correctness benefits, but also allows compilers to exploit the type information to make efficiency gains [TMC+96].

Object-oriented language designers have realized that they can no longer rely on the promise of faster machines to compensate for slow sequential code. At the same time, concurrent language designers have recognized the benefits of code modularity, reusability, and type-safety to large programming projects, particularly now that they can be implemented to incur less overhead. Researchers in both groups have realized that an object-oriented paradigm is an appropriate medium in which to combine the high-level software engineering and type-safety benefits already present with the improvement in execution speed produced by taking advantage of parallel and distributed architectures. Objects are clearly appropriate for encapsulating data on distinct processors, but can also be extended to handle concurrent execution. In particular, an individual object can concurrently execute more than one method, and many objects can execute their methods in parallel. Not surprisingly though, there are myriad interactions to consider when combining object-oriented and concurrent constructs, many relating to the tension between fast, unreliable low-level designs and slower, safer high-level designs.

1.2 Outline

This tension between high- and low-level design decisions means that the marriage of object-oriented programming with concurrency depends heavily upon the priorities of the designers, who must consider their audience of programmers and the features and functionality they will require. In this thesis, we are specifically interested in how to modify $\mathcal{LOOM}$, an experimental object-oriented language developed at Williams College in 1995 and 1996, to support concurrency. We will refer to this new language as Concurrent $\mathcal{LOOM}$. Though the original $\mathcal{LOOM}$ is a fairly small language, making it useful for experimentation, it supports an interesting combination of powerful features designed to ensure type-safety without restricting expressibility. The strong type system, a major priority in the language, was designed to help programmers avoid code that appeared to be correct at compile time but could break at run-time. We wish to maintain this spirit of safety by extending $\mathcal{LOOM}$ to support concurrency without introducing too many new, potentially abusable constructs and without introducing too much of the non-determinism that makes concurrent programs difficult to reason about. As much as possible, we would like to preserve the high-level of easily understandable, usable, and checkable constructs that motivated the original $\mathcal{LOOM}$ design. Our audience, then, is safety-minded object-oriented programmers who desire the added functionality of concurrency, with speed a secondary concern.

We will explore the issues involved in combining concurrency constructs with the object-oriented paradigm, comparing the choices made by a variety of concurrent object-oriented languages to motivate our decisions for Concurrent $\mathcal{LOOM}$. In Chapter 2, we explain the basics of the sequential $\mathcal{LOOM}$ language and provide some sample code to introduce the syntax. In Chapter 3, we review some basic properties of concurrent programming that will be important in our design. In Chapter 4, we discuss some early decisions to be made for concurrent objects and those made for Concurrent $\mathcal{LOOM}$. In Chapter 5, we continue to explore language decisions, focusing on communication and synchronization issues. In Chapter 6, we discuss conditional synchronization and one of the most pernicious interactions between concurrent and object-oriented constructs: "inheritance anomalies." In Chapter 7, we present informal semantics for type checking and evaluating Concurrent $\mathcal{LOOM}$ programs, focusing on new features and changes made to the original $\mathcal{LOOM}$ interpreter. In Chapter 8, we summarize the comparisons made throughout the thesis between Concurrent $\mathcal{LOOM}$ and other concurrent object-oriented languages, discuss future work, and draw conclusions. Two appendices follow, the first containing example Concurrent $\mathcal{LOOM}$ programs referred to in Chapters 5 and 6, and the second containing a complete grammar for Concurrent $\mathcal{LOOM}$.

Chapter 2

$\mathcal{LOOM}$ Basics

The programming language $\mathcal{LOOM}$ (derived from Language with Object-Oriented and Matching features) is a descendant of a number of earlier experimental languages developed at Williams College, most recently **PolyTOIL** [BSvG95]. **PolyTOIL** (Polymorphic Type-Safe Object-Oriented Imperative Language) is a statically-typed, object-oriented language designed to be both flexible and completely type-safe, unlike most popular object-oriented languages such as C++ [ES90] and Eiffel [Wei95]. It is based on two previous languages, **TOOPL** [Bru94] and **TOIL** [vG93, BvG93]. **TOOPL** (Typed Object-Oriented Programming Language) is a functional, statically-typed language with typical object-oriented features, including classes, objects, methods, inheritance, and subtypes. **TOIL** is an imperative language with these same features, and **PolyTOIL** extends the flexibility of the **TOIL** type system with support for parametric polymorphism. An important feature of this parametric polymorphism is a form of bounded quantification called matching, a relation between types similar to subtyping. Recognizing the complexity and redundancy of supporting both subtyping and matching, $\mathcal{LOOM}$ [Pet96] was designed with a similar syntax and semantics to **PolyTOIL**, with the exception that the subtype relation was removed in favor of retaining only the more expressive matching relation. $\mathcal{LOOM}$ also supports a module system to provide a higher level of abstraction than classes and to allow for separate compilation of program components in the future.

2.1 Object-Oriented Features

$\mathcal{LOOM}$ supports features typical of most object-oriented languages. The simple `points` program in Figure 2.1 illustrates $\mathcal{LOOM}$ syntax and basic object-oriented functionality. The `type` section of the program is used to declare the types of objects, classes, functions, and procedures (i.e., any non-primitive type needed in the program). The object types `PointType` and `ColorPointType` serve as interfaces for point and color point objects. They reveal publicly available (or `visible`) operations called methods and the number and types of method parameters. Implementation details such as state, stored as instance variables, and method bodies are not revealed by object types, a feature that allows such details to be changed without affecting clients relying only on the interface. Objects of type `PointType` have two functional methods, `getx` and `gety`, which return integers, and one procedural method, `move`, which takes two integer parameters. Objects of type `ColorPointType` include all the methods declared in object type `PointType`, as well as a functional `getColor` method and a procedural `setColor` method.

The `const` section contains definitions of classes, functions, and procedures that will be constant in the program. Since $\mathcal{LOOM}$ is statically typed, these constants all must have associated types. The class definitions in this section are used as templates for creating objects. They define and initialize instance variables and define implementations for both the `visible` method signatures declared in object types and for `hidden` methods available only for use by objects created by the class. In the `points` program, the `PointClass` and `ColorPointClass` implementations do not have explicit types associated with them because their types, which would otherwise be declared as class types in the type section, can be built by the $\mathcal{LOOM}$ interpreter on the fly. (This was an optimization added to remove some of the redundancy involved in including nearly identical object type and class type declarations in the `type` section.)

The `PointClass` implementation has two instance variables, `x` and `y`, both initialized to 0, and bodies for three visible methods, `getx`, `gety`, and `move`. The `ColorPointClass` inherits from `PointClass`, illustrating the code-reuse benefits of object-oriented programming. Color points inherit all of the instance variables and methods found in points, and have an additional instance variable, `color`, and two additional methods, `getColor` and `setColor`. `ColorPointClass` is known as a subclass of `PointClass`, and `PointClass` is a superclass of `ColorPointClass`. Subclasses that inherit methods can modify these method implementations, but cannot change the types of these methods as declared by the superclass.

The `var` section of the program lists the names and types of the variables to be used in the program body. The variable `num` is of type integer, a primitive type, while `p` and `cp` are of the user-defined types `PointType` and `ColorPointType`. Finally, the actual $\mathcal{LOOM}$ program body is found between the `begin` and `end` statements at the end of the program. Objects are instantiated by applying the `new` statement to a class definition. `PointClass` is used to instantiate `p`, since it has methods whose names and types match the type declared for `p`, `PointType`. Similarly, `ColorPointClass` is used to instantiate `cp`. The last two statements in the program body are method invocations, or messages, indicated by the `objectname.methodname` form. These statements invoke the `methodname` operation on the `objectname` object. The method signatures in the object types `PointType` and `ColorPointType` provide the only information about `p` and `cp` available for use in the program body. Thus, the only way to update or obtain information about the hidden

```
program points
type
   PointType=ObjectType
        getx, gety: func():integer;
        move: proc(integer,integer);
   end;
   ColorPointType=ObjectType include PointType
        getColor: func():integer;
        setColor: proc(integer);
   end;
const
   class PointClass
        var  x=0:integer; y=0:integer;
        methods visible
                getx= function():integer begin return x; end;
                gety= function():integer begin return y; end;
                move= procedure(newX, newY:integer)
                        begin x := newX; y := newY; end;
   end;
   class ColorPointClass inherit PointClass
        var color=0:integer;
        methods visible
                getColor = function():integer begin return color; end;
                setColor = procedure(newC:integer)
                             begin color := newC; end;
   end;
var
   num: integer;
   p: PointType;
   cp: ColorPointType;
begin
   p := new(PointClass);           -- Create a new point object
   cp := new(ColorPointClass);     -- Create a new color point object
   num := p.getx();                -- Get the x coordinate of p
   cp.move(2,3);                   -- Move cp to (2,3)
end;
```

Figure 2.1: A Simple *LOOM* Program

internal state of these objects is to invoke the methods with messages. The `PointType` object `p` receives a `getx()` message, which will return an integer representing the `x` coordinate of `p`. The `ColorPointType` object `cp` receives a `move()` message with the two parameters indicating the new position to which it should be moved. When the program terminates, `num` will be assigned to 0, and the `x` and `y` instance variables of `cp` will hold 2 and 3, respectively.

2.2 Matching and MyType Features

In addition to the subclass relation based on inheritance, many object-oriented languages also support a subtype relation based on the type system. This relation, denoted by $<:$, holds between two types if one type subsumes the other. In practice, it allows objects of one type to masquerade as objects of another type, for instance by assigning an object of type `ColorPointType` to a variable of type `PointType`. Since color points support all the methods found in points, any message sent to a point could also be sent to a color point. The advantage of such a relationship is that it allows for a useful form of polymorphism known as subtype polymorphism. A programmer can define data structures and functions that automatically operate on many types of objects by virtue of these objects being subtypes of the original type used to define the structure or function. The disadvantage of this relation is that many languages and programmers confuse or conflate the subtype hierarchy with the subclass inheritance hierarchy. While many times two classes in the subclass relation generate objects whose types are in the subtype relation, this is not always true. The result of this confusion is that certain object-oriented languages, such as Eiffel, sacrifice type-safety in order to support the more expressive programs that can be written by identifying the inheritance hierarchy with the subtype hierarchy. Conversely, other languages, such as Trellis/Owl [MK87], sacrifice some expressiveness in order to remain type-safe in situations where subclasses do not generate subtypes. Some languages even suffer from both problems, such as C++, which employs a fairly rigid type-system but allows programmers to explicitly cast types freely and unsafely.

To understand this problem, note that while two classes might be in the subclass relation, if one is allowed to change the types of method parameters and return types in the subclass, objects created by the two classes may fail to be in the subtype relation. Intuitively, one would think that an object of type τ would be a subtype of another type σ if the types of the parameters and return types of the methods of τ were all subtypes of the corresponding parameters and return types of methods of σ. However, while such a *covariant* relationship is necessary for the return types of methods in order for the subtype relationship to hold, it is the opposite, *contravariant* relationship that must hold for parameters. That is, the types of the parameters of the methods of σ must be subtypes of the method parameters of τ. If languages do not enforce this contravariant parameter type rule, one can write programs that compile correctly but break at run-time by trying to perform an operation on a contravariant method parameter that does not support the operation.

In practice, this rule can be very limiting for programmers who want the reuse benefits of subclassing to translate easily into intuitive, subtype relationships so they can take advantage of subtype polymorphism. This problem was exacerbated in **PolyTOIL** with the inclusion of **MyType**, the type identifier associated with the type of an object (i.e., the type of `self`). Like the `Current` identifier in Eiffel, **MyType** is always "anchored" to the object in which it appears. Thus, if a class definition inherits a method with a reference to **MyType**, this reference represents the type of an instance of the subclass, not the superclass. Consequently, any class that inherits a method with a parameter of type **MyType** will fail to be a subtype of the type of objects created by the parent class, since this parameter type will automatically change *covariantly* in the subclass.

The more general *matching* type relation was added to **TOOPL** and **TOIL** for use in type-checking classes, and later proved to be extremely useful in **PolyTOIL** and $\mathcal{LOOM}$ for allowing both expressive and type-safe programs using **MyType**. In $\mathcal{LOOM}$, an object of type τ matches an object of type σ, denoted $\tau <\!\# \, \sigma$, if τ extends σ. More simply, $\tau <\!\# \, \sigma$ if an object of type τ could be generated by a subclass of a class generating objects of type σ. In practice, any message sent to an object of type σ could also be sent to an object of type τ.[1] After writing a number of programs using this relationship, it became evident that nearly any program written using subtyping and parametric subtype polymorphism could also be written using the **MyType** construct and parametric match-bounded polymorphism. In addition, more types can be proven to be in the matching relation than the subtype relation because of the contravariant parameter rule for subtyping. Presenting programmers with both of these relationships seemed complex and redundant. As a result, the $\mathcal{LOOM}$ language was developed as a kind of subset of **PolyTOIL**, supporting matching but not subtyping. $\mathcal{LOOM}$ also disallows the changing of the types of inherited methods, mostly because this ability was not particularly useful with the subtype relation, given the contravariant parameter rule.

There are two advantages to using the matching relationship instead of subtyping. First, one can create container objects that are instantiated using a type variable parameter that bounds the type of objects used by the container. These match-bounded type variables allow the same kind of generic, polymorphic data structures supported by subtype polymorphism in a type-safe way. However, unlike the automatic occurrence of subtype polymorphism, types intended to be used in a match-bounded polymorphic way must be explicitly tagged with the type constructor $\#$. We know an object e has type $\#\sigma$ if e has type τ and $\tau <\!\# \, \sigma$. We refer to types of the form $\#\sigma$ as "hash" types. Such explicit tagging makes the intention of polymorphic structures more obvious. For instance, in Figure 2.2, we

[1] Note that this does not hold true for the so called "hash" types discussed below when **MyType** appears in a contravariant position in the message. For more information, see [Pet96].

```
type
  printable = ObjectType print:proc() end;
  nodetype [T<#printable] = ObjectType
     put_right:proc(mytype);
     get_right:func():mytype;
     set_val:proc(#T);
     get_val:func():#T;
     print:proc();
  end;
  doublenodetype [T<#printable] = ObjectType include nodetype
     put_left:proc(mytype);
     get_left:func():mytype;
  end;
const
  class node(T<#printable; n:#T)
     var value=n:#T;
         right=nil:mytype;
     methods
      visible
        put_right = procedure(other:mytype) begin right := other; end;
        get_right = function():mytype begin return left; end;
        set_val=procedure(new_val:#T) begin value:=new_val.clone(); end;
        get_val=function():#T begin return value.clone(); end;
        print = procedure() begin value.print(); end;
  end;
  class doublenode(T<#printable; n:#T) inherit node;
     var left=nil:mytype;
     methods
      visible
        put_left = procedure(other:mytype) begin left := other; end;
        get_left = function():mytype begin return left; end;
     end;
var anode: #nodetype;
begin
  anode := new(node(printable, x));      -- assuming existence of x
  anode := new(doublenode(printable,y)); -- assuming existence of y
end.
```

Figure 2.2: Generic Node Class

define a generic `node` class whose elements have a type `T` that matches `printable`. To instantiate a `nodetype` object using the `new` command, we would include two parameters in the command, a parameter `T` of a type that supported a `print` method and any element `n` whose type matched `T`.

Second, as we have noted, more types can be shown to be in the matching relation than in the subtype relation, so matching often provides more expressive programs. Notice that with the use of **MyType** as a parameter in the `put_right` method of `node`, the subclass `doublenode` fails to generate objects that are subtypes of `nodetype`. However, `doublenode` does generate objects whose types match objects of type `nodetype`. The **MyType** in objects of type `nodetype` refers to `nodetype`, while in objects of type `doublenodetype`, it is automatically updated to refer to `doublenodetype`. This means that the variable `anode` of type `#nodetype` can be instantiated using either `node` or `doublenode`. Such an instantiation would not be type-safe using subtyping if we (erroneously) assumed `doublenodetype` was a subtype of `nodetype` simply because `doublenode` is a subclass of `node`. Thus, the matching relationship allows a more expressive program, in this case.

2.3 $\mathcal{LOOM}$ as a Base for Concurrency

Type-safety, matching, match-bounded polymorphism, and the **MyType** construct are the features that distinguish $\mathcal{LOOM}$ from many other object-oriented languages. These simple yet robust features, along with the relatively small size of the language, also make $\mathcal{LOOM}$ an attractive choice for the base of a concurrent language. Fortunately, the rather detailed type-checking and matching algorithms of the language remain almost entirely unaffected by the addition of concurrency. Thus, in the remainder of this thesis, we will be able to concentrate on the basic object-oriented features of $\mathcal{LOOM}$ and how they are affected by the addition of concurrency. Only a rudimentary understanding of matching and type-checking concepts are assumed. Interested readers are referred to [Pet96] for more detailed information about type-safety and matching aspects and the addition of modules to the language.

Chapter 3

Concurrency Basics

3.1 Introduction

Parallel, distributed, and concurrent programming are buzz words often misunderstood by programmers used to sequential programming languages. A sequential computer program consists of data declarations and sequentially executed instructions used to model a problem. Such a program is frequently referred to as a process, since it can be executed on a system with a single processor that allocates memory for the data. While sequential programs can be used to model a wide variety of problems, certain problems consist of naturally concurrent components. Other problems are so time- or data-intensive that a single process running on a single processor is far too slow to solve them. Parallel, distributed, and concurrent languages all address these concerns by allowing programs to be written with more than one process to take advantage of potentially many processors.

Parallel languages require an architecture of many physical processors, generally all within the same computer, that can be used to run multiple processes. Parallel programs are usually written to take advantage of this architecture by establishing an explicit correspondence between processes and physical processors. Distributed languages also require that multiple processors be available for explicit assignment, but usually on distinct computers. These computers are loosely connected so that processes can communicate, usually by exchanging messages. However, in a concurrent language, few assumptions are made about the underlying architecture available to programmers. Instead, programmers create processes in abstract parallelism by assuming that each process is allocated its own separate physical processor. In reality, each process might have its own processor (distributed or parallel), may only have its own thread in an operating system running on a single processor, or all processes might share the power of one processor, with a hardware or software scheduler interleaving their instructions. These details are controlled by underlying hardware and software, not by the kinds of explicit process control constructs available in parallel and distributed languages.

One advantage of concurrent programming as opposed to parallel or distributed programming is that the abstraction of processes frees the programmer from many low-level architecture details. A concurrent programming language is therefore portable to more architectures. For instance, a concurrent language with implicit support for distributed architectures could be used to implement an airline reservation system with many computer terminals all accessing a common database. Although a distributed language might be faster in this situation, a concurrent program would be insulated from future architecture changes and upgrades. A concurrent program could also be used to implement an operating system, where multiple user processes, or threads, would share a single processor. In this case, having time-shared or pre-empted threads running on a processor avoids blocking of ordinary sequential processes, since the demands of a human user are often made at a rate much slower than the processor's available cycles.

A second advantage of the concurrent programming abstraction as compared to parallel or distributed programming is that it is conceptually easier for a programmer to reason about. Even if a concurrent program is executed on a single processor by a scheduler interleaving the instructions of various processes, it is easier to think about every process executing simultaneously on its own identical processor, an abstraction not usually possible in parallel or distributed languages when only a fixed number of processors are available. Another advantage to this approach is that a programmer does not have to worry about the rate at which processes are executed. Since the programmer does not know the order in which the statements in his processes are executed, he does not need to be concerned with low-level details like processor speed, allocation, or load-balancing and how these details might influence process interaction. Indisputably, there are many advantages to parallel and distributed programming as well, most obviously greater speed and network capabilities, respectively. However, putting aside speed and distribution concerns and focusing on the simplest way to allow a programmer to handle multiple processes safely, the concurrent paradigm is the clear choice. Thus, a concurrent design, rather than an architecture-specific parallel or distributed design, made the most sense for extending $\mathcal{LOOM}$ to handle programs that require multiple processes.

By thinking of each process as being executed on its own abstract processor, a programmer only has to consider two types of interactions between processes. First, two or more processes might contend for the same resource, such as a communication channel, memory cell, or object. A concurrent language must therefore support implicit or explicit constructs to prevent more than one process at a time from accessing this resource, and to ensure that all processes have fair access to the resource. Second, two processes might need to communicate with each other in order to exchange information. This is usually achieved by allowing processes to test and set shared variables or to send messages to each other. Since processes are executed at unpredictable speeds, communication also implies process synchronization. In order for one process to perform an action and another to detect it, implicit or explicit constructs must be available to ensure that events actually happen in the right order.

Thus, while the concurrent programming abstraction frees a programmer from a number of low-level architecture concerns, the step from a single, sequential process to many potentially interacting sequential processes still requires a language designer to provide constructs that allow a programmer to control these interactions. By implicitly including the semantics of contention resolution or communication in the existing constructs of a sequential language, a designer can take some of this responsibility off of programmers at the expense of allowing them less direct control over processes. Conversely, by providing new, explicit constructs, a language designer gives more control to programmers. In this case, programmers may be able to write faster code, but must take more care to ensure they have used the constructs properly and handled all potential interactions. To illustrate this tradeoff, we will briefly explore some common constructs for specifying concurrent execution and allowing processes to communicate and synchronize. These constructs are also necessary in designing concurrent object-oriented languages, and the tradeoffs between implicit and explicit constructs are even more important because of the software engineering concerns of object-oriented programming. Readers not familiar with concurrent programming are referred to [GM88] and [BA90] for more detail.

3.2 Specifying Concurrent Execution

The most obvious new feature needed to move from sequential programming to concurrent programming is the ability to create multiple processes. While a concurrent program, like a sequential program, usually begins execution with a 'main' process, a concurrent programmer also needs to be able to execute other independent processes. A simple way to achieve this goal if one only has a single processor is with *coroutines.* Coroutines are similar to procedures, but one can transfer control from one coroutine to another in the middle of the routine body. If control is later passed back to the first coroutine, execution resumes in this routine body where it left off, rather than at the beginning. For Concurrent $\mathcal{LOOM}$ we do not assume that we are limited to one processor, and would like to avoid the explicit process switching details inherent in using coroutines.

Another method for specifying concurrent execution is to use the *fork* and *join* constructs popularized by the UNIX operating system [GM88]. A fork statement creates a new process that begins executing concurrently with the invoking process. If the invoker needs to synchronize with the new process, it can use a join statement, which delays the invoker until the new process is finished. This method allows for true concurrent execution, but programmers must be certain they understand the consequences of concurrent execution of all the processes they fork. Other languages, such as Communicating Sequential Processes (CSP) [GM88], support a *cobegin* or *parallel* statement, in which a set of statements is listed to be executed concurrently. Synchronization is implicit in the fact that the cobegin statement terminates only after all of the constituent statements in the set have terminated.

A third popular way to create multiple processes is to identify processes with sequential procedures. In some languages, rather than explicitly activating procedures with fork or cobegin statements, each procedure in a program is started automatically as a process when the program is executed. The disadvantage to such an organization is that only one instance of each procedure can be active. Languages such as Ada [BA90] address this problem by providing constructs to create multiple instances of procedural processes. Either of these options is a more modular way of expressing concurrent execution than scattering fork or cobegin statements around a program. In an object-oriented environment, this option potentially can be extended to identifying processes with methods and message invocation with process creation. Another possibility is to identify objects with processes. In this case, an object process would be activated by receiving messages and then executing them one at a time, or by spawning new processes to execute multiple messages concurrently. We will explore the advantages and disadvantages of all of these possibilities in Chapter 4.

3.3 Shared Variables

Once one has decided on a means to specify concurrent execution of multiple processes in a language, the next step is to determine how processes communicate and synchronize with one another to exchange information and to control access to shared resources. If one assumes that programs will consist of components with access to the same store, a simple way of achieving these goals is to allow processes to atomically test and set shared variables. One process can determine the status of another by atomically testing a variable that both reference, and can inform other processes of its status by atomically setting the variable. When two or more processes share a resource, there are two types of synchronization that may need to take place.

