Polymorphic Types for Constructing Concurrent Objects and Layered Communication Protocols

by

Robert Gregory Lavender

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

APPROVED:

Dennis G. Kafura, Chairman

Richard E. Nance

J. A. N. Lee

James (Sean) D. Arthur

Marc Abrams

Scott F. Midkiff

Philip E. Cannata

May 1993
Blacksburg, Virginia

C.2

Polymorphic Types for Constructing Concurrent Objects and Layered Communication Protocols

by

Robert Gregory Lavender

Committee Chairman: Dennis G. Kafura

Computer Science

(ABSTRACT)

Polymorphic type abstractions are proposed for constructing concurrent objects and layered peer-to-peer communication protocols, both of which exhibit inherently asynchronous behavior. The type abstractions are shown to be both expressive and efficient in the context of a statically typed object-oriented language. Where appropriate, the utility of the type abstractions is illustrated by demonstrating their usefulness in concurrent programming using the Actor model. The results of this work have direct applicability to problems in concurrent programming, distributed systems, and communication protocol architectures.

An extensible, polymorphic type abstraction for structuring concurrent method execution in a strongly typed object-oriented language is introduced. The type abstraction is called a *polymorphic lambda type*. A lambda type is an abstraction for a procedure that is based in part on λ-abstraction in the λ-calculus. The lambda type is a key component of a concurrent object model that allows methods defined in a class to be instantiated as *lambda objects*. Lambda objects are used to represent some aspect of behavior and they represent a first-class execution environment. The first-class nature of lambda objects facilitates the construction of more powerful computational abstractions, primarily those requiring asynchronous interaction and concurrent execution. Through a series of refinements, lambda objects are shown to be as expressive as traditional procedures with little extra cost for call setup and invocation.

Concurrent objects require synchronization control. A type abstraction called a *behavior set* is introduced for specifying synchronization constraints in a strongly typed concurrent object-oriented language with Actor-style concurrency semantics. The behavior set abstraction offers a solution to structuring synchronization control that coexists with an inheritance mechanism, thereby avoiding the *inheritance anomaly*. The inheritance anomaly

is the name given to a conflict that arises when trying to reuse a class specification that incorporates synchronization constraints. The fundamental cause of the inheritance anomaly is explicated using the formalism of the Calculus of Communicating Systems. The solution of the inheritance anomaly is important since reuse of class specifications incorporating synchronization constraints is severely restricted by traditional synchronization approaches.

Concurrent object-oriented languages also require mechanisms to cope with the potential delay between the time at which a method is asynchronously invoked and the time at which a result value is computed. In general, the value returned as the result of an asynchronous invocation is a handle to the eventual result value, if any, of the asynchronously invoked method. The handle is called a *future* because it represents a value that is yet to be computed. The introduction of a *polymorphic future type* into a statically typed language with traditional procedure call semantics is problematic—futures should be expressible as user-defined types, extensible via inheritance, and efficiently realized. The temptation to introduce new syntax for expressing future types is strong; however, polymorphic types and *type conversion* operators permit the expression of futures as synchronized continuation objects. Futures imply the need for synchronization control between the concurrent operation producing the result value represented by the future and one or more threads that may be waiting on this value.

Finally, *polymorphic service access points* and the inheritance structure of a set of types representing protocol machines are introduced. The protocol structure is specifically designed to support concurrent objects requiring asynchronous peer-to-peer communication, such as Actors. An implementation is presented that illustrates the use of object-oriented structuring techniques in the implementation of a communication protocol architecture, resulting in a type structure that reflects a layered protocol architecture while providing an efficient implementation. Performance data is provided that demonstrates that the inheritance-based protocol structure incurs a modest 2% overhead in comparison with a traditional layered protocol structure.

Acknowledgements

The past four years have been spent trying to convince Dennis Kafura that my ideas and work were of dissertation quality. In trying to convince Dennis, I often convinced myself that I needed to do more thinking, and he often convinced me to abandon a particularly odd idea. These discussions usually took place over lunch, or in route to and from lunch. I think that for both of us the casual discourse we were able to achieve being out of the environs of McBryde Hall contributed significantly to the formulation of the ideas presented here.

Dick Nance has been an academic mentor and a professorial role model. He provided guidance through the first two years of my graduate study. Early on, when I felt the temptation to leave school and return to a safe and lucrative industrial position, I sought his counsel. Invariably the decision to continue my studies was my own, but Dick's observations concerning the benefits of an academic life allowed me to explicate my true feelings and goals.

Whereas Dennis and Dick have been my academic guideposts, Sean Arthur has contributed to the realization that an academic lifestyle is more than teaching, conducting research, and rushing to publish papers. His Faulknerian nature and sense of humor, in conjunction with his commitment to excellence, differentiates him from his peers.

J. A. N. Lee provided a thorough reading and critique of this dissertation. Any grammatical mistakes or factual errors can only be attributed to my neglecting to take all of his advice. I especially valued the comments and opinions of Marc Abrams and Scott Midkiff, since younger faculty are quick to recognize new concepts and formulate new relationships.

Some of the implementation supporting this dissertation was done at MCC, at the invitation of Phil Cannata and Chris Tomlinson. Phil provided the project environment, equipment, and threats that facilitated my progress. Chris acted as my unofficial advisor during the time I was away from the University.

Bob Slaski, President of Open Networks, has been a friend and professional mentor. Our mutually beneficial relationship financed much of my graduate education. More importantly,

the work I did at Open Networks allowed me to keep a pragmatic perspective on my research activities at the University, resulting in a desire to produce something useful.

Benjy Cline, Ashley Engelund, Ben Keller, Keung Hae Lee, and Cindy Dominguez-Williams have been supportive friends at various times during my graduate studies. Benjy's witty sarcasm and healthy skepticism matches my own. His competitive academic nature helped motivate me in the courses we took together, during quals, and in the final dissertation race. I beat him to a defense, but he beat me in course work, quals, and in turning 40. Ashley's friendship, love, and encouragement up through my Masters thesis and qualifying exams contributed to my eventual progress. She is to be credited with offering emotional support and helping to convince me that I was capable—the process was painful, but I learned a lot about myself. Conversations with Ben on the seemingly confused and immature state of the discipline of Computer Science as a science, helped me keep a healthy perspective on my work and the published literature. Keung Hae will recognize in this work some extensions to his original ideas. He also imparted to me some of the wisdom of Korean philosophy, which I find applicable in daily life. Cindy offered encouragement, entertained my rantings and ravings during spells of writer's block, and provided editorial assistance.

My best friend Rebecca has been very patient throughout. Her belief in my abilities and her desire for my time forced me to eventually put aside books and papers on mathematics and theoretical computer science, and finish the work I had already started. Without her as a motivating factor, I would still be trying to wrestle with marginal theoretical topics instead of the considerably more tractable topics discussed within this document.

Ultimately, my parents can take credit for my accomplishments. When I was young they provided an environment that emphasized education. My mother instilled in me a desire to read and my father directed me towards science.

Final thanks go to the people that wrote all the freely available software I used to do my research. Foremost in mind is the Free Software Foundation for providing the excellent GNU software tools, particularly: gcc, g++, gdb, emacs, and ghostview. Donald Knuth, Leslie Lamport, Oren Patashnik, Van Jacobson, and Tomas Rokicki created the publicly available TEX tools that facilitated the presentation of my ideas. Marshall Rose is to be commended for originally conceiving of the idea of OSI over TCP, implementing the ISO Development Environment, and making this valuable research tool freely available.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The science of computation is firmly grounded in trying to understand the structure and processing of information and in devising symbolic representations that allow the mechanical manipulation of information. Much of modern theoretical computer science is concerned with the formal *analysis* and *synthesis* of different computational structures, many of an algebraic nature. Formal logical systems and their computational interpretations, type theory, denotational semantics, etc., all benefit from the interplay of depth-first analytic reasoning and breadth-first synthetic reasoning. Experimental computer science is mostly concerned with synthesizing macroscopic structures in order to produce working systems. Because of real-world constraints and the large scale nature of the problems, a complete formal analysis is elusive. Lacking a complete theory, general principles derived from experience combined with generalized knowledge gleaned from diverse formal theories provide the methodological foundation for building real systems.

At present, object-oriented languages offer highly sophisticated structuring mechanisms for synthesizing software systems; i.e., classes, inheritance, and polymorphism. Inheritance and polymorphism are emerging theoretical concepts that are currently realized by language structuring mechanisms that must be applied in a principled manner to specific problem domains. This dissertation is the result of a depth-first analysis of the type structure of concurrent object-oriented programming and layered communication protocols, resulting in a reformulation of the underlying structures using polymorphic types and class inheritance. The contribution of this dissertation is a type-based programming infrastructure that can be used to synthesize concurrent and/or distributed object-oriented systems. In this respect, this work is theoretically motivated, but presented in terms of programming abstractions that can be used to construct non-trivial software systems.

1.1 Motivation

Given a problem domain for which a significant portion of the computation is inherently *concurrent* and dominated by *asynchronous communication* among components, what are appropriate abstractions for representing computational objects and structuring their interactions? How are sophisticated programming language constructs to be used in constructing a solution that is both expressive and efficient? The following two issues are the motivation for this dissertation:

- given a model of concurrency, how are synchronization constraints expressed and concurrent execution facilitated.
- given a model of communication, how is communication to be structured and efficiently realized?

Diverse research has been done with respect to developing formal specifications for distributed systems, and establishing verification techniques for such specifications. Research efforts are also focused on bridging the "semantic" gap between specifications and their implementations. Certainly, progress has been made with respect to automatic program generation; however, the state of the art in many problem domains still depends on highly competent programmers developing functionally correct implementations—irrespective of the existence or non-existence of formal specifications. To be clear, formal specification and verification of some, or all, of a system's properties are an integral part of any serious software development activity. However, the fact remains that given a specification, much creative work is yet to be done to realize an effective implementation. The research presented here offers a polymorphic, extensible type-based infrastructure that is a step across the bridge between specification and implementation.

In order to narrow the scope of the problem, the following constraints are applied:

- The Actor model [3] is adopted as the primitive concurrency model since communication among peer objects is based on asynchronous message-passing.
- The ISO Reference Model for Open Systems Interconnection (OSI) [92] is adopted as the architectural basis for layered communication due primarily to its asynchronous, peer-to-peer model of service.

- The language for expressing type abstractions is assumed to have the attributes of a strongly typed object-oriented language with the ability to express user-defined types, extend types using inheritance, and define polymorphic types or type templates. Although any one of a number of languages could in principle be used, C++ [58] has sufficient expressive power, the syntax is widely understood, and it is generally efficient.[1]

- The system context is assumed to be equipped with programming facilities for process creation as well as local and remote interprocess communication. A microkernel supporting multiple lightweight kernel threads within a single process is ideal, (e.g., Mach [2] and its derivatives, or Solaris [165]).

The two core issues presented above represent a difficult and varying set of problems. The contributions of this thesis can only be considered a small, but important, step towards a complete solution. The primary contribution of this work is the definition of a set of type abstractions that capture the essential character of control and data relations that arise in representing and structuring:

- asynchronous interaction among concurrent objects requiring synchronization, and

- asynchronous communication based on layered protocol machines.

Collectively, the type abstractions provide the infrastructure for constructing concurrent and/or distributed applications in a heterogeneous computing environment (e.g., open systems). To this end, the results reported motivate additional work for programming language designers, concurrency theorists, and implementors of concurrent and distributed systems.

1.2 Basic Concepts

In the science of programming, structural reasoning occurs using two principal cognitive processes: *specialization* and *generalization*, which Polya [142, p. 12] defines as duals of one another:[2]

[1] C++ is heavily influenced by Simula–67 [20, 48], which is the progenitor of many of the fashionably "new" ideas in object-oriented programming.

[2] This list is not meant to be complete; for example, one could easily include *induction* and *analogy* as well.

Specialization is passing from the consideration of a given set of objects to that of a smaller set, containing the given one.

Generalization is passing from the consideration of a given set of objects to that of a larger set, containing the given one.

Specialization often precedes generalization since consideration of special cases leads to identification of general properties. An abstraction is a formalized representation of a set of properties that is the result of the process of generalization. In particular, mathematicians define abstract algebraic systems as representations for sets of objects and operations that exhibit common structural properties, thereby facilitating their ability to reason using generalities, which can then be applied to specific concrete instances.

The field of computer science, heavily influenced by the methodology of mathematics, applies specialization, generalization, and abstraction in a similar fashion; however, the emphasis is on the concepts of *control* and *data*, which are central concepts in computer science due to the *operational* and *constructive* nature of the discipline. In subsequent chapters, the terms generalization and abstraction are used frequently. Generalization usually refers to a cognitive process and an abstraction usually refers to the symbolic language construct resulting from the process of generalization (as in a data abstraction). The use of the term abstraction in this sense follows Liskov and Guttag [116].

1.2.1 First-Class Types

The mathematical concept of *type* is central to programming. Computer scientists are primarily interested in characterizing and constructing information systems that facilitate effective computations involving symbolic representations of typed objects. A type is a *domain* where membership in the domain is determined by the truth value of the conjunct of a set of predicates denoting assertions regarding specific properties of the domain that define its (algebraic) structure. A type domain may consist of integers, real numbers, strings, functions, or more complex objects built up from these basic typed objects using domain constructors, such as products and sums, to construct new type domains.

Of particular interest to later considerations is the degree to which an abstraction is treated as a *first-class* type in a programming language. The term first-class under most

interpretations means a language construct that may, in general, be:

1. passed as an argument to a function or procedure,

2. returned as a value by a function or procedure, and

3. used as a component of an arbitrary data structure.

For example, a language construct for representing a mathematical function is considered first-class if it meets the above criteria. In many languages, the procedure abstraction that is used to represent a function does not typically have all of the above attributes. In examining procedure abstraction in diverse language settings, one might also consider the degree to which procedures exhibit the properties of an abstract data type; in which case, the following additional attributes would be required for an arbitrary procedure abstraction to be considered first-class:

4. its representation is encapsulated, and

5. it is manipulated only by a set of well-defined operations.

The procedure construct in most languages provides a protected scope, hence the first property is satisfied. The second property is perhaps the most important since it usually only applies to abstract data types. A abstract data type is partially defined by its operations. The focus of the following chapters is on abstract data types that represent concurrent operations and continuations in an object-oriented language setting; henceforth, for our purposes the term "first-class" shall be applied to any language construct with all five properties.[3]

1.2.2 Lexical Closures

A critical construction in programming is the *lexical closure* or *binding contour*. A closure is a representation for the scope and the bindings of a set of identifiers that are

[3]The abuse of terminology in Computer Science is unfortunate, but established. It would be more precise to use the term *first-order abstraction* to denote a construct with attributes 1–3, and the term *second-order abstraction* to denote a first-order abstraction with the addition of properties 4 and 5. This terminology would be more appropriate since the intended meaning is that of a basic abstraction upon which are defined operations that enable manipulation of the basic abstraction at a higher conceptual level. This terminology would then be consistent with the general interpretation given to a sequence of higher-order relations.

the basis for *environments*. Scope is a spatial (static) concept and binding is a temporal (dynamic) concept. The scope of an identifier is defined by the lexical closure in which it is statically declared. A particular binding of an identifier is equal to the lifetime of the containing closure during execution.

Closures are the basis for the common block and procedure abstractions. The primary purpose of the typical `begin-end` construct is to allow the programmer to express the scope of a set of variable names accessible by a privileged set of expressions and statements. The ability to define private and structured data locales in this way facilitate program development, debugging, and maintenance.

A procedure construct permits parameterization of a lexical closure. The procedure extends the usefulness of a block by allowing a block to be evaluated with different sets of variable bindings, resulting in a theoretically infinite number of different block instantiations. If the statements and expressions defined in a block are viewed as denoting a specific mathematical function, then a procedure construct extends the computational power of the block construct by allowing the evaluation of the block with all argument values from the domain of the function.

A basic assumption underlying both the block and procedure abstraction is the fact that the control structure being generalized is sequential. Dahl and Hoare, writing (over 20 years ago) on the advantages of the class construct in Simula–67 over the block in Algol, make the key observation about the disadvantage of the block as a realization for a lexical closure [47, p. 179]:

> It [the block] has the disadvantage that a program which creates a new block instance can never interact with it as an object which exists and has attributes.

The problem of course is the stack-oriented nature of blocks in Algol, and its descendants, versus the heap-oriented nature of lexical closures realized by the `class` construct in Simula–67, and its descendants. Stack-based environments are sufficient for expressing sequential nested control, and can be efficiently realized because of explicit hardware support for stack operations. However, stack-based environments present problems in trying to express non-nested, non-sequential control. For example, ad hoc threads packages added to a strongly typed sequential language must often subvert the procedure call mechanism and type checking in order to effect concurrent execution.

1.2.3 Data Type Abstraction

The continuing trend in programming languages begun with Simula–67 and CLU [115, 116], is to recognize the importance of first-class data abstractions and typed procedure abstractions as fundamental language constructs necessary for expressing solutions to non-trivial computational problems.

Functional languages, starting with Lisp and elegantly realized by Scheme [40] and ML [128], extend the notion of procedural abstraction even further by allowing procedures, or lambda expressions, to be treated as first-class language constructs.

Today, many modern imperative and functional language implementations provide syntactic constructs supporting the expression of data and procedural abstractions. However, many languages provide relatively rudimentary control abstractions. Scheme, with the `call/cc` mechanism for explicit continuations, Simula–67 with coroutines, and Argus, a descendant of CLU for distributed programming [113], are notable exceptions.

Data abstraction in current object-oriented languages is the result of the gradual recognition over time of the central place of the concept of type. Danforth and Tomlinson present an overview of the use of type abstraction in object-oriented languages [49]. The object-oriented programming paradigm originated in the modeling and simulation community with Simula–67, and the paradigm became more widely known with the introduction of the Smalltalk programming language and environment [63]. The current interest in object-oriented languages is the result of the natural progression of the early ideas of data and procedural abstraction, and the desire to provide more expressive language constructs supporting extensibility of types and data abstractions through subtyping and hierarchical composition. It is generally recognized that Simula–67 was the first programming language to employ the class abstraction and a class inheritance mechanism as a means for expressing static hierarchical composition of abstract data types [47]. In this work, the class is adopted as the primary syntactic construct for representing data abstractions principally because the class construct and the class inheritance mechanism in an object-oriented language permits the definition of first-class, polymorphic types. The class permits the expression of generalized types that may be specialized using inheritance, while type polymorphism permits the expression of generalized class templates that may be instantiated as specific types.

1.2.4 Control Abstraction

A *control abstraction* is a representation for some component of the execution environment of a program. A control abstraction is often realized using a data abstraction mechanism, but it is useful to distinguish a control abstraction as distinct from a data abstraction.

Most programmers are familiar with the following basic abstractions for representing aspects of control: *labels*, *procedures*, and *coroutines*. The most sophisticated control abstraction likely to be found in statically-typed imperative languages, is the coroutine. More sophisticated but less well-known control abstractions are *continuations*, *threads*, and *futures*, which are discussed in a subsequent chapters.

Labels are abstractions for control transfer using `goto`. Since publication of Dijkstra's paper[53] advocating the abolishment of the `goto` statement from high-level languages, labels are rarely visible in programs since control flow in most sequential algorithms is structured using `for`, `while`, and `repeat` constructs, or by a procedure call construct (implicit `goto`). A common situation for the use of labels in a high-level language is in the implementation of finite state automata that often require non-structured transfers of control to labels representing states in the automata. For example, event-driven control flow in protocol machines is often implemented using the `goto` statement because of the need for un-structured transfer of control.

Procedures are ubiquitous in imperative and functional languages. A procedure abstraction is often realized by a stack-based structure; however, a slight extension to the basic block abstraction is required since argument bindings and a return context must be accounted for by the procedure abstraction. It is not difficult to trace the historical development of the procedure concept as an abstraction—Turing Machines use sets of machine configurations, in the form of macros, for commonly executed tape movements. Procedural abstraction in its current manifestation in most imperative-style programming languages is a direct derivative of the procedure abstraction in Algol. Landin [109] was the first to recognize the correspondence between the procedure abstraction in Algol and untyped lambda abstraction in Church's λ-calculus [34], thus relating the imperative and functional styles for representing computable functions. A key feature of the type-free λ-calculus is the emphasis on free and bound variables and the scope of a variable within a lexical expression.

Like lambda abstraction in the λ-calculus, procedure abstraction facilitates the expression of the scope and extent of variables in a portion of program text, with the addition of strong typing.

A coroutine is a procedure that has a data or control dependency on another procedure. Coroutines have utility in structuring control flow between co-dependent procedures in a peer-to-peer calling relationship rather than the traditional client-server calling relationship. The most commonly cited example, and the original motivation for the coroutine concept described by Conway [44], is the cooperative relationship between routines representing the lexical analysis and the parsing phases in a language translator.

Simula–67 is one of the few early languages to provide direct support for the expression of coroutines within the language. The use of coroutines in Simula–67 is unique because of the existence of the class abstraction. Dahl and Hoare [47] investigated the combination of data abstraction using classes and control abstraction using coroutines. Interest in coroutines following Conway's original paper and their use in Simula–67 led to several language designs that included syntactic constructs for structuring coroutine relations [121]. Explicit control is also possible between coroutines given the ability of a coroutine to overtly cause itself to be suspended and transfer execution control to some other coroutine. Cooperating procedures may implement an explicit token passing protocol whereby each procedure decides when to give the control token to another procedure. Coroutines are interesting because they represent a combination of the primitive concepts of a continuation and a thread, which will be considered in a subsequent chapter.

1.3 Outline

The remainder of this dissertation is organized into six chapters. The concepts presented cover a seemingly wide range because of the intent to analyze and then synthesize new programming structures. To those so inclined, it is possible to skip around in the reading by following the chapter dependency graph depicted in Figure 1.1.

Chapter 2 introduces the concept of a function as an object using a *polymorphic lambda type* as a first-class representation for a function. Examples are given of the use of the lambda type in a sequential setting, which is then extended to a concurrent setting. The primary purpose of the lambda type is to serve as a semantic basis for implementing op-

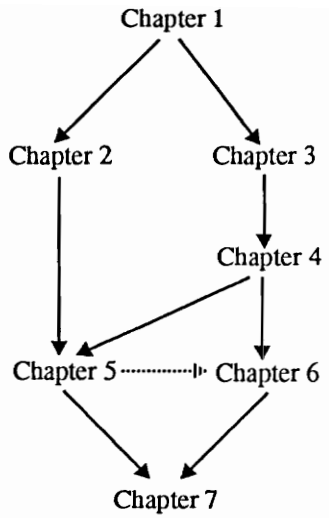

Figure 1.1: Chapter Inter-Dependencies.

erations that are concurrently executed using lightweight execution threads; however, this aspect of the lambda type is deferred until Chapter 5.

Chapter 3 introduces a set of type abstractions for expressing synchronization control that are compatible with inheritance in a strongly typed object-oriented language. Synchronization control in concurrent object-oriented languages is problematic because the inheritance mechanism and synchronization mechanism often interfere with one another. A problem called the *inheritance anomaly* arises when trying to reuse a class specification that contains synchronization code. The problem of structuring classes so that synchronization constraints may be reused requires an understanding of the phenomenon underlying the inheritance anomaly. The inheritance anomaly is first explained in terms of traditional synchronization using monitors. Subsequently, Milner's Calculus of Communicating Systems (*CCS*) is used as a tool to illustrate the underlying cause of the anomaly. Finally, a solution called a *behavior set* is introduced as an inheritance-compatible mechanism for expressing synchronization constraints. Behavior sets are shown to be specifically useful for structuring synchronization in the context of the Actor paradigm for concurrent computation.

The synchronization mechanisms introduced in Chapter 3 are used to construct a general mechanism for the handling of the result of an asynchronous operation. Chapter 4

introduces the concept of a *polymorphic future type*. The handling of the result value of an asynchronously invoked, concurrent operation is problematic because the procedure invocation mechanism in a stack-based execution model assumes sequential execution, meaning that a result value is immediately available on return from the procedure. Futures provide a synchronized rendezvous mechanism that permits the delayed binding of a result value of a concurrent computation. Polymorphic futures differ from similar solutions, reviewed in this chapter, by allowing arbitrary, user-defined future types.

Lambda types and future types are the fundamental building blocks for constructing different concurrent programming paradigms. The utility of these type abstractions is demonstrated in Chapter 5 where inheritance is used to extend the lambda type to construct a set of types for representing the components of the Actor model. Constructing components of the Actor model using polymorphic lambda types and polymorphic future types suggests a more fundamental view of concurrent computation of which actor programming is but one specialization.

Chapter 6 is concerned with an object-oriented approach to structuring layered peer-to-peer protocol machines. The synchronization concepts introduced in Chapter 3 and the polymorphic future type introduced in Chapter 4 are essential components. The primary focus however is the introduction of a new approach to structuring protocol machines based on a *vertical partitioning* that results in a *non-monolithic* protocol stack. A non-monolithic protocol stack facilitates concurrent network-based applications. The type abstractions introduced in this chapter offer new results on structuring layered communication protocols. The principal contribution is a design and efficient implementation, called OOSI, that is based on polymorphic service access points and a type hierarchy of protocol machines implementing the services of the upper layers of the ISO Reference Model. The OOSI implementation forms the basis for sophisticated communication among concurrent objects in a heterogeneous distributed system.

Specific suggestions for future work are made at the conclusion of each of the individual chapters. Chapter 8 concludes by summarizing the principal results of this work.

Chapter 2

Representing Functions as Objects

A class-based abstraction for representing functions as objects that permits the treatment of procedures as first-class types is introduced in this chapter. Function objects are derived in part from lambda abstraction in the λ-calculus, the operational semantics of applicative-order procedure evaluation, and the concept of a first-class closure. Function objects are defined as instances of a *polymorphic lambda type* that allows the specification of an arbitrary function as a derived type. The polymorphic lambda type is parameterized by the type of the co-domain of the function being represented

Through a series of refinements, the polymorphic lambda type is shown to be as expressive as a common procedure abstraction for representing a function. The polymorphic representation introduces little additional overhead with respect to call setup and invocation for both sequential and concurrent execution. The polymorphic lambda type is an essential building block for systems in which asynchronous interaction among concurrent objects dominates the computation; for example, Actor-style computations. The concepts introduced in this chapter are a prerequisite to the presentations in Chapters 4 and 5.

2.1 Abstractions for Computable Functions

An object-oriented approach to representing a function is motivated by the desire to treat the closure for the function as a first-class, polymorphic type. The purpose of this section is to briefly review the syntactic and structural aspects of programming abstractions for representing computable functions.

In this discussion, the term *function* is used to denote a mathematical entity f of the class $\mathcal{C}(F)$ of computable functions, while the terms *procedure*, *method*, and *lambda* are used to denote the programming language representation of an effective algorithm for computing f. The intent here is not to review the exact nature of computable functions as mathematical

objects; rather, at issue is the definition of a representation suitable for expressing *structural* aspects of procedures and the closure required for procedure execution. The main focus is the definition of a polymorphic type that captures the structural characteristics of a procedure while preserving familiar procedure semantics. A primary goal is is that either sequential or concurrent procedures may be specified using the same basic abstraction. The polymorphic nature of the abstraction arises as a result of the variability in the type of the result value produced by a procedure.

The distinction between a computable function and the symbolic representation of an effective procedure for the function is at the core of programming. The type-free λ-calculus is sufficient for representing the *general* attributes of computable functions—particularly the notions of symbolic representation by λ-abstraction, free and bound variables, and computation by substitution of bound variables and expression reduction.[1]

To motivate the reason for seeking a new programming abstraction for representing functions when seemingly expressive ones already exist, a series of examples are presented using the *factorial* function. The factorial function is used as a canonical example for expressing concepts in this chapter because of its simplicity, and the fact that the function may be represented equivalently by distinct algorithms—one iterative and the other recursive.

Using standard notation, the factorial function may be expressed as a map from the domain N of natural numbers (non-negative integers) to the co-domain N:

$$factorial : N \rightarrow N$$

This notation emphasizes that the factorial function is a member of the *type domain* $N \rightarrow N$ of unary functions on N. Nothing more can be derived from this notation, except by intuition. Alternatively, the algorithmic nature of the factorial function may be expressed as:

$$n! \stackrel{\text{def}}{=} n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1, \qquad n \in N \text{ and } 0! = 1.$$

Unlike the preceding representation, this one suggests an explicit iterative algorithm for computing the value of the factorial function for some value of n, from which one can derive the recursive formula $n! = n \cdot (n-1)!$.

[1] Barendregt [13] provides the authoritative analysis of the type-free λ-calculus; however, the presentation by Hindley and Seldin [74] is more instructive.

In contrast to standard mathematical notation, the expression of the factorial function using λ-abstraction in the type-free λ-calculus provides explicit information of an algorithmic nature. The λ-abstraction for the recursive algorithm is often written as:

$$factorial \stackrel{\text{def}}{=} \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \cdot factorial(n-1).$$

It is commonly known that the above formula is actually an equation, not a definition, since the definition is defined in terms of itself. To be technically precise, the definition should be written using an additional level of abstraction followed by self-application:

$$\begin{aligned} F &\stackrel{\text{def}}{=} \lambda f.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \cdot f(n-1) \\ factorial &\stackrel{\text{def}}{=} (F)factorial \end{aligned}$$

In the type-free λ-calculus, the syntactic form $(\lambda x.M)N$ is called an *application*, where the term N is applied to the abstraction $\lambda x.M$, subject to the substitution rules of β-conversion [13]. The additional level of abstraction is needed to avoid recursion in the definition. The need for self-application of the form $f \stackrel{\text{def}}{=}$ (λ-abstraction)f is a result of the syntax and semantics of anonymous function abstraction in the λ-calculus. The underlying semantics [65, 158], not discussed here, require this type of precision.[2] The observation of interest is that an abstraction for representing a function in the λ-calculus necessarily includes two distinct bindings: the function itself and the arguments to the function.

In general, λ-abstraction emphasizes the expression of three important structural aspects of a representation for a computable function:

- the syntactic representation of an algorithm for computing the function (in this case, a recursive algorithm).
- the notion of free and bound variables and the scope of a variable; i.e, the *closure* for a function.
- the evaluation of an application by the process of substituting values for bound variables and reducing the resulting expression (β-conversion).

[2] A general solution for expressing recursion using untyped λ-abstraction depends on the $\mathbf{Y}$ combinator (e.g., see Stoy [159]): $\mathbf{Y}\lambda f.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \cdot f(n-1)$, where $\mathbf{Y} = (\lambda h.(\lambda x.h(x,x))(\lambda x.h(x,x)))$.

It is commonly known that the computational emphasis of the λ-calculus is what makes it so appealing as the meta-language for programming language semantics. For this reason, functional programming languages almost exclusively adopt the λ-calculus as their semantic foundation. Scheme [40], an applicative-order, lexically scoped language, adopts a syntax similar to the λ-calculus. The following Scheme definition is a direct transliteration, into prefix normal form, of the preceding λ-abstraction:

$$(\textbf{define}\ \textit{factorial}\ (\textbf{lambda}\ (n)\ (\textbf{if}\ (=\ n\ 0)\ 1\ (*\ n\ (\textit{factorial}\ (-\ n\ 1))))))$$

The evaluation of $n!$ in Scheme is expressed as the application of a value to the factorial expression; i.e., $(\textit{factorial}\ n)$. Applicative-order reduction semantics dictate that n is evaluated prior to evaluating the factorial expression. The difficult computational issues associated with this form of function abstraction arise with the implementation details of the process for evaluating an application of argument values to a λ-abstraction. To facilitate the evaluation process, many languages (like Scheme) adopt prefix normal form syntax and applicative-order evaluation strategies. Furthermore, the language processor is responsible for deducing type information, establishing the lexical closure required to compute the function, and detecting improper usage.

Implicit in the definition of a λ-abstraction is the *environment* in which the expression is to be evaluated, since all free and bound variables must have bindings before evaluation may occur. The scope and substitution rules for variables requires careful consideration. Lexical scoping and static binding, although restrictive, are rational choices widely adopted in programming language design.

The fact that an expression in the pure λ-calculus is untyped is an aspect of the primitive nature of the type-free λ-calculus. There is no difficulty in syntactically augmenting the lambda notation with type information so that the bound variable n is restricted to the domain N and the functional parameter f is restricted to the domain $N \rightarrow N$:

$$F \stackrel{\text{def}}{=} \lambda f : N \rightarrow N.\lambda n : N.\textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \cdot f(n-1)$$

Many strongly typed imperative-style programming languages inherit their procedure semantics from the typed λ-calculus using applicative-order (call-by-value) rather than normal-order (call-by-name) argument evaluation. The existence of the assignment statement and the potential for side-effects presents no small difficulty in specifying an adequate

denotational semantics. However, the essential idea of expressing a function as an abstraction consisting of a set of statements constituting an algorithm for the computation of the value of the function, and the addition of a *signature* clause allowing a distinction between bound and free variables, is recognized as significant in its impact on statically typed programming languages. Generally, each procedure P is identified by a type signature having a schema not unlike the following:[3]

$$P(a_0 : T_0, a_1 : T_1, \ldots, a_n : T_n) : T_{n+1}$$

The schema is used to specify a type signature for a procedure P denoting some function of type $T_0 \times T_1 \times \ldots \times T_n \rightarrow T_{n+1}$. Each parameter symbol a_i is declared to be of type T_i and is called a *formal parameter* of P. Each symbol T_i denotes the type domain over which each a_i may range. Each a_i is a symbol lexically distinct from every other a_j, $i \neq j$, occurring in the type signature. There is no such restriction on each T_i. For any T_i and T_j, $0 \leq i, j \leq n+1$, $i \neq j$, occurring in the specification of P, T_i and T_j may be the same syntactic symbol. P evaluates to a value of type T_{n+1} representing a value from the co-domain of the function represented by P. As a special case, the symbol $T_\perp$ denotes a type domain consisting of a single uninformative value, denoted by $\perp$. The following schema denotes a procedure representing a side-effecting function that takes no arguments and returns no value:

$$P(a : T_\perp) : T_\perp$$

The occurrence of procedures with this type signature is commonplace in imperative-style languages with side-effects.[4]

The declaration of a type signature serves, at a minimum, the following purposes:

- it conveys correct use information to the user of a procedure.
- it informs the language processor of the correct usage, enabling the detection of type errors by static analysis of the syntax.
- it assists the language processor in generating target machine instructions to establish the environment for the procedure, perform the invocation, and bind the result value into the environment of the caller.

[3] Variations are common; for example, $T_{n+1}\ P(T_0\ a_0, T_1\ a_1, \ldots, T_n\ a_n)$; is the schema used in C++.

[4] For example, in ANSI C and C++ the equivalent type signature is: `void P(void);` hence, $T_\perp \equiv$ `void`.

The exact syntactic form of a type signature varies according to the language, but all syntactic forms are meant to convey the same type information concerning the argument and result value types. Continuing with our canonical example, the factorial type signature may be written as:

$$factorial(n : N) : N$$

A pragmatic problem exhibited by this example is that few programming languages actually allow the explicit type restriction of the domain of integers to the subset N. The absence of a high degree of type discrimination forces the specification of a less rigorous type signature defined on the set of integers I:

$$factorial(n : I) : I$$

Thus the responsibility for complete type checking rests partially with the programmer. Additional type restrictions must often be explicitly programmed, usually by implementing a predicate that asserts additional conditions by checking the argument values bound to formal parameters at run-time. The mechanism used to communicate a run-time type violation to the caller of a procedure is often less than satisfying. A typical recourse is to return some constant from the co-domain that is interpreted as an error value (e.g., -1). If a language supports an exception mechanism, an exception might be generated.

2.2 Generalizing Functions Using Classes

The generalization of a function using a class construct is one way to define a representation for a mathematical function that makes explicit the environment in which a procedure representing the function may execute. An advantage of programming in many applicative-style languages is that procedures are represented by first-class constructions, such as the lambda form in Scheme. In traditional statically typed imperative-style programming languages, including object-oriented languages, procedures are not treated as first-class type abstractions for functions; instead, procedures are treated as second-class, weakly-typed abstractions.

A traditional procedure abstraction, unlike other type abstractions, is rarely treated syntactically or semantically as an explicit type and therefore cannot be manipulated as such within a program. For example, it is typically the case that the only operations

defined on a procedure are one for taking the address of a procedure and one for calling the procedure. Both operations are primitive and immutable in the sense that the programmer has no explicit control over the semantics of either operation. Traditionally, the run-time system generated by a compiler for a imperative language implicitly defines the semantics of these two operations and generates the necessary instructions to carry them out. For a sequential language environment, such actions on the part of the compiler are warranted since procedure invocations are nested and are therefore inherently sequential. Assumptions made by a compiler about sequential execution due to nested control flow restricts the programmer's ability to adequately structure control flow for concurrent execution.

The *class* in an object-oriented language is a syntactic construct used to define a protected closure. The closure represents a lexical environment governed by a set of scope rules that restrict access to a fixed set of typed operations. The closure consists of a set of variable bindings that are accessible only by the set of operations defined in the class declaration.[5] A class declaration therefore enables the specification of arbitrary abstract data types. The class environment provides the encapsulation for the set of variables defining an abstract data structure and exports a set of type signatures denoting operations defined on the encapsulated data. A distinguishing feature of a class is that an inheritance mechanism allows a class to be incrementally extended. A significant feature of the class extension mechanism is the ability to associate new meanings (implementations) to class methods by *overloading*, a form of *ad hoc* polymorphism [29].

A function is commonly represented using a parameterized procedure abstraction, whose environment is by default defined as a block. An alternative is to specify the environment of a function using a class, provided that all the properties of procedure abstraction (e.g., parameterization, local scope, etc.,) can be represented. There are three distinct advantages to a class-based approach to function abstraction:

- representing a function by a class abstraction allows the instances, or ***function objects***, to be treated like other first-class types in the language; i.e, passed as a parameter, returned as a value, placed in a data structure, and most importantly, heap rather than stack allocated.

[5] Under some circumstances, the scope rules may be relaxed to allow access by operations not explicitly defined in the class declaration; e.g., the `friend` declaration in C++.

- the class permits the specification of user-defined operations on the function object, such as pre-conditions and post-condition predicates.
- the class inheritance mechanism permits specialization.

A class-based generalization for procedures enhances the expressive power of an object-oriented language by permitting definition of higher-order operations on procedures in the same manner as operations defined on other types in the language. The enhanced expressive power is representative of applicative-style interpretive languages, such as Scheme and the Common Lisp Object System (CLOS) [100], while preserving strong typing and static binding properties. An advantage over applicative-style languages is that the cost of procedure instantiation and invocation is minimal since a class-based function abstraction is amenable to common compiler optimizations, such as the inlining of methods.

2.2.1 Functions as Objects

To demonstrate a simple example of defining a function as a class instead of a traditional procedure abstraction, an iterative solution to the factorial function is defined using the syntax of C++. The purpose of this example is to convey a few key ideas. Once these ideas are established, the examples become more sophisticated.

In Figure 2.1, a `Factorial` class is defined as a generalization for the factorial function. Four operations are defined as inline class methods, and one operation is defined as a global inline procedure:

`lambda` — a method that implements an iterative algorithm for computing the factorial function. The name `lambda` is chosen for this method since it corresponds to a function expression, like the *lambda form* in Scheme corresponds to λ-abstraction.

`Factorial` — a *class constructor* method that establishes the closure required by the `lambda` method.

`operator()` — an overloaded "function call" operator representing an *apply* operation.[6]

[6] In the subsequent discussion, the keyword `apply` is used as a synonym for the C++ method call operation, denoted by `operator()`.

```
class Factorial {
protected:

   int n, result;

   inline void lambda() {          /* iterative factorial */
         result = 1;
         while (n > 1) {
            result = result * n;
            n = n - 1;
         }
   }

public:

   inline Factorial(int i) { n = i; }          /* closure constructor */

   inline Factorial& operator()() {          /* "apply" operator */
         assert(n >= 0);                     /* precondition predicate */
         lambda();                           /* evaluate n! */
         return self;                        /* self = *this */
   }

   inline operator int() { return result; }          /* type conversion operator */
};

inline int fact(int n) { Factorial f(n); return f(); }          /* "applicator" */
```

Figure 2.1: Iterative Factorial Class.

`operator int` — a *type converter* operator that "converts" an object of type `Factorial` to its integer result value.

`fact` — an *applicator* procedure that constructs a `Factorial` function object and "applies" it by invoking the `apply` operator for that object.

The set of methods collectively define the computation of the factorial function. The advantage of using a class to represent a function is that it becomes possible to separate, and explicitly define, the otherwise implicit actions that a compiler would perform in setting up and executing a procedure call. The inlining of the class methods eliminates overhead associated with the definition of these actions as separate methods.

An instance of the `Factorial` class is an object containing the two integer variables names `n` and `result`, required in the body of the `lambda` method. Since `n` and `result` are within the scope of the `lambda` method, they are an implicit part of the closure created by the `Factorial` constructor. The variable `n` represents the argument value to the factorial function, while `result` is an accumulator. In a traditional procedure abstraction, `n` would be bound to an explicit stack location and `result` would be bound to a stack location, or register. The class allows the treatment of arguments and the result as explicit instance variables, which may be bound to either stack allocated or heap allocated storage locations. The advantage of heap allocation is that the nested dependency inherent in the block structured procedure abstraction is eliminated, and so the environment of a function object may outlive the context in which is was created.

The `Factorial` class publicly exports only the class constructor method `Factorial`, the `apply` operator, and the integer type conversion operator. The `lambda` method is in a protected scope. The `Factorial` constructor performs the process of completing the closure for a particular instance. The closure for a class-based function consists of three separate bindings:

- global variable bindings,
- instance variable bindings, and
- argument value bindings.

Global variable bindings are determined statically at compile time. Instance variable bindings are resolved when an object is instantiated. The `Factorial` constructor is defined such

```
int x;
{
   Factorial f;                    /* instantiate new function object */

   f.n = 5;                        /* substitute argument value */          5
   f.operator()();                 /* "apply" f.lambda() to f.n */
   x = f.operator int();           /* convert f to its integer result */
}
```

Figure 2.2: Factorial Applicator Partial Inline Expansion.

that instance variable and argument value bindings are completed simultaneously since construction corresponds to argument substitution. The class instance variables represent the argument vector for the factorial function, in addition to a slot for the result value that is eventually computed.

Like application in the λ-calculus, the purpose of the applicator `fact` is to effect the substitution of an argument value into the environment of a `lambda` method representing the factorial function and then *evaluate* the function by invoking the **apply** operator. For the `Factorial` class depicted in Figure 2.1, these actions are performed separately by the different class methods. The constructor creates the closure and substitutes the value bound to its argument `i` for the value associated with `n`. An invocation of the inline `fact` applicator as in the following statement:

```
int x = fact(5);
```

is semantically equivalent to the partial inline expansion depicted in Figure 2.2. The **apply** operator causes the following actions:

1. verification of a precondition predicate defined on the value bound to `n`.
2. evaluation of the `lambda` method representing the algorithm for computing the factorial function.
3. return of *self* to the caller.[7]

[7]The notion of "self" in C++ is denoted by the keyword **this**, which is a pointer to an object. Dereferencing the **this** pointer produces the object in question, which will be denoted in the discussion by the identifier `self`.

The `apply` method serves the sole purpose of applying the factorial function represented by the `lambda` method to its argument. In this respect, the `apply` method is similar to traditional procedure call since it performs tasks that are usually implicit operations performed by the compiler run-time.

The `assert` statement implements the precondition predicate guaranteeing that the integer argument value bound to `n` at the time of closure formation is in the domain of non-negative integers. At this point, the exact semantics of the `assert` statement are not important; basically, the evaluation process aborts if the precondition fails.

The body of the `lambda` method implements an iterative algorithm for computing $n!$, using `result` as the accumulator. This method takes no arguments and returns no result. The side-effects are limited to the enclosing lexical environment, which consists of bindings for `n` and `result`.

The type converter operation denoted by `operator int` effects the return of the result of computing the `lambda` method. The result type of the `apply` operator is a reference to the `Factorial` object on which the `apply` operator is invoked. The value denoted by `self` is returned to the point of invocation of the `apply` method. The return of the value denoted by `self` as the result of the `fact` applicator causes implicit integer type conversion using `operator int` since the object denoted by `self` can "convert" itself to an integer value. The conversion results in a binding of the value of `result` as the result value of the `fact` applicator. Note that resuming at the caller's continuation with the value of `self` places the responsibility of obtaining the value of `result` with the client of a `Factorial` object. This technique is in contrast to traditional procedure return semantics, in which the result value of an operation is directly available at the continuation point that is the target of a return statement. The benefits associated with separating the return of a result value from an invocation will become obvious in later sections addressing asynchronous method invocation, where return semantics must be defined in a more relaxed manner.

For cases like the `Factorial` class, an optimizing compiler can perform inline substitution of the entire invocation of the `fact` applicator, resulting in code that is as efficient as a traditional procedure call, and perhaps better, since no procedure call setup overhead is incurred. The complete inline expansion of the `fact` applicator is semantically equivalent to the code sequence depicted in Figure 2.3.

```
int x;
{
   Factorial f;

   f.n = n;                                                    5
   if (f.n < 0)
      abort();

   f.result = 1;
   while (f.n > 1) {                                           10
      f.result = f.result * f.n;
      f.n = f.n - 1;
   }

   x = f.result;                                               15
}
```

Figure 2.3: Factorial Applicator Complete Inline Expansion.

The amount of non-control related stack space required for each invocation of `fact` is equal to the amount of space required to represent the two integer values denoted by `n` and `result`. In practice, both values are likely candidates for register allocation. Similarly, the value denoted by `self` is often allocated in a register. Generally, the inline expansion of the `lambda` method is undesirable since it may result in unnecessary code size growth if many different invocation sites exist in a program. If the `lambda` method is not defined inline, then call setup overhead is required to execute the `lambda` method; however, the associated overhead is no more than a traditional procedure call.[8]

The `Factorial` class can also be written using the recursive algorithm for computing the factorial function. The recursive implementation is given in Figure 2.4. The key observation is the specification of the local `Factorial` instance `f` in the body of the `lambda` method. Recursion is implemented by explicitly instantiating another `Factorial` instance and then invoking its `apply` operation.

This example highlights an important implication of the representation of a function as an object: a programmer is able to explicitly control the manner in which resources are used. In Figure 2.4, the `Factorial` instance `f` in the body of the `lambda` method is allocated on

[8] In C++, one stack slot is required to hold the `this` pointer passed implicitly as the first argument to all non-static class methods. If the `this` pointer is passed in a register, then no non-control stack space is required.

```
class Factorial {
protected:
   ...

   inline void lambda() {                                          5
         if (n > 1) {
            Factorial f(n - 1);        /* recursive instance */
            result = n * f();
         }
         else                                                      10
            result = 1;
   }

public:
   ...                                                             15
};
```

Figure 2.4: Recursive Factorial Class.

the stack since it is locally declared, but it need not be stack allocated. A `Factorial` object could be heap allocated, or allocated out of a specially managed pool of function objects. In a sequential factorial algorithm, there is no real advantage in heap allocation because so little stack space is required. However, one can easily conceive of cases in which it makes more sense to allocate function objects from non-stack space (e.g., a garbage collected heap space).

2.2.2 Specialization of Function Objects

An observation to make concerning the previous class abstraction of the factorial function is that the iterative and recursive classes have essentially the same overall syntactic structure. The only difference between the two is in the algorithm associated with the `lambda` method. To enhance the basic factorial abstraction, class inheritance is used to factor out the common structure of the two implementations. The factorization process presented here is a precursor to a more general factorization of the structure of function objects.

Figure 2.5 depicts an *abstract* `Factorial` class from which is derived both an `IterFact` class and a `RecrFact` class.[9] The common structural elements from the respective factorial

[9] An "abstract" class in C++ is a class in which one or more methods are declared *pure virtual*, meaning

```
class Factorial {
protected:
   ...
   virtual void lambda() =NULL;          /* "pure" virtual method */
                                                                          5
public:
   ...
};

class IterFact : public Factorial {      /* iterative solution */          10
   void lambda() {...}
};

class RecrFact : public Factorial {      /* recursive solution */
   void lambda() {...}                                                    15
};
```

Figure 2.5: A Specialized Factorial.

abstractions include the instance variables and the `apply` method. The `lambda` function in the `Factorial` class is defined as a pure virtual method, hence its implementation is deferred to a derived class. The `virtual` attribute attached to a type signature is a signal to the compiler that the method may be specialized by a derived class. In this case, the binding of an implementation to the `lambda` method is deferred to a subclass of the `Factorial` class. All that remains for a subclass to do is provide an implementation for `lambda`. The `IterFact` and `RecrFact` abstractions provide the iterative and recursive algorithms for their respective `lambda` methods.

An important concept demonstrated by this example is that specialization of a function using the basic class inheritance mechanism is warranted because of the structural similarities between functions. The reason for factoring the structural similarities is that the common structure can be reused by different implementations. Class inheritance is usually associated with reuse and specialization of more traditional abstract data types; the use of inheritance between classes representing single functions is unique since the "data type" in this case is a function type.

---

that there is no implementation associated with the type signature of the method. A signature defined with `=NULL` denotes a pure virtual method.

2.3 A Generalized Function Type

In the preceding section, the `Factorial` class illustrated the basic technique for representing functions as objects. The class-based abstraction for the factorial function was shown to be as expressive and efficient as traditional procedure abstraction in that both iterative and recursive algorithms can be expressed, and method inlining preserves efficiency. The factorial example is effective in demonstrating the basic expressive power that comes from representing functions as objects, but we can observe from the example that there are common structural aspects that can and should be factored.

In this section, the representation of a function as a class-based type is made even more expressive. The enhanced expressiveness arises as a result of introducing a more general abstraction for functions that defines additional operations over function objects. The new abstraction arises from a factorization of the common structure that is apparent when representing functions as objects; whereas λ-abstraction in the λ-calculus is the process of representing a particular mathematical function as a λ-expression. The uniqueness of the new abstraction is that it generalizes the structural aspects of functions. Class inheritance and polymorphism are combined so that concrete objects are realized as type specializations of a polymorphic type representing functions.

Structurally, procedure invocation may be viewed as a composite operation that consists of the following primitive operators:

construction — an operator for binding typed argument values into the environment of a function.

application — an operator for initiating the evaluation of a function.

resolution — an operator that converts a function object into a value from the co-domain of the function.

The type signatures of the higher-order functions on function objects, denoted by f, corresponding to these operators are as follows:

- $cons(f : T_f, a_0 : T_0, a_1 : T_1, ..., a_n : T_n) : T_f$
- $apply(f : T_f) : T_f$

- $result(f : T_f) : T_r$

A function object is denoted by the formal parameter f. In general, the value denoted by f is implicit and need not be explicitly specified in the type signature since it is determined by context.[10] The type T_f represents the type of a function object, while the type T_r, represents the result type. The *cons* operation binds the argument values $a_0, a_1, ..., a_n$ to the environment of the function object f and returns the function object with the new variable bindings as its result. The *apply* operation evaluates the function object f. The evaluation of a function object results in the function object itself as the result. The *result* function transforms the function object f to a value of type T_r, which represents the result of evaluating the function object represented by f in the environment created by *cons*.