First, if two or more processes perform operations on a shared data resource, such as a buffer, a programmer must ensure that all potential interleavings of instructions executed by these processes produce a correct result. If this property does not hold, a programmer must identify those *critical regions* of code that must be executed without interleaving other statements. This will ensure that these regions are executed in *mutual exclusion* from one another. For instance, the program fragment for a `put` process in Figure 3.1 puts integer items into a buffer stored as an array. If two processes execute the `put` procedure at the same time, different results are obtained by different interleavings of the two procedure statements executed by each process. The `atomic` keyword surrounding these two statements is thus added to ensure that other statements cannot be interleaved between their execution. Second, there may be situations in which the state of a shared resource should prevent the execution of an operation by some process. For instance, a full buffer should

```
procedure put (item: integer)
  begin
    atomic {
      size := size+1;
      buffer[size] := item; }
  end;
```

Figure 3.1: Mutual Exclusion Example

not allow a process to put an item into it. In such situations, the shared resource should be able to delay the operation until other processes have altered the state of the resource, for example by removing an item from the buffer. [GM88] calls this type of synchronization *condition synchronization*, a term we will adopt.

Two of the most well known, structured uses of shared variables for process synchronization are semaphores, introduced by Dijkstra [GM88], and monitors. While other constructs exist, we introduce only these to give a flavor for the issues involved in using shared variables. Semaphores are non-negative integer variables that support two atomic operations, `signal` and `wait`, that determine the number of some group of mutually exclusive regions that can be active concurrently. If s is a semaphore, `signal(s)` executes `s := s + 1`, while `wait(s)` delays until `s > 0` and then atomically executes `s := s - 1`. To implement mutual exclusion using semaphores, a programmer identifies critical regions and then appends a wait operation to the beginning of each and a signal operation to the end of each. Mutually exclusive critical regions use the same semaphore. If a semaphore s is initialized to be 1, then only one critical region protected by s can be active at a time. To allow a fixed number x of critical regions to be active at a time, s can be initialized to x. To implement condition synchronization, a shared variable representing the condition can be protected by a semaphore. A process indicates that a condition is true by signaling the semaphore and delays until the condition is true by waiting on the semaphore. While semaphores solve both the mutual exclusion and condition synchronization problems, they are difficult to use. It is easy to write programs that deadlock when some process waits on a semaphore that is never signaled. It is also difficult to guarantee fairness of execution when many processes access a shared resource. It is easy to forget a `signal` or a `wait` around a critical region, and easy to forget to protect a statement that references a shared resource with a semaphore.

Monitors provide a more central way of protecting shared resources by packaging the resource data and operations into a single structure. The data, representing the state of the resource, is hidden from processes, which can only access the resource via its operations. Mutual exclusion is achieved automatically by only allowing one monitor operation to proceed at a time. Condition synchronization can be implemented in a number of ways. One common way is to use `signal` and `wait` on special condition variables. Waiting on a condition variable causes the current operation executing in the monitor to delay and release its exclusive control of the monitor. Signaling a condition variable causes the current operation to suspend and activate another operation waiting on the variable. If no other operations are waiting, the operation that signals continues execution. Monitors enforce a larger grain of concurrency than semaphores by automatically making every operation in the monitor a critical region. However, they are a more organized model for protecting shared resources and are more easily modifiable since their hidden data representations can be changed without affecting a client's use of the public operations.

3.4 Message Passing

While semaphores and monitors require separate constructs for communication and synchronization of processes, both of these operations can be achieved by allowing processes to send messages to each other. Furthermore, processes that send and receive messages do not need access to a shared store as they do to access shared variables. Thus, message passing is more useful for distributed or potentially distributed systems, though it requires more overhead than simple variable tests and updates. Two processes can communicate if one process sends a message to the other with some information (usually included in the form of parameters) and the other process receives the message with this information. Unconditional synchronization occurs automatically, since a process cannot receive a message until it has been sent. As a result, mutual exclusion is easily programmable by making a shared resource its own process and letting the process control receipt of messages from multiple clients. Condition synchronization can be programmed by conditioning the sending and receiving of messages with boolean tests, sometimes called *guards*.

The two main issues to be resolved when designing a language using messages are how senders and receivers identify one another and how they synchronize. In some languages, such as CSP, senders and receivers must explicitly name each other. However, for certain applications, particularly client-server programs, messages need to be sent and received based on content, rather than process identity. To address this need, the Linda language [GM88] uses pattern matching to allow processes to insert and remove message-type entities called tuples from a pool called a tuple space. In Actor-based languages [Agh87], processes send and receive messages with mailbox queues, so that a server process can receive messages from multiple clients. This identification choice usually depends on the target applications, though in some languages, more than one protocol is supported.

The same choice holds true for the second issue, synchronization of message sending and receiving. When a process sends a message, it doesn't necessarily have to wait for another process to execute a corresponding receive. The message could be buffered by the receiving process to be handled when it was ready, and the sender could proceed immediately. This form of communication is known as asynchronous message passing. Alternatively, the sender could wait for the receiver to accept the message before proceeding, a protocol known as synchronous message passing. Once this rendezvous has been established, the sender and receiver can proceed concurrently. In some protocols, such as Remote Procedure Call or Ada tasks, an extended rendezvous can be established between the sender and receiver by delaying the sender until the receiver explicitly releases the sender from the rendezvous. Both synchronous and asynchronous message passing imply that receivers are blocked until they are sent a message, but some languages also allow a non-blocking receive statement to test if a message has been sent. Languages such as Ada also allow senders and receivers to synchronize conditionally by sending and receiving guarded messages and to prioritize a number of alternative sends or receives.

Languages differ on which form of communication to support. At some point, most programs require processes to synchronize so that senders and receivers can share information about their current states. Synchronous communication is preferable in this case, and usually faster as well. Asynchronous communication is usually slower because of the overhead of managing message buffers, but in many applications considerable gains in overall program speed are made by avoiding unnecessary extended rendezvous. Note also that asynchronous message passing can be programmed with synchronous messages by explicit buffering of messages by receivers, while synchronous message passing can be implemented with asynchronous messages by following every send statement with a receive statement to wait for an acknowledgment. Thus, some languages support synchronous communication, others asynchronous, and others both, depending on the target applications and whether designers want to make programmers explicitly design the protocol they don't support. In Chapter 5, we discuss communication and synchronization issues in concurrent object-oriented languages. Message passing will be especially important here, since method invocation, a form of message passing, is already used in most object-oriented languages. First, we turn our attention to some of the early decisions to be made in designing a concurrent object oriented language, including the specification of concurrent execution.

Chapter 4

Early Design Decisions

4.1 Language and Architecture Orientation

The most basic decision to be made when designing a concurrent language is the target architecture or architectures. Although concurrent languages are designed to manipulate abstract processes, many also support process constructs geared toward some abstract architecture or operating system structure. A concurrent language can be designed to exploit multiple operating system threads on a single processor, parallel processors in the same machine, multiple processors distributed among many machines, or any combination of these architectures. The advantage of such an approach is that knowledgeable programmers can speed up their code to exploit a specific type of architecture, if they so desire. In other concurrent languages, most decisions involving process and processor management are left up to the underlying operating system, compiler, or interpreter. The advantage to this approach is portability: the fewer assumptions a program makes about the underlying architecture, the more architectures on which it can be run. In addition, the fewer special constructs a language supports, the less opportunity for programmer error in using these constructs. The decision usually depends on the class of problems the language is designed to solve and whether or not the designers have a specific type of architecture in mind for programs written in their language.

In Java [GJS96], a popular object-oriented language, the concurrency primitives are designed around the assumption that a programmer might only have one processor to work with, and that the underlying operating system might implement either time-sliced or preempted threads. Java provides a thread class, from which other classes can inherit, to implement concurrency. The language designers chose to give programmers a great deal of control over the abstract thread life-cycle. Programmers can explicitly cause threads to wait, sleep, or suspend, and then notify them to resume execution. By using knowledge about the underlying architecture, a programmer can exercise more control over threads to utilize processor resources efficiently. For instance, if a programmer is aware that the operating system is implemented with preemptive threads, he can program his threads to yield to other threads of the same priority so that all threads get a chance to execute. Although this gives a programmer more control over the perhaps limited resources available to him, it also places a lot of responsibility on him to manipulate potentially many threads correctly.

In his concurrent extension to Eiffel [Mey97], another popular object-oriented language, Bertrand Meyer makes fewer assumptions about the underlying architecture. A process might be controlled by its own processor or an operating system thread. The only distinction programmers are required to make is whether an object is handled by the same abstract processor as its creator or a different one. To do this, programmers attach the keyword ‘separate’ to object or class declarations to indicate that an object or each instance of an object should be handled by a newly-spawned process. Since processors are abstract resources, there are an unlimited number. Of course, if only one physical processor or thread is actually available, the distinction is meaningless, though certainly not harmful. The programmer must also create a ‘Concurrency Control File’ to specify the actual physical machines that are available to a program at run-time for running processes or threads. This is a task that could be automatically handled by some operating systems or compilers to preserve the process abstraction. Thus, the programmer is not entirely free from hardware details, but at least these details are all gathered in one modifiable file.

In Concurrent $\mathcal{LOOM}$, one of our goals is to preserve the high-level insulation from architecture concerns supported by the object-oriented programming paradigm. We do not want to support constructs geared toward a specific abstract architecture, as in Java, because such constructs are not portable. Further, the finer control an inexperienced concurrent programmer has over processes or threads, the more potential for error, either through misuse of certain constructs or through a failure to anticipate certain undesirable execution paths. Integrating concurrency demands that we at least support the concept of abstract processes, but the less responsibility a programmer has in manipulating these processes, even the minimal extent in Eiffel, the better. This viewpoint unquestionably sacrifices speed for safety, but our hope is that smarter compilers and operating systems will compensate somewhat for this deficiency in the future with more advanced processor and thread allocation schemes.

Currently, $\mathcal{LOOM}$ is implemented as an interpreted language using Standard ML of NJ (SML-NJ), which supports a variety of concurrency primitives. Our goal in designing Concurrent $\mathcal{LOOM}$ is to handle as many of the concurrency details as possible implicitly with the interpreter, rather than explicitly with new language constructs. As long as such decisions can be made without inhibiting expressibility, we believe the language will be easier and safer for programmers to use. In addition, any improvement in the handling of concurrency primitives at any level below the interpreter will translate automatically into improvements for Concurrent $\mathcal{LOOM}$.

4.2 Object Models

The next decision to be made when designing a concurrent object-oriented language is how to integrate multiple processes into a sequential object model. This decision depends to a certain extent on whether one is designing from scratch, modifying an existing language, or extending an existing language. Creating a new language generally allows for the most freedom, changing an older language allows somewhat less freedom, and extending a language the least freedom, since changes must be backward compatible. In all of these cases though, the most major changes necessary to support concurrent execution involve either implicit or explicit language constructs to protect object state from unpredictable concurrent access. More than one object at a time may call a method in another object, or one object may call multiple methods on another object without waiting for one method to finish before beginning to execute another. If an object can process multiple methods concurrently, any interleaved execution order of the statements in these methods must produce a satisfactory result. If this is not possible, either the programmer must identify and protect the critical regions of code that must be executed without interleaving statements from other regions of code, or the language designer must build in this facility. If explicit constructs are provided, a programmer can dictate the size of critical regions as entire objects, individual methods, or groups of statements within methods. The finer control the programmer has, the faster his code can run, but the more danger of error from unanticipated execution paths.

Michael Papathomas [Pap95] identifies three general approaches used by language designers to structure concurrent objects. In the *orthogonal* approach, concurrency is integrated into a language independently of the object model. Sequential objects are accessible by concurrent threads without any implicit protection. Thus, language designers provide programmers with explicit constructs such as semaphores or monitors for manipulating processes as well as for controlling concurrent access to objects. Languages such as Smalltalk-80 [YT87], Trellis/Owl [MK87], and Emerald [BHJL86, BHJ$^+$87] all follow this approach, and provide a variety of constructs to implement mutual exclusion and conditional synchronization. This approach generally provides the finest control and is the easiest for a language designer extending an existing language to implement, since objects are not modified. However, it is the most dangerous as well, since programmers must identify and properly protect every critical region in every object.

In the *heterogeneous* design approach, two types of object are provided: sequential, or passive, objects, that cannot be invoked concurrently, and concurrent, or active, objects, which implicitly or explicitly handle concurrent execution. In Java for instance, sequential objects can be made active if their class definition inherits from the 'thread' library class. To implement mutual exclusion, programmers can explicitly manipulate threads to protect critical regions or declare methods to be *synchronized.* If a thread invokes a synchronized method in an object, no other threads can access the object until the method finishes executing. This construct provides less fine control than explicit thread manipulation, but leaves fewer execution paths open and is easier to program. This approach is somewhat harder to implement as a language extension, since existing object structures may need to be modified.

Finally, in the *homogeneous* approach, all objects are considered active objects, automatically supporting concurrent operations. Mutual exclusion is generally provided at the method level, as in Java synchronized methods, or object level, similar to monitors, rather than the statement level, in an effort to use object structure to incorporate concurrent execution. Objects may implement conditional synchronization by delaying the receipt of messages in order to prevent certain messages from being processed at certain times. In POOL [AvdL90, Ame87, Ame89] and Meyer's Eiffel objects, only one method in an object can be activated at a time, automatically making the entire object a critical region. In POOL objects, the programmer specifies conditions for the acceptance of each method in a special object routine that executes during the object's lifetime. These conditions generally involve tests of values of object instance variables. There is no explicit control of processes within the methods. In Eiffel, there is no loop body, but method calls indicate when they are acceptable through the pre-condition tests of the sequential version of the language. In either case, delayed method invocations are saved until the object can handle them. This approach is the hardest to implement as a language extension, since all objects must be modified to handle concurrent execution.

Papathomas argues convincingly that this last approach is the most useful. In the orthogonal approach, objects have no implicit protection from concurrent execution. The default is no protection, so programmers must identify both what requires protection in programs and how to implement it properly. The heterogeneous approach is somewhat better, since some of this protection can be built into the language design. However, supporting two types of objects is cumbersome to the designer and confusing to programmers who must distinguish and decide between them. The homogeneous method is thus better for two reasons. First, providing only one type of object is conceptually simpler for both designers and programmers. A programmer's code also has greater potential for reusability if only one object type is available. Second, by integrating concurrency control into object structure, language designers can choose to take some of the responsibility of concurrency control off of programmers. At the expense of less fine control over threads or processes, designers can automatically allow only one method to be active in an object at a time and leave programmers responsible only for identifying condition synchronization for method calls.

Clearly, the homogeneous approach seems most appropriate for Concurrent $\mathcal{LOOM}$. We are under no obligation to make the language backward compatible. However, the fewer changes and additions to the language, the better. The homogeneous approach meets both of these concerns. Although it requires the most modification of the semantics of an existing language, the resulting language can be much simpler for programmers to use than languages created using the orthogonal or heterogeneous approaches. Concurrent $\mathcal{LOOM}$

is aimed at programmers who are familiar with sequential object-oriented programming, but not necessarily concurrent programming. A concurrent object-oriented language would probably not be the choice of an experienced concurrent programmer interested in performance. Rather, the target audience is programmers interested in the software engineering benefits of the object-oriented paradigm with the added bonus of concurrent execution. The homogeneous object approach allows for the simplest presentation of concurrent objects with the least potential for misuse of concurrent features.

4.3 Concurrent Granularity

In the same paper, Papathomas also identifies three classifications for concurrent objects according to the number of internal processes or threads of control they can manage. This decision dictates the granularity of the concurrency in a language. In *sequential* objects, only one process can be active in an object at a time. This approach enforces a coarse grain of concurrency, much like monitors, as entire objects are the smallest entity that can be executing concurrently. This approach minimizes the number of potential execution paths in a program, simplifying the responsibilities of programmers at the expense of fine control over many processes. As mentioned above, the POOL family of languages and Meyer's Eiffel are two examples of languages that enforce the sequential operation of objects. A third example is Ada [BA90, GM88], whose task structures represent objects. Like POOL objects, Ada tasks can indicate the conditions under which a method call is acceptable. In all of these languages, if no explicit conditions are given in a body or some kind of pre-condition, a method call is always acceptable as long as no other method is currently active.

The second classification Papathomas discerns is *quasi-concurrent* objects. Like sequential objects, quasi-concurrent objects only support one active process at a time. However, quasi-concurrent objects can manage multiple processes and interleave their execution, somewhat like monitors that allow signaling and waiting within monitor procedures. It is up to the programmer to explicitly delay one process in order to allow the execution of another. This method provides somewhat finer control at the expense of more potential interference within an object. A programmer must be certain that when a delayed process is resumed, the state of its object is consistent with the way it was left before the delay. An example of such a language is Hybrid [KNP88], whose domains represent objects. To interleave activities within a domain, a method invokes a 'delegate' call on another method within the domain. The calling method is delayed until the called method returns. This delegation does not change the granularity of concurrency (there is still only one method active at a time within an object), but gives the programmer more control over exactly what the single active process is doing.

Finally, *concurrent* objects are the class of objects providing the finest granularity in a concurrent object-oriented language. Concurrent objects support an unlimited number of active processes. These processes can be created and destroyed explicitly by the programmer, or implicitly by the language whenever a method call is made. The more active processes in an object, the faster a program can accomplish its tasks. However, as noted in the discussion on the orthogonal integration of objects and concurrency, the programmer must be provided with constructs that manipulate processes in order to protect critical regions of code. Such constructs must be used properly to prevent unpredictable execution orders that alter object state in undesirable ways. Smalltalk-80, Trellis/Owl, and Emerald were the three orthogonal concurrent object-oriented languages mentioned. These languages are also in the concurrent object class, as programmers can use the process manipulation tools provided to launch as many processes within an object as they desire. Java is another example of a language supporting concurrent objects, which can be controlled using both process manipulation tools and synchronized methods and statements.

Just as we selected the simple, homogeneous approach to integrating concurrency with objects, we look for the simplest concurrent object classification for Concurrent $\mathcal{LOOM}$. This is clearly the sequential classification. At a first glance, the quasi-concurrent class of languages may not seem too complicated, but consider a chain of delays and activations of processes within an object. It becomes very difficult to reason about the state of an object as delayed processes get nested deeper and deeper. Reusability is also questionable, since a programmer who inherits from a class that implements such objects must make sure any methods he changes or adds will preserve the state of the object under process delays and restarts. The proper manipulation of multiple processes is complicated and prone to error. For Concurrent $\mathcal{LOOM}$ we prefer to minimize the number of potential programmer errors. However, as we will see in Chapter 5, a strict sequential approach does not allow for objects to call methods on themselves, a common operation in many programs, most notably recursive programs. We will thus relax this approach a bit to allow for such 'self-inflicted' [Car95] method calls.

By design, objects encapsulate and protect state, so it is a natural extension to raise the granularity of concurrency to the object level, making objects critical sections. The state of an object is automatically protected without any additional effort by a programmer when only one process is active in an object. Also, programmers are spared the detail of process creation, since object creation can be used to identify a single process with an object. Adding a body or pre-conditions to implement condition synchronization does not require thread manipulation, but only simple tests of instance variables or method parameters to further limit the acceptance of method calls. Such a design minimizes the number of nondeterministic execution orders in a program, making it easier to reason about. Though this choice may seem extremely limiting at first, we will see that the existence of multiple objects, each potentially allocated to its own processor, can form the basis for a very powerful, yet easy to understand, computation model. The key will lie in the number of concurrently active objects in a program and the speed and frequency with which they can communicate.

Chapter 5

Communication and Synchronization

So far, we have determined that Concurrent $\mathcal{LOOM}$ objects will be uniformly active entities, each supporting a single thread of control. Externally, Concurrent $\mathcal{LOOM}$ objects will look very similar to sequential $\mathcal{LOOM}$ objects, since their essential functionality as repositories of operations for hidden data is the same. We do not need explicit language constructs for creating or destroying processes, since object creation can include the former and garbage collection the latter. The next major decision to be made then is how a Concurrent $\mathcal{LOOM}$ program will communicate with the objects it creates, and how these objects will communicate and synchronize with each other.

5.1 Shared Variables vs. Message Passing

By identifying objects with processes in the homogeneous object model and restricting each object to support only one process, the choice of a communication mechanism for objects is greatly simplified. $\mathcal{LOOM}$ objects already communicate by sending messages in the form of method invocations to one another, so using message passing facilities in Concurrent $\mathcal{LOOM}$ is a logical extension. Since a one-to-one correspondence exists between objects and processes, sending a message to an object is equivalent to sending the message to its associated process. Using message passing in this way allows mutual exclusion to be included automatically in the language semantics, simply by allowing only one message to execute inside of an object at a time. When an object receives a message, the associated method can take control of the object process and hold it until it has finished executing. If some process sends a message to an object whose associated process is already handling another message, then the object will not begin to execute the new message until the current message is completed. In this way, the object state is automatically protected from simultaneous updates. Using shared variables for process communication and messages for object communication would be redundant given the identification of objects with processes, as well as limiting for the future possibility of distributed programming in $\mathcal{LOOM}$.

5.2 Synchronous vs. Asynchronous Message Passing

While the choice between shared variables and message passing was fairly simple, the same is not true for selecting a protocol or protocols for message passing. In Chapter 3, we examined some of the tradeoffs between synchronous and asynchronous communication, noting that either could be explicitly implemented by programmers in terms of the other, so that a language designer could choose to implement one or both protocols depending on his priorities. For Concurrent $\mathcal{LOOM}$, our first priority is to make the language simple to understand for programmers unaccustomed to concurrent programming. However, we have already limited the potential concurrency of Concurrent $\mathcal{LOOM}$ programs a great deal by allowing only one active process in an object at a time. The potential concurrency in Concurrent $\mathcal{LOOM}$ programs, and consequently the speed with which they execute, will depend heavily on the communication protocol we select. We must choose wisely.

Although we have distinguished between synchronous and asynchronous message passing, we also noted that there were further subtleties within these categories, depending on how long, if at all, senders of messages wait for receivers to acknowledge receipt and vice versa. We will consider three of the most useful message passing protocols in concurrent programming: *past*, *now*, and *future* [YSTH87]. Past-type message passing is what most people would consider "true" asynchronous message passing. The sender sends a message to the receiver and does not wait for the message to be received. It is called past-type because the action of sending the message terminates before the receiver actually processes the message. It allows a high degree of concurrency because senders and receivers can proceed quickly in parallel if multiple processors are available for computation.

Now-type message passing is a common form of synchronous message passing. The sender sends a message to the receiver and waits not only for the receiver to receive it, but also for the associated method to send some information back. This is somewhat like a procedure or function call, but in this protocol, the method may return information to the sender before completing all of its instructions. With this *early return* option, extended rendezvous between a sender and receiver can be avoided, so that the sender and receiver can operate concurrently for part of the time.

Finally, future-type message passing is a kind of combination of past- and now-type message passing. In this protocol, a sender sends a functional message to a receiver, expecting a result to be returned. However, the sender does not require the result immediately, so it does not wait for the result to be returned. (Hence the name 'future', for future need of the value.) Instead, the sender continues computation and later checks for the result

```
method M
  begin
    statements in rendezvous;
    RETURN value;
    statements after rendezvous;
  end;
```

Figure 5.1: Early return in a method

when it is actually required for some computation. If the receiver has returned a result at this point, the sender can use it. Otherwise, the sender must now block until the result is available. Future-type message passing is also known as *wait by necessity*, since the sender is only forced to synchronize with the receiver when it needs the value of the result.