The *cons* function corresponds to the class constructor for a function object. In general, a constructor method is specific to each class and cannot be factored. The *apply* function corresponds to the `apply` operator, which can be generalized over all function objects. The *result* function can also be generalized given that function objects are defined as instances of a polymorphic type parameterized by the type of the result value.

2.3.1 The Polymorphic Lambda Type

A *polymorphic lambda type*, represented by the `Lambda<T>` class definition in Figure 2.6, is a type abstraction for function objects. The `Lambda<T>` class arises from a generalization of the structural aspects of procedure abstraction and procedure invocation semantics. Instances of the `Lambda<T>` class are called *lambda objects* to denote the correspondence to λ-abstraction and to emphasize that functions are represented as objects. The key aspect of the `Lambda<T>` class template is that it is parameterized by a type parameter `T` as part of its definition. In general, every function has a result value of some type. The result type is represented in the `Lambda<T>` class by the instance variable `result`, whose type is defined by the generic type parameter `T`. In addition to a result value, every `Lambda<T>` instance has an associated algorithmic procedure, denoted by the method identifier `lambda`. The pure virtual type signature defines a method that must be given an implementation by a derived class. Because of the existence of a pure virtual function, each instance of the class

[10] In C++, the `this` pointer corresponds to the function object denoted by f.

```
template<class T> class Lambda {
protected:

   T     result;              /* generic result binding */

   virtual void lambda() =NULL;          /* "pure virtual" lambda */

   virtual bool precond() { return TRUE; }        /* default precond predicate */

   inline void eval() {              /* evaluator method */
         if (!precond())
            abort();
         lambda();
   }

public:

   inline Lambda& operator()()   { eval(); return self; }
   inline operator T()           { return result; }

#define resultis(x)              do { result = (x); return; } while (FALSE)
};
```

Figure 2.6: The Polymorphic Lambda Type.

template `Lambda<T>` is an "abstract" class.

The `Lambda<T>` class defines a set of methods that are required to implement operations on general function objects. The `precond` method represents a virtual precondition predicate, defined by default to evaluate to `TRUE`. A specific function abstraction that inherits from `Lambda<T>` will typically redefine this predicate to operate over local variables. The pure virtual `lambda` method has no associated implementation and is defined by a derived class to implement a particular function. The `apply` method, denoted by `operator()`, causes the evaluation of the `lambda` method by invoking the inline `eval` method. The `eval` method is a method for effecting evaluation. Evaluation in this context consists of checking a precondition predicate and evaluating the `lambda` method if the predicate evaluates to `TRUE`. In the default case, the predicate always evaluates to `TRUE`. Finally, the generic type converter, `operator T`, is used to effect the type conversion of a lambda object to the value of type `T` denoted by the `result` variable. A particular implementation bound to the `lambda` signature is responsible for binding a value to `result` at some time during

```
class Factorial : public Lambda<int> {
protected:
   int n;
   bool precond() { return n >= 0; }        /* redefine precond */
   void lambda() {...}                      /* iterative or recursive algorithm */
public:
   inline Factorial(int i) { n = i; }       /* construct closure */
};
inline int fact(int n) { Factorial f(n); return f(); }          /* applicator */
```

Figure 2.7: A Factorial derived from Lambda.

its execution.[11] The `result` value is obtained by invoking the parametric type converter procedure `operator T()`.

As an example, the `Factorial` class example is re-defined in Figure 2.7 as a specialization of the `Lambda<int>` class. The `Factorial` class inherits from the `Lambda<int>` class, since the result type of the factorial function is an integer type. Parameterizing the `Lambda<T>` template with an `int` type causes the instantiation of both a `result` variable of type `int` and an `operator int` type converter. As before, a specific instance of the factorial function may be computed by invoking the applicator `fact`. The applicator constructs a unique `Factorial` lambda object and invokes the `apply` operator, inherited from the `Lambda<T>` class.

The generalization offered by the `Lambda<T>` class permits the factoring of the result type information only; it is not generally possible to factor the argument type information used in the constructor because of the `Lambda<T>` class has no information concerning the number and types of the arguments required by any derived class. Hence, the constructor is specific to each derived class and not subject to factorization; however, it is possible to omit a constructor that requires no arguments.

[11] To facilitate binding a result value, the `resultis` macro is used to effect a `return(`*expression*`)` statement.

2.3.2 The Meta Lambda Type

In practice, it is useful to have an additional level of abstraction in the lambda type inheritance hierarchy so that lambda objects may be manipulated uniformly. A *meta* type is introduced as the super type of all `Lambda<T>` instances. Defining higher-order operations as part of a type abstraction for functions enables the flexible implementation of advanced control operations; for example, asynchronous procedure invocation, remote procedure call, and synchronization control. Additional operations that are defined on lambda objects include: a *guard* predicate [56], a *postcondition* predicate, and *before* and *after* operations [100].

The `MetaLambda` class is defined as an abstract class that factors out those methods common to all `Lambda<T>` classes, and introduces additional higher-order operators on lambda objects. Figure 2.8 depicts the `MetaLambda` class representing the Meta Lambda type. The class consists of all virtual methods. The default implementation associated with each `MetaLambda` method may be redefined by a derived class to effect non-standard behavior.

`~MetaLambda` — a null *destructor* required for proper subtype destruction.

`guard` — an execution guard on a lambda object.

`precond/postcond` — a predicate that asserts a pre-condition or post-condition on a lambda object.

`preabort/postabort` — an abort method triggered by the failure of a pre-condition or post-condition.

`before/after` — prologue and epilogue operations that assert the default pre-condition and post-condition predicates using the `assert` macro.[12]

`lambda` — the NULL algorithm of the lambda object.

`eval` — the evaluator that executes the sequence `before-lambda-after` to effect the computation of the function represented by the lambda object.

[12]The `assert` macro is defined to allow textual substitution of its argument p into the form p##.

```
class MetaLambda {
protected:

   virtual ~MetaLambda () {}      /* null virtual destructor */

   virtual bool guard ()          { return TRUE;}
   virtual bool precond ()        { return TRUE; }
   virtual bool postcond ()       { return TRUE; }

   virtual void preabort()        { abort(); }
   virtual void postabort()       { abort(); }

#define assert(p)                 do { if (!p##cond()) p##abort(); } while(FALSE)

   virtual void before ()         { assert(pre); }
   virtual void after ()          { assert(post); }
   virtual void lambda ()         =NULL;
   virtual void eval ()           { before(); lambda(); after(); }
};
```

Figure 2.8: The Meta Lambda Type Abstraction.

The `guard`, `precond`, and `postcond` predicates return `TRUE` by default. The `preabort` and `postabort` methods define `abort` as the default action to be taken if a pre-condition or post-condition fails. The `assert` macro is provided simply to facilitate the use of the `precond` and `postcond` methods. The `before/after` procedures are similar in purpose to the *:before* and *:after* methods in the Common Lisp Object System [100].[13] The `eval` method is used by the `apply` operator defined in the `Lambda<T>` class for effecting evaluation of the `lambda` method. In conjunction with the ability to overload all methods defined in the `MetaLambda` class, this set of "meta methods" enables complete specialization of the execution of a lambda object.

Note that if the default method implementations are not redefined, then inlining and optimization by the compiler can reduce the `eval` method to a single procedure call on the `lambda` method. Furthermore, the inlining of the `eval` method into the `apply` operation of, say the `Factorial` example in Figure 2.7, results in only the `lambda` method being called. In the case that the `apply` operation itself is also inlined, then the factorization

[13] Stroustrup, commenting on the history of C++, notes that an early version of the language included special class level `call` and `return` methods for effecting prologue/epilogue behavior, but these ideas were dropped because he was unable to convince others of their utility [162, p. 277].

```
template<class T> class Lambda : public MetaLambda {
protected:

   T     result;              /* generic result binding */

public:

   inline Lambda& operator()()   { eval(); return self; }
   inline operator T()           { return result; }

#define resultis(x)              do { result = (x); return; } while (FALSE)
};

class Lambda<void> : public MetaLambda {
public:

   inline Lambda& operator()()   { eval(); return self; }
};
```

Figure 2.9: Redefined Lambda Type Abstraction.

of a normal procedure call into primitive components is reduced at run-time to at most a single procedure call on the `lambda` method, and no procedure call if the `lambda` method can also be inlined. Hence, the cost in terms of efficiency for the enhanced expressiveness is negligible.

The definition of the `Lambda<T>` class is redefined in Figure 2.9 as a subclass of the `MetaLambda` class with all methods factored out, with the exception of the `apply` operator and the type conversion operator. Since both operators are dependent on the type parameter `T`, they cannot be factored. Two significant observations about the `MetaLambda` class are to be made:

- the process of evaluation is subject to redefinition, thereby providing a high degree of flexibility at minimal cost in terms of efficiency.
- encapsulating the evaluation process in the `eval` method enables uniform execution of all `lambda` methods since an underlying execution mechanism (e.g., a system threads package) need only consider one type of object—the `MetaLambda` type.

A special case arises with the `Lambda<T>` type when `T` is the `void` type ($T_\perp$), since no result value is required and type conversion to `void` is effectively meaningless in the absence of a

result value. To handle the special case, the `Lambda<void>` class is also defined in Figure 2.9.

A principal result of creating a multi-level abstraction for representing functions is that the supertype-subtype relationship presents a simplified two-level execution model. Execution of a simple operation, namely `eval`, results in the execution of any number of specific operations that effect the diverse computations of an application. Non-trivial consequences are that uniform execution provides opportunities for optimization and simplifies the debugging process since the execution model hinges on a unified conceptual idea—executing a `before-lambda-after` sequence for a given `Lambda<T>` instance. The `before-lambda-after` sequence defined here is similar in concept to the execution model of meta-circular interpreters used in sequential functional languages, such as the one used by Scheme [60]. An important difference is that concurrent execution is supported since the execution of the `eval` method may be scheduled as a separate thread of control.

Threads typically have no specific knowledge about the procedures they are to execute, for example the number and types of arguments, or the type of the return value. To enable a procedure to execute as a separate thread of control, the location of the procedure and the associated argument values must be provided to the thread creation mechanism. The thread mechanism must then establish a calling context for the procedure that will be executed at some later point, and also allow for the return of a result value. In the case of a C++ threads mechanism, it is also necessary to provide "self" (the implicit `this` pointer) as a reference to the object on which a method is being invoked. The significant advantage of the polymorphic lambda type approach is that all lambda objects are executed uniformly by executing the `eval` method defined in the `MetaLambda` class. Hence, initiating a lambda object as a separate thread of control simply requires providing a typed pointer to a `MetaLambda` object and the location of the `eval` method. There are no additional arguments since any arguments have already been bound in the environment of the lambda object as part of closure construction. The method dispatching mechanism generated by the compiler guarantees that the correct methods are executed in the event that they have been redefined by a derived class. Since the closure for the `lambda` method is pre-established as part of object construction, the correct environment is accessed. In effect, a threads mechanism is reduced to a simple virtual machine that repeatedly executes `eval` methods, which in turn effect the execution of the specific implementations attached to a particular `lambda` method.

The handling of a result value is already accounted for by the enclosing environment of a `lambda` method. The result value is obtained by invoking the type converter operation on the lambda object. The benefits of this approach are significant since the demands on the underlying threads mechanism are minimal—it need only execute an `eval` method with the following (implicit) type signature:

```
void eval(MetaLambda* self);
```

In addition to simplifying the use of a threads mechanism, the implementation of various user-defined scheduling schemes is potentially simplified since the scheduling mechanism need only concern itself with scheduling `MetaLambda` objects.[14]

2.3.3 Concurrent Lambda Objects

In this section, a further elaboration of the lambda object concept is provided as evidence of its ability to exceed the expressiveness of traditional procedure abstraction. The `Lambda<T>` and `MetaLambda` classes provide the basic mechanisms required to treat functions as objects. However, an even greater utility can be obtained by using these same abstractions to implement *concurrent* lambda objects. Computation in a concurrent object system is carried out by cooperating objects that at any one time are executing one or more of their methods concurrently with the method executions of other objects in the system. A complement to the `MetaLambda` abstraction is to define a class containing the operations necessary to enable concurrent execution of an arbitrary lambda object. In Figure 2.10, the `LambdaThread` class is introduced as a specialization of the `Lambda` class.

The underlying threads mechanism is system dependent, but the `LambdaThread` class offers a system independent interface. The primary purpose of this abstraction is to allow an individual `lambda` method, such as the one defined by the `Factorial` class, to be treated as a concurrent operation. The `LambdaThread` class defines the well-known multi-programming primitives introduced by Dennis and Van Horn [51]:

[14] A fundamental limitation in any concurrent language environment is the degree of multi-threading that can be achieved. For example, in the traditional UNIX process model, the scheduler, the input/output model, and the process signaling model all conspire to limit the use of truly independent threads. The new multi-threaded kernels (e.g., Mach and Solaris) offer significant advantages over traditional UNIX since the kernel views threads as schedulable lightweight processes.

```
class LambdaThread {
private:
   ...

public:                                                                    5

   static void fork(MetaLambda* lambda); /* fork "eval" method */
   static void join(MetaLambda* lambda);
   static void kill(MetaLambda* lambda);
   ...                                                                     10
};
```

Figure 2.10: The Lambda Thread Class.

`fork` — fork the `eval` method of a lambda object.

`join` — join with a lambda object thread.

`kill` — terminate a running lambda object thread.

The exact semantics of the primitives are not germane at this point; essentially, the fork primitive schedules the `eval` method for execution as a thread, the `join` primitive facilitates waiting on the completion of the execution of a lambda object, and the `kill` method allows the caller to terminate an ongoing lambda object computation.

As an example of a concurrent lambda object, a concurrent factorial is presented in Figure 2.11. Note that the `Factorial` class and the `fact` applicator are unchanged. The only change is a new the inline applicator `cfact`, which is a "forking" applicator that instantiates a heap allocated `Factorial` object instead of a stack allocated object, forks a lambda thread, and returns a reference to the newly created lambda object to the caller. The caller may then later `join` with the lambda object and then implicitly or explicitly invoke the `operator int` type converter to obtain the result. Alternatively, the `operator int` method inherited from the `Lambda<int>` class by the `Factorial` class could be redefined to implicitly join before returning the value denoted by `result`, as follows:

```
operator int() { LambdaThread::join(this); return result; }
```

The following statements suffice to initiate a concurrent factorial and later join with it to obtain the result:

```
class Factorial : public Lambda<int> {
protected:

    int n;

    bool precond() { return n >= 0; }

    void lambda() {...}

public:

    inline Factorial(int i) { n = i; }
};

inline int fact(int n) { Factorial f(n); return f(); }          /* sequential applicator */

inline Factorial& cfact(int n) {                    /* concurrent applicator */
    Factorial *f = new Factorial(n);
    LambdaThread::fork(f);
    return *f;
}
```

Figure 2.11: A Concurrent Factorial Lambda Object.

```
Factorial& f = cfact(100);        /* initiate concurrent 100!  */
  :
int x = f;        /* join and obtain result */
```

Overloading the type conversion operator to provide a join-style rendezvous semantics is interesting, but is problematic when more than one client thread wishes to join with a lambda object and obtain the result. The problem is that without automatic garbage collection support, it is impossible to know when to deallocate the heap allocated lambda object. In the absence of garbage collection, the most opportune time to deallocate a lambda object is when the type conversion operator is applied to obtain a result; however, this strategy precludes multiple joins. The problem of obtaining the "future" result of a concurrently executing lambda object is the topic of Chapter 4, so further discussion is postponed.

```
template<class T> class RemoteLambda : public Lambda<T> {
protected:

    Opr         opr;
    Args        args;
    Result      res;

    void before()       { ...; encode(); ...; }
    void after()        { ...; decode(); ...; }
    void lambda()       { ...; rpc(opr, args, res); ...; }      /* effect RPC */

    virtual int encode() =NULL;
    virtual int decode() =NULL;
};
```

Figure 2.12: The Remote Lambda Class.

2.3.4 Other Lambda-derived Abstractions

The existence of the `before` and `after` methods facilitate the expression of additional prologue and epilogue computations as part of the default `eval` implementation of the `MetaLambda` class. Two enhancements beyond asserting pre-conditions or post-conditions are the expression of a remote procedure call abstraction and a mutual exclusion abstraction for a lambda object representing a critical section.

The principal observation to be made is the flexible way in which the basic operation abstractions embodied in the `MetaLambda` class may be specialized to effect a wide range of execution behavior while maintaining a consistent and common interaction paradigm derived from the structure of the basic `before-lambda-after` execution model associated with lambda objects.

2.3.4.1 The Remote Lambda Abstraction

The `MetaLambda` and `Lambda<T>` abstractions facilitate the expression of remote procedure call semantics where the lambda object acts as a "proxy" for an operation residing on a remote node. The actual details of remote procedure call (RPC) are not discussed here since the concept is widely known and the implementation details vary depending on the operating system [19].

The `RemoteLambda<T>` class is introduced in Figure 2.12. The key feature is that the

before/after methods inherited from the MetaLambda class are redefined to perform the marshaling and un-marshaling of the arguments and result of an RPC. A derived class is required to provide an implementation for the pure virtual encode/decode methods to effect any translation from the language or machine dependent data representation to an appropriate independent data representation suitable for use in a distributed heterogeneous environment. The decode operation invoked by the after method has the responsibility of providing the decoded RPC result value, if any, and binding this value to the result instance variable of the Lambda<T> class so that it may be obtained by the invoking client when the type converter is applied.

The lambda method effects the actual remote procedure call, which is system dependent. The eval operation inherited from the MetaLambda class requires no change since all operations specific to remote invocation are abstracted in the lambda, before, and after methods that are executed by default.

2.3.4.2 The Mutex Lambda Abstraction

Overloading the before and after methods is also ideally suited to representing procedures requiring mutual exclusion, which can be implemented using a semaphore whose use is defined in terms of prologue/epilogue actions. An critical section can be implemented as a lambda method, while the prologue and epilogue semaphore actions can be expressed using the before and after methods inherited from the MetaLambda class. The MutexLambda<T> class depicted in Figure 2.13 guarantees that the execution of a lambda method is performed with mutual exclusion in force.

The implicit constructor for a MutexLambda object instantiates a Semaphore object that provides atomic P and V operations, corresponding to Dijkstra's P and V semaphore operations. The exact details of the Semaphore class are system dependent; but conceptually, binary semaphore semantics are effected. The before and after methods are the natural location for the P and V operations since guaranteeing mutual exclusion in many instances should appear as an implicit operation and a guarantee must exist that the V operation on the semaphore is always invoked on exit from a critical section.

```
template<class T> class MutexLambda : public Lambda<T> {
protected:

    Semaphore     sem;      /* protected semaphore */

protected:

    /* ensure mutual exclusion */

    void before() { assert(pre); sem.P(); }
    void after()  { sem.V(); assert(post); }
};
```

Figure 2.13: The Mutex Lambda Class.

2.4 Summary

To summarize, treating functions as objects permits several key extensions not realizable with ordinary procedure abstraction:

- the variable bindings making up the execution environment are part of the closure formed when an object is instantiated. Since local variables are represented as class instance variables, a function object may be heap allocated.
- since a function is represented as an object, the semantics of procedure call are specialized by selectively redefining the default method implementations. A key concept is that the binding of arguments, evaluating a function, and obtaining a result value are all separated into distinct operations.
- type conversion plays a central role in treating functions as objects. A lambda object may compute its result value concurrently with the execution of the caller and render that result value on demand in response to the invocation of a type conversion operation. Hence, type conversion operators permit the implementation of alternate continuation points for binding result values.
- all of the above may be achieved within the type system defined by the language and with relatively high efficiency using an optimizing compiler that supports efficient inlining.

2.5 Future Work

The use of a data abstraction to form a closure for a concurrent operation is the basis for threads in Modula–3 [133]. The approach shown here is capable of representing both sequential and concurrent procedures with the same degree of conceptual economy. In Modula–3, a `closure` type is defined that has a predefined `apply` operation that must be redefined with an operation that will eventually be forked as a separate thread, much like the `eval` method defined by the `MetaLambda` class. A restriction on the Modula–3 *apply* operation is that it may only take one argument. The lambda object approach to representing functions is believed to be more general than the Modula–3 approach; however, an in-depth comparison is warranted.

Coplien [46] defines the concept of *functors* in C++, which are described as function objects. The functor idea is not polymorphic and is not as extensively developed as the polymorphic lambda type presented here; furthermore, the concurrent case is not addressed. Similarly, the Ellie language [8] for fine-grained parallel object-oriented programming defines all types, including functions, as objects. Ellie is specifically designed for concurrent programming. The inference to be drawn is that the idea of representing a function as an object is a result of the combination of ideas from functional and object-oriented programming. A clear semantic model is required in order to fully understand this integration in a concurrent setting. Milner has initiated this research area by giving a formal model of "functions as processes" using the π-calculus, which can emulate the λ-calculus [127]. This is an exciting area of research and the work presented here can serve as a practical basis for validating the developing theory.

The issue of exception handling in the context of lambda objects has not been deeply investigated. The exception handling mechanism for C++ is still in the experimental stage. Based on the proposed mechanism [58], there does not appear to be any difficulty in supporting exceptions with sequential lambda objects. Exceptions in the context of concurrent lambda objects is another matter altogether which requires a general investigation.

Representing a function as an object that provides a first-class closure offers the potential for implementing persistent procedures. The first step in achieving persistence is to have access to the execution environment of the procedure. However, the static nature of C++ presents several pragmatically difficult obstacles. Clearly, the functions as objects approach

is a step in the right direction; however, C++ in its current incarnation is almost certainly the wrong language environment for achieving a satisfactory solution for making procedures persistent. In addition, explicit operating system support for persistence is required to realize a highly efficient implementation.

Chapter 3

Inheriting Synchronization Constraints

This chapter begins by presenting a new technique for structuring traditional synchronization control by demonstrating how to implement monitors using class inheritance. The purpose of this demonstration is to illustrate an application of the *principle of separation*. The second half of the chapter then introduces a new synchronization mechanism, called a *behavior set*, for specifying synchronization constraints in a strongly typed concurrent object-oriented language with message passing concurrency semantics. The behavior set abstraction offers a solution to structuring synchronization control that coexists with an inheritance mechanism, thereby avoiding the *inheritance anomaly*. The inheritance anomaly is the name given to a general conflict that arises when inheriting synchronization constraints. The fundamental cause of the inheritance anomaly is explicated using the formalism of Milner's Calculus of Communicating Systems. The solution of the inheritance anomaly is important since reuse of class specifications incorporating synchronization constraints is severely restricted by traditional synchronization approaches.

3.1 Introduction

The problem of finding an adequately expressive programming abstraction for structuring synchronization control first arose in computer science in the context of structuring multi-programmed operating systems, such as Dijkstra's "THE" system [54]. Virtually all imperative-style language mechanisms for expressing synchronization control can trace their specific programming abstraction directly to either semaphores (Dijkstra [52]), conditional critical regions (Brinch Hansen [22] and Hoare [75]), monitors (Dijkstra [55] and Hoare [76]), path expressions (Campbell [28] and Andler [9]), or serializers (Hewitt [72]).

The advantages and disadvantages of each control abstraction are familiar to designers of concurrent programs, and the diverse language mechanisms for realizing synchroniza-

tion control among competing processes, or threads of control, have been evaluated and contrasted since their introduction [11, 21, 117, 179]. However, the fact remains that synchronization control mechanisms continues to be of interest in concurrent programming, and a generalized, expressive, reusable mechanism is difficult to construct.

A revival of interest in programming abstractions for synchronization control comes not from the operating systems community, but from the concurrent object-oriented programming language community. Objects are natural units of concurrency because of encapsulation; therefore, concurrent object-oriented languages are interesting because of their potential for facilitating the expression of solutions to problems in parallel and distributed computing. A fresh look at synchronization control is required in concurrent object-oriented languages because of the potential to reuse code through structural inheritance. In order to facilitate an understanding of synchronization control in the context of inheritance, and develop an appreciation for the behavior set solution, it is useful to review briefly the rôle of synchronization control as it has been traditionally realized.

The results presented in this chapter come about as a result of applying the *principle of separation*. The application of this principle to the inheritance-synchronization problem leads one to first reconsider the techniques used to implement traditional condition synchronization. From a pedagogical perspective, the monitor, having been directly influenced by the class construct of Simula–67, is the synchronization abstraction best suited to an initial consideration of synchronization control in a concurrent object-oriented language. In approaching synchronization control from this familiar perspective, the basic conflict that arises when inheriting synchronization constraints is more easily characterized. Interestingly, the inheritance-synchronization conflict is a manifestation of the same underlying structural conflict that leads to the nested monitor problem [66, 96, 119, 139, 180].

3.2 Traditional Synchronization Control

The goal of this section is to illuminate the problem in structuring traditional synchronization control, using the classical monitor construct for examples because of its familiarity. The term "classical monitor" means a modular language construct guaranteeing mutual exclusion and providing synchronization via *condition* variables. According to Brinch Hansen [24], the monitor construct originated with Dijkstra's concept of a *secretary* [55], inherited

```
monitor arbiter;

   ... /* local data */

   var c : Condition;                                          5

   procedure acquire;
         do predicate → c.wait; od
         ...
   end                                                         10

   procedure release;
         ...
         c.signal;
   end                                                         15

   ... /* initialization */
end
```

Figure 3.1: Basic Monitor Structure.

its syntactic structure from the class construct of Simula–67 (augmented with private data), and received condition variables from Hoare [76].