Since our goal in Concurrent $\mathcal{LOOM}$ is simplicity, providing three different communication protocols seems like overkill. At the same time, ease of use dictates that we want to avoid making programmers simulate the protocols we don't support on their own. As a first approach, the safest thing to consider would be now-type messages, since they most limit the amount of concurrency out of the three protocols. Languages such as Ada [GM88], POOL [Ame87, Ame89, AvdL90], and POBL [Jon96] all provide now-type message passing. Like Concurrent $\mathcal{LOOM}$, each of these languages allows only one process per object. Concurrency is achieved by using the early return protocol in methods, as illustrated in Figure 5.1. A caller of method M is suspended until M has executed the statements preceding the `RETURN` statement, as in a sequential procedure call. However, after M returns (a possibly empty) value, the caller resumes execution concurrently with the statements in M after the `RETURN`. Thus, the `RETURN` does not necessarily signal the end of the method, only the end of the rendezvous between the caller and the method. To achieve the highest degree of concurrency, a programmer can move the return statement as close to the beginning of a method as possible, simulating asynchronous communication if the return statement is the first statement of the method.

However, there are a number of problems with this approach. First, there are non-trivial rules about exactly when and where one can move a return statement in a method, which Jones attempts to formalize for POBL. Early returns thus require programmers to do a lot of fine tuning and a lot of careful checking to make sure the concurrency they gain by moving a return statement will produce correct results. Second, as Liskov et al.. note in [GM88], combining synchronous communication with a static process structure often introduces a great deal of delay. Callers must wait not only for their message to gain control of the receiving object's process, but also for the associated method to return. If an early return is not possible in a method, this can be a non-trivial amount of time. Programmers thus find themselves writing elaborate protocols on their own to simulate some form of wait by necessity in order to avoid this problem. As Liskov et al. conclude, this indicates that the synchronous protocol alone is not adequate.

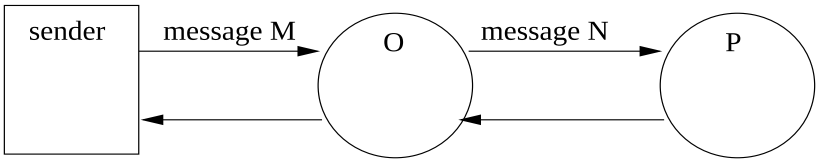
Another option would be to provide asynchronous past-type message passing instead of or in addition to synchronous message passing. For instance, in the Actor family of languages [Agh87, AH87, Agh90], asynchronous communication is the only protocol used. Actors are akin to objects, and each actor is implemented with a *mailbox*, a queue used to hold incoming messages. However, this option seems immediately undesirable for Concurrent $\mathcal{LOOM}$, since programmers would always have to simulate synchronous communication, an operation we noted is frequently necessary in concurrent programs. In addition, it is not clear how one would send an asynchronous message to a functional method that returns a result, since this result needs to be available to the sender. Providing both past- and now-type communication seems reasonable then, since programmers could use asynchronous past-type messages for procedural methods that did not need to wait for the receiver, and synchronous now-type messages for all other types of messages. However, this solution neither prevents programmers from (ab)using early returns nor solves the delay problem for functional methods.

5.3 Wait by Necessity

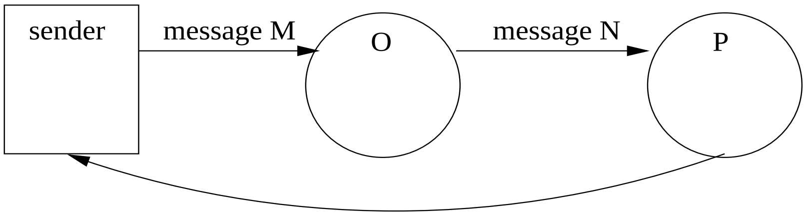
This latter problem brings us to the possibility of using the third message protocol, future-type, or wait by necessity. A number of languages, including ABCL/1 [YSTH87], POBL, and two implementations of concurrent Eiffel, one by Denis Caromel [Car93] and the other by Bertrand Meyer [Mey97], use some form of wait by necessity, either alone or together with some other protocol. The basic idea behind all wait by necessity implementations is the use of proxies. While a sender can send a procedural message asynchronously without any problem, it seems that sending a functional message requires that the sender wait for the result. However, we can actually relax this requirement by noting that it is only necessary that some object (perhaps itself) that the sender knows about wait for the result. Rather than waiting for the result itself, though, the sender can assign or create some proxy object to wait for the result and then continue computation immediately after sending the message. If and when the actual value of the result is needed for some computation later, the computation can query the proxy object for the value, usually by sending it a message.

In POBL, which we noted uses a synchronous protocol with early returns, a restricted version of future-type message passing is allowed in the form of delegation, illustrated in Figure 5.2. If the return value of method M in object O is the result of some functional message N sent to some other object P, the message and return statement can be replaced by a `delegate` statement. The effect is to pass the responsibility to object P for returning the result, freeing up object O from waiting for the result, which object O does not need for any computations. By delegating the responsibility to P, object O finishes processing

```
object O {                                     object O {
  method M                                       method M
    begin                                          begin
      ...                  ====>                     ...
      return P.N();                                  delegate P.N();
    end;                                           end;
end; }                                         end; }
```



Return Statements



Delegation

Figure 5.2: Delegation

method M faster and can begin to accept other messages sooner. Delegation is restricted, however, because it is only the receiver O that gains anything from the operation. The sender of the message must still wait for the result, regardless of whether O or P is the one sending the result.

To truly implement wait by necessity message passing, a more radical departure from now- or past-type message passing is necessary. A good first illustration of wait by necessity is ABCL/1, a language based on the Actor model of computation but with a number of important differences. ABCL/1 provides now, past, and future-type message passing, but the designers note that both now and future-type can be implemented in terms of past-type. In particular, consider using wait by necessity in Figure 5.3, where object O wants to send a functional message `P.N()`, which returns some value, say an integer or an object, to be assigned to `y`. To do so, it can create a *proto-object* (our syntax for proxy object), `yproto`,

```
object O {
  method M
    begin
      y := P.N();   // Don't wait for y
      ...
      a := y+1;     // Wait for y if necessary (assuming y is an integer)
      ...
      b := y.S()    // Wait for y if necessary (assuming y is an object)
    end;
end; }
```

Figure 5.3: Wait by Necessity

whose job will be to wait for the result of `P.N()`. The message can be sent asynchronously to P with `yproto` as the *reply destination* for the result. Then, `y` can be assigned to `yproto` and computation can proceed immediately.

Now, the proto-object will actually have two responsibilities. First, it must wait for the value returned from `P.N()`, since `P.N()` was sent with `yproto` as the reply destination. Second, it must handle requests from later computations that need to use the value of `y`. For instance, if `y` is an integer, we need the value of `y` to evaluate `y+1`. Similarly, if `y` is an object, we need its value before we can send the message `y.S()`. If `P.N()` has returned a value to `yproto` when M reaches an instruction requiring the value of `y`, `yproto` can provide the result and computation proceeds normally. If not, `yproto` can store the request in a queue and reply to it later when `P.N()` has actually returned its value. The computations that request the value of `y` from `yproto`, such as `y+1` or `y.S()`, must wait until their requests are answered.

The advantage of such an approach over delegation and now-type synchronous message passing is that senders of functional messages do not have to immediately wait for a reply. Senders can continue computation concurrently with the object receiving the message as long as the result is not needed in some computation. Wait by necessity is most useful when the result of a message is not needed for quite awhile after the message is sent, and the computation of the message result will be slow. Of course, if the result is immediately required or the computation of the message result is simple, the overhead incurred by communication using the proxy proto-object may be slower than synchronous message passing. In our example, if there are a number of instructions between the `y := P.N();` statement and the statements referencing the value of `y`, and `N` is a non-trivial method, we have clearly gained over synchronous communication.

Another benefit of true wait by necessity is that delegation comes for free. For example, consider the simple symbol table program in Appendix A.1 and assume we are using wait by necessity message passing. This program, adapted from a similar program in [Jon96], allows one to `insert` and `search` for string `data` sorted by integer `keys` in a table organized in a tree structure. In the `search` method, if the key `k` being searched for is not equal to the first key inserted into the table's tree, either the `left` or `right` table element is searched recursively. If `k` is less than the current key, the table elements to the left are searched using the statement `answer := left.search(key)`; otherwise, the table elements to the right are searched using `answer := right.search(key)`. Notice that this method never actually needs to reference the value of `answer` in a computation; it simply returns it. Thus, the current table element being searched returns `answer` immediately without ever having to wait for it's value to be calculated. The `left` or `right` table element is responsible for the calculation, and the first table element is free to accept other messages. Thus, we have automatically achieved the delegation-style message passing used in POBL, and the table can process messages in a pipeline fashion. In this case, both the sender of a search message and the table receiving it gain. The sender can send the message using wait by necessity, and the receiver can return the answer this way as well.

Though ABCL/1 was useful in our research for understanding how to define wait by necessity, the language itself is different enough from $\mathcal{LOOM}$ that we also looked elsewhere for concurrent objects that communicated with wait by necessity. Both Denis Caromel and Bertrand Meyer chose to extend Eiffel, a language more similar to $\mathcal{LOOM}$, to support concurrency using wait by necessity. Caromel's was the first of these two and the simpler in many respects. In his language, wait by necessity is the only communication protocol. Procedural messages are thus asynchronous by default, and functional messages operate in much the same way described for ABCL/1. Waiting occurs only when the value of a message not yet returned is needed in a computation. (Caromel calls this result an awaited object; we will continue to use the term proxy proto-object.) Specifically, if the proto-object is an object, a computation involves sending it a message. If the object is a base type such as an integer or boolean, a computation is any base operation defined on it, such as addition or comparison. Assigning to a variable that refers to a proto-object does not require waiting, nor does returning such a variable or passing one as a parameter.

Recall that in Meyer's Eiffel, programmers distinguish between objects handled by the same or different processes using the 'separate' keyword. This changes the semantics of message sending somewhat. First, consider message sending involving two separate objects. In this case, the semantics are the same as Caromel's Eiffel: procedural messages are sent asynchronously and functional messages are sent using wait by necessity. However, if a message sender and receiver are handled by the same process (i.e., are not separate), message sending must be synchronous in both cases. These semantics are automatically provided by the language, so a programmer must be clear about what type of message passing is being used for each message sent, since the syntax of all messages is identical. Unfortunately, this approach is further complicated by a number of 'separateness consistency rules' governing assignment of separate objects to non-separate objects and separate messages involving reference type arguments. As we have already noted in Chapter 4, the distinction between separate and non-separate objects breaks the abstraction of concurrent processes. The complicated semantics of message passing seems to be another argument against this approach.

Overall, Caromel's approach to communication seems simpler. His language also provides a mechanism for explicit synchronization, so that programmers do not have to design certain synchronous protocols. A 'wait' keyword attached to any object causes computation to delay until the object is no longer awaited (i.e., if it is a proto-object, until its value has been computed.) This feature appears useful for delaying the result of a functional method, but cannot be used to delay a procedural message, since such messages return no result. However, Caromel's language as a whole is not satisfactory because, like Java, it uses the heterogeneous object model, whereby objects are only concurrent if they inherit from a special `PROCESS` class. Meyer's language also addresses an issue that Caromel's does not: reserving objects for exclusive use by a method. This is an extension we will discuss in the future work section of Chapter 8.

5.4 Procedural Messages in Concurrent $\mathcal{LOOM}$

Having considered the advantages and disadvantages of common approaches for message passing protocols, the easier of the two decisions to be made regarding protocol selection in Concurrent $\mathcal{LOOM}$ involves procedural messages. We have noted that a procedural message does not require that any result be returned to the sender, so that sending a procedural message asynchronously allows the sender and receiver to proceed concurrently. Thus, the choice of asynchronous past-type message passing as a default for communication with procedural messages will allow maximum concurrency. This is an important consideration given that we have limited the grain of concurrency in Concurrent $\mathcal{LOOM}$ by identifying objects with processes.

To illustrate the power of this protocol, consider the producer-consumer program in Appendix A.2. Producer-consumer problems are canonical concurrent programming exercises used to test the ability of a language to allow multiple processes to access a shared storage space, in this case a buffer, safely, fairly, and without excessive delay. In Concurrent $\mathcal{LOOM}$, we assure safety (i.e., mutual exclusion) by allowing only one process to be active in an object. So, a producer object can `put` an item in a buffer object and a consumer object can `get` an item from the buffer, but the buffer can only process one such message at a time. With asynchronous messages though, a receiver such as a buffer can be swamped with more than one message at a time. As in the ABCL/1 protocol then, objects will require some sort of mailbox message queue in which to hold messages while they are busy. This approach ensures the second property, fairness, since messages sent to an object can then be processed in FIFO order. Centralizing all incoming messages in this way is an important property, since in languages such as Ada, each individual method within an object queues messages sent to it. If the next message to be executed by an object is chosen non-deterministically from among all methods with queued messages, some methods may be ignored for an excessively long time.

Finally, notice that the last two lines of the program are procedural messages commanding the producer to `produce` items and the consumer to `consume` them. By using an asynchronous protocol for these messages, the producer does not need to wait for the consumer's `consume` method to finish before starting, avoiding unnecessary delay. In fact, if the producer had to wait, the program could deadlock, since the consumer would never be able to get anything out of the initially empty buffer. Since the `produce` message is sent immediately after the `consume` message, both the producer and consumer objects can be active concurrently, sending their requests to the buffer object, so neither object has to wait an excessively long time for access to the buffer. In this program, the programmer has no idea in what order the `put` and `get` messages generated by the producer and the consumer will actually end up in the buffer's mailbox. This depends on how the underlying architecture chooses to handle the producer and consumer processes. The programmer can only guarantee that the buffer does not accept `put` messages when it is full or `get` messages when it is empty. We will return to this example in Chapter 6 to discuss how these conditions can be met using the `presynch` section introduced there.

While the advantages of asynchronous procedural messages are clear, we also noted that there are often situations in which a sender and a receiver of a message need to synchronize. To prevent programmers from having to explicitly design a synchronous protocol for these situations, we also choose to provide synchronous procedural messages. This is easily accomplished syntactically by prefacing a procedural message call with a `wait` keyword. Semantically, this has the effect of delaying the sender until the receiver has received, processed, and returned from the message. Notice that this is the same semantics used for procedural messages in regular $\mathcal{LOOM}$. However, since we use a message queue to handle asynchronous messages, synchronous messages will also have to be placed in the queue to preserve the proper ordering of messages. Thus, the act of placing a message in a queue is sufficient for asynchronous message passing, but synchronous message passing requires that the sender wait until the message comes to the front of the receiver's message queue and is processed. We will discuss mailbox semantics in more detail in Chapter 7.

Unlike Meyer's Eiffel, we prefer the different semantics of synchronous communication to be explicitly labeled with different syntax. In addition, we choose to leave the structure of methods in $\mathcal{LOOM}$ as is by not allowing for an early return option in method bodies, such as in POOL or POBL. This means that procedural messages are either strictly asynchronous or strictly synchronous. Preventing early returns sacrifices the added concurrency they provide in favor of avoiding the problems we noted with programming with them. We also question the reusability of early returns in large programs, since different objects might need to synchronize with a method in another object in different ways. With our implementation, the sender of a procedural message gets to choose exactly how to synchronize with the receiver. The message is responsible for determining synchronization, and the receiving method is simply a black box procedure. Notice also that our `wait` keyword is not the same as Caromel's. We choose to wait on messages, rather than objects, since messages are the source of synchronization. As we will see in the next section, this convention will also allow us to delay functional messages in an identical way.

To illustrate the usefulness of the `wait` protocol, we consider another canonical concurrent programming problem, the Dining Philosophers, illustrated in Appendix A.5. In this problem, we have 5 philosophers who divide their time between thinking and eating. The table they eat at is set with 5 plates and 5 forks. However, the philosophers eat spaghetti all the time, so they each require two forks to eat. If all the philosophers grab a fork, no one gets to eat (i.e., deadlock), and if the philosophers take turns eating in a non-deterministic way, some philosopher could be excluded for an unreasonable amount of time (i.e., unfair starvation.) The goal is to allow all 5 philosophers to eat their meals in a fair way. In the solution we present, we have a `ForkClass` object that keeps track of which of the 5 forks is currently in use, and a `PhilClass` for each philosopher with a `lifecycle` method that invokes `think` and `eat` methods in a loop corresponding to the three meals of the day. Note that a philosopher cannot eat until he has successfully obtained the two forks next to his plate. Thus, the call to the `takeforks` method in a philosopher's `lifecycle` method is done synchronously with a `wait` statement, guaranteeing that the call to the `eat` method that follows will not be invoked until the philosopher has both forks.

We also note here the `for` loop used in the main program to launch the philosophers into their lifecycles. The procedural message statement in this loop, `philos.get(i).lifecycle (i,forkset)`, is an asynchronous message by default, so no philosopher has to wait for another philosopher to finish entirely before starting. In the next chapter, we discuss how contention for the forks is handled with the `presynch` section and how fairness is achieved.

5.5 Functional Messages in Concurrent $\mathcal{LOOM}$

Having made our decision for procedural messages, we would like to make a congruent decision for functional messages to maintain a simple, consistent semantics. Fortunately, our priorities are still the same. We want to select a protocol that will maximize potential concurrency, given that we have minimized it everywhere else. At the same time, we want to prevent programmers from having to encode protocols on their own. The obvious choice for maximizing concurrency is a wait by necessity protocol. We choose the semantics similar to Caromel's Eiffel, where only messages sent to objects and base operations on primitive language types force waiting on variables that have not yet been calculated. With this implementation, we get the automatic advantage of delegation, as well as the benefit of not having to wait when performing simple assignments or passing parameters that may be proto-objects. Our implementation will be in the spirit of the ABCL/1 description of coding wait by necessity in terms of asynchronous past-type message passing.

A message sent using wait by necessity will create a proto-object and send the message asynchronously to the receiver with the proto-object designated as the reply destination. This message will be placed in the mailbox message queue of the receiver like any other message. Whatever variable the sender uses to hold the value of the message will be assigned to the proto-object, and the sender will continue computation. For instance, if `P.N()` is a functional message, its associated proto-object might be explicitly assigned to a program variable in an assignment statement, or function as an actual parameter in another method call. Later computations needing this variable will query the associated proto-object for the value. As soon as the receiver of the message sends its response to the proto-object, the proto-object can respond to requests made for this value. More detail on this implementation will be provided in Chapter 7. For now the semantics and a high level description of the underlying implementation will suffice.

As a simple example of the power of wait by necessity, consider again the producer-consumer program of Appendix A.2. In the consumer class, the `consume` method consists of a single loop to `get` integer items out of a buffer. Notice that the `item` to which these integers are assigned is never used in any computations in the loop. Thus, the messages in the loop will be logged successively in the buffer's mailbox as desired, but the consumer never has to wait for the buffer to process these messages. The `consume` method can finish its business correctly and rapidly, and free up its associated object to accept other messages.

Finally, as in the case for procedural messages, we note that there may be situations in which a programmer wants to send a synchronous functional message. Fortunately, we can accomplish this with the same syntax and high level semantics used for procedural messages. Syntactically, we can simply preface a functional message with a `wait` statement. Semantically, the meaning is also analogous to the procedural case. Rather than creating a proto-object to wait for the result of the message as we did for wait by necessity semantics, we send the message to the receiver and delay the sender until the receiver has processed the message. For procedural messages, this meant waiting until the receiver method associated with the message terminated (i.e., returned an empty value). For functional messages, this means waiting until the associated method returns a value. Notice this does not mean that the value returned by the receiver can't be a proto-object. Indeed, the sender doesn't care about what the receiver returns; its only concern is the synchronization: that the receiver has, in fact, received and processed the message. For instance, back in Figure 5.3, we could replace `b := y.S()` with `b := wait y.S()`. Computation would not proceed until `y` had processed the `S` message and `b` had obtained the result. However, the result could be either an actual value or a proto-object. Until `b` was needed in a computation, it wouldn't matter. Again, more specific implementation details for sending `wait` messages will be provided in Chapter 7.

To summarize, the default for both procedural and functional messages is a form of asynchronous message passing. For procedural messages, the default is past-type and for functional messages, the default is future-type (wait by necessity). To support synchronous message passing, both procedural and functional messages can be prefaced with a `wait` statement. No matter what the protocol, all messages are handled uniformly by receiver objects through a FIFO message queue. Thus, the sender of a message controls synchronization and the receiver controls fairness, and neither has to worry about how the other does this. The choice of asynchronous default message passing was made to maximize potential concurrency. Notice this may not necessarily translate into faster code in all cases due to the overhead of managing message queues and using proto-objects. We believe the uniformity and flexibility of this protocol is straightforward for programmers to understand and use. Essentially, programmers need only determine which messages should be synchronous and tag them with a `wait` statement. The underlying implementation takes care of the rest of the communication and synchronization issues.

5.6 Self-Inflicted Messages

The only limitation of this method so far is a problem encountered in any concurrent object-oriented language that only allows one method at a time to be active in an object. An object can only be activated by receiving a message from another object or the main program, both of which are independent, or external, processes. Once an object has accepted such a message, it cannot accept any other messages until it finishes processing the current message. However, after an external message activates a method in an object, this method body may send messages as well, including messages calling itself or other methods in its containing object. Such a message, termed a *self-inflicted message* by Luca Cardelli [Car95], is immediately problematic, since our semantics dictates that an object cannot accept another message until it finishes processing the current message. In this case, the object will never finish processing the current message if it cannot process the self-inflicted message, and deadlock is the result.

Clearly though, such self-inflicted messages are perfectly reasonable in a sequential object-oriented language. In particular, they are vital for recursive programs. To handle them in a concurrent language, we must recognize that their semantics is different than messages coming from external processes. In designing his distributed object-oriented language Obliq, Cardelli notes that self-inflicted messages are part of the computation requested by the current external message, not independent messages to be handled in their own computation. With this recognition, it becomes obvious that self-inflicted messages cannot be treated like normal messages in our implementation. They should not be put in the object's message queue to be processed later, not only because they should be processed before any other external messages already on the queue, but also because they must be handled before the current message continues its computation. What this means is that self-inflicted messages are actually like ordinary procedure or function calls within a method body. By evaluating messages an object sends to itself as functions or procedures, we achieve the desired semantics and avoid this form of deadlock. We will define this semantics more precisely when we discuss message evaluation in Chapter 7.

Unfortunately, this solution does not address another fairly useful class of programs: those involving mutually referential objects. In particular, deadlock is still possible if a method in some object `A` sends a message to another object `B`, which in the course of evaluating this message sends a message back to object `A`. If the method in object `A` cannot finish execution until the message sent to object `B` is evaluated, and similarly the method called in object `B` cannot finish until the message it sent to object `A` is evaluated, deadlock results. For instance, consider the Concurrent $\mathcal{LOOM}$ pseudo-code in Figure 5.4, and suppose we have two objects `A1` and `B1`, of types `Atype` and `Btype`, created using classes `A` and `B`, respectively. Further, suppose the instance variable `b` in object `A1` refers to `B1`, and the instance variable `a` in object `B1` refers to `A1`. If the method `M` is invoked on `A1`, it sends the message `wait b.P()`. When `B1` in turn sends the message `wait a.N()` in method `P`, deadlock results because `a` refers to `A1`. The object `A1` is still occupied waiting for the result of `b.P()`, and will continue to do so until `B1` returns this value, which is impossible until `a.N()` is evaluated. However, using asynchronous and wait by necessity messages prevents this form of deadlock in some situations, since the mutually referential objects may not have to wait for each other. The example in Figure 5.4 does not deadlock if one or both `wait` keywords are removed. In languages that support an early return option in methods, this form of deadlock can be avoided in the same way by placing a return statement before a mutually referential call. Both of these techniques provide reasonable solutions to this form of deadlock, but are only applicable if waiting is unnecessary. We will discuss this problem further in Chapter 8.

Mutually referential objects aside, it is worth noting that by using this semantics for self-inflicted messages, if we were to send all non-self-inflicted messages using the `wait` keyword, we would get exactly the same semantics as regular $\mathcal{LOOM}$, since synchronous messages act like procedure or function calls as well. Concurrency is only achieved when asynchronous procedural messages or wait by necessity functional messages are sent, since these allow sender and receiver processes to execute concurrently. This allows old sequential $\mathcal{LOOM}$ programs to be evaluated by the Concurrent $\mathcal{LOOM}$ interpreter simply by changing regular messages to `wait` messages everywhere. However, this semantic similarity is only true if there is no conditional synchronization, as specified in the `presynch` and `postsynch` sections described in the next chapter.

```
class A
  var
    x: integer;
    b: Btype;
  method M
    begin
      x := wait b.P();
    end;
  method N
    begin
      return 7;
    end;
end;
```

```
class B
var
  y: integer;
  a: Atype;
method P
  begin
    y := wait a.N();
    return y;
  end;
end;
```

Figure 5.4: Deadlock in a Mutually Referential Program

Chapter 6

Conditional Synchronization

With our communication protocol in place, objects can communicate rapidly and synchronize easily when necessary. Senders of messages do not have to worry about when or if their message is processed, since receiving objects hold incoming messages in a FIFO queue. In this way, a sender knows that its message will be processed before any other messages that arrive at the receiver after its message. However, while this protocol is sufficient for many applications, it will not work for all. As we have noted in the producer-consumer example, some messages are not acceptable under certain conditions, for instance a `put` message to a bounded buffer object when the buffer is full. This is the problem of conditional synchronization: a receiver can only accept (i.e., synchronize with) a message from a sender if certain conditions are true. Such conditions will usually depend on the state of receiver, and perhaps message parameters or information about the sender of the message. In order to support conditional synchronization, we must determine what constitutes a condition, how and where to specify conditions, and how to modify FIFO message queues to handle messages that do not meet conditions when they reach the head of a message queue.

6.1 Specifying Conditional Synchronization

Techniques for specifying conditional synchronization may be classified into two general categories [MTY93]: *guards* and *accept sets*. These techniques are used in most concurrent object-oriented languages to write the *synchronization code* necessary to meet the *synchronization constraints* of concurrent objects. The guard category involves associating a (possibly empty) boolean condition with each method in an object. Before accepting an incoming message, the object tests the condition of the method associated with the message to determine if the message can be processed. In languages that accept messages explicitly, rather than automatically through a message queue, programmers can also guard individual message acceptance statements, providing somewhat finer control. Accept sets allow an object to specify the next set of messages that it can accept, depending on conditions that the object tests after a message is accepted. The object is initialized with some default set of acceptable messages, and calculates a new set based on the conditions after each incoming message is processed. Some languages allow each method to specify the next accept set at the end of its body, while other languages specify general conditions in a separate section of the class that are evaluated after all messages. Thus, while guards are tested before a message is accepted and accept sets are calculated after, both use properties of the object state to determine which messages an object can accept. Below, we discuss the use of guards and accept sets in more detail in some representative concurrent object oriented languages. We will illustrate these constructs with a running example of a bounded buffer object with methods `put` and `get`, buffer size denoted by an instance variable `size`, and buffer capacity denoted by a constant `MAX`.

6.1.1 Guards

CSP, ABCL/1 and Ada are typical examples of languages that use explicitly guarded message acceptance. In ABCL/1, objects are written as scripts of message patterns followed by associated actions. Each message pattern can also be followed by an optional boolean constraint testing object state variables and the sender of the message. Messages sent to an object must match one of the message patterns and pass any associated constraints in order to execute the associated action. Similarly, in Ada task bodies (similar to object implementations), there are no methods, but simply message acceptance statements with associated actions. Message acceptances can be guarded by tests of task variables (instance variables). Both languages also support a 'select' statement for message acceptance, which allows an object or task to select incoming messages from among a number of optionally conditioned message patterns. This technique is powerful because a single object can accept the same message under different conditions with different associated actions. For instance, in Figure 6.1, adapted from [BA90], we show a fragment of an Ada task body for the bounded buffer example. The `select` statement in the loop allows the buffer to choose continuously and non-deterministically between incoming `put` and `get` messages, which are handled by the `accept` 'entries'. The `put` entry accepts `put` messages if `(size < MAX)` and the `get` entry accepts `get` messages if `size>0`. If the buffer has pending messages of both types when both conditions are true, the language implementation selects one non-deterministically.

A variation on this technique is used in both POOL and Caromel's Eiffel. These languages are more similar to Concurrent $\mathcal{LOOM}$ in that they are class-based and objects are created as instances of classes. In POOL, each class has an associated 'body,' a routine that explicitly controls the single process associated with each object instance of the class. Thus, each object instance of a class has the same body, which begins executing when the object is created. If no body is specified in a class, a default body that simply processes incoming messages in FIFO order is executed. To specify conditional synchronization, a body

```
loop
  select
    when (size < MAX)
      accept put(item: in Integer) do ...   // put action
  or
    when (size > 0)
      accept get(item: out Integer) do ...  // get action
end loop
```

Figure 6.1: Guards for an Ada Buffer

```
BODY DO
  if (size = 0) then
     ANSWER(put)
  else if (size = MAX) then
     ANSWER(get)
  else
     ANSWER(any)
  fi
 OD
YDOB
```

Figure 6.2: A POOL Buffer Body

can include explicit accept statements for messages guarded by boolean tests of instance variables. A body can also choose from a number of messages using a select statement. In Caromel's Eiffel, each object instance of a class that inherits from the `PROCESS` class also has an associated default routine, called 'live', that controls the object process by accepting messages in FIFO order. As in POOL, this routine can be redefined to alter the FIFO policy by accepting messages conditionally. For instance, in Figure 6.2, adapted from [AvdL90], we define a body for a bounded buffer in POOL. The body executes in an infinite loop for the life of the buffer object, testing the `size` of the buffer before choosing to `ANSWER` an incoming message. In this solution, we get around using a `select` statement by first specifying the conditions under which only one of `put` or `get` is acceptable. If neither of the boundary conditions `(size = 0)` or `(size = MAX)` holds, either type of message is acceptable. We could also use a `select` statement as in the Ada example.

6.1.2 Pre-Conditions

Meyer employs yet another guard technique in his concurrent version of Eiffel by making use of the existing pre-condition construct in the language. Rather than specifying a body to handle the conditional synchronization of message acceptance, Meyer chooses to guard methods, not message acceptance. This decision means that every message sent to a particular method of an object must pass the same test. In sequential Eiffel, method pre-conditions are used to make sure an incoming message meets certain conditions, specified in the associated method by boolean tests of object instance variables and method parameters. If an incoming message fails to meet the pre-condition of the associated method, an exception is raised. To extend pre-conditions for use in a concurrent environment, Meyer had to address the problem of tests involving 'separate' objects that could be shared by a number of other objects, such as a buffer shared by producer and consumer objects. A pre-condition testing a shared object (i.e., a message sent to that object) that returns true is not reliable, since other objects can modify the shared object at any time, invalidating the condition. A test that returns false is also unreliable, since the condition might become true later if another object modifies the shared object.

In Concurrent Eiffel, Meyer changes the semantics for method parameters and pre-conditions involving separate objects to solve these problems. First, any message sent to a method with separate object parameters waits until these objects are available, that is, not executing messages from other objects. Then, the receiver 'reserves' the parameters for exclusive use for the duration of the method's execution. This means that the reserved objects will accept messages only from the receiver while the method is being executed. Thus, pre-condition tests involving separate objects passed as parameters will be guaranteed to hold for this duration. Second, if a pre-condition test involving a test of a separate object parameter fails, rather than raising an exception, the message will simply wait until the test succeeds. Success is only possible if the object parameter is modified, so the receiver unreserves it to allow other objects to do so. Later, after waiting for some implementation-determined amount of time, the receiver can attempt to execute the message again, accepting it if the pre-conditions are met this time. In this way, pre-condition tests of separate objects can control conditional synchronization. However, pre-condition tests of all other entities keep their old semantics of raising exceptions, since no amount of waiting would produce any changes in their values.

For instance, in Figure 6.3, adapted from [Mey97], we define separate producer and consumer classes that `put` and `get` things from our standard buffer. We also assume that the buffer has publicly available boolean instance variables `full` and `empty` describing the status of the buffer. The `produce` and `consume` methods of the producer and consumer classes are parameterized by separate buffer objects. Before a producer can `produce`, it automatically reserves the buffer parameter `b` and waits for the pre-condition test of `b`, `require not b.full`, to be true. Once the producer has reserved `b`, the value of `b.full`

```
separate class producer is
  produce(b: separate Buffer; item: integer) is
    require
      not b.full
    do
      b.put(item)
    end
end

separate class consumer is
  consume(b: separate Buffer) : integer is
    require
      not b.empty
    do
      Result := b.get
    end
end
```

Figure 6.3: Eiffel Producer and Consumer Using a Buffer

will remain unchanged unless the `produce` method modifies or unreserves `b`. If the test fails, the producer will unreserve `b` so that another object, such as a consumer, can modify the buffer. The language implementation determines when the producer tests the pre-condition again. Similarly, if a consumer wants to `consume`, it reserves the buffer parameter `b` and waits for `not b.empty` to be true.

Thus, to implement conditional synchronization in Meyer's Eiffel, programmers use the old pre-condition idea with some new semantics. Programming this way fulfills the design-by-contract principle that motivates programming in Eiffel, applies the same tests to the same messages, and keeps the language uncluttered with bodies or other new constructs. These are all simplifying features that would be nice to duplicate in Concurrent $\mathcal{LOOM}$. However, this technique is somewhat awkward in some cases. Any time conditional synchronization is necessary using a separate object, a new method must be written if this object is not a parameter in the enclosing method. The producer-consumer example is also somewhat cumbersome to code, since the producer and consumer clients of the buffer must control their access to the buffer, rather than depending on the buffer to handle these accesses itself. The buffer has no way to control message acceptance, except with pre-conditions that test its `size`. However, the `size` of the buffer is simply an integer, so pre-conditions that test it raise exceptions, rather than causing waiting or synchronization. In addition, all separate objects passed as parameters are automatically reserved, regardless of whether they are shared by anyone else. This is a convenience to programmers, since they don't have to worry about when to reserve parameters, but it also adds unnecessary overhead when reserving is not necessary. We will return to the idea of reserving objects when we discuss future work in Chapter 8.

6.1.3 Accept Sets

Svend Frølund [Frø92] designed a theoretical language that uses guards in yet another way, which turns out to be equivalent to accept sets. Rather than organizing guards by message acceptance, as in Ada and POOL, or by method, as in Eiffel, he organizes them by condition, so that the same condition testing the state of an object can be applied to more than one method. He creates a separate section in his language for synchronization code, where guards prevent rather than enable methods, with statements of the form `condition prevents method`. As long as a condition is true, the associated method or methods should not be acceptable. For instance, in the buffer example, the constraints for `put` and `get` would be specified as `(size >= MAX) prevents put` and `(size = 0) prevents get`. He also allows guards to test whether or not a particular method is disabled, and provides an `all-except` keyword to use for preventing all methods except one. Notice that this formulation of guards does not necessarily make a one to one correspondence between method names and conditions, so that the guards do not act as pre-conditions. The advantage to such a formulation is that synchronization constraints can be incrementally modified in subclasses just by adding new ones. By contrast, if guards organized by method or message are used and the synchronization constraints of a subclass change, the guards associated with superclass methods or message acceptance conditions often have to be re-written.

To implement Frølund's proposal, one could collect the conjunction of all the disabling guards that apply to a particular method, negate them, and use them as a pre-condition for the method. Another solution is to just evaluate all guards when an object is created, building a list of currently acceptable methods based on the initial state of the object. Then, after any message to the object is accepted, one can re-evaluate all the guards to calculate a new list. In this way, the disabling guards function somewhat like post-conditions in a sequential language such as Eiffel, since they determine message acceptability after method invocation, rather than before. Thus, Frølund argues that disabling, rather than enabling, conditions are appropriate for guards evaluated like post-conditions. In Eiffel, new post-conditions in a subclass must be more restrictive than the post-conditions of the superclass, so Frølund suggests that one should only disable more methods in a subclass.

Laurent Thomas [Tho94] proposed a similar construction for his experimental language DooJi, which supports multiple threads per object. Synchronization constraints in DooJi are expressed using 'behaviors', which are named 'method sets', and 'activation conditions', which specify when a behavior is in effect. Behaviors function as accept sets, since any message sent to an object must be a member of the current behavior in order to be ac-

```
behaviors=
  full={get}
  empty={put}
  partial=full+empty
computed by
  (when (size = MAX) -> full
   when (size = 0)   -> empty
   else              -> partial)
```

Figure 6.4: Behaviors and Activation Conditions in DooJi

cepted. For instance, in Figure 6.4, adapted from [Tho94], we illustrate the synchronization code for the bounded buffer in DooJi. The behavior 'full' names the method set `[get]`, and the activation condition `(size=MAX)` makes 'full' the current behavior for the buffer. As long as 'full' is the behavior, the buffer can only accept `get` messages. Similarly, the buffer can only accept `[put]` messages if it is 'empty', and either message otherwise. Essentially, Thomas gives names to accept sets, whose enabling conditions can be evaluated after a message is accepted to determine the new accept set. Method sets can be specified using `ALL` or `ALL-MINUS` keywords to indicate that all object methods or all except some specified methods are associated with a behavior. Behaviors and activation conditions are inherited separately from each other and from methods to allow for selective updating of each section in subclasses. Behaviors and activation conditions in a subclass can also refer to their respective components in a superclass using `super` and `super.behaviorname` keywords, making incremental changes easier to specify when method sets or conditions change in subclasses. However, unlike Frølund, Thomas makes no provision for restricting the combination of subclass and superclass behaviors and activation conditions in a subclass. To enforce more restrictive accept sets in DooJi subclasses, references to the `super` activation condition section and to `super.behavior-name`'s would have to be controlled.

We will compare all of these methods for specifying conditional synchronization in the next few sections. However, before beginning to make decisions for conditional synchronization in Concurrent $\mathcal{LOOM}$, we must consider how the addition of the new constructs to handle this feature will interact with other language features, particularly inheritance.

6.2 Inheritance Anomalies

Until now, we have happily combined concurrency with object-oriented features without causing any undesirable interactions. With the homogeneous object model and one process per object, objects and processes are identified. This also means that communication between objects and between processes are unified with message passing. However, we have also happily ignored one major feature of object-oriented programming: inheritance. Code reuse and extensibility is an important part of object-oriented design, and any good object-oriented language, sequential or concurrent, must support it well. So far, the Concurrent $\mathcal{LOOM}$ design has included the introduction of only one new visible language keyword, `wait`, optionally attached to messages. All other changes have included only hidden semantic and implementation details. It is not too difficult to see that inheritance in Concurrent $\mathcal{LOOM}$ as it stands should not produce any bizarre interactions with concurrency. However, the fairly major addition of conditional synchronization to the language is likely to add new, visible constructs subject to class inheritance. We must consider the effects of these new constructs on reuse and extensibility. Clearly the most obvious way to allow reuse and extension of conditional synchronization code for class methods is directly through the inheritance hierarchy of classes. However, there are two problems with this approach. First, in some languages, it is useful to have generic synchronization protocols that can be applied to many different classes. In this case, one needs to be able to separate generic conditional synchronization code from specific classes to allow reuse. For example, in DRAGOON [Pap95], a language that supports multiple threads per object, one can write both regular classes for use in defining objects and 'behavior classes' for defining generic synchronization protocols for message acceptance (i.e., FIFO, readers-writers, etc.). A regular class can inherit from another regular class in the usual way supported by sequential object-oriented programming, but it can also inherit from a single behavior class to use a synchronization protocol.

For instance, in Figure 6.5, adapted from [Pap95], we define the interface for a `BUFFER` with `put` and `peek` methods. The `peek` method is like `get`, except that it does not actually remove an item from the buffer. Next, we define a behavioral class, `READERS-WRITERS`, with a protocol that allows only one writer method, designated by `WOP`, to be active at a time, and many concurrent reader methods, designated `ROP`'s, to be active as long as no `WOP` methods are active or waiting. The class `RWBUFFER` implements a bounded buffer with this protocol by inheriting from the `BUFFER` class and using the `READERS-WRITERS` protocol to rule message acceptance. The `put` method is designated a writer and the `peek` method is designated a reader. In Concurrent $\mathcal{LOOM}$, we choose not to address generic synchronization protocols because we believe they would be of minimal use, given the strict FIFO message acceptance protocol already in place. Even a simple readers-writers protocol is not possible in Concurrent $\mathcal{LOOM}$ objects due to the identification between objects and processes. Conditional restrictions in Concurrent $\mathcal{LOOM}$ will likely depend on specific class instance variables or method parameters, so synchronization code will have to remain in regular classes.

Unfortunately, the second problem that arises when trying to inherit synchronization code is not as easily dealt with. By choosing to keep synchronization code within classes, such code is automatically inherited by subclasses. However, the synchronization con-

```
class BUFFER
  introduces
    procedure put(item: in integer);
    procedure peek(item: out integer);
end;

behavioral class READERS-WRITERS is
ruled WOP, ROP;
where
  per(WOP) <=> active(WOP) + active(ROP) = 0;
  per(ROP) <=> (active(WOP)=0) and (requested(WOP)=0);
end;

class RWBUFFER
  inherits BUFFER
  ruled by READERS_WRITERS
  where
    put=>WOP,
    peek=>ROP
end;
```

Figure 6.5: Behavioral Classes in DRAGOON

straints of a subclass might be different than those of its superclass if methods or instance variables are added or modified. This means that even if the body of a superclass method is unaltered by a subclass, its synchronization code might need to be changed. Worse, the synchronization constraints of subclass methods might require the introduction of new instance variables. Not only might superclass synchronization code need to test these variables, but also superclass method bodies might have to be completely re-defined to operate on them, just so that a subclass method can implement the appropriate synchronization code. Thus, if synchronization code is not specified carefully, extensive re-definition is necessary in subclasses. These problems, dubbed *inheritance anomalies* [MWY90], have received much attention by researchers of concurrent object-oriented programming languages. They are pernicious enough so that the first versions of the POOL language chose to exclude inheritance entirely because a satisfactory solution could not be found.

In [MTY93], the authors identify three general types of inheritance anomalies, which they illustrate with bounded buffer examples that we describe briefly here. The first, *state partitioning*, can occur when synchronization constraints are specified using explicit message accept sets, like the behaviors in Thomas' DooJi. The problem is that new methods in a subclass must be accounted for in the accept sets of the superclass. Even if an old method is not modified by the subclass, its associated accept sets might change, since a new method can partition the state associated with an accept set. For instance, consider defining a subclass of the DooJi example in Figure 6.4 with a method `get2` that removes two items from the buffer. As illustrated in Figure 6.6, this addition partitions the `partial` state into `one-element`, in which only `put` and `get` messages should be accepted, and `partial`, in which `put`, `get`, and `get2` messages should be accepted. The behavior for `full` must be re-defined and a new behavior for `one-element` is added, while `empty` and `partial` remain the same. All behaviors are re-calculated in the subclass, so the behavior for `partial`, `(full+empty)`, will reflect the addition of `get2` to the behavior for `full`. The new activation condition for `one-element` is added easily enough by reusing the old conditions via the `super` keyword. Even though the method bodies for `put` and `get` are unchanged by the addition of `get2`, the synchronization code describing the accept sets that go into effect following their execution had to be updated.

In languages where accept sets are calculated explicitly within each method, rather than in a separate section, re-definition is even more extensive for state-partitioning anomalies. Every method body must be updated to account for new states and methods in order to specify the next accept set. Frølund's proposal does better, since adding one new disabling guard, `(size<1 disables get2)`, does the trick without having to modify any part of any old method. State partitioning anomalies are generally not a problem with guarded methods when the guards are specified separately from methods, as in Frølund's proposal, or associated explicitly with methods. In the latter case, a `get2` method body and associated guard `(size>1)` could simply be added to a bounded buffer without affecting any other methods. However, when guards are associated with message acceptances, as in POOL

```
behaviors=
  // empty and partial are inherited and re-calculated
  full={get,get2}
  one-element={put,get}
computed by
  (when (size = 1)    -> one-element
   else               -> super)
```

Figure 6.6: State Partitioning Anomaly in DooJi

bodies or Ada tasks, the entire body or task must be re-written just to add in new code for accepting messages to new methods.

A second type of anomaly, *history sensitiveness*, can occur with guarded methods. The problem here is that the acceptance of messages for a new method in a subclass might depend on previous acceptances of messages by other methods, that is, on the object's history. For instance, consider the pseudo-code in Figure 6.7, where we add a method `gget` to the bounded buffer with the same method body as `get`, but which is not acceptable immediately after a `put` message. The only way for `gget` to know the history of the buffer object's acceptance of other messages is with the addition of a boolean instance variable `after-put`. The `put` method body must be changed to set `after-put` to `true` and the `get` body changed to set it to `false`. This allows `gget`'s synchronization code to test `after-put` to determine if the last message accepted was a `put`. In this case then, it is the old methods' bodies, not their synchronization code, that must be altered.

Notice that if the method accept set proposals of Frølund or Thomas are used instead, the same problem occurs. A new instance variable would have to be added for the disabling guards or activation conditions to test, and the `put` and `get` method bodies would have to set it appropriately. However, if accept sets are specified for each method, as in the pseudo-code in Figure 6.8, only the accept set for `get` must be modified to add `gget`. Thus, while this latter variety of method accept sets does not solve the anomaly in this case, it only requires the re-definition of the superclass synchronization code. Unfortunately, the accept set code is inside the method body, so the entire body for `get` must be rewritten. A better solution would move this code to a separate section, to be inherited and modified separately from the method body. The ability to inherit superclass accept sets with a `super` keyword, as in DooJi, would also be useful to reduce the amount of re-writing necessary. Separation would also help reduce state partitioning anomalies, since only accept sets would have to be modified.

Finally, a third anomaly, *state modification*, can occur in languages that allow multiple inheritance. In a sequential language, the methods from one class that a subclass inherits from should not be affected by the methods of another class it inherits from (assuming

```
class buffer
  void put(integer item) when (size<MAX) = begin ... end;
  integer get() when (size>0) = begin ...  end;
end;

class history-buffer inherit buffer
  void put(integer item) when (size<MAX) =
    begin ... after-put = true; end;
  integer get() when (size>0) =
    begin ... after-put = false; end;
  integer gget() when (!after-put && size>0) =
     begin ... end;
end;
```

Figure 6.7: History Sensitive Anomaly with Guards

```
class buffer
  void put(item:integer) = begin ... end;
              accept-set: (size=0)   => [put]
                          (size=MAX) => [get]
                          otherwise  => [put,get]
  integer get() = begin ... end;
              accept-set: (size=0)   => [put]
                          (size=MAX) => [get]
                          otherwise  => [put,get]
end;

class buffer2 inherit buffer
  // put inherited as before

  integer get() = // code as before
               accept-set: (size=0)   => [put]
                           (size=MAX) => [get,gget]
                           otherwise  => [put,get,gget]
  integer gget() = // code and accept sets as in get
```

Figure 6.8: History Sensitive Anomaly with Accept Sets

```
class buffer
  void put(item: integer) when (size<MAX) = begin ... end;
  integer get() when (size>0) = begin ... end;
end

class locker
  void lock() when (!locked) = begin ... end;
  void unlock() when (locked) = begin ... end;
end

class lockbuffer inherit buffer, locker
  void put(item: integer) when (!locked && size<MAX) = begin ... end;
  integer get() when (!locked && size>0) = begin ... end;
```

Figure 6.9: State Modification Anomaly with Guards

there are no name conflicts). However, consider Figure 6.9, adapted from [MTY93], where we create a class that inherits from the bounded buffer class and from a locker class with methods `lock` and `unlock`. Instances of locker class suspend acceptance of all messages except `unlock` after accepting a `lock` message by setting a boolean variable `locked` to true. When an `unlock` message is accepted, `locked` is set to false. To maintain these semantics in the locked buffer subclass if guards are used, the guards of the `put` and `get` methods must be modified to test whether or not the object is locked before accepting a message. If accept sets are calculated in each method, as was done in Figure 6.8, the synchronization code describing the accept set that goes into effect after an `unlock` message is accepted must be modified to include `put` and `get`. In either case, the old method synchronization code must be changed even though the method bodies were not changed.