A monitor is a *passive* object—it does not possess an independent thread of control; rather, monitor procedures are executed using the thread of control of a client process. Consequently, interaction with monitors is synchronous and the monitor construct is best suited to problems that can be formulated in terms of a shared memory process model; hence, a monitor may be used to protect and arbitrate concurrent access to any type of shared data resource. Protection is achieved by encapsulating local data that represents synchronization control information and possibly information about the resource. Arbitration is achieved by *implicit* mutual exclusion and *explicit* condition synchronization via user-defined condition variables. The structure of a simple arbitration monitor is shown in Figure 3.1. Although the monitor abstraction may be used to represent a variety of different types of resources and access policies, the general structure is as depicted.[1]

The structure of the condition synchronization portion of a monitor is of primary interest here. Monitor procedures are commonly defined in terms of complementary `acquire` and

[1]The syntax and semantics of the classical monitor examples shown in this chapter are adapted from Andrews [10]. The do–od construct represents an iteration block.

`release` operations. The typical structure of an `acquire` operation is to test a predicate defined in terms of the local state information, and if the predicate is false, relinquish mutual exclusion in the monitor by waiting on a condition variable. Similarly, the `release` operation awakens a delayed process so that it may attempt to re-acquire mutual exclusion in the monitor. Assuming *signal-and-continue* semantics on condition variables [78], a signaling process exits a monitor prior to any awakened process re-entering the monitor. While it is not always the case that the `signal` primitive is executed as the last statement of a `release` action, commonly used monitors appear to have this "signal-at-end" structure.

In general, a monitor is a programming abstraction that encapsulates the representation of either:

type 1: a combined control and data abstraction, or

type 2: a pure control abstraction.

A type 1 monitor is also an abstract data type for the resource that it controls. A type 2 monitor encapsulates only synchronization control information for an external resource represented by some other data abstraction. The classical `BoundedBuffer` monitor, depicted in Figure 3.2, combines both the data used to represent synchronization control and the data used to represent a bounded buffer; therefore, it is a type 1 monitor. The operations of the `BoundedBuffer` monitor combine statements for condition synchronization as well as statements for modifying the state of the representation for a bounded buffer.

The classical `ReadersWriters` monitor, depicted in Figure 3.3, is an example of a monitor that is a pure control abstraction. A monitor of this type encapsulates only condition synchronization data and does not also represent an abstract data type of the resource for which synchronization control is provided.

Generally, type 1 monitors are prone to deadlock as the result of nested monitor calls. The deadlock situation arises precisely because of the failure to cleanly *separate* the synchronization control actions from the actions of the resource, and the fact that mutual exclusion is in effect while a thread of control has not returned from a monitor procedure [21]. An important observation concerning the structure of a monitor is that a clean separation of condition synchronization code from the actual monitor operations for both type 1 and type 2 abstractions is critical in structuring overall synchronization control to facilitate reuse.

```
monitor BoundedBuffer;

   var buf[1:n] : T;
   var front, rear, count : Integer;
   var not_full, not_empty : Condition;                                    5

   procedure get (var result : T);
         do count = 0 → not_empty.wait(); od
         result := buf[front];
         front := (front mod n) + 1;                                       10
         count := count − 1;
         not_full.signal();
   end

   procedure put (value : T);                                              15
         do count = n →  not_full.wait(); od
         buf[rear] := value;
         rear := (rear mod n) + 1;
         count := count + 1;
         not_empty.signal();                                               20
   end

   /* initialization */

   front := rear := 1;                                                     25
   count := 0;
end
```

Figure 3.2: Classical Bounded Buffer Monitor.

```
monitor ReaderWriter;

   var readers : Integer;
   var writing : Boolean;
   var oktoread, oktowrite : Condition;                                    5

   procedure request_read();
         do writing → oktoread.wait(); od
         readers := readers + 1;
   end                                                                     10

   procedure release_read();
         do writing → oktoread.wait(); od
         readers := readers − 1;
         if readers = 0 → oktowrite.signal(); fi                         15
   end

   procedure request_write();
         do readers > 0 or writing → oktowrite.wait(); od
         writing := true;                                                20
   end

   procedure release_write();
         writing := false
         oktowrite.signal();                                             25
         oktoread.broadcastl();
   end

   /* initialization */
                                                                         30
   readers := 0;
   writing := false;
end
```

Figure 3.3: Classical Readers-Writers Monitor.

```
class Condition;

    ...     /* condition queue */

    function empty() : Boolean;                                          5
        ...
    end

    procedure wait();               /* FIFO queueing */
        ...                                                              10
    end

    procedure signal();             /* signal-and-continue semantics */
        ...
    end                                                                  15

    procedure broadcast();          /* signal all waiters */
        ...
    end
end                                                                      20
```

Figure 3.4: A Condition Variable Class.

3.2.1 Condition Synchronization

Condition synchronization refers to a predicate-based mechanism whose evaluation determines whether or not a process may execute a shared set of statements and/or expressions. The key to effecting condition synchronization is the predicate that determines whether a client process is granted mutual exclusion. Ideally, a condition synchronization mechanism permits programmer control over the definition of the predicate and the actions to be taken depending on the truth value of the predicate. For example, whether a client thread that enters a monitor procedure may continue, or be forced to relinquish mutual exclusion. Note that in order to determine whether a client process can acquire exclusive access to a resource, its thread of control must first enter a monitor procedure, and remain blocked until access is granted.[2]

In the previous examples, and in all subsequent examples, condition variables represent instances of the `Condition` class, defined in Figure 3.4. For discussion purposes, the condition `wait` primitive provides FIFO queuing and the condition `signal` primitive provides

[2]Some implementations permit conditional entry, which allows a client to first determine whether entry into the monitor would cause the client to become suspended.

signal-and-continue semantics. The advantages and disadvantages of alternative condition signaling schemes are well-known and adequately discussed by Andrews [10] and Howard [78]. Briefly, signal-and-continue semantics allow the signaling process to continue until exiting the monitor before allowing an awakened process to resume in the monitor.[3] Signal-and-continue semantics are distinct from the *signal-and-exit* semantics originally proposed by Hoare in his seminal paper [76]. Signal-and-exit semantics cause the signaling process to immediately relinquish control to the signaled process, implying that the entry predicate need not be re-checked since the signaled process is guaranteed exclusive access following a signal. The type of signaling semantics chosen is in part driven by the structure of the condition synchronization code found on entry to request procedures. With signal-and-continue, the condition that was true when a signal is issued may no longer be true when a signaled process re-enters the monitor procedure on which it is blocked. Hence, it is incumbent upon each request procedure to structure the predicate that determines condition synchronization within a loop so that a resumed process always re-asserts the predicate before proceeding.

The use of the `broadcast` primitive is applicable in the case of the readers-writers problem since the semantics for signaling reader processes is different for writer processes. A broadcast signal has the same semantic effect as iteratively signaling each process blocked on a condition. Depending on the organization of the condition queue, broadcast signals can be made more efficient than iterative calls to the `signal` primitive.

3.2.2 Separating and Inheriting Synchronization Control

This section focuses on the structural aspects of synchronization control in monitors from the perspective of specialization and reuse. A common feature of a large class of monitors is the mostly independent structure of the condition synchronization code with respect to the rest of a monitor procedure. The only part of synchronization that is dependent on the local data of a monitor is the predicate that is tested as part of a request action. The central idea in this section is that the code for expressing condition synchronization should be separated from the body of a monitor procedure. The need to distinguish and separate the elements of condition synchronization from monitor procedures has been recognized

[3] A `signal` primitive under signal-and-continue semantics is equivalent to the `notify` primitive employed in Mesa [108].

```
monitor ProducerConsumer

    consumer, producer : Condition;

    pure function consumable() : Boolean;                                    5

    pure function producible() : Boolean;

    procedure consumer_ingress();
        do not consumable → consumer.wait(); od                              10
    end

    procedure consumer_egress;
        producer.signal();
    end                                                                      15

    procedure producer_ingress();
        do not producible → producer.wait(); od
    end
                                                                             20
    procedure producer_egress();
        consumer.signal();
    end

end                                                                          25
```

Figure 3.5: Abstract Producer-Consumer Structure.

by others; notably, Bloom [21] and Kessels [101]. The approach presented here is similar in concept, but distinct in technique since the underlying premise is that synchronization control code should be structured in a manner that facilitates reuse.

A number of synchronization problems are classified in terms of a generalized producer-consumer relationship, where access to a resource by competing producer and consumer processes is arbitrated by a monitor. Both the bounded buffer and readers-writers problems may be viewed as specializations of a more general producer-consumer relationship. The similarity of the `BoundedBuffer` and `ReaderWriter` monitors derives from the common synchronization structure exhibited by both abstractions, despite the fact that the `BoundedBuffer` monitor encapsulates its data resource while the `ReadersWriters` monitor is pure synchronization. In the case of readers-writers, consuming is a non-destructive operation on some resource while in the case of the bounded buffer, consuming is a destructive operation on an resource.

Assuming an object-oriented perspective, it is logical to consider factoring the common arbitration structure into a higher-level abstraction. Figure 3.5 depicts an abstract `ProducerConsumer` monitor representing a factorization of the synchronization code common to both the `BoundedBuffer` and `ReaderWriter` monitors. This abstract monitor contains:

- two condition synchronization variables: `consumer` and `producer` of type `Condition`.
- two functions, `consumable` and `producible`, that are declared *pure*. A function declared pure has an undefined implementation—an implementation must be defined by each derived monitor.
- two `ingress/egress` procedures representing the synchronization operations that effect the acquisition and release of mutual exclusion in a derived monitor.

This general `ProducerConsumer` monitor is used as a generalized base construct that may be specialized through inheritance by other monitors, or classes, that require this type of synchronization control. A derived construct (monitor or class) must provide an implementation for each condition predicate by overloading the appropriate predicate definitions. Instances of `ProducerConsumer` are incomplete in isolation since there is no default implementation associated with the respective `consumable/producible` predicates. The implementation of the respective `ingress` and `egress` procedures may also be overloaded through inheritance. The default implementation provided here effects a fair arbitration policy between a producer process and a consumer process—each `egress` procedure by default signals the alternate condition.

Figure 3.6 depicts a `BoundedBuffer` class as a specialization of the abstract `ProducerConsumer` monitor. The use of a class to represent the bounded buffer data abstraction rather than a monitor, as was done formerly in the classical case, is a result of the fact that synchronization control is now effected by a combination of procedural abstraction and inheritance. The operations of the `BoundedBuffer` class invoke the inherited `ingress/egress` methods to effect synchronization control. The `ingress` methods in turn invoke the derived overloading of the pure condition predicates. Note that the `pure` attribute defines the associated monitor operation to be exempt from mutual exclusion.

```
class BoundedBuffer inherits ProducerConsumer;

   ...              /* buffer representation */

   overload function consumable() : Boolean;                          5
         consumable := count > 0;
   end

   overload function producible() : Boolean;
         producible := count < n;                                     10
   end

   procedure get(var result : T);
         consumer_ingress();
         ...                          /* access buffer */             15
         consumer_egress();
   end

   procedure put(value : T);
         producer_ingress();                                          20
         ...                          /* update buffer */
         producer_egress();
   end

   ...        /* initialization */                                    25
end
```

Figure 3.6: Specialized Bounded Buffer.

The `consumable` and `producible` predicates are each provided with an implementation that expresses the boolean conditions that arbitrate access to the monitor by competing producer and consumer processes. Representing the synchronization constraints as independent procedures allows the inherited base monitor operations to invoke the overloaded implementation provided by a derived class. The use of separate procedures to represent synchronization constraints is essential in permitting specialization of constraints in this manner. A disadvantage of defining condition synchronization predicates as explicit procedures is that data values not defined as part of the monitor abstraction cannot be used unless they are made part of the lexical closure of the predicate through explicit parameterization.

A call to one of the `ingress/egress` procedures effects mutual exclusion, which is then released on return from the procedure. In other words, the class presents to the client a normal abstract data type for a bounded buffer, and the operations of the data type acquire and release mutual exclusion by executing the appropriate `ingress` and `egress` monitor procedures. This technique of combining a monitor and a data abstraction via inheritance allows synchronization control to be inherited, but maintains the necessary separation between mutual exclusion and condition synchronization implemented by the monitor and the data manipulation code implemented by the data abstraction. An added benefit of this structure is that the principal condition leading to the nested monitor problem, namely, failure to release mutual exclusion when making another monitor call, can generally be avoided since mutual exclusion is only in effect during an `ingress` or `egress` operation.

Figure 3.7 depicts a `ReaderWriter` class that also inherits its synchronization operations from the `ProducerConsumer` monitor. The `ReaderWriter` class provides an overloading for the condition predicates and the `producer_egress` procedure to allowing all blocked readers to proceed by issuing a condition queue `broadcast` instead of a `signal`.

The following observations are to be made concerning the structure of these new implementations for classical monitors:

- the explicit synchronization code is removed from the body of the `request/release` procedures and is encapsulated by separately defined, inherited predicates.
- the derived class is obligated to define the synchronization constraints in terms of predicates represented by the condition procedures.

```
class ReaderWriter inherits ProducerConsumer;

    var readers : Integer;
    var writing : Boolean;

    overload function consumable() : Boolean;
        consumable := not writing;
    end

    overload function producible() : Boolean;
        producible := readers = 0 and not writing;
    end

    overload procedure producer_egress();
        producer.signal();
        consumer.broadcast();
    end

    procedure request_read();
        consumer_ingress();
        readers := readers + 1;
    end

    procedure release_read();
        readers := readers - 1;
        if readers = 0 → consumer_egress(); fi
    end

    procedure request_write();
        producer_ingress();
        writing := true;
    end

    procedure release_write();
        writing := false;
        producer_egress();
    end

    /* initialization */

    readers := 0;
    writing := false;
end
```

Figure 3.7: Specialized Readers-Writers Monitor.

```
class Semaphore {
private:

   sem_t          sem;     /* system dependent semaphore type */

public:

   inline Semaphore() { ... }        /* initialize sem */
   virtual ~Semaphore() {}           /* null virtual destructor */

   inline int acquire() { return P(sem); }
   inline int release() { return V(sem); }
};
```

Figure 3.8: A System Dependent Semaphore Type.

3.3 Extensible Synchronization Control Types

In the previous examples, the `monitor` construct is an integral part of the language structure. Considerable syntactic liberties were taken to illustrate extending a generalized monitor using inheritance and ad hoc polymorphism to achieve overloading of procedures. The discussion of the detailed structure of synchronization control types was deferred until now to permit a clear presentation to develop an appreciation for the need to separate synchronization constraints, expressed as predicates, from the body of monitor procedures. The monitor types presented in this section are defined using the syntax of C++, particularly `virtual` member functions, which can be overloaded by derived classes to effect new behavior.

3.3.1 A Polymorphic Mutex Type

The `Semaphore` type presented in Figure 3.8 is an abstraction that implements straightforward mutual exclusion based on an underlying system dependent semaphore mechanism. On construction of a `Semaphore` object, a `sem_t` instance is implicitly created and bound to `sem`. The exact definition for the `sem_t` type varies on a per system basis; hence, the details are omitted. The inline `Semaphore` operations `acquire` and `release` map to corresponding `P` and `V` operations defined on `sem`.

The polymorphic type `MutexScope<T>`, shown in Figure 3.9, is provided to allow the

```
template<class T> class MutexScope {
protected:

    T*          m;      /* mutex type */
    bool        self;   /* self constructed? */

public:

    inline MutexScope()             { self = TRUE;  m = new T(); m->acquire(); }
    inline MutexScope(T* s)         { self = FALSE; m = s; m->release(); }
    virtual ~MutexScope()           { m->release(); if (self) delete m; }
};
```

Figure 3.9: A Polymorphic MutexScope Type.

programmer to write critical sections using block scope rules to automatically acquire and release mutual exclusion through explicit construction and implicit destruction of a mutual exclusion object declared within a block. In C++, the constructor of a local object is automatically called when the scope in which the object is declared is entered. Automatic destruction is performed when the scope is exited. For example, the `MutexScope<Semaphore>` constructor for the object denoted by the variable `mutex` in the following code fragment is automatically called on entering the scope of the block, which attempts to acquire mutual exclusion. Likewise, the destructor is automatically called on exit from the block, thereby releasing the mutual exclusion lock already held.

```
{       /* enter critical section by explicit MutexScope construction */
    MutexScope<Semaphore> mutex;
    :
}       /* exit critical section by implicit MutexScope destruction */
```

Tying the mutual exclusion protocol with the scope rules for implicit construction and destruction of an object guarantees that mutual exclusion is always released by the thread that acquired it, even if the block is exited before reaching the end.

3.3.2 An Abstract Producer-Consumer Monitor Type

A C++ version of the Producer-Consumer monitor is illustrated in Figure 3.10 as a class. The structure of the `PCMonitor` class follows closely the structure of the abstract

```
class PCMonitor : public Semaphore {
protected:

    Condition consumer, producer;

public:

    /* "pure virtual" condition synchronization predicates */

    virtual bool consumable()       =NULL;
    virtual bool producible()       =NULL;

    /* ingress/egress operations */

    virtual void consumer_ingress()     { while (!consumable()) consumer.wait(); }
    virtual void producer_ingress()     { while (!producible()) producer.wait(); }

    virtual void consumer_egress()      { producer.signal(); }
    virtual void producer_egress()      { consumer.signal(); }
};
```

Figure 3.10: A Class-Based Producer-Consumer Monitor.

`ProducerConsumer` class depicted earlier in Figure 3.5. Monitor semantics are realized by inheriting mutual exclusion from the `Semaphore` type and extending basic mutual exclusion by providing the condition queues, predicates, and methods required for condition synchronization. Two local condition queues are defined: the `consumer` variable represents the queue on which consumer threads block and the `producer` variable represents the condition queue on which producer threads block. The `Condition` class implements a structure not unlike that shown previously in Figure 3.4.

The primary components of the `PCMonitor` class are the pure condition predicates and the complementary pairs of `ingress` and `egress` operations for consumers and producers. The `consumable` and `producible` predicates have no associated local implementations, thus a derived class is obligated to provide a specific implementation for each predicate. The `consumer_ingress` operation defines the method used by a consumer thread to gain "consume" access. The determination of whether or not a consumer thread can acquire access is solely dependent upon the truth value of the separately defined `consumable` condition predicate, whose implementation is defined by a derived class. If the predicate evaluates to `FALSE`, then the requesting consumer thread is forced to wait in the `consumer` condi-

tion queue. The consumer_egress operation is used by a consumer thread to indicate that consuming has completed. Execution of the consumer_egress operation results in a signal that allows a blocked producer thread to proceed. Condition signaling obeys signal-and-continue semantics; hence, the condition predicate is defined in a while statement to guarantee that the invariant assertion is not violated by a consumer thread that simultaneously attempts to acquire access.

Similarly, the producer_ingress operation implements the synchronization for producer threads. The producible predicate defines the synchronization condition for allowing a producer thread to enter the monitor. The producer_egress operation is used to exit the monitor and signal all consumer threads waiting on the consumable condition queue. Broadcast signaling of consumer threads can be defined by simply overloading this method in a derived class.

Figure 3.11 depicts two specializations of the MutexScope<T> type: Consumer and Producer condition synchronization classes, where the type parameter T is defined as the PCMonitor type. The Consumer type extends the semantics of mutual exclusion to include condition synchronization using the consumer_ingress and consumer_egress operations. The invocation of the Consumer class constructor results in the automatic construction of its parent MutexScope<PCMonitor> object, which attempts to acquire mutual exclusion as part of construction. Following acquisition of mutual exclusion by the parent constructor, the consumer_ingress operation is invoked on the mon object to perform condition synchronization. Destruction of a Consumer object causes the reverse actions: the consumer_egress operation is invoked on the mon object and the destruction of the inherited MutexScope<PCMonitor> object releases mutual exclusion. A similar process occurs when constructing a Producer type: the producer_ingress and producer_egress operations are invoked on construction and destruction, respectively, to enforce the producer condition synchronization discipline.

Inheriting from the MutexScope<T> class in this manner allows the incremental construction of diverse condition synchronization disciplines and all but eliminates the chance of violating the synchronization protocol. Given the fact that the condition synchronization predicates of the PCMonitor class and the respective ingress/egress operations may be completely redefined, alternative synchronization policies are easily structured by incremen-

```
class Consumer : public MutexScope<PCMonitor> {
public:

   Consumer(PCMonitor* mon) : MutexScope<PCMonitor>(mon) { mon->consumer_ingress(); }

   virtual ~Consumer() { mon->consumer_egress(); }
};

class Producer : public MutexScope<PCMonitor> {
public:

   Producer(PCMonitor* mon) : MutexScope<PCMonitor>(mon) { mon->producer_ingress(); }

   virtual ~Producer() { mon->producer_egress(); }
};
```

Figure 3.11: Producer and Consumer MutexScope Types.

tally extending the basic types to obtain the desired behavior.

3.4 Inheriting Synchronization Constraints using Behavior Sets

The previous sections illustrated the application of the principle of separation to the traditional monitor construct. The identification and separation of condition synchronization predicates enables the specification of general synchronization control that can then be specialized using type inheritance. As mentioned previously, the monitor is *passive* object that is used synchronously by other threads of control. This section is concerned with the inheritance of synchronization constraints by objects that are *active* (possessing their own thread of control), and which are used asynchronously by sending messages.

3.4.1 The Inheritance Anomaly

In a concurrent object-oriented language, one would like to be able to inherit methods and their synchronization constraints without compromising the flexibility of either the inheritance mechanism or the synchronization mechanism. A problem called the *inheritance anomaly* [122] arises when synchronization control is implemented within a class and an attempt is made to specialize behavior through inheritance.

In standard object-oriented models all the methods declared in a class definition are

always available for execution by a client regardless of the internal state of an object of that class. When the methods of an object are subject to concurrent execution, it is incumbent upon the implementor of those methods to provide the appropriate synchronization control mechanisms so that instance variables are kept consistent by guaranteeing synchronization constraints through a combination of mutual exclusion conditions and execution ordering conditions.

The class implementor typically provides within the implementation of each method the code necessary to determine whether an object is in a state in which execution of the requested method is appropriate. To avoid unnecessary errors, the client of an object must either known a priori the legal execution orders, or verify that the object is in a state that allows a method to be executed; for example, by invoking a predicate provided by the just such a purpose.

A non-standard object-oriented model can be defined in which the collection of methods in a class definition are partitioned into subsets with respect to the state of an object. Depending on the object state, only a subset of the methods declared in the class definition are available for execution. In a non-concurrent object-oriented language, objects with such semantics may or may not be useful since there is only one thread of control. In a concurrent object-oriented language, mechanisms based on such semantics provide a natural and elegant means for expressing synchronization. The ability to enable and disable the execution of a particular subset of methods in an object's interface is similar to the effect achieved by the common monitor abstraction in which the execution of an operation for a data resource is potentially delayed, subject to the granting of access rights by a monitor.

The concurrent behavior of an object is captured in part by the method definitions and in part by the mechanism used to guarantee condition synchronization constraints. The anomaly arises because the inheritance mechanism and the synchronization mechanism interfere with one another, limiting the ability of the subclass to reuse the method implementations of the superclass. The interference arises when the synchronization constraints of a derived class differ sufficiently from the assumptions made by the parent class to effectively invalidate some or all of the parent class methods, thereby inhibiting the degree of reuse obtainable via inheritance. Furthermore, the anomaly has been observed across a spectrum of concurrent object-oriented languages regardless of the kind of synchronization

mechanism employed [6, 25, 99, 134].

3.4.2 Defining Concurrent Object Behavior

In this section, the notion of concurrent object behavior is formalized using Milner's Calculus of Communicating Systems (*CCS*) [126]. Using *CCS behavior equations* to specify and reason about the observable behavior of concurrent objects, the nature of the conflict in inheriting synchronization constraints is explicated. Using behavior equations as a formal representation of concurrent object behavior results in the definition of *behavior sets* and a *behavior function*. A behavior set represents a subset of the methods defined as part of an object's interface. The behavior function is a mapping from the domain of object states to the domain of behavior sets. A behavior set is used to represent the synchronization constraints of actor-like concurrent objects and may be extended via inheritance. By expressing relevant object states, behavior sets, and the behavior function as first-class, inheritable, and mutable entities in a concurrent object-oriented language, the conditions are defined that must be met in order to inherit synchronization constraints free of known anomalies.

In speaking of the behavior of an object, the principle issue is the set of messages an object will accept at some point in time, or alternatively, the set of methods that are visible in the interface of the object upon receipt of a message. From this perspective, the behavior of an object is its *observable behavior* since what is relevant is how the object appears to those clients that use the object. This notion of observable behavior is motivated by the similar notion described in [126]. A difference in this work is that the machinery of *CCS* is used in specifying and reasoning about the observable behavior of individual objects, not complete systems of objects.

In dealing with concurrent objects, the relationship between the state of an object as captured by its instance variables and the subset of methods that define its observable behavior is critical. This relationship is precisely what defines the behavior of a concurrent object. In order to understand concurrent object behavior, this relationship must be investigated.

3.4.2.1 Specifying Behavior

The behavior of an object may be defined by a set of *behavior equations* that capture the states of an object and the subset of methods visible when the object is in a particular state. The *state* of an object is defined to be the set of values assigned to the instance variables of the object, sometimes called a *configuration*, at any point in time. For example, the behavior of an object that maintains some prescribed linear order over a collection of items, and whose size is bounded, may be succinctly defined by the following equations:[4]

$$
\begin{aligned}
A_0 &\stackrel{\text{def}}{=} \texttt{in}(x_1).A_1 \\
A_i &\stackrel{\text{def}}{=} \texttt{in}(x_{i+1}).A_{i+1} + \overline{\texttt{out}}(x_1).A_{i-1} \qquad (1 \leq i < n) \\
A_n &\stackrel{\text{def}}{=} \overline{\texttt{out}}(x_1).A_{n-1}
\end{aligned}
$$

This set of equations defines all behaviors of a bounded linear order and is similar to an example given by Milner [126]. In each equation, the name on the left-hand side denotes an agent whose behavior is defined by the right-hand side. The notation A_k, $0 \leq k \leq n$, is used here as an abbreviation for the term $A(x_0, x_1, \ldots, x_k)$, which introduces $x_0, x_1, \ldots, x_k$ as bound variables with scope restricted to the right-hand side of the defining equation. As a special case, the variable x_0 denotes the null element; hence, A_0 describes the behavior of an empty linear order.

In the behavior definition of the A_i agent, the *summation combinator* $(+)$ conveys that agent A_i offers both the in and $\overline{\texttt{out}}$ operations simultaneously to a client wishing to interact. If the in operation is chosen, the *prefix combinator* (.) requires that an agent accept an input value denoted by the bound variable x_{i+1} and then behave according to the equation defined by A_{i+1}. Similarly, if the $\overline{\texttt{out}}$ operation is chosen, the agent outputs a value denoted by x_1 and then assumes the behavior defined by agent A_{i-1}. In general, agent A_i becomes agent A_{i+1} following an in operation and agent A_{i-1} following an $\overline{\texttt{out}}$ operation, with the behavior of agents A_0 and A_n being special cases. From this perspective, the behavior equations define the operations offered by an agent as well as a replacement behavior. The notion of replacement behavior is a fundamental aspect of the Actor model [3], thus it is reasonable to use behavior equations for specifying and reasoning about the behavior of

[4] In *CCS*, input and output ports are distinguished using the overbar label notation; hence, the label $\overline{\texttt{out}}$ denotes an output port and the label in denotes an input port.

individual actor-like objects.

Although a generic bounded linear order is specified, the above set of behavior equations is isomorphic to a set of equations representing a bounded buffer accepting put and get operations, a stack accepting push and pop operations, or a queue accepting enqueue and dequeue operations. The isomorphism is realized through an application of the *CCS* relabeling operator (/) to yield the desired name substitution:

$$\begin{aligned} \text{Buffer} &\equiv A_0[\texttt{put/in}, \texttt{get/out}], \ldots, A_n[\texttt{put/in}, \texttt{get/out}] \\ \text{Stack} &\equiv A_0[\texttt{push/in}, \texttt{pop/out}], \ldots, A_n[\texttt{push/in}, \texttt{pop/out}] \\ \text{Queue} &\equiv A_0[\texttt{enqueue/in}, \texttt{dequeue/out}], \ldots, A_n[\texttt{enqueue/in}, \texttt{dequeue/out}] \end{aligned}$$

The isomorphism is achievable at this level of abstraction since there is no concern for the actual semantics of the generic in and $\overline{\texttt{out}}$ operations; e.g., whether the $\overline{\texttt{out}}$ operation returns values according to FIFO or LIFO semantics. To maintain generality, the equations describing a bounded linear order are used with the understanding that the isomorphism may be applied at any time.

3.4.2.2 Object States and Behavior Sets

Behavior equations may be viewed as defining independent agents representing the various value configurations of an object. In this section a model is defined that captures the essential elements used in developing a programming abstraction to represent a collection of behavior equations.

The model associates with each A_i a state σ_i and a set β_i called the *observable behavior set*. For a given behavior equation A_i, the observable behavior set β_i is constructed from the non-restricted prefix operations on the right-hand side of behavior equations. By non-restricted prefix operations, is meant those operation names that do not appear within the scope of the *CCS* restriction operator, thereby removing those operations from the set of observable behaviors. The collection of all states is given by the state set:

$$S = \{\sigma_0, \sigma_1, \ldots, \sigma_n\}.$$

The set of all possible observable behavior sets is the powerset:

$$B = \mathcal{P}(M), \text{ where } M = \bigcup_{i=0}^{n} \beta_i.$$

To complete the model, a function is needed that relates states to behavior sets.

3.4.2.3 Mapping Object States to Behavior Sets

A function $f_\beta : S \rightarrow B$ is defined that maps elements of the state set to elements of the powerset of observable behaviors. It may be argued that in developing an abstract data type of a bounded linear order in the standard model, a programmer implicitly defines a mapping from S to B. More precisely, each σ_i is *always* mapped to a single element in B, namely $\{\texttt{in}, \overline{\texttt{out}}\}$. The function f_β is called the *behavior function* since it defines the observable behavior of an object in any given state. In the standard model f_β is defined as:

$$f_\beta(\sigma_i) = \{\texttt{in}, \overline{\texttt{out}}\} \qquad (0 \leq i \leq n)$$

That is, objects in the standard model alway have the same observable behavior regardless of the object state.

The claim is made that the definition of f_β in the standard model is not natural because $f_\beta(\sigma_0) = \{\texttt{in}, \overline{\texttt{out}}\}$ forces us to write the method implementing the $\overline{\texttt{out}}$ operation in such a way that an underflow condition is detected. A similar situation occurs for $f_\beta(\sigma_n)$. What is needed is a more natural mapping for f_β. A more natural mapping from S to B can be given as follows:

$$\begin{aligned} f_\beta(\sigma_0) &= \{\texttt{in}\} \\ f_\beta(\sigma_i) &= \{\texttt{in}, \overline{\texttt{out}}\} \qquad (1 \leq i < n) \\ f_\beta(\sigma_n) &= \{\overline{\texttt{out}}\} \end{aligned}$$

This new mapping corresponds to our intuition about the observable behavior of a bounded linear order.

Consider implementing an object in an object-oriented language that represents a bounded linear order as a class-based abstraction. The mapping f_β suggests that the abstraction depends on distinguishing three behaviors, which are dependent on the synchronization conditions: empty, not empty, and full. Clearly, these conditions appeal to our intuition about the expected behavior of an abstraction representing a bounded linear order. When the abstraction is implemented, a class that exports `in` and `out` methods will be defined. Both methods must either explicitly or implicitly implement a synchronization mechanism consistent with the behavior equations previously formulated.

Suppose that a new constraint on the behaviors is introduced. For example, the behavior given by the A_1 equation is defined as being different from the $A_2, \ldots, A_{n-1}$ behaviors because a new operation is introduced that augments the behavior sets of agents $A_2, \ldots, A_{n-1}$. In distinguishing the A_1 behavior, a new partitioning must be defined that is different than the one previously formed. it is now necessary to distinguish the conditions empty, full, singleton, and somewhere in between. Specializing the previously implemented abstraction using inheritance implies that it is necessary to redefine the mapping given by f_β. Redefining the mapping means that the domain S, the codomain B, and the mapping of elements in S to elements in B must all be changed. If the linear order type abstraction has been implemented in such a way that these components are implicitly embedded in the methods, then the inheritance anomaly is encountered. That is, because the components of the mapping f_β are implicitly embedded in the implementation, the only way the mapping may be redefined to effect a new synchronization policy is by reimplementing the methods in which the mapping components are embodied. Hence, any benefit of reuse by specialization is defeated. The solution is to separate the components of the mapping f_β from the method implementations and define them as explicit (first-class) constructions.

3.4.3 Inheriting Concurrent Object Behavior

The types of concurrent object-oriented systems of interest are composed of objects with properties similar to those described in the Actor Model [3]. Each actor possesses its own thread of control and communicates with other actors via message passing. Although the Actor model supports fine-grained concurrency, the considerations here are restricted to inter-object concurrency, which is achieved using a become operation. The become operation results in a *replacement behavior* with its own thread of control, which may run concurrently with the behavior that initiated the become operation.[5]

The concept of *behavior abstraction* was previously proposed in ACT++ [99] as a mechanism for capturing the behavior of an object. Upon initial examination, behavior abstraction seems powerful since synchronization can be achieved naturally by dynamically

[5]Chapter 5 provides a more complete presentation of the Actor model and its realization. The key idea is that concurrency is achieved by a combination of asynchronous message passing and creating replacement behaviors using the become operation.

modifying the visibility of the object interface using the become operation. The efficacy of this mechanism and its degree of interaction with the ACT++ inheritance mechanism has been examined by others and been found to have serious limitations [122, 138]. The most serious limitation occurs because a behavior abstraction is not easily extensible. Enabled sets [174] improve on the notion of behavior abstraction by promoting the control of the visibility of an object's interface to a dynamic mechanism which can be manipulated within the language; i.e., enabled sets are first-class entities. The flexibility offered by enabled sets suggests combining behavior abstraction and enabled sets to realize a behavior set. The mechanism that captures the concept of a behavior set has the following properties:

- it is a natural extension of formal methods for specifying concurrent object behavior,
- it is free from known inheritance anomalies,
- it can be expressed entirely within the language, and
- it can be enforced efficiently at run time.

The syntax of ACT++ is used in the following sections to illustrate how to express elements of the object state set S, elements of the observable behavior powerset B, and the behavior function f_β, such that concurrent behavior may be defined and inherited free from known anomalies

3.4.3.1 Expressing Concurrent Behavior

To represent concurrent object behavior within the ACT++ language, three first-class constructs are defined:

1. *state functions* representing some or all of the elements of the state set S,
2. a *next behavior function* representing the function f_β, and
3. *behavior sets* representing elements of the observable behavior powerset B.

In the example shown in Figure 3.12, each construct is expressed in an ACT++ class definition of a bounded linear order.

```
template<class T> class LinearOrd : public Actor {
private:
   ...            /* private instance variables */

protected:        /* instance variables and methods visible to subclasses */

   BehaviorSet B0, Bn, Bi;

   virtual bool empty() {...}
   virtual bool full() {...}

   virtual BehaviorSet nextBehavior() {
      if (empty())
        return B0;
      else if (full())
        return Bn;
      else
        return Bi;
   }

public:    // methods visible to subclasses and clients

   void in(T x) {
     ...
     become nextBehavior();
   }

   T out() {
     ...
     become nextBehavior();
   }

   LinearOrd() {    // construct initial "empty" object
     ...
     B0 = &in;
     Bn = &out;
     Bi = B0 + Bn;
     become nextBehavior();
   }
};
```

Figure 3.12: The Linear Order Class.

In this example, two predicates `empty` and `full` are used to distinguish three states: empty, full, and neither empty nor full. Although not shown, the predicates are computed based on implementation dependent instance variables representing the actual number of elements in the linear order. Both functions are used by the `nextBehavior` method, which maps the current object state to a behavior set represented by an instance of the `BehaviorSet` class. There are three behavior sets defined: B_0, B_n, and B_i. The B_0 and B_n behavior sets correspond to the previously expressed behavior equations A_0 and A_n, respectively. The B_i behavior set is used in this class abstraction to collectively represent the observable behaviors of the intermediate behavior equations $A_1, A_2, \ldots, A_{n-1}$. Each behavior set is initialized in the class constructor. Instances of the `BehaviorSet` class are first-class objects as evidenced by the overloading of the binary addition operator, `operator+` denoting set union when applied to two behavior sets; In the example, B_i is formed as the union of the behavior sets B_0 and B_n.

3.4.3.2 Inheriting Concurrent Behavior

To substantiate the claim that the inheritance anomaly is avoided, a new class called `HybridLinearOrd` is derived from the `LinearOrd` class. The main feature of the `HybridLinearOrd` class is that a new method is introduced that forces a change in the mapping given by f_β. The new method allows a client of an instance of the `HybridLinearOrd` class to atomically read a pair of elements instead of a single element. The method cannot simply invoke the `out` method twice since the `out` method executes a `become` operation after each invocation. Due to the concurrency in the system, another object may have its `out` request executed before the second `out` is processed. The specification of this new object is given by the following behavior equations:

$$
\begin{aligned}
A_0 &\stackrel{\text{def}}{=} \texttt{in}(x_1).A_1 \\
A_1 &\stackrel{\text{def}}{=} \texttt{in}(x_2).A_2 + \overline{\texttt{out}}(x_1).A_0 \\
A_i &\stackrel{\text{def}}{=} \texttt{in}(x_{i+1}).A_{i+1} + \overline{\texttt{out}}(x_1).A_{i-1} + \overline{\texttt{outpair}}(x_1, x_2).A_{i-2} \qquad (2 \leq i < n) \\
A_n &\stackrel{\text{def}}{=} \overline{\texttt{out}}(x_1).A_{n-1} + \overline{\texttt{outpair}}(x_1, x_2).A_{n-2}
\end{aligned}
$$

The behavior equations for a hybrid linear order differ from the equations specifying the behavior of a linear order only in the addition of the choice of an $\overline{\texttt{outpair}}$ operation

in the definitions of the A_2 through A_n behaviors. There are two effects of this refinement: first, the $\overline{\texttt{outpair}}$ operation must be added to the appropriate observable behavior sets and a new powerset B' must be computed; second, since the A_1 behavior is now distinguished and since $B' \supset B$, a new mapping f'_β is required:

$$\begin{aligned} f'_\beta(\sigma_0) &= f_\beta(\sigma_0) \\ f'_\beta(\sigma_1) &= f_\beta(\sigma_1) \\ f'_\beta(\sigma_i) &= f_\beta(\sigma_i) \cup \{\overline{\texttt{outpair}}\} \qquad (2 \leq i \leq n) \end{aligned}$$

Clearly there is cause to reuse the implementations of the `in` and `out` methods defined in the `LinearOrd` class. In addition, since the new mapping f'_β relies on f_β it is also possible to reuse the `nextBehavior` method. However, both the B_i and B_n behavior sets must be redefined to include the method representing the $\overline{\texttt{outpair}}$ operation. A new instance of the `BehaviorSet` class must also be defined to include the methods representing the `in` and $\overline{\texttt{out}}$ operations defined in the A_1 behavior equation.

The definition of the `HybridLinearOrd` class shown in Figure 3.13 inherits from the `LinearOrd` class and introduces the following:

- a new state function `singleton`,
- a new behavior set `One`,
- a redefinition of the behavior function `nextBehavior`, and
- a new method `outpair`.

The `singleton` function corresponds to distinguishing the agent A_1 from the $A_0, A_2, \ldots, A_n$ agents, and the new instance of the `BehaviorSet` class corresponds to the behavior set associated with agent A_1. The B_i and B_n behavior sets are augmented in the class constructor with the `outpair` method corresponding to the enlarged codomain B'. Thus, the inherited `nextBehavior` method can be trivially redefined to correspond to the new mapping f'_β by adding a check for the state corresponding to agent A_1 and invoking the superclass behavior function `LinearOrd::nextBehavior` for all other states.

Inheriting concurrent behavior requires that the superclass methods may be reused *and* the mapping given to f_β by the superclass may be specialized. This means the elements of

```
template<class T> class HybridLinearOrd : public LinearOrd<T> {
protected:

   BehaviorSet B1;

   virtual bool singleton() {...}

   BehaviorSet nextBehavior() {
      if (singleton())
        return B1;
      else
        return LinearOrd::nextBehavior();
   }

public:

   Pair<T> outpair() {
     ...
     become nextBehavior();
   }

   /* Note: base class constructor is executed first */

   HybridLinearOrd() {
      B1 = Bi;
      Bi = Bi + &outpair;
      Bn = Bn + &outpair;
      become nextBehavior();
   }
};
```

Figure 3.13: The Hybrid Linear Order Class.

S, the elements of B, and the function f_β must be representable in the language and the representations must be:

1. first-class,
2. inheritable, and
3. mutable.

The inheritance anomaly occurs in previous formulations of this problem precisely because the behavior sets and the behavior function, as they occurred in the superclass, were neither first-class nor mutable.

State functions representing elements of S, instances of the `BehaviorSet` class representing elements of B, and the `nextBehavior` function representing f_β have these properties. All are first-class language entities inheritable by a subclass. Instances of the `BehaviorSet` class are mutable by a subclass since they are within the scope of a `protected` clause. The `empty` and `full` predicates representing object states and the `nextBehavior` method representing the behavior function are mutable because they have the `virtual` attribute. These methods are also within the scope of a `protected` clause, which hides them from clients of a subclass but allows subclass specialization.

The usefulness of the behavior set mechanism as just presented has been implemented in ACT++ and demonstrated on a multiprocessor [131]. The implementation runs on both a 10 processor Sequent Symmetry and a uniprocessor Sun SPARC workstation, using the PRESTO threads package [16].

3.5 Future Work

This chapter has not addressed proving the correctness of programs that inherit synchronization constraints. The specification of pure virtual synchronization predicates implies that the abstract producer-consumer monitor cannot by itself be proven correct. At best, only a proof schema can be given that must be completed when a derived class provides an implementation for each condition predicate. An advantage however is that the abstract class already defines the structure of the derived monitor, which is sufficient to allow the formulation of a proof template that can then be parameterized by the actual assertion for

a derived class. A consequence is that much of the proof structure can be defined a priori with guidelines for completing the proof in much the same way as one would complete the condition synchronization predicates. For example, a universally quantified assertions for mutual exclusion and liveness can be defined for a generalized producer-consumer relationship. When the generalized condition synchronization structure is refined into a more specific structure, it will be necessary to refine the mutual exclusion and liveness quantifiers to deal with the specific nature of the relationship (e.g., a strong readers preference). Intuitively, it seems that one could inherit proof structure; however, this is currently a hypothesis that requires further investigation. A logical starting point is to co-develop a generalized synchronization inheritance structure for common synchronization problems and the associated proof schema structure, which could then be specialized further for application specific synchronization requirements. The superposition approach used in Unity proofs [30] seems to be relevant to the type of proof structure being suggested.

The behavior set concept is a modest contribution to the more general problem of inheriting synchronization constraints. The use of *CCS* here is descriptive rather than prescriptive. A general solution requires a prescriptive formalism that allows the declarative specification of synchronization constraints and the automatic generation of behavior sets.

Chapter 4

A Polymorphic Future Type

Concurrent object-oriented languages require mechanisms to cope with the potential delay between the time an operation is asynchronously invoked and the time at which a result value is returned. In general, the result value returned to the continuation point of an asynchronous method invocation is a handle to the eventual result value, if any, of a concurrently executing computation. The handle is commonly called a *future* since it represents a value ready at some future time.

In a statically typed object-oriented language, an operation has a corresponding type signature that identifies the type of the result value. The introduction of a future into a statically typed language with traditional procedure call semantics should allow futures to be expressed as first-class, user-defined types. Clearly, a language can be extended with new syntax that permits the expression of futures; however, if a language type system supports polymorphic types and the definition of explicit type conversion operators, then it is reasonable to attempt to define futures as polymorphic types that are expressed within the type system of the language. Furthermore, futures represent the rendezvous site for separate threads of control that must be synchronized. The results of the previous chapter are used to structure the synchronization control for futures.

4.1 Introduction

Coroutines, message send operations, remote procedures, and local parallel procedures are operations that may be asynchronously invoked, resulting in two concurrent execution environments: one for the caller and one for the called operation. All asynchronous operation invocation mechanisms share the fundamental requirement for a complementary mechanism supporting the eventual return of a result value when the asynchronously invoked computation has completed. Traditional call/return semantics rely on an implicit

continuation point to which the result of a procedure call is bound when the called procedure terminates. The nested block structure of Algol-like sequential languages guarantees the existence of the continuation point at the time at which a procedure returns. In the concurrent case, no such assumptions can be made since the continuation point following the invocation point of an asynchronous operation is not necessarily the continuation point to which a result value is to be returned—in some cases, the context of the invocation point may no longer even exist.

To address this problem, a mechanism is required that makes no assumptions about returning the result of a concurrent computation to the *continuation* point of the invocation; but instead, allows for an explicit and flexible rendezvous mechanism. A concurrent computation must be able to create a result value which one or more concurrent computations can obtain independently. Independent interaction necessarily implies that condition synchronization be employed by the result mechanism to guarantee its consistency. In addition, the result mechanism must have a lifetime independent of either producer or consumer computations, thereby implying the need for garbage collection.

Various mechanisms have been proposed within specific language contexts to address this problem. A challenge is to define a type abstraction that is syntactically convenient, expressive, and with minimal execution overhead. The primary goal is to define a mechanism that is useful in a statically typed concurrent object-oriented languages, in particular, C++. The remainder of this chapter defines the concept of a *polymorphic future type* and demonstrates how they are implemented in C++. Subsequent chapters utilize the primitive polymorphic future type for coping with the return value of asynchronously invoked operations.

4.2 Concurrent Method Execution

A principle feature of a concurrent object system is the ability to construct user-defined objects that contain operations possessing an independent thread of control when invoked. The set of instance variables encapsulated by the object represent a shared state among the concurrent operations defined by the object. Figure 4.1 contrasts the difference between synchronous and asynchronous interaction of a client object with a server object.

Part (a) depicts the traditional synchronous, stack-based, request/reply interaction be-

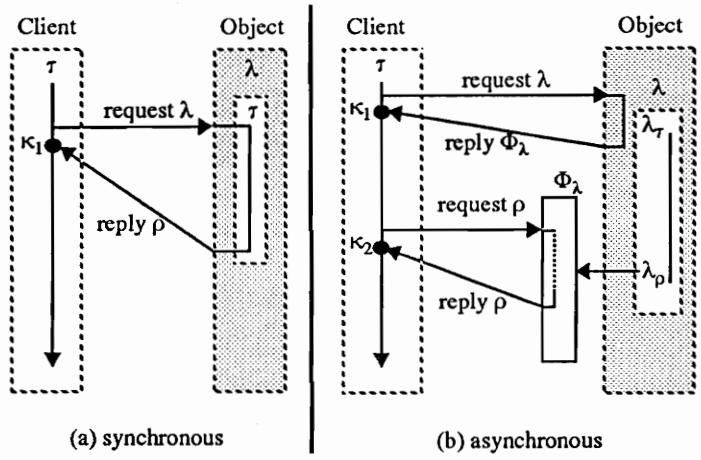

Figure 4.1: Method Execution Model.

tween a client and a method defined as part of an object. The client synchronously invokes the method denoted by λ with the appropriately typed argument values. The request operation establishes an implicit *continuation* point, κ_1, at which control will resume when the method returns. The called method is executed using the *thread*, τ, of the caller. Upon completion of the λ method, control resumes at κ_1 with the result value denoted by ρ. The advantage of this interaction model is that it is simple, efficient, and amenable to common optimization techniques, such as inlining. The disadvantage is that the nested, stack-based model is inherently sequential.

Part (b) depicts an asynchronous interaction between a client and an object method. The invocation establishes an implicit continuation point, κ_1, for the client. The method invocation mechanism uses the client's execution thread, τ, to create a new thread environment, λ_τ, within which the λ method will execute. The invocation mechanism then resumes at the continuation point κ_1 of the client. The client thread τ and the λ_τ thread may then execute concurrently.

The result of an asynchronous invocation is a value denoted by an object Φ_λ. When a client resumes execution at κ_1, it has a capability to manipulate the Φ_λ object as a though it were the result value, ρ, of the λ operation. The Φ_λ object represents a *future*, from which a client can request the future result, λ_ρ, of the execution of the corresponding λ operation. At a later point in time, the original client thread, or any other thread that has access to the

future, may request the result of the λ operation from the Φ_λ object, depicted as a request for ρ. If the result from the concurrently executing λ method has not yet been bound to the future, then the client thread blocks in the Φ_λ object. When the result is computed and the thread τ_λ terminates, the client's continuation point κ_2 is resumed with the result value ρ. In effect, the future is "converted" to a value, either implicitly or explicitly, when the result of the corresponding computation is produced. Subsequently, the Φ_λ object and the thread τ_λ are subject to garbage collection.

The asynchronous interaction model is of interest here. To elaborate on the concepts just introduced, the remainder of this section provides a brief review of continuations, threads, and different language mechanisms for implemented the concept of a future.

4.2.1 Continuations

The concept of a *continuation* was first developed by Strachey and Wadsworth in formulating a denotational semantics that could account for jumps [160, 125]. Informally, a continuation is an abstraction that represents the locus of execution at which computation will resume upon completion of the current computation step. Formally, a continuation is a function of one argument that represents the rest of the computation of a program. The argument to a continuation is the global state of the computation up to the point of the continuation. In denotational semantics, two important classes of continuations are defined: *expression continuations* and *command continuations*. Implicitly, an expression continuation is the control point to which an intermediate value obtained from evaluating an expression is returned. A command continuation is the control point, following a statement, to which is passed the new state resulting from the current command. For example, in evaluating the imperative assignment statement:

```
int x = fact(5);
```

the evaluation of the function represented by the procedure call expression `fact(5)` results in its value being supplied to the expression continuation that represents the right-hand side of the assignment statement. This expression continuation is the function that changes the value bound to the storage location denoted by the integer variable `x`. The command continuation represents the rest of the program immediately following the assignment statement,

which receives as an argument the new state resulting from the assignment. In both cases, the continuation is a *normal continuation* since the control point is returned to under normal (non-error) conditions. An example of a non-normal continuation is the control point in an exception handling block that is executed in the event of an exceptional condition occurring during the evaluation of a procedure.