6.3 Proposed Solutions

A number of language designers, including those whose proposals were discussed in the last section, have devoted time to providing partial or total solutions to one or more types of inheritance anomalies in their languages. Despite the range of solutions, nearly all agree that some immediate improvement in avoiding inheritance anomalies is made by separating the specification of synchronization code for methods from the method bodies. In effect, this allows separate inheritance of method bodies and method synchronization code. For some anomalies, this means only the synchronization code section of a subclass must be modified. Notice that when guards are used, state partitioning and state modification anomalies are handled nicely by separating method guards. The former is not a problem with guards at all, and the latter requires only that guards, not method bodies, be modified in the subclass. This separation does not solve the latter anomaly, but at least makes it easier to fix. However, using a separated guard section in classes will not avoid history sensitiveness anomalies, since these require changing superclass method body code, not guard code.

Although Meyer's use of pre-conditions seemed like a good, simple way to handle conditional synchronization, it is now clear that this solution is flawed in two ways. First, state-partitioning and state modification anomalies require that pre-conditions of inherited methods be modified, since pre-conditions act as guards. Some improvement would thus be possible by adding a separate section to classes where the pre-conditions of all methods are declared. Second, it does not handle history-sensitiveness anomalies well, since adding a history sensitive method to a subclass requires that the bodies of other methods be modified. The body and live routines in POOL and Caromel's Eiffel afford some improvements on both of these problems. Conditional synchronization is specified in one place, and history-sensitiveness can be handled by adding boolean flags to the body or routine to be set after certain messages have been accepted. However, this approach requires that the entire routine be rewritten in a subclass whenever any change, however minor, is necessary.

In Frølund's proposal, we noted that disabling guards solve state partitioning anomalies, and they also solve certain state modification anomalies. For instance, in the locked buffer example of Figure 6.9, one new guard, `locked prevents all-except unlock`, specifies the synchronization constraint for the subclass. However, we noted that this proposal does not handle history sensitive anomalies, and Matsuoka et al. demonstrate that it does not handle all state modification anomalies. The history sensitive anomalies are particularly problematic, since we have shown that method bodies, rather than synchronization code, for superclass methods need to be redefined. Thomas' proposal suffers from the same problem of not being able to specify history sensitive programs without modifying superclass method bodies. In all three forms of inheritance anomaly, we have noted that explicit accept sets associated with individual methods require redefinition in subclasses of superclass method synchronization code, but not of superclass method bodies. Thus, redefinition is always necessary using accept sets, but is at least somewhat less problematic for history sensitive problems when accept sets are associated with individual methods, rather than specified with the disabling conditions or behaviors of Frølund and Thomas.

Of all the proposals we have seen, only Matsuoka et al., who categorized the three forms of inheritance anomaly, present a method that eliminates or minimizes all three categories. In passing, we note that Meseguer [Mes93] manages to avoid the problem altogether by removing synchronization code completely with a concurrent rewriting logic, but this technique requires a radical departure from $\mathcal{LOOM}$ structure and semantics. The key observation that Matsuoka et al. make is that neither guards nor accept sets alone will minimize all inheritance anomalies. Thus, the authors propose an extension to a version of the ABCL language that includes a form of both, to be used as desired by programmers

```
class buffer
  synchronizers
    (size<MAX) enables put;
    (size>0) enables get;
  methods
    void put(item: integer) = begin ... end;
    integer get() = begin ... end;

class buffer2 inherit buffer
  synchronizers
    // Old synchronizers inherited
    (size>1) enables get2;
  methods
    // Old methods inherited
    integer get2() = begin ... end;
end;
```

Figure 6.10: State Partitioning Solution with Synchronizers

in a separate section of a class. All synchronization code is automatically inherited by a subclass and can be selectively overridden.

The basic solution includes two types of constructs: *synchronizers* and *transitions*. Synchronizers are an enhanced version of Frølund's guards, but they enable rather than disable methods. They take the form `condition enables method`. Conditions can be grouped into named guards and methods can be grouped into named method sets, as in DooJi, improving reuse and readability. As in DooJi, an overridden method set in a subclass can refer to a method set in its superclass using a `super` keyword followed by the method set name. Method sets support the `all-except` qualifier, as well as qualifiers for all the methods in a class and all non-inherited methods. Synchronizers effectively solve state partitioning anomalies and some state modification anomalies, but not history sensitiveness anomalies. For instance, in Figure 6.10, adapted from [MTY93], we illustrate the addition of the `get2` method to the bounded buffer. The important thing to notice is that the separate synchronizer `(size>1) enables get2` takes care of the new synchronization constraints for the subclass without having to modify any old constraints or methods.

Transitions are enhanced accept sets that can be used instead of synchronizers and minimize, but do not solve, some history sensitive anomalies. They can also be used to solve some forms of the other anomalies. A transition is associated with an individual method or method set and is executed immediately after the execution of an associated method to alter the acceptable set of methods in an object. A variety of operations are available for modifying the accept set, including enabling and disabling individual methods, replacing the set entirely, and saving and restoring the previous accept set. Thus, transitions differ from the accept sets proposed by Thomas and Frølund, because transitions are explicitly associated with individual methods. We noted in Figure 6.8 that accept sets of this type required only that the synchronization code of the `get` method be redefined when the `gget` method was added to the bounded buffer. Since transitions are separated from method bodies, this re-definition is localized in the transition and method set sections.

In Figure 6.11, adapted from [MTY93], we illustrate the transitions solution to the `gget` problem. The default transition in the `buffer` applies to all methods except for methods with their own transition. Thus, in the subclass `buffer2`, we add one new transition for `put` to disable `gget` once, and the rest of the synchronization is handled by the inherited default transition. This does not solve the history sensitive anomaly, since we must still update synchronization code associated with a method of the superclass whose body does not change. However, it does provide a better solution than guards or Frølund's or Thomas' accept sets, which all require that method bodies be changed.

Separate inheritance rules for method sets, synchronizers, and transitions allow for fine grain updates and minimize extensive re-definition in subclasses. Synchronizers are most useful for handling state partitioning anomalies, while transitions are needed to handle history sensitive programs. Both constructs can be used to handle handle state modification anomalies. Frølund's solution to the locked buffer example can be coded similarly using synchronizers. Matsuoka et al. also present a more complicated lock example that is not solvable with Frølund's proposal but is solvable with their proposal by using method sets and transitions. The drawbacks to this proposal are its somewhat overwhelming array of features and the need for programmers to choose between synchronizers and transitions when specifying synchronization constraints. However, this proposal will heavily influence our design of conditional synchronization code for Concurrent $\mathcal{LOOM}$.

6.4 Concurrent $\mathcal{LOOM}$ Solution

We have seen a number of different proposals for implementing conditional synchronization in a concurrent object-oriented language, each with advantages and disadvantages. Of all the proposals, we feel Meyer's solution in Eiffel of using method pre-conditions is the most simple and elegant. However, we question the automatic reservation of object parameters, and the distinction made between semantics for pre-conditions that test separate objects and all other entities. Unfortunately, this solution is also a disaster when combined with inheritance. On the other end of the spectrum, Matsuoka et al. introduce flexible synchronization mechanisms that handle inheritance anomalies gracefully and with minimal re-definition in subclasses, but these mechanisms require extensive changes and additions to the base language. We reject the POOL bodies and Caromel Eiffel routines because these

```
class buffer
  method sets:
    mset EMPTY     {put}
    mset FULL      {get}
    mset PARTIAL   EMPTY | FULL
  methods
    void put(integer item) begin ... end;
    integer get() begin ... end;
  transitions:
    transition default {
      become EMPTY when (size == 0);
      become FULL when (size == MAX);
      become PARTIAL otherwise; }
end

class buffer2 inherit buffer
  method-sets:
    // msets for EMPTY and PARTIAL inherited and re-calculated
    mset AFTER-PUT  {gget}
    mset FULL        super FULL | AFTER-PUT;
  methods
    integer gget() = // code and as in get
  transitions:
    transition put
      disable_once AFTER-PUT;
```

Figure 6.11: History Sensitive Anomaly with Transitions

usually must be entirely rewritten in subclasses. We consider the Frølund and Thomas proposals for accept sets to be inferior to the Matsuoka et al. proposal because they do not handle history sensitive programs as well. For Concurrent $\mathcal{LOOM}$, our goal then is to find a protocol for conditional synchronization that retains some of the simplicity of Meyer's solution, but provides more of the flexibility found in the Matsuoka et al. solution for dealing with inheritance anomalies.

Our first decision for Concurrent $\mathcal{LOOM}$ is fairly obvious given the preceding discussion. However we specify conditional synchronization, the code should be separated from method bodies to at least allow separate inheritance of method bodies and synchronization code. Even finer divisions, as in the Matsuoka et al. solution, may be beneficial as well. Next, we also agree with this proposal's decision to provide some form of both guards and accept sets, since neither construct handles all forms of inheritance anomalies alone. However, we would like to present a less elaborate design, at least initially, in keeping with our goal of making Concurrent $\mathcal{LOOM}$ simple to use and understand.

In comparing Meyer's Eiffel solution to that of Matsuoka et al., one thing that caught our attention was the similarity between pre-conditions and synchronizers, and between post-conditions and transitions. Pre-conditions are simply guards organized by method name, while synchronizers are guards organized by condition. Eiffel post-conditions are evaluated after a method to ensure the state of an object is consistent with changes purported to have been made by the method. Transitions are executed after a method to actually make the state of an object's acceptable method set consistent with changes made by the method. This similarity suggests that we can use a pre- and post-condition organization to specify conditional synchronization, but by removing these constructs to their own section in the object. Such an organization is advantageous given our audience of object-oriented programmers likely familiar with pre- and post-conditions already. Further, if we separate the pre- and post-conditions from each other, each can have its own inheritance hierarchy, separate from that of method bodies and each other.

For our pre-condition section, we must decide whether to organize our guards by method name or condition. In the former design, used by Meyer in Eiffel, each method that requires conditional synchronization is associated with a boolean test of object instance variables and method parameters. A message for the method is accepted only if the condition is true. In the latter design, used by Frølund and Matsuoka et al., a boolean condition allows or prevents the acceptance of one or more messages. The former choice seems more obvious for Concurrent $\mathcal{LOOM}$ given that this is how pre-conditions in a sequential language such as Eiffel are organized. However, with the latter choice, it is easy to incrementally modify synchronization code in a subclass by simply adding a new condition to be applied to one or more methods. Unfortunately, organizing guards by condition does not allow the condition to test specific method parameters, a feature we would like to have, given that this is possible with ordinary pre-conditions.

We can solve this dilemma by organizing pre-conditions by method name, so that conditions are specific to a method and can test parameters. If a subclass needs to incrementally modify a superclass method pre-condition, a programmer must rewrite the precondition, but can use a `superpre` keyword to inherit the superclass pre-condition. A new condition can be combined with the `superpre` condition using any boolean operation. This facility minimizes the amount of redefinition necessary when modifying a method's pre-condition. Notice too that the grain of inheritance becomes even finer if we organize pre-conditions by method name. A subclass will automatically inherit the pre-conditions of the superclass that are not modified, but a programmer can also selectively update pre-conditions for certain methods. As Meyer noted, organizing pre-conditions by method name also requires that methods reserve shared objects in some way if their pre-conditions need to test them.

For our post-condition section, we need to allow programmers to update the set of acceptable messages after a message has been executed, in order to handle history sensitive methods. We have seen that organizing accept sets by condition, as in Thomas' and Frølund's proposals, does not allow history sensitive programs to use inheritance without extensive re-definition. Both Meyer's post-conditions and Matsuoka's transitions are organized by method name, an organization nicely consistent with our choice for pre-conditions, and which handles history-sensitive programs better. The simplest operations we can use to alter an object's method accept set are enabling and disabling methods. In fact, disabling a method can be thought of as setting the pre-condition for the method to be strictly false, while enabling a method as restoring the old (possibly empty) pre-condition for a method. Thus, an object can have both pre-conditions and post-conditions, unlike the Matsuoka et al. model where programmers must select between synchronizers and transitions. Pre-conditions and post-conditions have the same responsibility – determining when a message is acceptable – but in different ways. In the transitions designed by Matsuoka et al., one can perform a number of operations beyond enabling and disabling methods, as well as enabling and disabling conditionally. For our implementation, we choose to stick with simple, unconditional enabling and disabling for the present, with the potential for adding these other possibilities in the future. Further study is necessary to assess the usefulness of conditional post-conditions, given that conditional pre-conditions are available. We also do not implement the method sets in their proposal, since they are largely a convenience that can be used to associate the same accept set with a group of methods. We leave this as a future optimization, as well.

More specifically, the pre-condition section for Concurrent $\mathcal{LOOM}$ classes is indicated by the `presynch` keyword. Each visible method in a class, new or inherited, can have a pre-condition statement. The method name and its parameter names and types must be explicitly written to match the signature of the method as written for the method body. This allows the condition to test the parameters of the method. This signature is followed by a boolean condition that tests the properties of instance variables and the method parameters. For example, in Figure 6.12, we illustrate the `presynch` section for a bounded buffer in Concurrent $\mathcal{LOOM}$. The pre-conditions for both `put` and `get` test the `size` of the buffer

```
presynch
  put(item:integer) : (size < MAX);
  get() : (size > 0);
```

Figure 6.12: Presynch Section for Concurrent $\mathcal{LOOM}$ Buffer

to determine whether the associated message is acceptable. A method's pre-condition can refer to the pre-condition of a parent method with a `superpre` call that operates on pre-conditions using the same syntax and semantics as a `super` method call. For instance, the expression `superpre.put(item)` in the pre-condition section for `put` in a subclass of the buffer example would invoke the superclass pre-condition for the `put` method on the parameter `item`. Currently, we have not made provision for reserving shared objects, so tests of such objects are not necessarily reliable, and in fact can cause unnecessary deadlock in some situations. We also leave open the possibility of testing the identity of the sender of a message in pre-conditions, as allowed in ABCL/1, and forcing pre- and post-conditions to become less and more restrictive in subclasses, respectively. We will discuss the addition of these features further in Chapter 8.

Similarly, the post-condition section is denoted by the `postsynch` keyword. Since we do not as yet allow for conditional enabling and disabling of methods, the name of a visible method without its associated parameters is sufficient for specifying a post-condition for a method. The method name is followed by an `enable` list containing the names of other methods to be enabled following a call to the method, and a `disable` list containing the names of methods to be disabled. For example, in Figure 6.13, we illustrate the `postsynch` section for a subclass of the buffer with the history sensitive `gget` method. We `disable` `gget` after a `put` message is accepted and `enable gget` after a `get` message is accepted to implement the desired synchronization constraint. There is also a `superpost` keyword, which can be included before the enable and disable lists to inherit the enable and disable lists of a parent post-condition. A message to a disabled method is never acceptable, while a message to an enabled method must pass any pre-condition tests for the method in order to be accepted. Both `presynch` and `postsynch` sections have their own inheritance hierarchies. Both are automatically inherited by subclasses, and pre- and post-conditions for individual methods are replaced if they are rewritten. We will illustrate how Concurrent $\mathcal{LOOM}$ handles all of the inheritance anomalies discussed with complete examples in Section 6.6. We will discuss type-checking and evaluation semantics for the `presynch` and `postsynch` constructs in more detail in Chapter 7.

```
postsynch
  put : disable gget;
  get : enable gget;
```

Figure 6.13: Postsynch Section for Concurrent $\mathcal{LOOM}$ Buffer

```
script {
  =>put(item) {
    if (size == MAX) then
      select =>get()
    store item }
  =>get() {
    if (size == 0) then
      select =>put(item)
    return current}
}
```

Figure 6.14: Message Acceptance and Delay in ABCL/1

6.5 Message Queue Semantics

With conditional synchronization in place, now we must consider the effect of pre- and post-conditions on an object's message queue and its FIFO ordering policy. In particular, what happens to a method at the front of a message queue that fails its pre-condition test or is disabled? Clearly the only way such a message will ever be acceptable is if the object first processes some other method that alters the object state or method accept set. Thus, we must relax our FIFO policy to allow for this to happen. In some languages, such as ABCL/1, a message that cannot currently be accepted can cause the object to wait for a specific message whose execution will allow the unacceptable message to be accepted. After the message desired arrives and is executed, the waiting message is executed and FIFO acceptance is resumed. This protocol is useful for applications that require fine control over message ordering. For instance, in the pseudo-code in Figure 6.14, adapted from [YSTH87], we illustrate an ABCL-like script for a buffer object that processes incoming `put` and `get` messages. If the buffer is full when a `put` message arrives, the script waits for a `get` message using the `select =>get()` statement. The buffer is prevented from answering any other other messages until a `get` message arrives. Once a `get` message is processed, the `put` message is acceptable because the buffer is no longer full, and the associated action, `store item`, is executed. Similarly, a `get` message delays the buffer until a `put` message arrives if the buffer is empty.

The other possibility for handling unacceptable messages is to simply place such messages on another queue immediately and resume processing incoming messages. Once any other message has been executed, the queue of unacceptable messages can be checked, in order, for messages that may have become enabled or whose pre-conditions may have become true as a result. Once all methods on the unacceptable queue have been checked, FIFO execution of new messages is restored until the next message is executed. Essentially, the difference is whether or not an unacceptable message can specify the messages that will make it acceptable. The second protocol might be faster for some applications, since an unacceptable message does not hold up other messages by controlling the next acceptable message. The first protocol might be faster in others, since the second protocol requires that the unacceptable message queue be checked after all method executions, even if none of the messages in the queue have been made acceptable.

For Concurrent $\mathcal{LOOM}$ we choose this latter protocol for simplicity, since it does not require the addition of any new constructs to the language. Unacceptable messages are stored in another queue and automatically examined in FIFO order after any message is executed to see if any have become acceptable. We will specify the semantics of this protocol more rigorously in the next chapter. We leave the possibility open for former protocol however, by noting that a method's pre-condition could also easily specify the methods whose execution might make its pre-condition true. Again, further study is needed to determine how useful such a protocol would be, given the ability to enable and disable methods in post-conditions. In general, we are trying to avoid overkill, but want to leave options open to handle new situations discovered in applications. We also note again that self-inflicted messages are not handled through the message queue, and thus do not test pre- or post-conditions when they are executed. This decision is consistent with the handling of pre- and post-conditions for self-inflicted messages in Eiffel [Wei95].

One final thing to note about both protocols, though, is that their disruption of the FIFO message queue introduces the potential for deadlock in programs. The first protocol automatically deadlocks a receiving object if an unacceptable message waits for a specific message that never arrives. Both protocols can block an object sending a `wait` or wait by necessity message if the message never passes a pre-condition test or remains disabled in the receiver forever, potentially causing deadlock in a program. The second policy then is more desirable for client-server applications, since clients, rather than servers, might deadlock. Deadlock-free languages are desirable but nearly impossible to design and often provide limited expressibility. We leave it up to programmers to write deadlock-free code.

6.6 Examples

Now that we have established a protocol for conditional synchronization, we will illustrate its usefulness in handling some of the canonical examples already introduced, as well as examples involving inheritance. The original $\mathcal{LOOM}$ language does not permit multiple inheritance, in accordance with a general agreement among language designers that this facility is more trouble than it's worth. Elaborate semantics must be designed to handle name conflicts between methods of inherited classes, and programmers generally require multiple subtyping or matching, not multiple inheritance. We retain this decision in Concurrent $\mathcal{LOOM}$, so state modifying inheritance anomalies will not be a problem.

First, we consider the original producer-consumer with bounded buffer problem introduced in Appendix A.2. Here, the synchronization constraints for a buffer are handled by a `presynch` section. A buffer object can only accept a `put` message if it is not full. This constraint is met if the instance variable `size` is less than the maximum buffer size, here constrained to be 10. The buffer can only accept a `get` message if it is not empty, so the test `(size > 0)` in the `get` pre-condition satisfies this constraint. No synchronization constraints are necessary for producer or consumer objects, since the buffer handles the synchronization of the incoming `put` and `get` messages from these objects. If a buffer receives a `put` message when it is full, the message is held in the buffer's unacceptable message queue until a `get` message is processed, decreasing the size of the buffer. Similarly, if the buffer receives a `get` message when it is empty, the message is held in the unacceptable queue until a `put` message increases the size of the buffer. In passing, we note that this implementation is much simpler than the one proposed by Meyer in [Mey97]. In his implementation, the producer and consumer are responsible for understanding the conditional synchronization constraints of the buffer. Both must obtain exclusive access to the buffer and then send it messages testing its size before sending it a `put` or `get` message. In this case, we prefer our implementation, since the producer and consumer do not have to worry about how the buffer handles incoming messages.

To illustrate the use of simple pre-condition inheritance, consider the producer-consumer program in Appendix A.3. This program defines a buffer class similar to the one in the program in Appendix A.2, with the addition of a `dummy` parameter in the `put` method and two subclasses. In the subclasses, the method bodies are the same, but the synchronization constraints change. The class `BufferClass2` inherits from `BufferClass` and adds a new instance variable, `new_iv`. We redefine the synchronization code for both `put` and `get`, but want to retain the old constraints as well. Thus, we use the `superpre` construct to invoke the pre-conditions of `put` and `get` in the original `BufferClass`. For the new `put` pre-condition, we also test the parameter `dummy`, and for the new `get` pre-condition, we test the new instance variable `new_iv`. The class `BufferClass3` inherits from `BufferClass2` without making any changes or additions to method bodies or instance variables. In addition, no changes are made to the pre-condition for `get`, so the old pre-condition from `BufferClass2` is automatically inherited. Only the pre-condition for `put` changes, this time by including `superpre.put` and testing the value of the `item` parameter.

We have already noted that a guard synchronization protocol where the guards are organized by method name and kept separate from method bodies, such as our `presynch` section, effectively handles state partitioning anomalies, and that state modification anomalies are not relevant without multiple inheritance. Now we consider a history sensitive program, found in Appendix A.4, to illustrate how Concurrent $\mathcal{LOOM}$ handles history sensitive anomalies. Recall the bounded buffer example discussed in Section 6.2, in which we add a `gget` method to the original buffer. The `gget` method removes an item from a buffer like `get`, but cannot be accepted immediately after a `put` message. In order to implement this program, we first add a new type, `BufferType2`, to the type section so that we can use buffer objects with the new `gget` method, and modify the buffer types used by `ProducerType`'s and `ConsumerType`'s to be of `BufferType2`. Next, we create a new class, `BufferClass2`, which inherits from the original `BufferClass`. The new `gget` method has a similar method body and the identical pre-condition to `get`. To support the history sensitive constraint, we add a `postsynch` section. We disable `gget` after a call to `put` and re-enable it after a call to `get`. As in the Matsuoka et al. solution, we must modify the synchronization constraints of inherited methods whose method bodies do not change, but at least we have separated the code out into a synchronization condition.

As a final example, consider again the Dining Philosopher's program introduced in Appendix A.5. We can now understand how to use synchronization constraints to make this program work correctly. The constraint for this program was that a philosopher could not eat until he picked up the two forks next to his plate. In particular, he should not pick up one fork unless he can pick up both, or deadlock may result. It is now clear that we can accomplish this with a pre-condition for the `takeforks` method in a `ForkClass` object. Philosopher number `num` cannot take his left fork (`forks.get(num)` in the fork array) or his right fork (`forks.get(num%5+1)`) unless he can take both. A philosopher who sends a `takeforks` message waits until the message is actually processed. With our message queue semantics, if one or both forks are not currently available, the `ForkClass` object saves the `takeforks` message on an unacceptable message queue and goes back to waiting for other messages. In the meantime, other philosophers will be sending `takeforks` and `giveforks` messages to the `ForkClass` object as well. Some `takeforks` messages will be acceptable, and others will be queued, while all `giveforks` will be acceptable by default. After any acceptable message is executed, the queue of unacceptable messages is examined to see if any have become acceptable. At some point, both of the forks needed by the sender of the first `takeforks` message in the unacceptable queue will be returned by some other philosophers who send `giveforks` messages. Being at the head of the saved message queue, this `takeforks` message will pass its pre-condition test and be executed.