In most language environments, normal continuations are implicitly created, used, and destroyed by the language run-time environment; they are not available to the programmer for explicit use—one need not be explicitly concerned about either the representation or existence of continuations. Control flow and expression evaluation in imperative-style, statically typed languages are made highly efficient due to the fact that the compiler can make static assumptions about continuation points and optimize the code accordingly. For a large class of sequential programming problems, these assumptions about sequential (nested) control flow are warranted. For programming problems requiring non-sequential control flow (e.g., backtracking, coroutining, and event handling), the implicit assumptions limit the programmer's ability to express solutions that closely represent the desired control flow. To program the required control, the programmer is often forced to adopt low-level programming tactics that subvert some aspect of the language, such as the block structure, resulting in convoluted and potentially non-portable programs.

Few languages provide explicit language support for continuations. Scheme is an exception in that it provides first-class continuations using the *call-with-continuation* primitive (`call/cc`) that permits the implementation of sophisticated control abstractions, such as exceptions and coroutines [68, 69]. Languages that do not directly support explicit continuations may provide primitive library routines that can be used to effect non-local jumps (e.g., `setjmp` and `longjmp` in C). Non-local jumps using these primitives can then be used to construct basic exception handling and coroutine mechanisms. The disadvantage to using library routines is that the implementations of the primitive routines are often machine and/or compiler dependent, thereby limiting the portability of programs using the primitives. Furthermore, the primitives are based on static assumptions about the procedure call mechanism that are often only appropriate within a specific language context.[1]

[1]For example, C++ requires that destructors for local objects be executed when exiting procedure scope. A naïve `longjmp` procedure will not know to execute the destructors, resulting in "memory leaks."

4.2.2 Threads

A *thread* is a control abstraction representing an independent execution environment for a computation. A thread is often defined in terms of an asynchronously invoked procedure that initiates a concurrent computation. The execution environment commonly consists of a collection of variables representing the program counter, general purpose and arithmetic registers, the stack frames representing the lexical closure for the procedure, and the return context. Threads can be used to support coroutine relationships or asynchronous procedure invocations; hence, a thread may be viewed as a primitive construct from which more elaborate control abstractions can be built.

A thread is sometimes described as a *lightweight process* since the execution information represented by a thread is similar to the execution information used by the operating system to represent processes. The distinction between a thread and a process is both operating system and language relative. In subsequent discussion, a process is considered to have a computational granularity equivalent to that of a UNIX process on a uniprocessor architecture. A thread is considered to have the computational granularity of a procedure in an imperative language like C++, Ada, or Modula–3, augmented with the necessary control information to allow preemptive scheduling.

Most procedural languages with concurrency semantics have an explicit mechanism for creating independent threads of control. The mechanism may be characterized in terms of whether or not it is semantically integrated into the language. The integrated language approach permits explicit thread creation using syntactic constructs semantically derived from the primitive `fork/join` multiprogramming constructs of Dennis and Van Horn [51]. For example, the relatively high-level `cobegin–coend` construct, introduced by Dijkstra, was used by Brinch Hansen in Concurrent Pascal [23] and has influenced similar constructs in other languages (e.g., `co–oc` in SR [64]). Other "modern" languages with specific syntax supporting concurrent execution threads are Ada [27] and Modula–3 [133].[2]

An approach distinct from the integrated approach is to augment a sequential language with a basic threads mechanism that is supported by the underlying operating system. A threads package or library is then provided for a particular language environment. Useful

[2]There are many others. PL/I [7] was one of the early high-level languages to employ language support for lightweight tasks. Simula–67's coroutines [20] are an example of a limited threads capability.

features provided by system threads packages include support for red-lined stacks to indicate stack overflow, and support for preemptive scheduling based on user-assignable thread priorities. Examples are the C threads library provided with Mach [45], the Lightweight Process (LWP) library provided with SunOS (Solaris) [163, 164], the POSIX threads library [85, 130], and the PRESTO threads package for use with C++ [14, 16].

In general, threads packages are highly system dependent and programs written using a specific threads package are often not portable across system platforms.[3] The variability and system dependency of thread packages led to the implementation of PRESTO for C++ programming on specific UNIX platforms. The PRESTO package is an experiment in providing an extensible threads package that allows the programmer to specialize the attributes of thread specific classes through class inheritance. PRESTO however is not as flexible as one would hope, since many of the polymorphic type features of C++ cannot be handled by the thread invocation mechanism. The basic problem is that method invocation in C++ is more sophisticated than a traditional procedure call; the result is that the expressive power of C++ is limited when using PRESTO managed threads. PRESTO, like most other threads packages, provides an untyped mechanism for indicating the procedure and argument values to be bound to a thread object. A thread invocation mechanism must obtain the activation record that is part of a normal procedure invocation. The language type system and procedure call mechanisms are necessarily bypassed in the attempt to provide a general asynchronous invocation mechanism that is capable of creating a "call state" as part of the environment for an independent thread of control. The fundamental problem is that the nested block structure and the stack-based execution model of imperative-style procedure invocation conflict with the low-level control required by a thread invocation mechanism. Ultimately, it is the immutable policies enforced by a compiler and the *ad hoc* nature of a non-integrated threads mechanism that hinders an elegant solution.

4.2.3 Future Mechanisms

The structure of a polymorphic future type for concurrent object-oriented programming in C++ is the primary contribution of this chapter. The role of the Φ_λ object introduced by the concurrent method execution model, presented in Section 4.2, is similar to the role

[3]The POSIX threads library is an attempt to overcome this defect.

of a *future* in MultiLisp, a *promise* in Argus, a *Cbox* in Concurrent Smalltalk, and a *future ref* in ES-Kit. The polymorphic future type introduced in this chapter has been directly influenced by this previous work; hence, a brief review of the other mechanisms is warranted.

4.2.3.1 Futures in MultiLisp

Halstead introduced the concept of a future as a control abstraction in MultiLisp, a concurrent Scheme implementation [67]. Futures are similar in concept to the `delay` and `force` constructs that together support lazy evaluation in sequential Scheme [40]. A `delay` expression evaluates to a lambda expression that can then be used as an argument to the `force` procedure to cause the evaluation of the delayed expression; for example:

```
(define foo (delay (factorial 50))
  ⋮
(...  (force foo) ...)      ;; evaluate (lambda() (factorial 50))
```

The expression `(factorial 50)` is not evaluated, but instead made into an implicit `lambda` that is maintained in the environment and eventually evaluated by `force`.

A `future` expression in MultiLisp is used in a manner similar to `delay`:

```
(define foo (future (factorial 50))
```

The result of a `future` expression is a value representing the parallel evaluation of the expression `(factorial 50)`, rather than a lambda expression that when forced, will evaluate to 50!. Thus, futures in MultiLisp provide the opportunity for expressing a high degree of parallelism in computing MultiLisp expressions. For example, construction of a *conscell* data structure using the `cons` function may be expressed as:

```
(define conscell (cons (future E1) (future E2)))
```

Since each `future` expression generates a concurrent computation, returning an immediate future value as the result, the `cons` expression immediately constructs a two element conscell containing two future values. When the respective evaluations of expressions E_1 and E_2 are complete, the futures in the conscell denoted by `foo` are implicitly replaced with the values of the concurrently computed expressions. Should a future value be referenced in another expression before the computation represented by the future has completed; e.g,

`(car foo)`, the evaluation of that expression blocks until the future is replaced with its value by the corresponding parallel computation.

In MultiLisp, once a computation is started for a future expression it is not possible to force the computation to abort. The computation will proceed to completion, even though the eventual result might not be needed. A more powerful control abstraction would permit the treatment of a future as an abstract data type allowing the specification of operations (e.g., `cancel`) on futures themselves and not the values they denote.

4.2.3.2 Promises in Argus

Futures in MultiLisp motivated Liskov to introduce the concept of a *promise* in conjunction with *call-streams* in Argus [118]. Argus is a language and a run-time system for programming distributed applications [113, 114]. The language is a direct descendant of CLU [115], both in its syntax and semantics. Unlike the dynamically typed MultiLisp, Argus is statically typed; hence, promises are strongly typed. A call-stream is an abstraction that combines the semantics of remote procedure call and message sending by allowing non-blocking procedure invocations. As with a future in MultiLisp, a promise is a structure representing the future result value of a concurrent computation. Unlike a future, a promise is able to maintain information about exceptions that might have been raised by the concurrent computation.

Syntactically, a promise is declared in a type expression that is consistent with a corresponding procedure type signature:

```
factorial:proc (n:int) returns(int) signals(overflow)
P = promise returns(int) signals(overflow)
```

A key feature of a promise is that it is consistent with the abstract data type concepts that CLU and its derivatives are founded upon—a promise is an instance of an abstract data type that is parameterized with type information related to the type of the result value encapsulated by the promise. The specification of a promise indicates the type of the result value of the corresponding procedure and any exception types. To claim a result from a promise, the `claim` operation is invoked on an instance of a promise type:

```
p:P := fork factorial(50)
```

```
  ⋮
x:int := P$claim(p)
```

If the result of the concurrent computation represented by `factorial` has not been produced at the time of the call to `claim`, the calling process blocks. If an exception is raised, the exception handling code for type `overflow` is automatically invoked.

4.2.3.3 Cboxes in Concurrent Smalltalk

Concurrent Smalltalk, a concurrent extension of sequential Smalltalk, employs a *Cbox* class to facilitate asynchronous method invocation [181]. Method invocation in Concurrent Smalltalk is expressed using a messaging paradigm where messages denoting methods defined in a class definition are sent to instances of the class to effect computation. An asynchronous invocation is distinguished from a synchronous invocation by the special "`&`" symbol appended to the name of a message sent to an object. For example, the following code fragment asynchronously sends the `factorial` message to the number object represented by the symbol `50`. The expression returns a Cbox object that is bound to the locally declared, dynamically typed `future` identifier.

```
|future|
  ⋮
future ← 50 factorial&.
```

Similar to previous future mechanisms, a Cbox object is the recipient of the eventual result of a concurrent computation. A result value is obtained from a Cbox by sending the `receive` message to a Cbox object. The `receive` method blocks if a result value has not yet been computed and bound to the Cbox object:

```
result ← future receive.
```

An elegant aspect of the first-class nature of Cboxes is that they can be used to implement and/or-synchronization. Cboxes may be placed into collections and a thread may then wait for all or some of the Cboxes in the collection to have result values pending using various and/or-synchronization messages (e.g., `receiveAnd:` and `receiveOr:`) sent to a Cbox instance representing the collection.

4.2.3.4 Future References in ES-Kit

An interesting application of the concept of futures in an imperative setting is the *future ref* used in the Experimental Systems Kit (ES-Kit), a distributed kernel and run-time system for distributed and parallel programming in C++ [110].[4]

A future ref in an ES-Kit application is an instance of a system defined `FutureRef` class; hence, a future ref object is a first-class type. A `FutureRef` instance is returned as the result of an asynchronous method invocation using an overloading of a non-standard *method call* operator, denoted by the symbol `->()`. In C++, operator overloading is a form of *ad hoc* polymorphism that allows a different implementation to be associated with certain operations when applied to instances of a class. The method call operator, when applied to an instance of a special system class, effects an asynchronous procedure call that immediately returns a `FutureRef` instance as a result. The type signature for this operation is as follows:

```
FutureRef operator->()(int id, int len, ...);
```

The arguments are a method identifier, the length of the argument vector, and an arbitrary argument vector.[5] Interestingly, the semantics given to this operator in ES-Kit are not consistent with the semantics defined by the C++ language [58]. The method call operator has no equivalent in standard C++. The ES-Kit method call operator is a combination of the standard C++ unary *member access* operator "->" and the binary *function call* operator "()". The access operator is most often used to implement "smart pointers" [161]; whereas, the function call operator is often used to implement an iterator as part of an *iteration abstraction* [116]. To obtain the binary method call operator, the ES-Kit environment requires a modified compiler; hence, the language is technically a semantic extension of C++.

If an object is defined as an instance of a user-defined class that is derived from the special system class defining the overloaded operator, then all method invocations in the derived class will return future refs as result values. In this regard, a future ref is similar

[4]The GNU C++ compiler [171] was begun by Tiemann at MCC as part of the Experimental Systems Project in an effort to make C++ usable for distributed and parallel programming.

[5]The method identifier corresponds to an entry in the method dispatch table for the object to which the operator is applied (i.e., the `vtable`).

to a MultiLisp future or an Argus promise in that the future type represents a handle for a concurrent computation. In C++, the invocation of a method of some object pointed to by `obj` and returning a result value of type `T` is specified as:

```
T result = obj->method(a1, a2, ..., an);
```

Under normal C++ procedure call semantics, a stack context is established for the method, with the usual pushing of argument values (including the value of `obj` as an environment binding context and return address.[6]) The call to `factorial` is then made and, in this case, an integer value is returned (probably through a register) and assigned to the storage location denoted by `result`.

A future ref, in conjunction with the overloading of `operator->()`, provides an asynchronous procedure call semantics. An invocation is specified as:

```
FutureRef future = obj->factorial(50);
  :
int result = future;        /* implicit type conversion */
```

The call to `factorial` initiates an asynchronous invocation and immediately returns a `FutureRef` instance that is bound to `future`. At some later point in the computation, the future ref is implicitly type converted to an `int` by assignment to `result`.

A limitation with future refs is that they are not general polymorphic types. Type conversion is limited to the set of primitive language types `char`, `int`, `float`, etc., and pointers to these types. It is straightforward to define the small set of type conversion operators required to handle the primitive language types. Arbitrary user-defined types however are not handled as elegantly as one would like—the untyped `void*` pointer is used to allow arbitrary user-defined types, with the burden placed on the programmer to explicitly perform the appropriate type conversion at run-time.

Unlike futures in MultiLisp or promises in Argus, a future ref in ES-Kit is first-class. Using an overloaded equality relation defined as an operation on the type, it is possible to determine whether or not two futures refs, and not the values they denote, are equivalent.

[6]The environment binding context is the first argument on the stack and is represented by the keyword `this`.

The comparison of two futures does not result in a blocked computation should the value of either future not yet be computed. For example, the following conditional tests whether or not the two future ref instances refer to the same concurrent computation, and not whether the values they denote are equivalent.

```
if (future1 == future2)
  ...
```

If it is really the values that are to be compared, then explicit type conversion can be used to force the futures to be evaluated. In the following example, the evaluation of two future refs to the integers they represent is forced by an explicit type conversion. The equality relation is then applied to two integer values when the computations denoted by the futures are both completed.

```
if ((int) future1 == (int) future2)
  ...
```

In contrast, the MultiLisp expression `(eq? future1 future2)` will block if the computations denoted by `future1` and `future2` have not both produced a value by the time the equality expression is evaluated. Since MultiLisp is dynamically typed, there is no equivalent to explicit type conversion; instead, a special equality relation and symbol must be defined to compare whether or not two futures refer to the same computation.

4.2.3.5 Summary

Table 4.1 summarizes the type characteristics of each of the future mechanisms just described. The characteristics of interest are: type resolution, type conversion, first-class properties, polymorphism. Type resolution is distinguished by whether or not the type of a value is determined at run-time (dynamic) or compile-time (static). Type conversion is categorized as either implicit, explicit, or both. A future is consider a first-class type if it is possible to specify user-defined operations as part of its representation. A future is polymorphic if it possible to define arbitrary future types. In general, dynamic type resolution implies polymorphism.

The following section introduces a polymorphic future whose type is determined at compile-time, that allows both explicit and implicit type conversion, is first-class, and is polymorphic.

Table 4.1: Comparison of Future Mechanism Type Characteristics.

Future Mechanisms	Type Characteristics			
	Resolution	Conversion	First-Class	Polymorphic
MultiLisp Futures	dynamic	implicit	no	yes
Argus Promises	static	explicit	no	no
C-Smalltalk Cboxes	dynamic	implicit	yes	yes
ES-Kit Future Refs	static	explicit/implicit	yes	no

4.3 Polymorphic Futures

As previously mentioned, a severe limitation of Future Refs is that only primitive language types are supported. A more powerful generalization would allow arbitrary user-defined types. To define a more powerful generalization, a type system supporting the declaration and use of polymorphic, or parametric, types is required. In this section, a new future mechanism is introduced that approaches the expressiveness of futures in MultiLisp, is as strongly typed as a promise in Argus, is represented by an first-class object like a Cbox in Concurrent Smalltalk, and improves on the type weaknesses of Future Refs in ES-Kit. A principle feature of this new mechanism is that it allows representation of arbitrary user-defined types as futures.

Figure 4.2 illustrates the definition of a polymorphic future type expressed as a template class parameterized by the generic type parameter `T`. A polymorphic future type is referred to as a "future of type `T`" and denoted by the type `Future<T>`. A concrete future type is an instance of `Future<T>`; for example, a `Future<int>` denotes a future of type integer. To illustrate the use of a polymorphic future type, consider the following type signature for a `connect` operation that performs an asynchronous network connection request returning an integer result once the connection is established, aborts, or is rejected:

```
Future<int> connect(...);        /* asynchronous network connect */
```

The type signature specifies that the type of the result of the network `connect` operation is a future object representing an integer.

A natural consequence of the representation of a future as a polymorphic type is that an instance of a specific `Future<T>` is implicitly convertible to a instance of type `T`. That is, a `Future<T>` instance can be used in place of an instance of type `T`—conversion will occur implicitly. The semantics of future type conversion are that a value of type `T` is

```
template<class T> class Future {
private:

   Capsule<T>* future;              /* future capsule */

   inline void* operator new(size_t)       { return NULL; }
   inline void  operator delete(void*)     { return; }
   inline void  operator &()               { return; }

public:

   inline Future()                         { future = new Capsule<T>(); }
   inline Future(const Future& f)          { future = f.future; future->refs++; }
   inline ~Future()                        { if (--future->refs == 0) delete future; }

   inline static Future create() { Future f; return f; }

   /* "producer" methods */

   inline void operator=(Future f) {
         Producer mutex(future);
         if (--future->refs == 0)
            delete future;
         future = f.future;
         future->refs++;
   }

   inline void operator=(const T& result) {
         Producer mutex(future);
         future->result = new T(result);
   }

   /* "consumer" methods */

   inline operator T()             { Consumer mutex(future); return *future->result; }
   inline T operator*()            { Consumer mutex(future); return *future->result; }
   inline const T* operator->()    { Consumer mutex(future); return future->result; }

   inline bool operator==(const Future& f) const { return f.future == future; }
   inline bool operator!=(const Future& f) const { return f.future != future; }
   inline bool exists() const { return future->result != NULL; }
};
```

Figure 4.2: Polymorphic Future Class.

produced if the value has been computed; otherwise, the type conversion to type T implicitly blocks awaiting the result. This semantics implies that synchronous invocation can be accomplished simply by requesting type conversion at the site of invocation; for example:

```
int result = connect(...);        /* implicitly blocks */
```

The connect operation returns a Future<int> object that is implicitly converted to an int since result is an integer type. The caller blocks at the expression continuation of the assignment statement awaiting the result value of the connect operation. Non-blocking, asynchronous behavior is achieved by explicitly using a Future<T> instance and later requesting the result value by implicit type conversion, as follows:

```
Future<int> future = connect(...);        /* async invocation */
  :
int result = future;        /* might block */
```

An inherent drawback to the use of futures in a statically typed language like C++ is that arguments to procedures are passed by value. Eager evaluation of procedure arguments forces the conversion of a future type to its result value, possibly blocking the procedure invocation awaiting the future value for an argument. Applicative-order evaluation implies that it is not possible within the C++ type system to pass a Future<T> instance to a procedure expecting a type T argument value without causing the evaluation of the future. If eager evaluation is not desired, it is necessary to explicitly specify an argument of type Future<T> in the type signature of the operation, thereby effecting call-by-need.

4.3.1 Future Capsules

Future objects represent continuations that will eventually be the site of a rendezvous between a thread computing the result value represented by the future object, and the client thread(s) that will eventually request the result. The potential for concurrent access to a Future<T> object dictates a requirement for synchronization control to guarantee that consumer threads block on the future until the operation serving the future produces the result. Consumer threads themselves may execute concurrently within a future. The condition synchronization of a future is similar to producer-consumer condition synchronization,

which is solvable using traditional synchronization techniques. A *capsule* is an object that is used by a future to encapsulate the eventual result value and provide synchronized access. The reason for instantiating a capsule object independent of a future object is primarily for efficiency. The synchronization employed by a future capsule requires mutual exclusion and condition synchronization. A novel approach is to define a capsule as part of a synchronization type hierarchy for monitors that uses inheritance to allow monitor specialization. Using inheritance, a capsule may be implemented as a subtype of a generalized producer-consumer monitor whose behavior is specialized to provide the specific synchronization semantics required by futures.

Figure 4.3 illustrates the relationship between a `Future<T>` object, a `Capsule<T>` object, and a result object of type `T`. As stated previously, the construction of a `Future<T>` object results in the implicit construction of a corresponding `Capsule<T>` object. The solid link from the `Future<T>` object to the `Capsule<T>` object denotes that the binding between the two objects is established at the time a `Future<T>` object is constructed. The `Future<T>` class is purposely separated from the `Capsule<T>` class so that a `Future<T>` instance may be used efficiently as either a return value or argument to a procedure. The dashed link between the `Capsule<T>` object and the object of type `T` is established when a value is computed and bound to the future capsule.

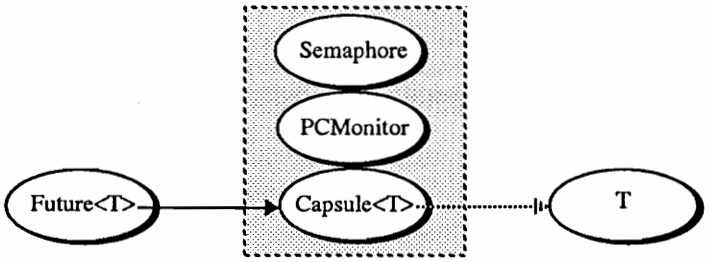

Figure 4.3: Future, Capsule, and Result Objects.

The shaded box surrounding the `Capsule<T>` object in Figure 4.3 depicts the composite object formed by a three tier inheritance structure. The inheritance hierarchy is *private*, meaning that only a `Capsule<T>` object is visible to a `Future<T>` instance. As depicted, the `Capsule<T>` class inherits privately from the `PCMonitor` class, which in turn inherits from the `Semaphore` class to enable mutual exclusion. The `PCMonitor` class defines a monitor that

```
template<class T> class Capsule : private PCMonitor {
private:

    friend class Future<T>;        /* for use only by a Future<T> */

    int    refs;                   /* reference count */
    T*     result;                 /* generic result binding */

    inline Capsule()               { result = NULL; refs = 1; }
    inline ~Capsule()              { delete result; }

    /* overloaded condition synchronization predicates/methods */

    virtual bool consumable() const { return result != NULL; }
    virtual bool producible() const { return TRUE; }

    virtual bool producer_egress() { consumer.broadcast(); }
};
```

Figure 4.4: Future Capsule Abstraction.

implements a producer-consumer synchronization discipline extended to provider concurrent consuming broadcasting to all blocked consumers. The actual mutual exclusion mechanism is system dependent. The efficiency of `Capsule<T>` objects is limited by the underlying system mechanism.[7]

At some point in time, a `Capsule<T>` instance will bind to a dynamically created instance of type `T`, denoted by the dashed arrow to the object labeled `T`. The object of type `T` represents the eventual value of a future. Figure 4.4 illustrates the template class that defines the polymorphic capsule type `Capsule<T>`. The generic type parameter `T` corresponds to the same type parameter of a `Future<T>` since instances of `Capsule<T>` are instantiated and bound to a specific `Future<T>` instance by the `Future<T>` constructor.

A concurrent producer thread binds a value of type `T` to a `Future<T>` instance by assignment, and one or more concurrent consumer threads access that value by using either the type conversion operator, the dereference operator, or the member access operator described in the last section. A `Future<T>` instance must use a synchronization control mechanism to block any consumer threads that attempt to access a future object before the

[7]Recent work by Bershad [17] has shown that mutual exclusion in support of multi-threading on uniprocessor RISC architectures can be made highly efficient even in the absence of explicit hardware support.

corresponding producer thread has bound a value to the future. Once the value is bound, all of the consumer threads can then access the future result value concurrently since a `broadcast` signal is delivered to the consumer condition queue.

4.3.1.1 Construction/Destruction

Construction and destruction of `Capsule<T>` types is only permitted by objects of type `Future<T>`. Placement of the constructor and destructor operations within the context of a `private` clause in the class definition, and declaration of the `Future<T>` class as a `friend` class, guarantees controlled construction and destruction of capsule objects.

Construction of a `Capsule<T>` instance occurs only as a consequence of the construction of a `Future<T>` object. The default constructor simply initializes the variable `result` to a null value and initialized the reference count `refs` to 1.

Destruction of a `Capsule<T>` type is caused by the destruction of the `Future<T>` object to which it is bound. Destruction simply deletes the type `T` object bound to `result`, if any. If no result is bound and consumer threads are blocked on the consumer condition queue awaiting the result, the default action is to terminate all blocked threads.

4.3.1.2 Condition Synchronization Predicates

As discussed in Chapter 3, inheriting synchronization constraints in concurrent object-oriented programming languages results in a conflict that arises when attempting to reuse methods that internalize condition synchronization constraints. As previously demonstrated, the conflict arises because the synchronization conditions are too tightly bound to the monitor operations. The solution outlined previously implements condition synchronization constraints as "virtual" synchronization predicates that are defined independently of monitor operations. The monitor operations defined in Chapter 3 utilize virtual synchronization predicates and the binding between the monitor operations and the predicates is thus relaxed. Defining synchronization conditions as virtual predicates allows the conditions to be easily redefined by derived classes, thereby facilitating the reuse of monitor operations defined by a base monitor class.

The two synchronization predicates, `consumable` and `producible`, are virtual methods inherited by the `Capsule<T>` class from the `PCMonitor` class. The `Capsule<T>` class provides

Table 4.2: Summary of Polymorphic Future Methods Attributes.

Methods	Attributes	Description
c'tor	public, inline, static	default constructor
c'tor&	public, inline, static	"copy" constructor
create	public, inline, static	used in place of private new
d'tor	public, inline	default destructor
operator new	private, inline, static	dynamic allocation prohibited
operator delete	private, inline, static	dynamic deallocation prohibited
operator &	private, inline	"address of" operator prohibited
operator =	public, inline, mutex	overloaded assignment operator
operator T	public, inline, mutex	type conversion operator
operator *	public, inline, mutex	overloaded dereferencing operator
operator ->	public, inline, mutex	overloaded member access operator
operator ==	public, inline, const	overloaded equality relation
operator !=	public, inline, const	overloaded inequality relation
exists	public, inline, const	existence predicate

an implementation for each predicate that is specific to the synchronization requirements of futures. The `consumable` predicate is used by `PCMonitor` operations to enforce condition synchronization on consumer threads; likewise, the `producible` predicate is used to enforce condition synchronization on producer threads. The `consumable` synchronization predicate is defined to be `TRUE` only if an object of type `T` has been bound to `result` (i.e., `result` is not nil). An object of type `T` is bound to `result` by a producer thread executing a `Future<T>` assignment operator. The `producible` predicate is defined to be the constant `TRUE` since it is always the case that a producer thread may bind an object type `T` to `result`, subject only to acquisition of mutual exclusion by the `Future<T>` assignment operator. Hence, by overloading the inherited condition synchronization predicates with future specific conditions, the behavior of the `PCMonitor` is specialized to meet the particular synchronization needs of arbitrary future types.

4.3.2 Future Methods

A key issue in the implementation of the `Future<T>` type is that the methods should be easily redefined through inheritance and the implementation should incur minimal execution overhead. Table 4.2 summarizes the method attributes of the `Future<T>` type.

The following observations are made concerning the methods defined in the `Future<T>` declaration depicted in Figure 4.2:

- all operators are defined as inline methods.

- all non-constant operator methods employ synchronization control to ensure consistent access to the value held by the future.

Method inlining results in faster code only if the compiler also performs efficient machine register allocation when replacing a procedure call with inlined code. Davidson and Holler [50] offer convincing empirical evidence that inlining improves performance more often than not, particularly for small procedures like those defined by the `Future<T>` template class. By declaring all `Future<T>` methods inline, an optimizing compiler can often reduce the run-time overhead associated with invoking `Future<T>` operations; thus, operations on future objects are generally more efficient than normal procedure calls, at the cost of increased code size at the call site.[8]

4.3.2.1 Construction/Destruction

As depicted in Figure 4.2, a `Future<T>` object consists of a single data item denoted by the *protected* instance variable `future`, which is a pointer to an object of type `Capsule<T>`. A `Future<T>` object is typically created by a procedure that returns the future as a result value of an asynchronous procedure invocation. The size of a `Future<T>` object is the amount of storage required to hold a pointer, which is typically no greater than four octets. Hence, a `Future<T>` object may be efficiently returned by a procedure through a register and passed by value as a procedure argument, requiring only one slot in the call frame. The lifetime, or extent, of a future object once created, is subject only to the number of threads that maintain a reference to the future. In the simple case, there are two references to a future:

- one for the producer thread that will eventually bind a value to the future, and
- one for the consumer thread that will eventually claim the value.

In both cases, the producer and consumer contexts dictate the lifetime of the future. When a future is no longer "live", due to the fact that all contexts in which it is referenced no longer exist, it is deleted. Ideally, an automatic garbage collector would maintain such

[8]The C++ front-end to the GNU gcc 2.x compiler [155] performs efficient method inlining and generates optimized native machine code on a wide range of architectures.

information and automatically delete future objects when they are no longer referenced. However, C++ does not currently provide automatic garbage collection of objects.[9] As an alternative, a simple and efficient reference counting scheme based on the controlled construction and destruction of `Future<T>` objects is adopted. This scheme is based on the ability to restrict `Future<T>` objects so that it is not possible to obtain a pointer to a future object. Furthermore, each time a `Future<T>` object is constructed or destructed (either explicitly or implicitly as a result of scope rules) the reference count is updated properly.[10]

Constructors: A `Future<T>` object may be constructed either by a declaration of a `Future<T>` instance or by calling the `create` method. The `create` method is used in place of the `new` operator for dynamically creating future objects. The need for reference counting dictates that pointers to `Future<T>` objects must be prohibited; hence, the `new` operator, the `delete` operator, and the "`&`" operator for taking the address of an object, are declared as private methods to prohibit public use. Note that it is still possible to define a `Future<T>` reference type, denoted by the type `Future<T>&`. C++ reference types present a problem. References are like pointers, but are more restricted. A common usage is in the definition of a *copy constructor* used to "clone" objects. For example, the `Future<T>` copy constructor is defined by the following type signature:

```
Future(Future& f);
```

A copy constructor requires a reference type parameter; hence, unlike `Future<T>` pointer types, it is not acceptable to prohibit `Future<T>` reference types. Detecting when a future object is referenced by a reference type is problematic since the future reference counting scheme is based on the orderly construction and destruction of future objects. Assignment of a future object to a future reference type is not subject to the same rules. In principle, the reference count could be updated by defining an overloading for the `Future<T>&` type conversion operator in the `Future<T>` class definition. However, this would only allow the detection of when a reference type was applied to a future object; there is no general

[9] A future version of the GNU C++ compiler will employ some support for a generation-scavenging garbage collector, based on work by Moss et al. [79].

[10] Reference counting schemes have limitations with respect to reference cycles; hence, situations in which a future references another future either directly or indirectly should be avoided. When used in a normal manner as the result of an asynchronous procedure invocation, cycles will not occur.

mechanism for detecting when the same reference is dropped since reference types do not have destructors. As long as a reference type is only used as an argument to a procedure, and the scope of the future object bound to the reference exists longer than the scope of the procedure, then object reference types may be safely used. No other use is deemed safe since it is potentially the case that the future object being referenced might be deleted by some other thread if the reference count reaches zero.

The default construction of a `Future<T>` type results in the initialization of the `future` instance variable to a null value. A `Capsule<T>` object is only instantiated when the future object is actually used, by invoking one of the `Future<T>` public operations. The decision to delaying the binding of a `Capsule<T>` object is an optimization to eliminate unnecessary overhead when constructing temporary future objects. The binding of a `Capsule<T>` object occurs immediately on invocation of the copy constructor. The copy constructor invokes the `clone` operation, causing the binding of a `Capsule<T>` object to the future object being cloned, if one is not already bound. The newly constructed future object is then bound to the *same* `Capsule<T>` object; that is, cloning is used to make future objects share the same `Capsule<T>` object.

Destructor: The destruction operation `~Future` decrements the `future` reference count and deletes the bound `Capsule<T>` object if the reference count is zero. The deletion of the `Capsule<T>` instance bound to `future` implies the destruction of the encapsulated type `T` object, if any.

4.3.2.2 Overloaded Operator Methods

The body of each operator method contains a local declaration of a `mutex` object that enforces either `Consumer` or `Producer` mutual exclusion. A `mutex` object of type `Consumer` enforces consumer synchronization while an object of type `Producer` enforces producer synchronization. Upon entry to any operator method, a `mutex` object of the appropriate type is instantiated for the `Capsule<T>` instance bound to `future`. For example, a method that returns the value of a future defines:

```
Consumer mutex(future);
```

Construction of the `mutex` object occurs automatically on scope entry and corresponds to Dijkstra's P operation combined with the appropriate condition synchronization predicates that determine whether or not to delay on a condition queue. The construction of a `mutex` object blocks if the synchronization condition for the specified type is not met. Implicit destruction occurs on scope exit and corresponds to the V operation combined with signal-and-continue semantics for the appropriate condition queue. For example, the destructor for a `Producer` instance releases mutual exclusion and all waiting consumer threads that they may continue by broadcasting to the consumer condition queue.

A semantically satisfying aspect of defining mutual exclusion and condition synchronization as a declaration of a `Mutex<T>` subtype whose extent is subject to normal procedure scope rules is that the synchronization protocol is correctly followed by all operations.

Assignment: The doubly overloaded assignment operator `operator=` is used to assign either a value of type `T` to a future object or another future. Since assignment updates the future, a `Producer` object is constructed upon entering scope to effect mutual exclusion. Once mutual exclusion is acquired, a new object of type `T` is constructed and assigned to the `result` variable of the `Capsule<T>` object referenced by the `future` variable.

A requirement of all objects of type `T` represented by a `Future<T>` is that the object must define a copy constructor, denoted by the type signature `T(const T&)`. The copy constructor allows an instance of type `T` to be constructed from any other instance of type `T`, as is done in the `new` expression of the assignment operator. Primitive types have an implicit copy constructor defined. User-defined types must explicitly specify the copy constructor. The assignment operator, in creating and binding a type `T` object, invokes `T`'s copy constructor with the value being assigned to the future. For example, the following fragment invokes the `operator=` method on the object `date` with the value 2001:

```
Future<int> foo = fact(5);        /* invokes foo.operator=(120) */
```

The assignment operator in turn creates an object of type `int`, binding it to the `result` component of the object bound to `future`:

```
future->result = new int(120);
```

The default copy constructor generated by the C++ compiler for integer types is invoked to create a new integer instance.

An alternate overloading of the assignment operator is provided to permit destructive assignment of future objects, denoted by the type signature: `operator=(Future&)`. The assignment of one future object to another results in both future objects sharing the same capsule object, with the appropriate adjustments being made for reference counting.

Type Conversion: Next to assignment, type conversion is the most important operation defined on polymorphic future types. The type conversion operator method `operator T` allows a consumer thread to obtain a value of type `T` by type converting an object of type `Future<T>` to an object of type `T`. The invocation of the type conversion operator results in the creation of a `Consumer` object that will block the consumer thread if the result represented by the future is not yet available.

Implicit type conversion is accomplished by using a future instance of type `Future<T>` wherever a type `T` value is expected. While implicit type conversion greatly facilitates the use of futures, it can result in unexpected type conversions if used naïvely. The most common unexpected implicit conversion occurs when passing a future instance to a procedure expecting a value of type `T`. Since argument evaluation is applicative-order, the type conversion operator will implicitly be applied to the future instance in an attempt to obtain a value of type `T`. For example, passing a `Future<int>` object to the function `foo` defined by the following type signature may cause the procedure call to block pending the conversion of the `Future<int>` to a value of type `int`.

```
void foo(int x);
Future<int> future;
  ⋮
foo(future);      /* invocation may block */
```

If the future is not yet ready, the call to `foo` is equivalent to the following sequence:

```
int x = future.operator int();      /* explicit type conversion */
foo(x);
```

The right side of the assignment forces the conversion of the future to an integer, possibly blocking before assigning the result to the temporary. The call to `foo` is then made with the integer value that is bound to `x`.

The static typing of C++ forces the explicit use of `Future<T>` types in procedure type signatures to effect call-by-need semantics. If lazy evaluation of a future is required, then a specific `Future<T>` type should be declared as a formal parameter. For example, the signature for the `foo` procedure can be rewritten as follows:

```
void foo(Future<int> x);
```

The body of `foo` can then effect the type conversion of the future denoted by `x` at the appropriate time.

Dereference: The dereference operator method `operator*` has the same semantics as the type conversion operator. Applying the dereference operator to a future instance results in an object of type `T`. For example, applying the dereference operator to a `Future<char*>` instance representing a future for a string will cause the formatted print statement in the following code fragment to block until the string represented by the future has been read:

```
Future<char*> futureString;
printf("value read is %s", *futureString);
```

Member Access: The member access operator method `operator->` allows a consumer thread to manipulate the result value of a future *through* the future by obtaining a constant pointer of type `T` to the result. The `const` attribute applied to the result value of type `T*` indicates that the pointer returned by the member access operator is restricted to invoking only `const` operations. For example, assuming a type `String`, on which is defined a `length` operation with the `const` attribute, the following code fragment allows the `length` operation to be called through the future without first explicitly obtaining the `string` object:

```
Future<String> futureString;
  ⋮
int len = futureString.length();
```

The restriction permitting access to only constant operations through the future is necessary to maintain the integrity of the future. Without the restriction, it would be possible to destroy the object encapsulated by the future using an explicit call to the `delete` operator defined by default on all C++ objects.

4.3.2.3 Predicates and Relations

Like the operator methods, the predicate and relation methods are defined inline. Unlike the operator methods, predicate and relation methods are "constant" operations. Constant operations do not require synchronization control because they do not require that the future dereference the encapsulated value. Predicates and relations may be invoked at any time and do not cause the calling thread to block.

Existence: The existence predicate `exists` is a constant member function that is used to determine whether or not the future can be evaluated (by type conversion, dereferencing, or member access) without blocking.

Comparison: The equality predicate `operator==` is used to compare two instances of a `Future<T>` type. The inequality predicate `operator!=` is the dual of the equality operator method. The comparison is between two `Future<T>` objects and not the future values they denote. For example, the conditional in the following code fragment will immediately evaluate to `FALSE`, regardless of whether or not the connect operation has returned a result.

```
Future<int> future1 = connect(...);      /* async network connect */
Future<int> future2 = connect(...);
  ⋮
if (future1 == future2)      /* evaluates immediately to FALSE */
```

Trying to compare two incompatible future types, e.g., `Future<foo>` and `Future<bar>`, is an type error that will be caught at compile time.

To compare a future instance with a value of the same type as that encapsulated by the future, evaluation of the future must first be done by applying either the type conversion operator or the dereference operator. Continuing the above example, the following conditional evaluates to `TRUE` if both future instances refer to the same integer value:

```
if ((int) future1 == *future2)
  ...
```

```
class Lambda<Future<T>> : public MetaLambda {
protected:

    Future<T>     result;

    virtual void lambda () =NULL;

public:

    inline Lambda& operator() () { eval(); return *this; }
    inline operator Future<T> () { return result; }

#define resultis(x)         do { result = (x); return; } while (FALSE)
};
```

Figure 4.5: Parameterizing a Lambda using a Future.

4.4 Concurrent Lambda Types using Futures

An important feature of the `Future<T>` type is that it can be used as the parameterized result type of a `Lambda<T>` type. The key idea is that the `Future<T>` type in conjunction with the `Lambda<T>` type are sufficient abstractions for concurrent computation. Since the `Lambda<T>` type implements a general closure for a procedural computation, all that is necessary to effect concurrent computation is a mechanism for a delayed result value and the ability to fork a thread.

For example, Figure 4.5 depicts a type `Lambda<Future<T>>` that defines a Lambda object whose `result` has type `Future<T>` and whose type conversion operator return a `Future<T>`. This type is generated by the compiler when the `Lambda<T>` type is parameterized with a `Future<T>` type. The `resultis` macro used by the `lambda` method for assigning a result value invokes the future assignment operator to bind the result to the future.

Consider again the factorial example of Chapter 2. As depicted in Figure 4.6, a concurrent factorial can be defined by providing a `Future<int>` type as the type parameter to the inherited `Lambda<T>` class. The inline `fact` function creates a thread that will schedule the `eval` method for execution and return a `Future<int>` as a result. The concurrent factorial is simply invoked as:

```
Future<int> future = fact(n);
```

```
class Factorial : public Lambda<Future<int>> {
protected:

    int n;

    void lambda() {...}

public:

    inline Factorial(int i) { n = i; }
};

inline Future<int> cfact(int n) {          /* concurrent applicator */
    Factorial *f = new Factorial(n);
    LambdaThread::fork(f);
    return *f;
}
```

Figure 4.6: A Concurrent Factorial using a Future.

```
    :
int result = future;
```

The implementation details of the `LambdaThread` class used to create a thread for a `Lambda<T>` object are not germane here; the skeletal structure was presented in Figure 2.10 on page 36. One of the previously described threads packages can be used to provide the underlying threads mechanism.[11] The `LambdaThread::fork` method simply forks the `eval` method inherited from the `MetaLambda` class to effect a separate thread of control.

The next chapter demonstrates more generally the expressive power of the `Future<T>` and `Lambda<T>` types by implementing a set of Actor types.

4.5 Future Work

The use of template classes in C++ allows the creation of polymorphic future types and polymorphic capsule types. An interesting area for future work is to implement the future, capsule, monitor, and mutex abstractions in Ada and Modula–3. Both Ada and Modula–3

[11] For example, the PRESTO threads package or POSIX threads.

provide generic types, and unlike C++, provide built-in support for multitasking. Ada implements a *CSP*-style rendezvous tasking model; Modula–3 implements a lightweight threads model like the one assumed here. Modula–3 also provides built-in support for garbage collection, which would facilitate the storage management of future types.

A useful experiment would be to compare and contrast all three implementations in terms of their structure and performance. The original Ada definition does not define an inheritance mechanism, so a non-inheritance based compositional relationship between capsules and monitors would have to be designed.[12] Modula–3 provides a subtyping mechanism, so the structure here should be directly translatable.

A more theoretical avenue of research is to explore the semantics of polymorphic future types from a denotational perspective. Section 4.2.1 of this chapter alluded to a relationship between continuations and futures. Continuations have mostly been explored in a sequential language setting, primarily in the context of Scheme. Continuations have been used to implement coroutine semantics [69], which is a more limited form of concurrency than assumed by future types. Given the close relationship between continuations and futures in asynchronous invocation semantics, there is the potential for achieving some useful theoretical results relating type conversion and synchronization in the context of polymorphic future types.

[12]The forthcoming Ada–9X definition does define an inheritance mechanism.

Chapter 5

Constructing Actors using Lambda and Future Types

This chapter demonstrates the utility of the previously introduced polymorphic lambda and future types for concurrent programming. A strongly typed Actor system is constructed using the lambda and future types as fundamental building blocks. Although the resulting system is not as flexible or extensible as a dynamically typed, interpretive Actor language, it is simple and can be used to program actor-style concurrent applications with real-world constraints. The primary conclusion to be drawn from this chapter is that first-class representations for procedures and futures are flexible abstractions for concurrent programming in a statically typed, imperative language, such as C++.

5.1 Introduction

Concurrent object-oriented programming languages (COOPLs) are interesting because objects are natural units for the expression of concurrency due to encapsulation of private data and the ability to arbitrate access to that data at a well-defined interface. The generality of the object concept presents a challenge when attempting to compare and contrast the salient features of different COOPLs. Within the design space of COOPLs, different models of concurrent object execution exist. The fundamental issues are the degree of inter-object and intra-object concurrency, and the coexistence of concurrency mechanisms with other object-oriented language mechanisms. Papathomas [138] provides a taxonomy by which COOPLs may be grossly categorized according to whether or not objects are endowed with concurrency properties. According to Papathomas' categorization, a language in which all objects embody concurrency semantics are classified as *non-orthogonal*. For example, a language based on the Actor model is non-orthogonal since all actors by default possess an independent thread of control. Languages in which concurrency properties are not directly associated with objects are classified as *orthogonal*. For example, adding a concurrency

mechanism to a sequential language to empower it with concurrent procedure execution is an example of an orthogonal COOPL. In general, the non-orthogonal approach dictates the definition of an entirely new language, while the orthogonal approach implies extensions to an existing sequential language. POOL–T [5, 6], Ellie [8], Hybrid [134], and ABCL/1 [183] are non-orthogonal. Concurrent Smalltalk[181, 182], Concurrent C++ [61], and ACT++ [97, 111] are orthogonal.

The use of the term orthogonal to describe a COOPL that is based on extensions to an existing sequential language is unfortunate since the connotation is that objects and concurrency are inherently at odds. This is not necessarily the case. Clearly, in the rush to empower a language with concurrency semantics, some designers choose an *ad hoc* approach; for example, the C++ task package and the PRESTO threads package [16] add a basic threads capability to C++, but the result is orthogonal with respect to the language type system and the inheritance mechanism since some features can not be safely used in conjunction with the threads mechanism. ACT++ is an attempt to synthesize a concurrent language framework by implementing the non-orthogonal Actor model using a sequential object-oriented language and an independent concurrency mechanism. The result is a set of programming abstractions that facilitate the expression of actor computations in a statically typed language; however, there are restrictions on the use of some class inheritance features because of orthogonality that arises in the integration of C++ and a threads mechanism. In ACT++, objects that represent actors possess a thread of control while others do not. In general, objects possessing an independent thread of control are called *active* objects, while objects that execute their methods by borrowing another object's thread of control are *passive* objects. In ACT++, the programmer indicates explicitly whether or not an object is active or passive by defining actor classes. Unlike the pure model approach, the addition of explicit concurrency mechanisms adds slight syntactic complexity since the programmer explicitly defines an object as either passive or active. The additional complexity can be sufficiently dealt with by defining appropriate programming abstractions that present conceptual uniformity. While a non-orthogonal language is typically semantically pure, a hybrid approach based on principaled extensions to an existing language can have a sound operational semantics. A distinct advantage of the hybrid approach is that it can be used as the implementation language for constructing an efficient virtual machine that

presents the programmer with a semantically pure COOPL.

The remainder of this chapter is organized as follows. Section 5.2 presents a primitive object model that supports the construction of different concurrent object-oriented programming paradigms based on the simple concept of "procedures as objects", or lambda objects, as introduced in Chapter 2. The first-class nature of lambda objects facilitates the construction of computational models requiring asynchronous interaction and concurrent execution. Section 5.3 reviews the components of the Actor model. A fresh perspective is offered that views an actor as a binding mechanism that uses a special form of condition synchronization. Section 5.4 presents a set of types that represent the components of an actor. The Actor type constructions provide a basis for constructing a new implementation of the ACT++ language originally defined by Kafura and Lee, and subsequently extended by Kafura and Mukherji [98, 131] with the behavior set mechanisms previously presented in Chapter 3. Section 5.5 discusses the benefits to be derived from constructing a new version of the ACT++ class library based on the lambda and future type abstractions. The primary advantage of a new implementation is the independence gained by having an execution model based on an operational semantics of lambda objects rather than a specific threads package. This separation of concerns provides semantic independence of a concurrency model from a particular threads implementation. Section 5.6 outlines directions for extending the ideas presented in this chapter.

5.2 A Lambda-based Object Model

This section introduces a primitive object model in which some subset of the methods defined for an object are defined as lambda objects. A goal of the model is to preserve the familiar object paradigm while enhancing the operational semantics of method execution to allow for the construction and concurrent execution of methods. The key feature of the model is that concurrent methods are realized by defining lambda objects as class-based extensions of the polymorphic lambda type introduced in Chapter 2.

Figure 5.1 depicts the structure of the object model. As in traditional object-oriented programming models, an object is a representation for an abstraction boundary. An object is defined primarily by the set of method type signatures that are publicly exported as part of its visible interface. The method type signatures denote actual procedures that

are invoked by the method dispatching mechanism to effect some behavior associated with the object. The distinguishing feature of the model depicted in Figure 5.1 is that object methods are divided into two distinct sets:

- a set of *primitive methods*, each denoted by M, and
- a set of *lambda objects*, each consisting of a primitive lambda method and a private state, denoted by σ, representing the environment bindings required by the method.

The distinction between the two is that lambda objects are instances of the polymorphic lambda type, which define "first-class" procedure types that are declared, constructed, and used like any other first-class language type. Lambda objects, as presented in Chapter 2, permit the definition of local procedure state and high-order operations over a procedure, independent of any data type encapsulating the lambda object. A lambda object represents the complete closure for the execution of the encapsulated λ method. The private state of a lambda object may be defined to contain all the information necessary to form the closure required for the execution of a particular λ. An important consequence of this fact is that the environment for a λ method need not be stack allocated, thereby allowing λ method execution to be independent of the calling context.

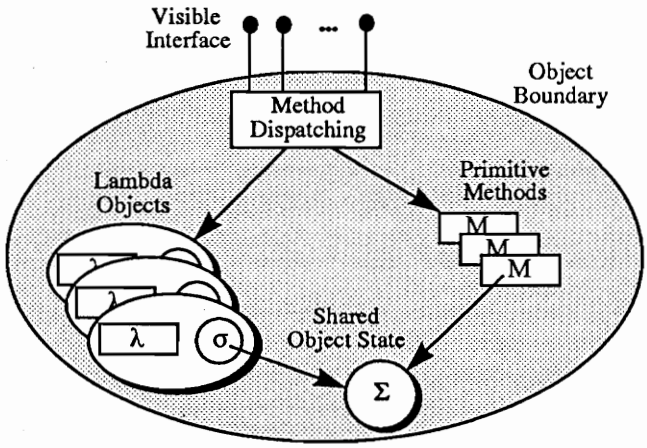

Figure 5.1: A Primitive Object Model.

Both primitive methods and lambda objects share the same object state, denoted by Σ. In a purely sequential setting, all methods are primitive and therefore the shared object

state is accessed serially, requiring no explicit synchronization. The method dispatching mechanism for the sequential case is based on traditional applicative-order, stack-based procedure invocation. Alternatively, the shared object state may be accessed concurrently by independently executing lambda objects. Synchronization is required to maintain a consistent shared state, so the concurrent λ methods and the method dispatching mechanism have to be more sophisticated, but it is not necessarily the case that the visible interface beyond the abstraction boundary need be any different.