Notice that this program is a good candidate for a number of potential language extensions we have proposed. The pre-condition for the `takeforks` method tests an array object by sending it messages. Recall that if this object is shared, the tests are unreliable unless the tester has reserved it. No other objects beside the `ForkClass` object use the array, so in this case the tests are reliable. However, it would be nice to have the option of reserving such an object just in case the implementation changes. The extension of pre-conditions to include the names of methods whose execution will make them acceptable is another useful option. In the current solution, `takeforks` messages on the unacceptable queue are automatically examined after both `takeforks` and `giveforks` messages are processed, but only a `giveforks` message can make a `takeforks` message acceptable. Of course, checking the queue of saved messages after a `takeforks` message does no harm in this particular program, but it might slow execution. Also, we note that Meyer presents a more elegant solution to this problem by using the reserved functionality of parameters. A philosopher can request to eat with two 'separate' fork object parameters, and the semantics of the implementation automatically require the eat message to wait until both fork parameters are available to be reserved. Both of these features could be added fairly easily to the Concurrent $\mathcal{LOOM}$ interpreter, discussed in the next chapter along with the informal semantics of the language.

Chapter 7

Informal Semantics

Our specification for Concurrent $\mathcal{LOOM}$ is now complete. We have selected object and process models, communication and synchronization protocols, and a conditional synchronization mechanism. Although a formal semantic description is available for the original $\mathcal{LOOM}$, there is none for Concurrent $\mathcal{LOOM}$, yet. In Appendix B, we provide a complete grammatical description for Concurrent $\mathcal{LOOM}$, where productions tagged with an asterisk indicate changes or additions to the original $\mathcal{LOOM}$ grammar. In place of a formal semantic description of Concurrent $\mathcal{LOOM}$, we give a more informal description of the semantics of the language from the point of view of a programmer. In addition, we describe at a high level the changes made to the $\mathcal{LOOM}$ interpreter to implement these semantics in our fully functional prototype Concurrent $\mathcal{LOOM}$ interpreter. This description is helpful for understanding how concurrent execution is actually handled. The $\mathcal{LOOM}$ interpreter was written in SML-NJ, Version 0.93. It essentially makes two passes over a $\mathcal{LOOM}$ program, one for type-checking and one for evaluating. For Concurrent $\mathcal{LOOM}$, we first updated the $\mathcal{LOOM}$ interpreter to run under ML version 109.23, which provides concurrency primitives. Below we discuss the high level semantics of type-checking and evaluating new language constructs and some of the implementation details necessary for the design.

7.1 Type-Checking

Fortunately, the type-checking algorithm for Concurrent $\mathcal{LOOM}$ required little modification due to the small number of new constructs introduced in the language. Our first new construct, the `wait` keyword for messages, is easily handled. A message tagged with a `wait` is subject to the same type-constraints as an untagged message, since the type-checking algorithm is not concerned with communication or synchronization. However, the `presynch` and `postsynch` sections required new type-checking rules. It is not too difficult to see that for both of these sections, the algorithm should check that the method names with pre- or post-conditions are visible methods in the associated class. A hidden method should not have a pre- or post-condition, since it cannot be invoked by an external process. For a post-condition, it is sufficient to check the methods listed in its `enable` and `disable` lists to make sure they are visible methods as well.

For a method pre-condition, recall that we include parameter names and types with the method's name. This means that this signature and the associated boolean expression can be type-checked just like a method definition, where the condition functions as the method body. The symbol table used for type-checking the method's class already contains the class' instance variables when a pre-condition is checked, so adding the parameters to the symbol table provides necessary and sufficient information to check the condition. In this way, the usual checks to make sure variables referenced in the condition exist and are used appropriately occur automatically. The only additional test we must perform is to ensure the type of the entire condition is boolean.

7.2 Concurrent ML Primitives

Before we discuss evaluating a Concurrent $\mathcal{LOOM}$ program, we need to consider the constructs for specifying concurrent execution provided by Concurrent ML (CML). These constructs will dictate how we actually implement the desired Concurrent $\mathcal{LOOM}$ semantics. A regular $\mathcal{LOOM}$ program is evaluated using a single process to interpret the components of the program. However, to evaluate a Concurrent $\mathcal{LOOM}$ program, we need more than one process. The main body of a program can be interpreted with a single process, but every object the program creates must have its own associated process. Most of the time, these object processes will be inactive, simply waiting for messages to arrive. We will also need a separate process to handle accesses to the shared store of the program, since multiple processes may request to access it at the same time. Like the object processes, the store process should handle requests to access or update the store one at a time in a FIFO order. To handle multiple messages arriving at objects or the store at the same time, we will also need some form of buffered, asynchronous communication mechanism. This will allow receivers to keep a FIFO list of incoming messages and prevent senders from having to wait for acknowledgments from receivers. Fortunately, CML has primitives for handling both of these requirements. We can *spawn* processes to execute functions, and use the *mailbox* construct for communication. A mailbox is an unbounded, buffered communication channel to which processes can send asynchronous (non-blocking) messages. Messages are held in FIFO order in a mailbox buffer until a process executes either a blocking or non-blocking receive statement on the mailbox to remove the first message. A blocking receive suspends the process executing the receive until a message arrives in the mailbox buffer, and a non-blocking receive immediately returns the first message in the buffer, or none if the buffer is empty. For our implementation, we will only need to use blocking receives.

7.3 Object Creation

The first semantic change we must make to the $\mathcal{LOOM}$ language is to associate a process with each object in a program to handle any messages sent to the object during the course of evaluation. The obvious time to make this association is when the main process, or the process of an object executing a method, creates a new object. Thus, in addition to all the other elements included in an object when it is created, we include a unique mailbox that will handle all its incoming messages. An object's mailbox is available to any process that has a reference to the object. To actually handle messages sent to the mailbox in the interpreter, the new object's creator spawns a new process, `waitformsg`. The new object's creator process continues execution and `waitformsg` executes concurrently, waiting for messages to be sent to the new object's mailbox and processing them in FIFO order.

In the interpreter, an object's `waitformsg` function executes for as long as the object is active in the program being interpreted. Garbage collection takes care of removing unreferenced objects and any processes associated with them. The first statement in the `waitformsg` function is a blocking receive on the object's mailbox, which effectively puts the associated process into a waiting mode. The function is reactivated when the mailbox receives a message. Once a message has been appropriately handled, either by being evaluated or queued because it failed a pre-condition test, the function calls itself recursively. The next message is received from the mailbox if and when a message arrives and the process repeats. The details of message sending and receiving are where the real work is done.

7.4 Sending Messages

Recall that when sending messages, four protocols are possible: asynchronous and synchronous procedural messages, and wait by necessity and synchronous functional messages. Synchronization issues aside, the semantics of sending Concurrent $\mathcal{LOOM}$ messages is similar to sending $\mathcal{LOOM}$ messages: evaluate any message parameters, find the receiver, and execute the receiving method body in the receiving method environment with actual parameter values bound to formal parameter names. The differences are that the receiver must be accessed through its mailbox, the receiver is responsible for executing the method, and the sender may or may not wait for this execution. In this section, we consider both types of synchronous messages and asynchronous procedural messages. First, consider the simplest cases of sending a non-self-inflicted procedural or functional `wait` message. As we noted at the end of Chapter 5, these semantics should yield the same results as those for a regular $\mathcal{LOOM}$ procedural or functional message. The sender evaluates any parameters, sends the method name and parameter values to the receiver, and waits for the receiver to evaluate the method body in the environment of the method and return the (possibly empty) result. The only difference is that the sender process evaluates the parameters and the receiver process evaluates the method body. No concurrency occurs, since the sender waits for the receiver to process the message.

However, to actually implement this protocol in Concurrent $\mathcal{LOOM}$, we must take into account that the receiver handles messages through a FIFO message queue, its mailbox. Thus, rather than simply finding the appropriate method in the receiver and executing it, the sender must send the message to the receiver's mailbox so that the receiver can execute the message in the appropriate order. Further, our current semantics dictate that the receiver does not know who the sender is or with what protocol the message was sent, but it must be able to send a result back to the sender when the message gets to the head of its mailbox queue. However, the receiver cannot simply send the result of the message back to the sender's mailbox. If the sender is the main program process, it has no mailbox, and if it is an object, it can't distinguish the result message from other messages sent to its mailbox while it is waiting. Instead, the sender must create a new mailbox, exclusively for the receiver to use for returning the result of the message. Thus, the message that the sender sends to the receiving object should consist not only of the desired method name and actual parameters, but also a new mailbox, the reply destination to which the receiver should send the result. After sending such a message, the sender then executes a blocking receive on this mailbox, suspending the current process until the receiver returns the result to this mailbox.

For instance, in Figure 7.1, we illustrate with Concurrent $\mathcal{LOOM}$ pseudo-code the semantics of sending a functional wait message. In class `A`, method `M` sends the message `wait b.N`, which returns an integer assigned to the variable `y`. When an object is created using class `A` and method `M` is called, the process evaluating `M` must send a message to the object `b` to evaluate `b.N`. The process evaluating `M` evaluates the parameters, `p1, p2,...,pn`, creates a new mailbox for the reply destination, `ReplyMailbox`, and sends a message to `b`'s `ObjectMailbox` with the name of the method it wants evaluated, `N`, a list of the parameters, and a reference to the `ReplyMailbox`. Then, it executes a blocking receive on the `ReplyMailbox`, which suspends the process until `b` returns the result of evaluating `N`.

Since methods are treated like functions in our implementation, we actually do this evaluation slightly differently for speed and simplicity, but the result is the same. The sender evaluates any message parameters in its environment, but does not simply send them off to the receiver, who would have to look up the method and bind the parameters into its environment when evaluating the method. This is because the method definition, consisting of parameters, body, and environment, is actually available to the sender of a message. Rather than having the receiver waste time looking this information up, the sender uses it. The sender binds the actual parameters into the environment of the method and sends the method name, method body, and this updated environment to the receiver. Thus, an object's `waitformsg` procedure does not receive method names and parameters, but method names, bodies, and environments with the parameters already bound. The method body is evaluated in this environment by the receiver. The method name is used

```
class A
  method M
    var
      y: integer;
      b: BType;
    begin
      y := wait b.N(p1, p2,..., pn);
    end;
end;
```

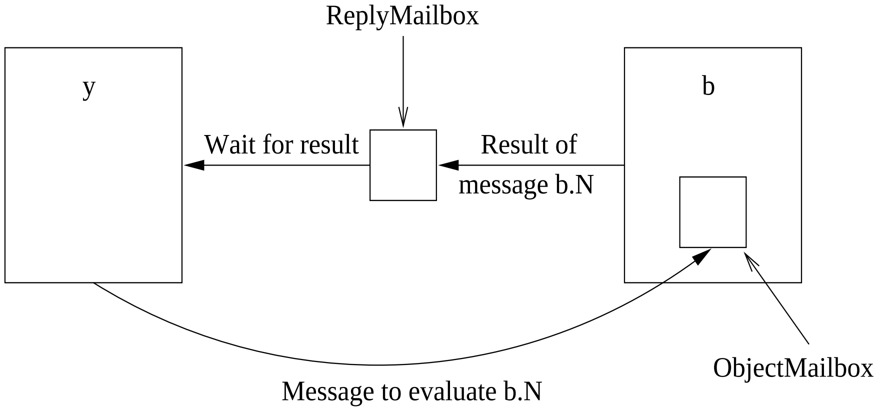


Figure 7.1: Synchronous Functional Message Example

to look up pre- and post-conditions, which we will discuss in Section 7.7. The remainder of the protocol is the same, though. The sender executes a blocking receive on the new mailbox created as the reply destination for the message, blocking the current process until the receiver's `waitformsg` process has executed the method and returned the result to this mailbox. This result, the return value of a functional method or null for a procedural method, is the result of the current expression being interpreted, and the sender's process continues.

Evaluating an asynchronous procedural message is only slightly more complicated. Semantically, we follow the same steps of evaluating and binding parameters, but the sending process does not wait for the message to be processed by the receiver and continues execution immediately. However, our semantics dictate that the receiver is not responsible for synchronization details, so the receiver does not know whether the message was sent synchronously or asynchronously. Thus, the receiver must always return its null result somewhere in case the sender is waiting. In our implementation for synchronous procedural messages, the receiver returned the result to a new mailbox created by the sender. For asynchronous procedural messages, the sender must do the same thing, but can spawn a new process that will wait for the result, rather than waiting itself. This process is simply a one line function that performs the blocking receive on the mailbox, to which the receiver will send the null result of the procedure. The sender does not care about when this result is actually returned and continues evaluation. The newly spawned process dies once the result has been received.

Evaluating a message that an object sends to itself is even easier. Recall that such self-inflicted messages cannot be sent through the object's message queue or deadlock results. We handle messages an object sends to itself like regular procedure or function calls. That is, parameters are evaluated in the current environment and bound into the environment of the associated method, and the method body is evaluated in this updated environment by the current process, which, of course, waits for the result. Self-inflicted messages do not evaluate pre- or post-conditions.

7.5 Sending Wait by Necessity Messages

The real complication in message sending is introduced with wait by necessity functional messages. In this protocol, the sender sends the message to the receiver and does not need the value of the result immediately, but may need it later for another computation. Recall that the semantics of this protocol require using a proxy proto-object to wait for the result of the functional method. The sender creates a proto-object to wait for the result and sends the message asynchronously to the receiver with the proto-object as the reply destination. Recall also in our discussion in Section 5.3 that the proto-object must wait for not only the result, but also requests from computations that need the result. The sender must spawn a new process associated with the proto-object that waits for both of these things. Then, the sender assigns the proto-object to the result of the message and continues execution. If and when a later computation needs the result of the message, it requests the result from the proto-object. The new process associated with the proto-object handles this request. If the result of the message that the proto-object represents has been returned, and it is not another proto-object, the result is provided immediately. If the result is not yet available, or is another proto-object, the computation requesting the result is blocked until the result is actually available.

Recall that computations that need the actual value of a proto-object include messages sent to objects and basic operations like addition and comparison on primitive types. However, assignment statements, actual parameters, and return statements that involve proto-objects are not considered computations and simply use the proto-objects as their values. This means that multiple program variables might refer to the same proto-object, affecting our semantics in two ways. First, after a computation requests the result of a program variable that refers to a proto-object, we cannot just replace the program variable with the result and destroy the proto-object. Other program variables might refer to the proto-object, so we instead rely on garbage collection to remove proto-objects that are no longer referenced. In fact, we do not even replace a program variable that refers to a proto-object with the actual value of the proto-object when the variable requests this value. In the spirit of wait by necessity, we use the value in the computation, and assume that the program variable may not ever be used again. Thus, we don't waste time replacing its reference to the proto-object with the real value. This decision means that a program variable that refers to a proto-object must request its value every time the variable is used in a computation. In applications where the result is only used in a computation once, this decision leads to faster code, while in others it leads to slower code. We could easily change this decision if most applications were demonstrated to be in the latter category. The second way our semantics are affected is that two or more objects that have a reference to the same proto-object might send the proto-object requests for its value at the same time. Thus, proto-objects, like regular objects, must be able to store and respond to multiple request messages in FIFO order.

For instance, Figure 7.2 illustrates these semantics with Concurrent $\mathcal{LOOM}$ pseudocode and a schematic representation of proto-object manipulation. In an object instance of class `A`, method `M` sends a functional message `b.N()` that returns an integer to the variable `y`, and then a procedural message `c.P(y)`. The first message is sent using wait by necessity, so `y` is assigned a proto-object, which we call `yproto`. The second message is sent asynchronously, and the parameter, `y`, refers to `yproto`, and thus does not need to be evaluated. Later in the method, the actual value of `y` is needed in the computation of `y+2`. This computation requests the value of `y` from `yproto` and the computation proceeds when it is available. However, at the same time, the message `c.P(y)` is being evaluated by object `c`. Since `y` was passed as a reference to `yproto`, the formal parameter `z` in method `P` refers to `yproto`

```
class A
  method M
    var
      x, y: integer;
      b: BType;
      c: CType;
    begin
      y := b.N();
      c.P(y);
      ...
      x := y+2;
    end;
end;
```

```
class C
  method P(z: integer)
    var w: integer
    begin
      w := z+3;
    end;
end;
```

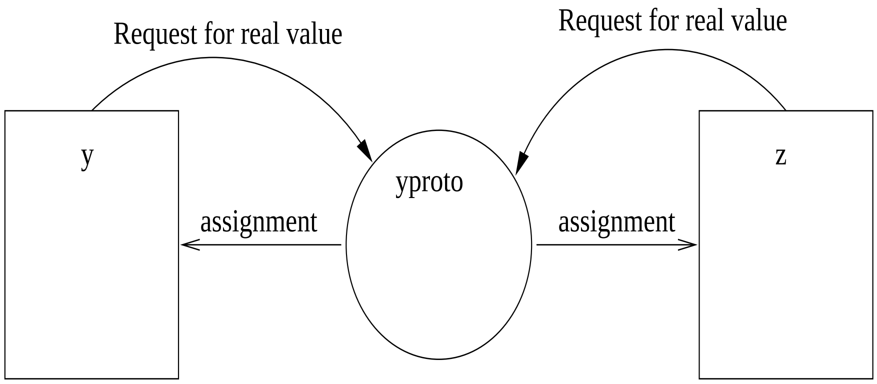


Figure 7.2: Proto-Object Manipulation

as well. Method `P` needs the value of `z` when it evaluates `z+3`, so it requests the real value of `yproto` as well. Thus, `yproto` is referenced by two variables in different objects, both of which will request its value in an undetermined order.

To implement this wait by necessity protocol, we follow the usual steps of evaluating parameters and evaluating the receiver to find the receiving object's mailbox. Next, we create a proto-object so that the sending process can provide it immediately as the result of the message and continue execution. The first responsibility of the proto-object is to wait for the result of the message being sent. Recall that receivers do not know the identity of senders or the protocol used to send incoming messages. They simply return the result of a message to the mailbox specified in the message. Thus, our proto-object needs a `ResultMailbox` to wait for the result of the message being sent. The second responsibility of the proto-object is to provide this result to any computation that requests to use it later. Since these requests can come concurrently from any object with a reference to the proto-object and should be handled in FIFO order, we also provide the proto-object with a `UseMailbox` to wait for requests. Thus, a proto-object is simply an entity with two publicly available mailboxes. To complete the evaluation of a wait by necessity message, the sending process spawns a new process associated with the proto-object to handle messages sent to these two mailboxes. The sending process sends the message to the receiver's mailbox with the proto-object's `ResultMailbox` as the reply destination, and continues evaluation by assigning the proto-object to the result of the message. The new process associated with the proto-object executes concurrently, waiting for incoming messages to its two mailboxes. We will discuss this process in the next section.

If a computation needs the real value of a program variable that refers to a proto-object, it sends a message to the proto-object's `UseMailbox`. It creates a new mailbox as the reply destination for the result and executes a blocking receive on it. If the proto-object has an actual value, it answers the request (in FIFO order) by sending this value to the reply destination mailbox, and the computation proceeds. If not, the computation is delayed until the actual value is returned to the proto-object's `ResultMailbox`. For instance, consider again the code fragment introduced in Figure 7.2. In Figure 7.3, we illustrate the use of proto-object manipulation in sending the wait by necessity message `b.N()`. The proto-object `yproto` is created, the message is sent asynchronously so that computation can continue immediately, and `yproto` is assigned to `y`. The `ResultMailbox` of `yproto` is provided as the reply destination for the message, so that `b` returns the result of `N` to this mailbox. Later, when the value of `y` is needed in the computation `y+2`, a new mailbox, `CompMailbox`, is created and a request is sent to `yproto`'s `UseMailbox`. As soon as `b.N()` has returned its result to `yproto`'s `ResultMailbox`, the request in `yproto`'s `UseMailbox` is answered by sending the result to `CompMailbox`.

Note now that when sending a message to an object using any of the four message sending protocols, the sender must check the object to see if it refers to a proto-object and wait for the real object if necessary. The sender needs to know the identity of the object so it can send a message to its mailbox. If the object is a proto-object, the sender can send a message to its `UseMailbox` to get the real object. In fact, introducing proto-objects into the interpreter means that any variable in a program could refer to one at any time. Thus, when we evaluate a computation involving a message send or an operation on a primitive type, we must check the operands to see if they refer to proto-objects and wait for the real values if necessary. Suitable changes were made to the interpreter to handle this forced evaluation.

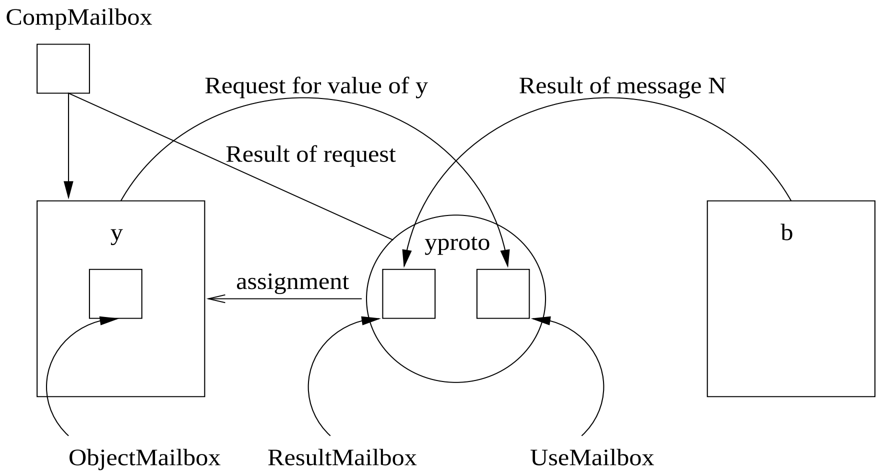


Figure 7.3: Mailboxes and Proto-Object Manipulation

7.6 Proto-Object Semantics

The semantics for messages sent to a proto-object's mailboxes are not difficult, but must be handled carefully. Clearly, a proto-object cannot do anything useful until it receives the result of the message which it represents, so it must wait until this happens. However, when a result arrives, the proto-object cannot simply commence returning it to all requesters of its value. Recall that return statements in functional methods do not force the evaluation of proto-objects. Thus, the value returned to a proto-object might be another proto-object. If this is the case, the current proto-object must send a request to the result proto-object to obtain its real value. The current proto-object waits for this value, just as it waited for the result of the message it represented. This technique is applied recursively to all proto-objects, since the result proto-object might have to wait for a real value as well if the function it represents returns another proto-object. Thus, each proto-object only has to wait once if the result of the message it represents is also a proto-object, but there could be a long chain of proto-objects all waiting. Eventually, the last proto-object in the chain will receive a non-proto result, which will be propagated up the chain.

To implement this protocol, the first statement in the function to handle proto-object messages in the interpreter is a blocking receive on the proto-object's `ResultMailbox`. When the `ResultMailbox` receives the result of the function it represents, it checks it to see if it is a proto-object. If so, the current proto-object sends a request to the `UseMailbox` of the result proto-object. The current proto-object uses its `ResultMailbox` as the reply destination for this message and then calls its message handling function again. This begins the execution of the blocking receive on the `ResultMailbox` again, this time waiting for the result of the other proto-object. This scenario could be repeated in the result proto-object as well, and again in its result until a non-proto-object was finally returned to the `ResultMailbox` of the last proto-object in the chain. Once the `ResultMailbox` of a proto-object obtains a non-proto result, the message handling function can commence the rest of its lifecycle, responding in FIFO order to requests for the result, made with messages sent to the `UseMailbox`. These requests can come from both other proto-objects in the chain waiting for a non-proto result, and from computations in the program that need the result. Both requests simply provide a reply mailbox for the proto-object to send the result to, so the proto-object need not distinguish between the two types of requests.

This protocol is illustrated schematically in Figure 7.4. The Concurrent $\mathcal{LOOM}$ pseudo-code shows three classes, `A`, `B`, and `C` and assumes they have associated types `AType`, `BType`, and `CType`. In Class `A`, method `M` sends a message `N` to an object `b` of type `BType`. The integer result is assigned to the variable `y`. By, default, `b.N()` is sent using wait by necessity, so `y` is assigned a proto-object, `yproto`, which waits for the result of `b.N()`. However, method `N` in class `B` sends a message `S` to an object `c` of type `CType` expecting an integer result, to be assigned to the variable `z`. Since `c.S()` is also sent using wait by necessity, `z` is assigned a proto-object, `zproto`. When `z` is returned to `yproto`, `yproto` recognizes that `z` refers to `zproto`. Thus, `yproto` sends a message to `zproto`'s `UseMailbox` requesting its actual value, specifying that it should be returned to `yproto`'s `ResultMailbox`, which waits for the result. Finally, when object `c` executes method `S`, it returns an actual integer value to `zproto`'s `ResultMailbox`. Then, `zproto` can begin responding to requests for its value held in its `UseMailbox`, one of which is the request from `yproto`. When `yproto` gets the result, it can be certain that it is not a proto-object, and begins returning it to any requesters of the value of `y`.

7.7 Receiving Messages

Having described object creation and message sending, we can now understand the semantics of how objects receive messages. Recall that Concurrent $\mathcal{LOOM}$ objects should process incoming messages in FIFO order, but that pre-conditions and disabling post-conditions can alter this order. Fortunately, CML mailboxes store messages in FIFO order, so sending a message to an object's mailbox guarantees this much automatically. Of course, mailboxes are unbounded, so applications in which a sender of messages might greatly outstrip a receiver could exhaust memory. Programmers can avoid this problem with judicious use of `wait` messages in such applications. Recall that in our high level semantics for message sending, a message consists of the method name, any actual parameters, and a reply destination mailbox. Conditional synchronization aside, evaluating a message is simply a matter of finding the appropriate method, binding the actual parameters into its environment, evaluating the method body, and returning the (possibly empty) result to the mailbox provided

```
class A                      class B                      class C
  method M                     method N                     method S
    var                          var                          begin
      y: integer;                  z: integer;                  ...
      b: BType;                    c: CType;                    return 22;
    begin                        begin                        end;
      ...                          ...                      end;
      y := b.N();                  z := c.S();
      ...                          return z;
    end;                         end;
end;                         end;
```

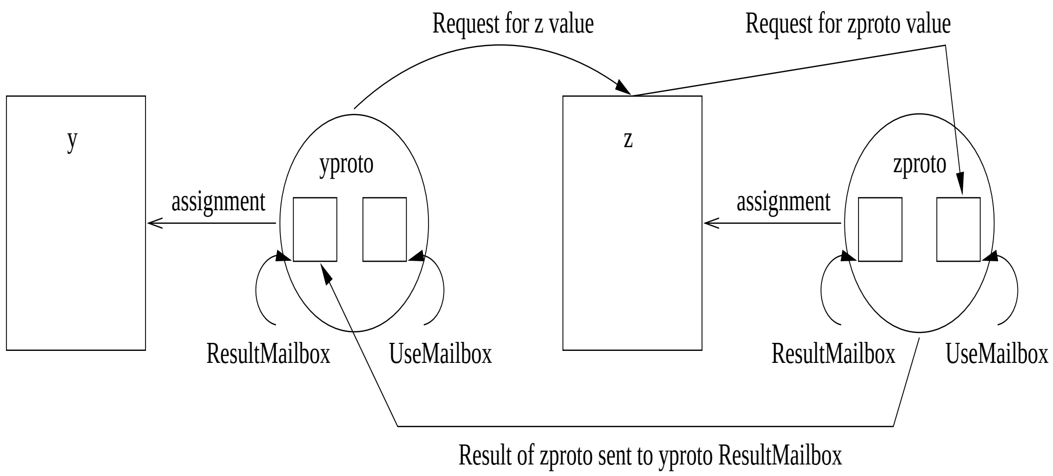


Figure 7.4: Proto-Object Evaluation

in the message.

To handle conditional synchronization, a message at the head of an object's mailbox queue must be examined to make sure it is enabled and passes its (possibly empty) pre-condition test before it is actually evaluated. When an object is created, all of its methods are enabled by default, but methods may become disabled by post-conditions after messages are evaluated. If a method is enabled, recall that we evaluate its pre-condition like a boolean function. We bind the values of actual parameters of the method into the environment of the pre-condition, since it might test these parameters, and evaluate it in this updated environment. These semantics are akin to finding the bindings of function parameters in the current environment and adding them to the environment of the function when evaluating a regular $\mathcal{LOOM}$ function. If the result of evaluating the pre-condition is true, the method is evaluated. Then, any post-conditions of the method are evaluated, first by disabling any methods in the `disable` list, and then enabling any in the `enable` list. These semantics dictate that if a method is both enabled and disabled by a post-condition, it will be enabled after the post-condition is evaluated. This can only happen if a programmer explicitly enables and disables the same method in a class definition. When contradictory post-conditions arise as a result of inheritance, the post-condition of the subclass overrides that of the superclass. We will discuss inheritance in the next section.

If a message is sent to a disabled method or fails its pre-condition, it is removed from the mailbox queue and placed at the end of an unacceptable message queue. The object waits for the next message to arrive in its mailbox and tries to evaluate it in the same way. Once a message has been evaluated successfully, we cannot simply continue processing messages that arrive in the mailbox queue, since there may be messages in the unacceptable message queue whose pre-conditions will now be true as a result of processing the message, or will be re-enabled by the accepted message's post-condition. Before we can check for any new messages, we must check all messages in the unacceptable message queue to see if they are executable. We start at the beginning of the queue with the first message put there. If it still cannot be processed, it remains at the head of the queue and we move on to the next message. If an acceptable message is found, it is removed from the unacceptable message queue and evaluated just like a message from the mailbox. Then, the checking of the unacceptable message queue starts over at the beginning of the queue to preserve FIFO ordering. When the end of the unacceptable message queue is reached, no more messages in the queue are executable, and the object returns to waiting for messages in its mailbox. This process continues for the duration of the object's existence.

To implement this protocol, recall that we use the `waitformsg` function spawned when an object is created to process messages sent to an object's mailbox. This means that a `waitformsg` function needs to know about the mailbox for the associated object, the pre- and post-conditions for the class that created the object, the list of currently disabled methods, and the queue of currently unacceptable messages. The first two items are available at object creation time, and the last two are both empty initially, so a `waitformsg` function is parameterized by these items when it is spawned. Recall also that in our implementation, a message sent to an object actually consists of the method name and body, the method environment with the actual parameters already bound in, and a reply destination mailbox. After a `waitformsg` function accepts such a message, it first checks the list of currently disabled methods to see if the method name is in it. If the method is currently disabled, the message is added to the end of the unacceptable message queue and `waitformsg` calls itself recursively to wait for the next message. If the method is not disabled, any pre-condition for the method is evaluated. If the condition is false, the message is added to the end of the unacceptable message queue and `waitformsg` calls itself recursively. If a message is not disabled and its pre-condition is true, the method body is evaluated in the environment sent in the message. Whatever the result of the message, it is then returned to the reply destination mailbox. After a message has been successfully processed, the object's `postsynch` section is checked for a post-condition for the associated method. If one is found, the list of disabled methods is updated. Then, the unacceptable message queue is checked for acceptable messages. When the end of the unacceptable message queue is reached, `waitformsg` calls itself recursively with the updated list of disabled methods and updated unacceptable message queue.

7.8 Inheritance

The final major addition made to the $\mathcal{LOOM}$ interpreter is code to handle evaluation of class definitions that inherit from superclasses. Methods and instance variables are updated using the same semantics as in $\mathcal{LOOM}$, but we must also handle inheritance of pre- and post-conditions. Recall that our inheritance semantics for both pre- and post-conditions dictate that each have a separate inheritance hierarchy. Further, all pre- and post-conditions are automatically inherited in a subclass, but individual conditions can be updated. When a pre- or post-condition for a method is updated, it replaces the old condition, but the new condition can refer to the old condition using the `superpre` and `superpost` constructs. For both pre- and post-conditions, the evaluation is essentially a matter of starting with the superclass conditions and checking each subclass condition against any superclass condition and making the appropriate adjustments for the subclass.

To update a pre-condition for a method, recall that pre-conditions are treated like boolean functions. Function definitions in $\mathcal{LOOM}$ evaluate to triples consisting of formal parameters, a body, and an environment. When a function is called, the values of the actual parameters are bound into the environment of the function and the body is evaluated in this environment. Thus, evaluating a pre-condition definition in a class essentially entails building a function triple. The parameters are given in the code, and we determine the body by automatically inheriting the (possibly empty) parent pre-condition unless the subclass redefines it. The real task is to build the environment. Clearly a pre-condition should be evaluated in the same environment as its associated method. Since we evaluate method definitions before pre- and post-conditions when evaluating a class definition, we can simply look up the environment of the associated method. The only complication is that a pre-condition can refer to a parent method using a `superpre` call. Thus, we also bind `superpre` to the pre-conditions of the parent class in the pre-condition environment. This is the same semantics used for creating the environments of inherited methods, which might make a call to a `super` method. In this case, `super` is bound to the methods of the superclass. This same process is applied to all pre-conditions defined in a subclass, so that old pre-conditions are automatically inherited unless new ones are defined.

To update a post-condition for a method, we use a similar process of checking a subclass post-condition against any superclass post-condition of the same name, automatically inheriting the superclass post-conditions of methods that are not redefined. Recall that the use of `superpost` in a post-condition means that the `enable` and `disable` lists of the parent class should be inherited, but that the subclass should override any conflicting superclass rule. That is, if a superclass post-condition enables a method but the subclass disables it, the method should be disabled in the subclass. If a post-condition in a subclass uses `superpost`, we must first combine the old and new `enable` and `disable` lists, being careful to resolve conflicts in favor of the subclass. Then, we update the post-conditions for the subclass by replacing the (possibly empty) post-condition for the method in the superclass with the revised `enable` and `disable` lists. For both pre- and post-conditions, the implementation of inheritance in the interpreter is essentially list manipulation.

7.9 Miscellaneous

These are the major changes that were made to the original $\mathcal{LOOM}$ semantics. In the interpreter, various functions had to be changed to reflect the new structure of objects and message sending. For instance, arrays are created as library objects in $\mathcal{LOOM}$, so their code had to be changed to reflect the new structure of objects. The most major changes were made in the main evaluation file, with functions added and altered to force the evaluation of proto-objects, unwind curried parameters, send messages using the various protocols, and evaluate pre- and post-conditions. Various CML libraries were imported to use mailboxes and spawn procedures, but all of the old $\mathcal{LOOM}$ interpreter code compiles in CML. The Concurrent $\mathcal{LOOM}$ interpreter is fully functional, and evaluates all old $\mathcal{LOOM}$ programs correctly when `wait` messages are used where necessary. However, two large programs require `wait` messages in places that are not obvious. We don't know why this is necessary, since we have not analyzed these programs thoroughly. This is a topic for future study, the subject of the next and last chapter, where we also summarize our findings and compare our language to some of the other languages that influenced our design.

Chapter 8

Comparisons, Future Work, and Conclusions

Having finished our design and specification of Concurrent $\mathcal{LOOM}$, we now review some of the features, good and bad, of other concurrent object-oriented languages that influenced our design. These languages were discussed in sections throughout the paper, but we collect them here for discussion as well for comparative purposes. Some of these languages have features we would like to consider including in Concurrent $\mathcal{LOOM}$, but did not have time to research thoroughly or add to the interpreter. We discuss these features in the Future Work section. Finally, we evaluate the Concurrent $\mathcal{LOOM}$ language in terms of its preservation of desirable object-oriented properties and its handling of important concurrency features.

8.1 Comparisons

A language can best be evaluated by comparison with its competitors. In theory, our goal was to design a concurrent object-oriented language that would be easy to use for object-oriented programmers not familiar with concurrency. In practice, we have a working prototype interpreter for our language. Neither $\mathcal{LOOM}$ nor Concurrent $\mathcal{LOOM}$ can hope to compete with large commercial languages given its slow interpreted implementation. However, we believe that for a theoretical concurrent object-oriented language, Concurrent $\mathcal{LOOM}$ has made some important improvements over other commercial and theoretical languages, and that these improvements are transferable to both other languages, and to a faster, compiler-driven version of Concurrent $\mathcal{LOOM}$. In designing our language, we turned first to a broad language comparison paper by Michael Papathomas [Pap95] to see how other languages, our theoretical competitors, handled concurrent object-oriented features. We followed Papathomas' recommendation in selecting the homogeneous object model as the best design for combining concurrency with objects. Given our goal of simplicity, we also selected a sequential design for the organization of objects, eliminating concurrent and quasi-concurrent objects as difficult to reason about and protect when shared. Papathomas' paper also brought the inheritance anomaly to our attention, a problem that influenced our selection of a conditional synchronization protocol later on. From there, we turned to existing concurrent object-oriented languages to study their design decisions, attempting to incorporate good features and avoid bad features we found in the process.

Though not a true object-oriented language, the Ada language [GM88, BA90] was one of the first languages to combine object-oriented with concurrency features. Unfortunately, early versions of the language (pre-Ada 95) were plagued by two problems. First, as Liskov et al. note in [GM88], the synchronous rendezvous communication protocol and the static process structure of one thread per task are too limited for many applications. Second, messages are queued by method within a task, rather than on a single queue in the task. This policy can lead to starvation of messages sent to an individual method if message queues are chosen for acceptance non-deterministically. If nothing else, responding to messages in FIFO order is made more difficult with this policy. In Concurrent $\mathcal{LOOM}$, we chose to offer both synchronous and asynchronous communication, one of the remedies proposed for the first problem, and to store messages sent to an object in a single queue to make FIFO execution easy. Asynchronous communication was chosen as the default because it increases potential concurrency, but of course it is not necessarily better or worse than synchronous communication. Since the application determines the most useful protocol, we wanted to provide programmers with both options.

Another early influential language was the POOL family of languages, developed by Pierre America [Ame87, Ame89, AvdL90]. Like Ada, POOL combines synchronous message passing with a single process associated with each object. Jones makes the same decision for POBL [Jon96]. We agreed with these designers on the decision to allow only one externally invoked method to be active in an object at a time. This decision automatically ensures the mutual exclusion property for protecting object state. All of these languages provide an early return option in methods to simulate asynchronous communication, but we objected to this choice for two reasons. First, as Jones attempted to formalize, there are specific rules about when one can move a return statement from the end of a method to an earlier location. This is a complication we did not want programmers to have to worry about. Second, we also wondered if different senders of messages might need to synchronize with the receiver method in different ways. We preferred to leave synchronization as the concern of the sender, using the default asynchronous protocols or a `wait` statement for synchronous communication, and preserve methods as black box functions within an object. Another problem with POOL was the use of the body routine in an object to control conditional message acceptance. This structure does not handle inheritance anomalies at all, since the entire body must be rewritten in a subclass that requires new conditional synchronization constraints. That is, there is no provision for incremental update of synchronization constraints.

Two languages that had a large impact on our design decisions were the concurrent extensions to Eiffel proposed by Meyer [Mey97] and Caromel [Car93]. We were impressed by the simplicity of Meyer's addition of a single new keyword to his language - the 'separate' keyword - but ultimately felt that the underlying semantics of his language was complicated. Designating objects as separate or not from their creator breaks the concurrency abstraction of associating each process (i.e., object) with its own processor. In addition, there are a number of non-trivial rules associated with separate objects concerning assignment and parameterization. Message sending is also somewhat complicated by different semantics for syntactically equivalent messages depending on whether the receiver is separate from the sender. We chose not to make any process/processor assignments possible in Concurrent $\mathcal{LOOM}$, and to provide a new construct, `wait`, to connote our synchronous messages. Meyer's decision to use pre-conditions for specifying conditional synchronization is appealing, but somewhat awkward due to the necessity of parameterizing objects that need to be tested. In addition, only pre-condition tests of separate objects can cause waiting, which we saw forced producer and consumer objects to control their synchronization with the shared buffer, rather than allowing the buffer to handle this synchronization itself. We also question the semantics of always reserving separate object parameters, since this is only necessary when an object such as a buffer is actually shared.

Both Meyer and Caromel selected wait by necessity as their default communication protocol for functional messages. We liked and adopted Caromel's semantics of not waiting for the evaluation of awaited, or proto, objects used in parameters, assignment statements, or return statements. In both languages, the proto-objects that result from wait by necessity are passed and assigned by reference, just like ordinary objects. Our semantics is a bit simpler because we do not have to handle Eiffel expanded classes, which induce copying of objects in assignment and parameter passing. However, Caromel's Eiffel does not handle inheritance anomalies well because it uses a 'live' routine to specify conditional synchronization. Like POOL bodies, live routines must be entirely redefined by subclasses that change or update synchronization constraints.

The other language that strongly influenced the Concurrent $\mathcal{LOOM}$ design was ABCL/1 [YSTH87] and its extension for handling inheritance anomalies, described in [MTY93]. The ABCL/1 language is based on the Actor family of languages [AH87, Agh87, Agh90], from which it inherits the idea of message queues. This language was helpful in formally distinguishing between past (asynchronous), now (synchronous), and future (wait by necessity) type message passing. The authors of [YSTH87] showed that now- and future-type could be easily implemented using past-type message passing, an implementation we chose to adopt using the mailbox structure in CML. In terms of handling inheritance anomalies, [MTY93] was the seminal paper. It defined three types of anomalies and made clear that neither guards nor accept sets alone are sufficient for handling all types of anomalies. Our solution to handling inheritance anomaly is similar to theirs, but provides fewer features and is organized in the form of pre- and post-conditions. The `presynch` pre-condition section solves state partitioning anomalies and the `postsynch` post-condition section minimizes re-definition for history sensitive anomalies.

8.2 Future Work

In the course of our study of other languages, we also found a number of features that we might like to add to Concurrent $\mathcal{LOOM}$, but have not had time to study in detail or implement yet. The first is the idea introduced by Meyer of reserving shared objects. If one object wants to be guaranteed exclusive access to another, it seems that such a mechanism is necessary. For instance, Meyer notes that a producer object might want to add two items to a buffer in a row. While one could of course add a `put2` method to the buffer, this defeats the purpose. Further, one may not have access to the buffer source code. This problem also showed up in pre-conditions that tested shared objects. The results of such tests are useless unless the tester has exclusive access to the shared object. In addition, if a shared object is updated, other objects with pending unacceptable messages that test the shared object in pre-conditions must be notified. If these objects are never notified, unnecessary deadlock is possible if their unacceptable message queues are never examined again in the usual way. We like Meyer's idea of reserving shared objects as parameters, but question whether it should be the default, particularly since we make no distinction between separate and non-separate objects. One possibility is a `reserved` keyword attached to either formal or actual parameters in method bodies. If a method's pre-condition test involved a reserved object and a message to the method failed the condition, the message would be placed in the object's unacceptable message queue as usual. The shared object could notify the message's object whenever it was updated and the object's unacceptable message queue could be checked for the message to see if it had become acceptable. Alternatively, queued messages with pre-conditions involving shared objects could just be re-tested periodically to see if they had become acceptable.

The portions of the language that are definitely open to improvement and optimization are the `presynch` and `postsynch` sections. As we noted, the authors of [MTY93] include many more features in their synchronizers and transitions than we do in our pre- and post-conditions, including named guards and method sets, and various ways of enabling and disabling methods. We wonder if some of this might be overkill, but leave the decision of which other features to include to further study. In the original ABCL/1, receivers are allowed to test the identity of senders in pre-conditions, an optimization that might be useful for some applications. We were also interested in Frølund's proposal to make disabling guards only more restrictive in subclasses, since this is generally the rule of thumb for post-conditions in sequential object-oriented languages such as Eiffel. It might be appropriate to only allow pre-conditions to become less restrictive and post-conditions more restrictive, as is done in Eiffel. We chose not to enforce such rules in Concurrent $\mathcal{LOOM}$ because we wonder if they should really apply to synchronization constraints.

In terms of message sending, we are fairly happy with our asynchronous protocols with `wait` option, but leave open the possibility for changes in message receiving. In particular, we noted in Section 6.5 that in some applications it could be useful to allow a method's pre-condition to specify the methods whose execution might make the method acceptable if it is currently disabled or fails its pre-conditions. In effect, this could allow the receipt of an unacceptable message to delay the execution of other messages until one arrived whose execution made it acceptable. This is the protocol adopted in ABCL/1, where an object script that accepts a message that cannot be executed causes the object to wait for another message whose execution will make it acceptable. For instance, we could specify that an unacceptable `put` message causes a buffer to wait for a `get` message. Of course, in our implementation, this happens anyway for this example, but other applications with more methods might want to prevent acceptance of all future messages except for some specific message when the current message is unacceptable.

The only unsatisfactory consequence of our message passing protocol is the deadlock that can result in programs with mutually referential objects, as discussed in Section 5.6. We noted that this problem can be avoided with our protocol in some situations by using asynchronous and wait by necessity messages to prevent objects from actually having to wait for the results of mutually referential messages. Languages that support early return statements in methods can also avoid this problem with judicious placement of return statements in mutually referential methods. We do not want to support early return statements in Concurrent $\mathcal{LOOM}$ for a number of reasons, discussed in Section 5.2. In addition, we do not believe that using early returns to solve this deadlock problem is any more powerful than using asynchronous and wait by necessity messages. Both designs allow programmers to prevent a method in one object from waiting for the execution of a method in the other. In programs where this is possible using one design, it should be possible using the other. Instead, it might be possible to add some optimizations to the Concurrent $\mathcal{LOOM}$ interpreter to detect mutually referential situations dynamically at run-time. One possibility might be to keep a list of the objects referred to in nested message invocations and develop semantics for circular references.

We designed our language conservatively with a minimum of features to make it simple to use, but also recognize that some programmers might want a more advanced version of the language to support some new features. For instance, both Meyer's Eiffel and ABCL/1 support a kind of express message that can bypass the message queue and interrupt any currently executing message. This feature is useful for real-time and critical applications that need the ability to handle emergency situations. Another feature would be a statement to launch procedures in parallel. We question the usefulness of such a feature given the ability to send asynchronous messages, but it might be useful for sending a group of messages asynchronously and waiting until all are finished to continue. We might also consider the fairly drastic change of relaxing our sequential object model of only one active method per object. We would consider the idea of supporting synchronized and unsynchronized methods in an object, similar to the functionality in Java [GJS96], but we would not want the granularity of concurrency at a level lower than methods. The final major revisions/goals for Concurrent $\mathcal{LOOM}$ are to extend the language to support distributed execution, write a compiler for it, and formalize the semantics.

In terms of more small scale future work, there are a few implementation details that need to be addressed. First, the execution of a cloning message is currently handled like any other message - that is, asynchronously by default. This means that a clone message is placed on the receiver's message queue and is not executed until other messages that arrived at the receiver before it are executed. This seems like the logical semantics, but should be investigated more thoroughly. We have also implemented `wait` messages using an extra period in the message send (i.e., `wait object..method()`) to avoid some reduce/reduce and shift/reduce conflicts in the grammar. This should not be too difficult to fix, but will require some massaging of the grammar that was not immediately obvious to us. In the last chapter, we noted that there were two large $\mathcal{LOOM}$ programs that required certain seemingly random messages to `wait` in order for them to work properly. Many old programs required `wait` statements on printing methods to make the output come out in a reasonable order, but we do not know why these two programs required additional waiting. These programs need to be studied more closely to isolate the problem. In general, more programs than the selected canonical examples and old $\mathcal{LOOM}$ programs we focused on need to be written in Concurrent $\mathcal{LOOM}$ to really get a good feeling for the expressiveness of the language.

8.3 Conclusions

Although we have not done an extensive formal analysis of Concurrent $\mathcal{LOOM}$, it is possible to evaluate the language more informally in two ways. First, since we have extended and modified an object-oriented language, how well are important properties such as modularity, reusability, and inheritance maintained? Second, how well do we handle desirable concurrent program features, such as the ability to specify mutual exclusion, fairness of process execution, and absence of deadlock?

To answer the first part, we have already noted that we only introduce three new constructs into our language: the `wait` keyword, and optional `presynch` and `postsynch` sections. The `wait` keyword is only a synchronization keyword for messages, and has no effect on object-oriented properties like modularity and inheritance. However, we have seen that the `presynch` and `postsynch` sections have a large impact on inheritance. Our design minimizes or eliminates inheritance anomalies by using separate inheritance hierarchies for the `presynch` and `postsynch` sections, which implement guards and accept sets, respectively. Reusability is also handled well, since subclasses automatically inherit pre- and post-conditions from superclasses. Subclasses can selectively update pre- and post-conditions for individual methods, and refer to the conditions of their parent class using the `superpre` and `superpost` constructs. Modularity is an issue we chose not to address for pre- and post-conditions because we did not think generic pre- or post-conditions, such as synchronization protocols like readers-writers, would be especially useful. In general, we tried to make the `presynch` and `postsynch` section inheritance semantics as similar as possible to method inheritance semantics in order to preserve desirable object-oriented properties.

In terms of concurrency features, Concurrent $\mathcal{LOOM}$ was designed specifically to handle two important program properties automatically: mutual exclusion and fairness. By allowing only one method to be active in an object at a time, the state of an object is automatically protected from simultaneous update, guaranteeing mutual exclusion. By organizing communication around mailbox queues, we also guarantee FIFO fairness of execution of messages within an object. Of course, we have no control over the fairness of execution of all the processes, and therefore the fair execution of messages in all objects. However, we assume that whatever architecture a programmer uses will implement some sort of fairness protocol for the various object processes. Another important property for concurrent programs, particularly client-server applications, is the absence of unnecessary delay. Our communication protocol was designed with this property in mind by making asynchronous procedural messages and wait by necessity functional messages the defaults for communication. Senders of either of these types of messages do not need to wait for the receiver to process the message and return a response. Only if the sender actually needs the result might waiting be necessary.

The last important concurrent program feature we consider is absence of deadlock. Concurrent $\mathcal{LOOM}$ does not guarantee that all programs written in the language will be deadlock-free. This responsibility is left up to the programmer. There are three situations in which a Concurrent $\mathcal{LOOM}$ program could deadlock. First, a sender could deadlock by sending a message using wait by necessity or the `wait` keyword to which the receiver never responded. This could only happen if the message always failed its pre-condition or its associated method remained disabled forever in the receiver, since FIFO execution of messages in the receiver is guaranteed. The second way deadlock is possible is in programs with mutually referential methods. While an object can call methods on itself, using the `self` or `super` keywords, we currently have no way of detecting mutually referential calls. Finally, we mentioned the third form of deadlock as a result of not handling `presynch` tests of shared objects. The second form of deadlock may be fixable with some interpreter optimizations, and the third by adding semantics for reserving objects, but we see no simple solution for the first form.

Despite the lack of modular pre- and post-conditions, which we did not think would be useful, and the potential for deadlock in programs, a flaw of many respected concurrent languages, we believe Concurrent $\mathcal{LOOM}$ is a simple, expressive, relatively safe concurrent object-oriented programming language. Introducing concurrency into a sequential language puts a lot of responsibility on programmers to understand how abstract processes might interact. We have tried to minimize this responsibility without making the language too restrictive. At the same time, we have sought to preserve all of the nice object-oriented properties found in the original $\mathcal{LOOM}$. A number of canonical concurrent programs are safely and elegantly handled by Concurrent $\mathcal{LOOM}$, but there are of course applications for which the Concurrent $\mathcal{LOOM}$ process structure is too restrictive (i.e., readers-writers applications). Nearly all of the old sequential $\mathcal{LOOM}$ programs work as is when evaluated by the Concurrent $\mathcal{LOOM}$ interpreter, and those that don't are fixed with `wait` messages. Our design was conservative for our audience of object-oriented programmers, but we leave open the possibility for new features that would increase expressiveness, in some cases trading our automatic safety guarantees for finer control over processes. Our belief is that the simpler the language, the better, and that new features should be added only when proven necessary.

Appendix A

Concurrent $\mathcal{LOOM}$ Programs

A.1 Symbol Table

```
program symbol_table
-- stores strings keyed by integers in tree structure
type
   tabletype = ObjectType
     insert:proc(integer, string);
     search:func(integer): string;
     clear:proc();
   end;

const
   class tableclass
     var data="": string;
         key=0: integer;
         left=nil: mytype;
         right=nil: mytype;
   methods
     visible
        insert = procedure(k:integer; d:string)
          begin
            if (key=0) then      -- if table empty, set 1st elt
               key := k;
               data := d;
            elsif (key=k) then   -- if keys match, set first elt
               data := d;
```

```
            elsif (k<key) then      -- insert data to left
               if (left=nil) then   -- create new left table element
                 left := clone();
                 left.clear();
               end;
               left.insert(k,d);
            else                    -- insert data to right
               if (right=nil) then  -- create new right table element
                 right := clone();
                 right.clear();
               end;
               right.insert(k,d);
            end;
          end;

        search = function(k:integer): string
          var answer: string;
          begin
             if (k = key) then      -- desired elt found
               answer := data;
            else
               if (k<key) then      -- delegate search to left element
                 answer := left.search(k);
               else                 -- delegate search to right element
                 answer := right.search(k);
               end;
            end;
            return answer;
          end;

        clear = procedure()
          begin
            data := "";
            key := 0;
            left := nil; right := nil;
          end;
   end;

var
   atab: tabletype;
```

```
   str: string;
begin
   atab := new(tableclass);
   atab.insert(3,"HI");
   atab.insert(6,"BYE");
   str := atab.search(6);
   printString(str);
end;
```

A.2 Producer-Consumer

```
program producer_consumer import ARRAY;

type
   BufferType = ObjectType
        put: proc(integer);
        get: func():integer;
        is_empty: func(): bool;
        is_full: func(): bool;
   end;

   ProducerType = ObjectType
        produce: proc(BufferType,integer);
   end;

   ConsumerType = ObjectType
        consume: proc(BufferType,integer);
   end;

const
   class BufferClass
   var
     size=0: integer;
     -- Make an array of 10 integers, each initialized to 0
     intarray = new(ARRAY::arrayclass(integer,0,1,10)) :
                                   ARRAY::arraytype[integer];
   methods
     visible
        put = procedure(item:integer)
          begin
                size := size+1;
                intarray.set(size,item);
                printString("Putting item ");
                printNum(item);
                newline(1);
          end;

        get = function(): integer
          begin
```

```
                printString("Getting item ");
                printNum(intarray.get(size));
                newline(1);
                size := size-1;
                return size;
          end;

        is_empty = function():bool
          begin return (size = 0); end;

        is_full = function():bool
          begin return (size = 10); end;

   presynch
        put(item:integer) : (size < 10);
        get() : (size > 0);
   end;

   class ProducerClass
   var
     i=1: integer;
   methods
     visible
        produce = procedure(buf:BufferType; ct:integer)
          begin
             for i:= 1 to ct do
                 buf.put(i);
             end;
          end;
   end;

  class ConsumerClass
   var
     i=1: integer;
     item=0: integer;
   methods
     visible
        consume = procedure(buf:BufferType;ct:integer)
          begin
             for i := 1 to ct do
```

```
                 item := buf.get();
             end;
          end;
   end;

var
   abuf: BufferType;
   consumer: ConsumerType;
   producer: ProducerType;

begin
   abuf := new(BufferClass);
   consumer := new(ConsumerClass);
   producer := new(ProducerClass);

   consumer.consume(abuf,5);
   producer.produce(abuf,5);
end;
```

A.3 Producer-Consumer with Presynch Inheritance

```
program producer_consumer import ARRAY;

type
   BufferType = ObjectType
        put: proc(integer,integer);
        get: func():integer;
        is_empty: func(): bool;
        is_full: func(): bool;
   end;

   ProducerType = ObjectType
        produce: proc(BufferType,integer);
   end;

   ConsumerType = ObjectType
        consume: proc(BufferType,integer);
   end;

const
   class BufferClass
   var
     size=0: integer;
     intarray = new(ARRAY::arrayclass(integer,0,1,10)) :
                                  ARRAY::arraytype[integer];
   methods
     visible
        put = procedure(item:integer;dummy:integer)
          begin
                size := size+1;
                intarray.set(size,item);
                printString("Putting item ");
                printNum(item);
                newline(1);
          end;

        get = function(): integer
          begin
                printString("Getting item ");
```

```
                printNum(intarray.get(size));
                newline(1);
                size := size-1;
                return size;
          end;

        is_empty = function():bool
          begin return (size = 0); end;

        is_full = function():bool
          begin return (size = 10); end;

presynch
     put(item:integer;dummy:integer) : (size < 10) & (item > 4);
     get() : (size > 0);
end;

class BufferClass2 inherit BufferClass
var
     new_iv = 1:integer;
presynch
     put(item:integer;dummy:integer) :
         superpre.put(item,dummy) & (dummy=0);
     get() : superpre.get() & (new_iv > 0);
end;

class BufferClass3 inherit BufferClass2
presynch
     put(item:integer;dummy:integer) :
         superpre.put(item,dummy) & (item>0);
end;

class ProducerClass
var
  i=1: integer;
methods
  visible
     produce = procedure(buf:BufferType; ct:integer)
       begin
          for i:= 1 to ct do
```

```
                 buf.put(i,0);
             end;
          end;
   end;


   class ConsumerClass
   var
     i=1: integer;
     item=0: integer;
   methods
     visible
        consume = procedure(buf:BufferType;ct:integer)
          begin
            for i := 1 to ct do
                item := buf.get();
            end;
          end;
   end;

var
   abuf: BufferType;
   consumer: ConsumerType;
   producer: ProducerType;

begin
   abuf := new(BufferClass3);
   consumer := new(ConsumerClass);
   producer := new(ProducerClass);

   consumer.consume(abuf,5);
   producer.produce(abuf,5);
end;
```

A.4 History-Sensitive Producer-Consumer

```
program producer_consumer import ARRAY;

type
   BufferType = ObjectType
        put: proc(integer);
        get: func():integer;
        is_empty: func(): bool;
        is_full: func(): bool;
   end;

   BufferType2 = ObjectType include BufferType
        gget: func():integer;
   end;

   ProducerType = ObjectType
        produce: proc(BufferType2,integer);
   end;

   ConsumerType = ObjectType
        consume: proc(BufferType2,integer);
   end;

const
   class BufferClass
   var
     size =0: integer;
     intarray = new(ARRAY::arrayclass(integer,0,1,10)) :
                                   ARRAY::arraytype[integer];
   methods
     visible
        put = procedure(item:integer)
          begin
                size := size+1;
                intarray.set(size,item);
                printString("Putting item ");
                printNum(item);
                newline(1);
          end;
```

```
     get = function(): integer
       begin
             printString("Getting item ");
             printNum(intarray.get(size));
             newline(1);
             size := size-1;
             return size;
       end;

     is_empty = function():bool
       begin return (size = 0); end;

     is_full = function():bool
       begin return (size = 10); end;

presynch
     put(item:integer) : (size < 10);
     get() : (size > 0);
end;

class BufferClass2 inherit BufferClass
methods
   visible
     gget = function(): integer
       begin
             printString("G-Getting item ");
             printNum(intarray.get(size));
             newline(1);
             size := size-1;
             return size;
       end;
presynch
     gget() : (size > 0);
postsynch
     put : disable gget;
     get : enable gget;
end;

class ProducerClass
```

```
   var
     i=1: integer;
   methods
     visible
        produce = procedure(buf:BufferType2; ct:integer)
          begin
             for i:= 1 to ct do
                buf.put(i);
             end;
          end;
   end;

   class ConsumerClass
   var
     i=1: integer;
     item=0: integer;
   methods
     visible
        consume = procedure(buf:BufferType2;ct:integer)
          begin
             for i := 1 to ct do
                if (i%2 = 1) then item := buf.get();
                else item := buf.gget();
                end;
             end;
          end;
   end;

var
   abuf: BufferType2;
   consumer: ConsumerType;
   producer: ProducerType;
begin
   abuf := new(BufferClass2);
   consumer := new(ConsumerClass);
   producer := new(ProducerClass);
   consumer.consume(abuf,5);
   producer.produce(abuf,5);
end;
```

A.5 Dining Philosophers

```
program dining_philosophers import ARRAY;
-- allows 5 philosophers to eat fairly using 2 of 5 forks
type
   ForkType = ObjectType
        takeforks: proc(integer);
        giveforks: proc(integer);
   end;

   PhiloType = ObjectType
        think: proc(integer);
        eat: proc(integer);
        lifecycle: proc(integer,ForkType);
   end;

const
   class ForkClass
   var
     num2=0:integer;
     -- Make an array of 5 booleans, each initialized to true
     forks = new(ARRAY::arrayclass(bool,true,1,5)) :
                                  ARRAY::arraytype[bool];
   methods
     visible
        giveforks = procedure(num:integer)
          begin
                num2 := (num%5)+1;
                forks.set(num,true);
                forks.set(num2,true);
                printNum(num);
                printNum(num2);
                printString(" returned");
                newline(1);
          end;
        takeforks = procedure(num:integer)
          begin
                num2 := (num%5)+1;
                forks.set(num,false);
                forks.set(num2,false);
```

```
            printNum(num);
            printNum(num2);
            printString(" taken");
            newline(1);
      end;

presynch
     takeforks(num:integer): (forks.get(num)) & (forks.get(num%5 + 1));
end;

class PhiloClass
var
  number=0:integer;
methods
  visible
     think = procedure(time:integer)
       var i:integer;
       begin
            printString("Philosopher ");
            printNum(number);
            printString(" thinking");
            newline(1);
            for i:=1 to time do
              printString("");
            end;
       end;

     eat = procedure(time:integer)
        var i:integer;
        begin
            printString("Philosopher ");
            printNum(number);
            printString(" eating");
            newline(1);
            for i:=1 to time do
              printString("");
            end;
        end;

     lifecycle = procedure(num:integer;forks:ForkType)
```

```
          var i:integer;
          begin
                number := num;
                for i:=1 to 3 do
                  think(4);
                  wait forks.takeforks(number);
                  eat(5);
                  forks.giveforks(number);
                end;
          end;
   end;

var
   forkset: ForkType;
   philos: ARRAY::arraytype[PhiloType];
   philo1: PhiloType;
   i:integer;

begin
   forkset := new(ForkClass);
   philo1 := new(PhiloClass);
   philos := new(ARRAY::arrayclass(PhiloType,philo1,1,5));
   for i:=1 to 5 do
     philos.get(i).lifecycle(i,forkset);
   end;
end;
```

Appendix B

Concurrent $\mathcal{LOOM}$ Grammar

```
* Indicates changes from original LOOM grammar

modulefile : [compunitlist [";"]]

compunitlist :
          interfaceUnit [";" compunitlist]
        | moduleUnit [";" compunitlist]
        | programUnit [";" compunitlist]

interfaceUnit : INTERFACE ID importlist typedeclist END

moduleUnit : MODULE IMPLEMENTS ID importlist abbrevlist constlist END

programUnit : PROGRAM ID importlist abbrevlist unitBlock

abbrevlist : [ABBREV abbrevdecllist [";"]]

abbrevdecllist : ID ["[" tfuncparamdecllist "]"] "=" typeExp
                 [";" abbrevdecllist]

nonemptyidlist: ID {"," ID}

idlist: [nonemptyidlist]

importlist : [IMPORT nonemptyidlist ";"]
```

```
typedecl :
          ID ":" typeExp
        | ID ["[" tfuncparamdecllist "]"] "<#" typeExp
        | ID ["[" tfuncparamdecllist "]"] "=" typeExp

typedeclist : typedecl {";" typedecl} [";"]

unitBlock : constlist varlist BEGIN stmtlist END

funcBlock :
         ":" typeExp constlist varlist BEGIN stmtlist RETURN exp [";"] END

constlist : [CONST constant {";" constant} [";"]]

constant :
         ID "=" funcExp
       | ID "=" exp ":" typeExp
*      | CLASS ID ["(" funcparamdecllist ")"] initializedvarlist methodlist
          presynchsection postsynchsection END
       | CLASS ID ["(" funcparamdecllist ")"] inheritexp END
       | FUNCTION ID "(" funcparamdecllist ")" funcBlock
       | PROCEDURE ID "(" funcparamdecllist ")" unitBlock

*inheritexp : INHERIT exp [MODIFYING idlist [";"]] initializedvarlist
                methodlist presynchsection postsynchsection

initializedvarlist : [VAR initializedvardecllist]

initializedvardecllist :
          ID "=" funcExp [";"]
        | ID "=" exp ":" typeExp [";"]
        | ID "=" funcExp ";" initializedvardecllist
        | ID "=" exp ":" typeExp ";" initializedvardecllist

varlist : [VAR vardecllist]

vardecllist : nonemptyidlist ":" typeExp
                        {";" nonemptyidlist ":" typeExp} [";"]

tfuncparamdecllist : nonemptyidlist "<# typeExp
```

```
                        {";" nonemptyidlist "<# typeExp } [";"]

funcparamdecllist :
          [nonemptyidlist "<# typeExp [";"]]
        | [nonemptyidlist ":" typeExp [";"]]
        | nonemptyidlist "<# typeExp ";" funcparamdecllist
        | nonemptyidlist ":" typeExp ";" funcparamdecllist

methodlist : [METHODS methodsection]

methodsection :
          HIDDEN methods [methodsection]
        | VISIBILE methods [methodsection]

methods : ID "=" funcExp {";" ID "=" funcExp} [";"]

*presynchsection : [PRESYNCH presynchlist]

*presynchlist : ID "(" funcparamdecllist ")" ":" exp [";"]
                  [presynchlist]

*postsynchsection : [POSTSYNCH postsynchlist]

*postsynchlist : ID ":" [SUPERPOST ";"] enablelist disablelist [";"]
                   [postsynchlist]

*enablelist : [ENABLE idlist ";"]

*disablelist : [DISABLE idlist ";"]

nonemptytypelist : typeExp {"," typeExp}

typelist : [nonemptytypelist]

methodtypes : [METHODS methodtypesection]

methodtypesection :
          HIDDEN vardecllist [methodtypesection]
        | VISIBLE vardecllist [methodtypesection]
```

```
objectmethods : [[METHODS] [vardecllist]]

typeExp :
          ID
        | nonidtypeExp
        | "(" ID ")"

nonidtypeExp :
          "(" nonidtypeExp ")"
        | ID "[" nonemptytypelist "]"
        | ID "<#" typeExp
        | MYTYPE
        | BOOLEANTYPE
        | NUMBERTYPE
        | REALTYPE
        | STRINGTYPE
        | TOPOBJTYPE
        | TOPCLASSTYPE
        | ARRAYTYPE "[" typeExp "]"
        | HETARRAYTYPE "[" typeExp "]"
        | FUNCTYPE "(" typelist ")" ":" typeExp
        | PROCTYPE "(" typelist ")"
        | CLASSTYPE [INCLUDE typeExp] varlist methodtypes END
        | OBJECTTYPE [INCLUDE typeExp] objectmethods END
        | "#" typeExp

stmtlist : {stmt ";"}

stmt :
          exp ":=" exp
        | WHILE exp DO stmtlist END
        | IF exp THEN stmtlist {ELSIF exp THEN stmtlist}
                              [ELSE stmtlist] END
        | exp "(" paramlist ")"
        | FOR exp ":=" exp [BY exp] DO stmtlist END

funcExp :
          FUNCTION "(" funcparamdecllist ")" funcBlock
        | PROCEDURE "(" funcparamdecllist ")" unitBlock
```

```
param : exp | typeExp

paramlist : [param {"," param}]

exp :
          ID
        | nonidExp
        | "(" ID ")"

nonidExp :
          "(" nonidExp ")"
        | INTEGER
        | REAL
        | BOOLEAN
        | STRING
        | SELF
        | NIL
        | TOP
        | ARRAYCLASS "(" typeExp "," exp "," exp "," exp ")"
        | HETARRAYCLASS "(" typeExp "," exp "," exp "," exp ")"
        | exp "(" paramlist ")"
        | SINGLEOP exp
        | exp INFIXOP exp
        | exp "." ID
*       | WAIT exp ".." ID
        | SUPER "." ID
*       | SUPERPRE "." ID
        | funcExp
        | NEW "(" exp ")"
*       | CLASS [INHERIT exp [MODIFYING idlist] ";"] initializedvarlist
            methodlist presynchsection postsynchsection END
```

Bibliography

[Agh87] Gul Agha. *Actors.* MIT Press, 1987.

[Agh90] Gul Agha. Concurrent object-oriented programming. In *Communications of the ACM Vol 33 (9)*, pages 125–141. ACM, 1990.

[AH87] Gul Agha and Carl Hewitt. Concurrent programming using Actors. In *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, 1987.

[Ame87] Pierre America. POOL-T: A parallel object-oriented language. In *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. In *Formal Aspects of Computing*, volume 1, pages 366–411, 1989.

[AvdL90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. *Proc. OOPSLA-ECOOP '90*, 25(10):161–168, October 1990.

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.

[BHJ+87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. In *IEEE Transactions on Software Engineering*, pages 65–76. IEEE, January 1987.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proc. OOPSLA '86*, pages 78–86. ACM, September 1986.

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994. An earlier version of this paper appeared in the 1993 POPL Proceedings.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *Proc. ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995.

[BvG93] Kim B. Bruce and Robert van Gent. TOIL: A new type-safe object-oriented imperative language. Technical report, Williams College, 1993.

[Car93] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–101, 1993.

[Car95] Luca Cardelli. Obliq: A language with distributed scope. In *Proc. POPL '95*, pages 286–297. ACM, 1995.

[ES90] Margaret A. Ellis and Bjarne Stroustrop. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Frø92] Svend Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proc. ECOOP '92*. Springer-Verlag, 1992.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley Longman, 1996.

[GM88] Narain Gehani and Andrew D. McGettrick. *Concurrent Programming.* MIT Press, 1988.

[Jon96] C.B. Jones. Accommodating interference in the formal design of concurrent object-based programs. In *Formal Methods in System Design Vol. 8(2)*, pages 105–122. Kluwer Academic Publishers, March 1996.

[Mes93] Jose Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar M. Nierstrasz, editor, *Proc. ECOOP '93*, pages 220–246. Springer-Verlag, LNCS 707, 1993.

[Mey97] Bertrand Meyer. *Concurrency, distribution, client-server and the Internet.* Prentice Hall, 1997. Text used comes from a book draft found at www.eiffel.com/doc/manuals/technology/concurrency/concurrency.html.

[MK87] J. Eliot B. Moss and Walter H. Kohler. Concurrency features for the Trellis/Owl language. In G. Goos and J. Hartmanis, editors, *Proc. ECOOP '87*, pages 171–180. Springer-Verlag, LNCS 276, 1987.

[MTY93] Satoshi Matsuoka, Kenjiro Taura, and Akinori Yonezawa. Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages. In *Proc. OOPLSA '93*, pages 109–126. ACM, October 1993.

[MWY90] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. Synchronization constraints with inheritance: What is not possible - so what is? Technical Report 10, Department of Information Science, University of Tokyo, 1990.

[Pap95] Michael Papathomas. Concurrency in object-oriented programming languages. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 29–68. Prentice Hall, 1995.

[Pet96] Leaf Petersen. *A Module System for LOOM*. Williams College Senior Honors Thesis, 1996.

[Tho94] Laurent Thomas. Inheritance anomaly in true concurrent object-oriented languages: A proposal. In *IEEE TENCON '94*, pages 541–545. IEEE, August 1994.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN 96 Conference on Programming Language Design and Implementation*. ACM, 1996.

[vG93] Robert van Gent. *TOIL: An imperative type-safe object-oriented language*. Williams College Senior Honors Thesis, 1993.

[Wei95] Richard Weiner. *Software Development Using EIFFEL*. Prentice Hall PTR, 1995.

[YSTH87] Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada, and Yasuaki Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, 1987.

[YT87] Yasuhiko Yokote and Mario Tokoro. Concurrent programming in Concurrent-Smalltalk. In *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, 1987.