The remainder of the present chapter is concerned with constructing actor programming abstractions based on an object model that includes lambda objects. An important observation is that this model does not require an extension to the syntax and semantics of the underlying language, but can be realized using the extensibility mechanisms of the type system and the normal expressive power of C++. The advantage of this approach is that the resulting extensions are compatible with existing language libraries and tools. The disadvantage is that the extensions are ultimately limited by the early binding inherent in a static type system.

5.3 Actors

Hewitt [70] introduced the Actor model and demonstrated the utility of message passing among actors as a means for expressing control abstractions. Subsequently, Clinger [41] defined the semantics of actors. The work of Agha [3] extended the basic Actor model with the idea of *replacement behavior* and made Hewitt's work accessible to a wider audience. The object-based approach to concurrency advocated by the Actor model continues to have wide influence on concurrent object languages and systems [184]. A significant feature of the Actor model is the treatment of behaviors, or procedures, as first-class constructions. Sussman and Steele developed Scheme [40] in part as an attempt to understand actor-style control structure in terms of continuations [156, 157]. Although Scheme is a sequential language, procedures, or lambda forms, are treated as first-class constructions.

The Actor model is a conceptually pure approach with a high degree of potential concurrency since every computation is carried out by an actor, and everything in the language is conceptually an actor. In practice however, efficiency concerns preclude the complete actor generalization—some elements are made primitive, such as integers. The execution

model is interesting because the establishment of the closure required to execute an actor operation is constructed in a lazy manner and is controlled in part by synchronization constraints local to each actor. For this reason, actor languages are most often implemented by a virtual machine based interpreter that supports dynamic typing and late binding at the cost of efficiency; for example, Acore [120] and Rosette [173]. The Rosette actor language adopts a virtual machine approach that implements primitive actor objects without the baggage of complete actors.[1] A major advantage of a language like Rosette is the coherent semantics that are achieved from a virtual machine design. The disadvantage is the cost in terms of run-time efficiency due to the interpreted aspect of the virtual machine, principally dynamic type checking.

An ambiguity arises in the Actor model when one asks the question: what *exactly* is an actor? A typical response is to simply enumerate the components defined by the model:

- a **message queue**, represented by a unique address and mailbox.
- a list of potential **tasks**, represented by messages.
- a set of potential **behaviors**, represented by scripts (procedures).
- a list of **acquaintances**, represented by local variables accessible by currently executing behaviors.

In an actor system, computation is effected by the behaviors. Figure 5.2 illustrates the configuration of a single actor, identified as α, which is prepared to bind the k^{th} message to the current behavior A_k. Behaviors are activated when the actor binding mechanism selects a message from the message queue, binds the message to a behavior, binds an environment, including acquaintances, to the behavior, and "activates" the behavior by binding a thread. As a behavior executes, it may send messages to other actors, create new actors, and compute a replacement behavior to process the next message in the message queue. As depicted in Figure 5.3, messages are sent using a `send` operation, actors are created using a `new` operation, and replacement behaviors are identified using a `become` operation.

[1] Similarly, for efficiency in sequential virtual machine based languages like Smalltalk [63] and SELF [176], not all objects are pure.

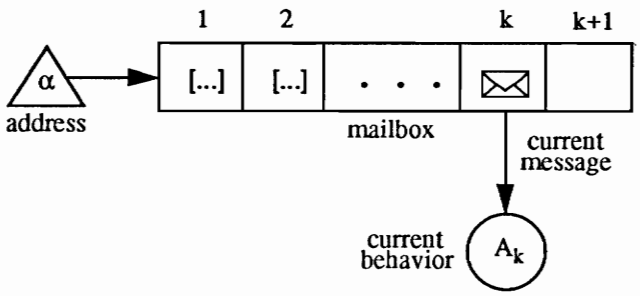

Figure 5.2: A Single Actor Abstraction.

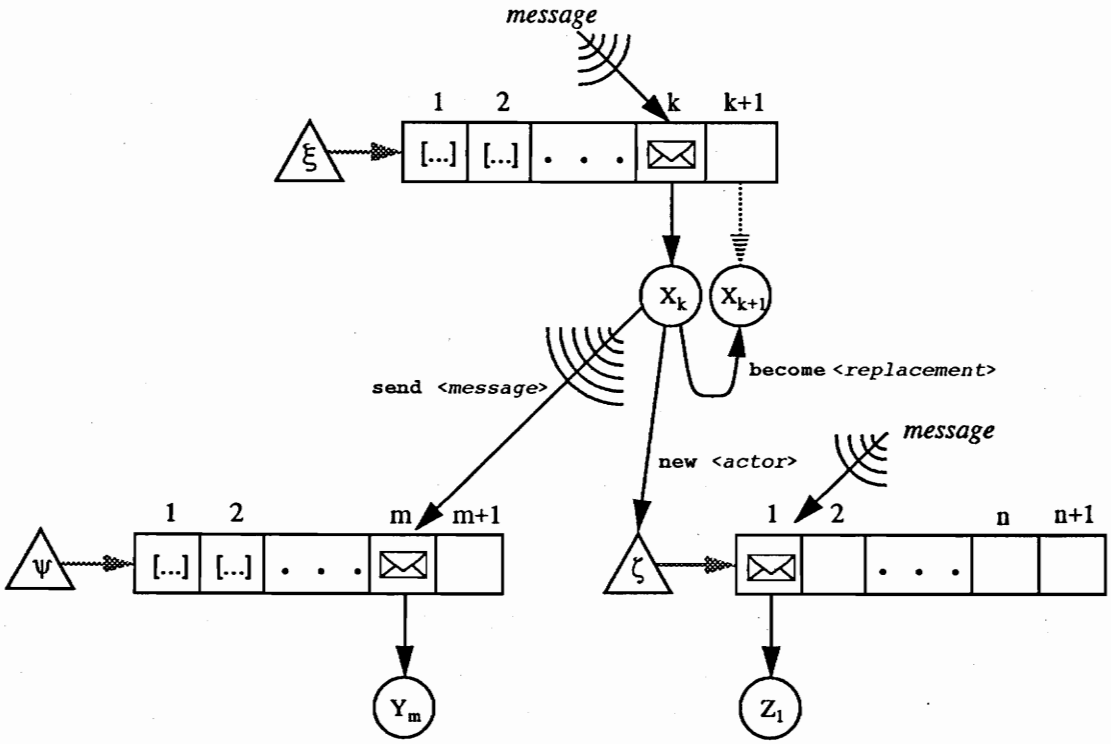

Figure 5.3: An Actor System in Action.

5.3.1 Actor Computation

The "actor-ness" in the Actor model is often associated with the address of the message queue component since the target of a message **send** operation is an actor, and messages are ultimately placed in a mailbox by the underlying communication system. An alternative perspective offered here is that an actor is a *synchronized binding mechanism* that effects the special relationship among the mail queue, the pending messages (tasks), the set of behaviors (procedures), and the acquaintances (environment). Put another way, an actor is an independent binding mechanism that "acts" to complete the unique closure for a potential computation and initiates the computation subject to local synchronization constraints. The actor binding mechanism may be represented by the higher-order function:

$$A(\lambda, \mu, \epsilon, \tau) \rightarrow P_\lambda$$

where λ denotes the behavior to activate, μ denotes the message (bound variable bindings), ϵ denotes the environment (free variable bindings), τ denotes the thread, and P_λ denotes the *process* representing the computation of the closure formed by binding λ, μ, ϵ, and τ. From this point of view, actors are perhaps best viewed as *constructors of computations*. In this sense, actors are similar to traditional data type constructors except that actors construct *active* procedure types rather than *passive* data types.

Computation occurs when a message is sent to an Actor. As typically implemented, a message contains an identifier for an operation along with a list of argument values, and perhaps a continuation to which the result of the eventual computation requested by the message is to be sent. When the actor synchronization conditions are such that the message can be bound to a behavior, the behavior associated with the identifier in the message is looked up, the argument values are bound to the behavior, the acquaintances are bound, a thread is bound, and the behavior is executed. A continuation, if one exists in the message, is also bound to the behavior. At some point in the execution of the behavior, the **become** operation is used to "unlock" the actor synchronization mechanism so that another message in the mail queue may be processed. In addition, the continuation is "satisfied" at some point in the execution by binding a result value.

As an alternative to the late binding policy, consider an early binding policy in which a *partial* closure for a behavior is formed as part of message creation. The partial closure

contains a binding for the procedure representing the behavior, the argument values, the continuation, and the acquaintances. In effect, a message contains a closure bound with everything required to execute a behavior except a thread. A thread could be bound early and placed in the idle state pending activation by the actor subject to local synchronization constraints. However, to conserve execution resources, it is more efficient to delay creating and binding a thread until the closure associated with a message can be scheduled for execution by the actor condition synchronization mechanism.[2]

Early binding has the effect of binding a behavior to an actor's acquaintance list at the time the behavior is created. The "normal" view of the actor replacement behavior is that the acquaintance list is cloned as part of each become operation so that as each new message is processed, each behavior is bound to this newly cloned acquaintance list. Early binding suggests that the become operation should not cause the actor to clone itself, but rather a behavior should determine whether or not to clone the actor and *re-bind* itself to this private clone. The distinction is subtle, but the consequences are not. Placing the responsibility for cloning with a behavior instead of the actor synchronization mechanism presents the opportunity to make an important optimization. If a behavior no longer accesses the acquaintance list after executing a become operation, cloning is unnecessary. In the normal case, the actor mechanism must always clone the acquaintance list since it does not know if the behavior that executed the last become operation will continue to access the acquaintance list or not. The actor has no choice but to clone to guarantee exclusive access for the next behavior. Allowing the behavior to make the choice about whether or not to clone the acquaintance list can significantly improve performance since extraneous copying of the acquaintance list can be eliminated.

5.3.2 Actor Synchronization

The purpose of the become operation is to effect concurrency by allowing an executing behavior to "unlock" the actor by computing a replacement behavior before the current behavior has completed execution. Upon completion of the become operation, the current

[2]To facilitate guaranteeing *fairness* [59], an actor system might allocate a fixed quota of threads to each actor synchronization mechanism. Thus, it makes sense to delay binding a thread until a behavior is to be executed.

behavior and the replacement behavior are allowed to execute concurrently. A replacement behavior should only be initiated if the current behavior has reached a point in its execution where it will no longer effect state change in an actor's acquaintance list, or it can create a private copy. In general, an *arbitrary* replacement behavior may be specified; for example, a queue might become a stack, or vice-versa. In practice however, a more restrictive semantics is often applied. The `become` operation is usually restricted to a subset of the behaviors attached to an actor as part of the actor definition. The behavior set concept introduced in Chapter 3 is intended to capture the restricted set of replacement behaviors. For example, a bounded buffer is restricted to the behaviors defined by the behavior sets `empty`, `full`, and `partial`, which implicitly define the set of legal operations that may be performed by an actor.

A deeper analysis of the `become` operation shows that it acts like a condition queue signaling primitive that has what might be called *signal-and-detach* semantics, as opposed to traditional condition queue signaling disciplines (i.e., signal-and-continue, signal-and-exit, signal-and-wait, etc.).[3] When the current behavior executes a `become` operation, the actor synchronization mechanism is signaled indicating that it may attempt to process another message. The current behavior "detaches" from the actor, perhaps cloning a private copy of the current state, and executes as a separate computation independent of the actor. The synchronization mechanism is then free to select a message from the queue and execute a new behavior. The normal view of message selection is that the *next* message in the mailbox is selected. However, it may be the case that the next message cannot be processed because the required operation cannot be executed due to current synchronization conditions. For example, a buffer actor can not allow a `get` operation to be performed when the buffer is empty.

A more flexible policy, which is adopted in some actor-based languages (e.g., Rosette [173]), is to allow the synchronization mechanism to peruse the mailbox and select *any* message that denotes an operation that can be executed with respect to current synchronization conditions. To make a selection, the synchronization mechanism must be able to determine whether or not the operation associated with a message can be legally executed.

[3] Hewitt and Atkinson [72] introduce the concept of a *serializer* in the Actor model for synchronization and contrast it to synchronization using traditional monitors.

The behavior set mechanism introduced in the latter half of Chapter 3 is one mechanism for expressing actor synchronization conditions. Behavior sets are a *centralized* mechanism that require a set of state predicates defined over the acquaintance list to determine the current behavior set. In general, a centralized scheme makes it difficult to include argument values from arbitrary messages as part of the state information used by the state predicates. Howard [78] illustrates a similar problem with centralized synchronization predicates in traditional monitors. An alternative is to define specialized predicates, or *guards*, over each behavior that are able to call upon the centralized predicates if needed. In selecting a message from the mailbox for execution, the synchronization mechanism can test the guard associated with each behavior represented by the message to determine whether or not the behavior can be executed. The result is a *distributed* synchronization mechanism since each behavior contains a predicate that determines whether or not it can execute. The distributed synchronization mechanism requires early binding of an actor's environment to each behavior sent as part of a message so that the guard defined on each behavior can access both its argument values as well as the centralized predicates defined over the actor's acquaintance list.

5.4 A Lambda-based Actor System

This section demonstrates how an actor system with early binding can be implemented using the polymorphic lambda and future types introduced in Chapter 2 and Chapter 4, respectively. The key idea behind using lambda and future types in constructing an actor system is that lambda objects are the fundamental unit of concurrent computation; hence, the actor synchronization mechanism and behaviors are represented as specializations of the `Lambda<T>` type. Behaviors are direct extensions of the basic polymorphic Lambda mechanism for representing first-class procedures. Polymorphic future types are used instead of explicit continuations for representing future result values. The threads of control associated with a specific actor are the thread bound to the current behavior and the thread associated with the actor synchronization mechanism.

There are three principal types used to construct Actor computations:

1. an `Actor` type.

2. a Message type.

3. a polymorphic Behavior type.

The Actor type provides an identity and mailbox to which Message objects are sent. A Message object is an encapsulator for a behavior object. The Behavior type is a subclass of Lambda<T>, meaning that all behaviors are lambda objects that require the definition of the pure virtual lambda method inherited from the Lambda<T> class. A Behavior object is generically referred to by its super type MetaLambda, which is the base class of all Lambda<T> types, as discussed in Chapter 2.

5.4.1 The Actor Type

The purpose of an actor type abstraction is to provide message queue synchronization and scheduling of behaviors. The Actor class depicted in Figure 5.4 encapsulates a unique mailbox to which messages are sent. Objects that are instances of any class inheriting from the Actor class inherit the send operation, which permits a message to be enqueued in the mailbox.

The Actor class is derived from the MetaLambda class that was introduced in Chapter 2 and depicted in its entirety in Figure 2.8. To facilitate the discussion in this chapter, an abbreviated MetaLambda class is depicted in Figure 5.5. The MetaLambda class provides a pure virtual declaration for the lambda method and default definitions for the virtual before, after, and guard methods. When an Actor object is constructed, a LambdaThread is created by the constructor to execute the inherited virtual eval method, resulting in the execution of the sequence before-lambda-after. In general, the lambda method executes until the actor is destructed, which kills the associated thread.

Unlike more homogeneous actor languages, programming with the Actor type allows the programmer considerable flexibility. For example, it is possible to redefine the synchronization and scheduling policy of the actor by simply redefining the implementation of the lambda method. The normal scheduling policy is to execute a behavior in the mail queue whose synchronization guard evaluates to TRUE, subject to any behavior set constraints. An alternative scheduling policy might favor a particular class of behaviors. For example, an actor with reader-writer semantics might wish to implement either readers priority or

```
class Actor : public MetaLambda {
private:

   MsgQueue     mailbox;          /* a basic message queue */
   bool         copied;           /* are we a clone? */                    5

   bool guard()  { return !mailbox.empty() && !mailbox.locked(); }

   void lambda() { ... }          /* synchronization/scheduling algorithm */
                                                                           10
public:

   /* copy constructor used for "cloning" */

   inline Actor (Actor& actor)    { copied = TRUE; }                       15
   inline Actor()                 { copied = FALSE; LambdaThread::fork(this); }

   ~Actor() { LambdaThread::kill(this); }

   inline void send (Message* msg) { mailbox << msg; }                     20

   inline bool cloned() { return copied; }

   /* compute next behavior set and unlock mailbox */
                                                                           25
   virtual void next()     { ...; mailbox.unlock(); }
};
```

Figure 5.4: The Actor Type.

```
class MetaLambda {
protected:

   virtual ~MetaLambda () {}      /* virtual destructor required */
                                                                           5
   virtual bool guard ()          { return TRUE;}

   ...            /* virtual pre/post predicates, assert macro */

   virtual void before ()         { assert(pre); }                         10
   virtual void after ()          { assert(post); }
   virtual void lambda ()         =NULL;
   virtual void eval ()           { before(); lambda(); after(); }
};
```

Figure 5.5: An Abbreviated MetaLambda Class.

```
class Foo : public Actor {
private:
   ...
   void lambda() { ... }          /* "foo" priority scheduling */
public:
   void foo() { ... }
   void bar()  { ... }
   ...
};
```

Figure 5.6: A User-Defined Actor.

writers priority. With a traditional synchronization mechanism, like the monitor presented in Chapter 3, multiple condition queues are required. With actors, a single queue holds all pending operation requests. A priority scheme is straightforwardly implemented on a per actor basis by redefining the scheduling algorithm implemented by the virtual `lambda` method. For example, the `Foo` actor depicted in Figure 5.6 redefines the `lambda` method inherited from the `Actor` class to allow `foo` operations to take priority over `bar` operations.

5.4.2 The Message Type

A message is simply a container for a generic `Behavior` object. As illustrated in the next section, each `Behavior` object is ultimately derived from the `MetaLambda` class. Hence, each instance of the `Behavior` class may be manipulated as a `MetaLambda` type by the actor scheduling mechanism.

The `Message` type depicted in Figure 5.7 consists of private mail queue linkage and a reference to a behavior object as a `MetaLambda` type. When the actor synchronization mechanism selects a message from the mail queue, the behavior is scheduled for execution by forking a `LambdaThread` to execute the `eval` method defined by default for all `MetaLambda` types.

```
class Message {
private:
   ...              /* linkage */
                                                                    5
public:
   MetaLambda* behavior;          /* a Behavior object */
   Message (MetaLambda* lambda) : behavior(lambda) { ... }          10
   ~Message ()   { ... }          /* dequeue from mailbox */
};
```

Figure 5.7: Actor Message Type.

5.4.3 The Behavior Type

Figure 5.8 depicts a polymorphic `Behavior` class parameterized by an actor type parameter `Actor` and a result type parameter `T`. When a behavior object is created, the actor to which the behavior is bound is passed as the argument of type `Actor` to the constructor. The `self` member variable represents the early binding of a behavior to an instance of type `Actor`. The `self` variable is subsequently used to refer to the acquaintances (instance variables and local methods) of the bound `Actor` object. When a behavior object is constructed, a new `Message` object is constructed and immediately sent to the `Actor` object to which the behavior is bound.

A behavior that produces a result value requires a future to provide the result to any other behaviors (threads) that may be waiting on the result. In this respect, the future serves the same purpose as a continuation. As discussed in Section 4.4 and depicted in Figure 4.5 of that section, the `Lambda<T>` type may take as its type parameter a `Future<T>` type. For example, a behavior that returns an integer value is defined as a subclass of the class `Lambda<Future<int>>`. The `Lambda<T>` type instance that results from substituting a `Future<int>` as the type parameter `T` is depicted in Figure 5.9. The `Lambda<T>` type is implemented such that the result of the `lambda` method, assigned to the future using the `resultis` macro, will be bound as the result value of the future and provided to any behavior that is able to rendezvous with the future object. Parameterization of the `Lambda<T>` type with a `Future<T>` type is a significant feature of a lambda-derived Actor system since it

```
template<class Actor, class T> class Behavior : public Lambda<T> {
protected:

    Actor*          self;      /* actor to which we are bound */

    /* bind to actor as self and send this object to the actor */

    inline Behavior (Actor* actor) : self(actor) { actor->send(new Message(this)); }

    ~Behavior () { if (self && self->cloned()) delete self; }

    /* cloning become copies actor, non-cloning become detaches the actor */

    inline void become()            { self->next(); self = new Actor(*self); }
    inline void become(void*)       { self->next(); self = NULL; }

    /* the default "after" method forces a non-cloning become */

    void after() { if (self && !self->cloned()) become(NULL); }
};
```

Figure 5.8: Actor Behavior Type.

implies that the `lambda` method of a `Behavior` object may be executed concurrently without any additional mechanism required for handling the result.

The `become` operations of the `Behavior` class are defined to allow a behavior to unlock the actor by executing the `next` operation on the `self` actor, and perhaps clone a private copy of the actor state. Unlocking the actor using the `next` method allows the actor scheduling mechanism to schedule a replacement behavior. If cloning is not required, a `become(NULL)` request will unlock the mailbox and detach the behavior from the actor without copying state information.

The virtual `after` method inherited from the `MetaLambda` class is defined to issue a `become(NULL)` request. The `after` method is always executed as part of the standard `before-lambda-after` sequence of the `eval` method that is executed for each `Behavior` object when the actor binds a thread to the behavior. The `after` method is used to effect an automatic `become` operation when "falling off" the end of a behavior so that the actor is never left in a locked state when the current behavior completes.

```
class Lambda<Future<int>> : public MetaLambda {
protected:
    Future<int>  result;              /* future result binding */
    virtual void lambda () =NULL;     /* defer implementation */
public:
    inline Lambda<Future<int>>& operator() () { eval(); return *this; }
    inline operator Future<int> () { return result; }
#define resultis(x)       do { result = (x); return; } while (FALSE)
};
```

Figure 5.9: A Lambda-Future Type.

5.4.4 A Bounded Buffer Actor

To illustrate a simple usage of the actor type abstractions just introduced, a simple bounded buffer actor is presented. Figure 5.10 depicts a `Buffer` class that inherits from the `Actor` class. The definition of the `Buffer` class is like any other class definition in C++ and no special syntax is required. In general, a class definition includes all of the following:

- acquaintances defined as local instance variable definitions.
- default constructors, including a copy constructor, and a virtual destructor.
- definitions for a set of nested *behavior* classes, inheriting from a `Behavior` template.
- inline *behavior constructor* methods corresponding to each of the behavior classes.

The acquaintance list is simply the set of instance variables defined in the class; in this case: `buf`, `max`, and `n`, representing an integer buffer, the maximum size, and the current number of elements, respectively. The default constructors must include a copy constructor to facilitate "cloning." The amount of state information to clone is a decision made on a per actor basis. In general, the entire set of instance variables may be cloned; however, to minimize copying, only those instance variables that may possibly be shared across concurrent behaviors need be cloned.

```
class Buffer : public Actor {
protected:

   int*          buf;     /* buffer */
   int           max;     /* maximum size */                                  5
   int           n;       /* number of elements */

   inline bool empty () const      { return n == 0; }
   inline bool full  () const      { return n == max - 1; }
                                                                              10
public:

   /* constructors/destructor */

   inline Buffer(int size) : max(size), n(0)          { buf = new int[size]; }  15
   inline Buffer(Buffer& b) : Actor(b)                { ... }        /* clone ourself */

   ~Buffer()                                          { delete buf; }

   /* behavior classes and their constructor methods */                        20

   friend class Get : public Behavior<Buffer, Future<int> > { ... };
   friend class Put : public Behavior<Buffer, void> { ... };

   inline Future<int> get()        { return *(new Get (this)); }              25
   inline void put(int x)          { new Put (this, x); }
};
```

Figure 5.10: Bounded Buffer Actor.

The most important components of an Actor subclass definition are the nested friend classes that define Behavior objects.[4] In Figure 5.10, two behavior classes are defined: the Get class representing the behavior that removes an integer from the buffer and a Put class representing the behavior that appends an integer to the buffer. Both behavior classes provide the Buffer type as the Actor type argument of the Behavior<Actor, T> template. The Get class defines the result type T as a Future<int> while the Put class defines the result type as a void. The result type corresponds to the type of the eventual result produced by the execution of the lambda method of a particular behavior. Corresponding to each behavior class is an inline behavior constructor method that constructs a new behavior object at each invocation. The get method constructs a new Get behavior object, while the put method constructs a new Put behavior object. As previously described, passing a Future<T> object as the result type of the Behavior class results in the binding of a Future<T> object as the result type of a Lambda<T> object. Hence, the get method returns a Future<int> object as its result by dereferencing the newly created Get object and implicitly applying the Future<int> type converter generated as part of instantiating a Lambda<Future<int>> type.

The following code fragment illustrates how to instantiate a new buffer actor and send a get message, a put message, and then evaluate the Future<int> returned by the get message.

```
Buffer buf(10);        /* create an integer buffer actor of size 10 */
Future<int> f = buf.get();        /* send a "get" message */
buf.put(5);        /* send a "put" message with the value 5 */
  :
int result = f;        /* evaluate the future returned by buf.get */
```

Syntactically, the above looks like normal C++ syntax for procedure invocation; however, each invocation results in an inline expansion that effects behavior object creation, partial closure binding, and message sending. To appreciate the semantics, the details of the Get and Put behavior classes must be examined.

[4] Nested classes in C++ ensure that name conflicts do not arise. A nested class name is qualified by the name of the containing class. The methods of a class declared as a friend have access to the private/protected components of the class in which the friend is declared.

Although seemingly complicated, the actions discussed in the following subsections are accomplished by a relatively small amount of code that is mostly expanded inline at the site of a method invocation. The objective is to allow the programmer to use normal C++ type declarations and method invocation syntax to effect actor instantiation, message passing semantics, and concurrent execution. The important observation is that almost all of the machinery is contained by default in the `Lambda<T>` and `Future<T>` types. The `Behavior<Actor, T>` type is simply the "glue" that effects the special configuration of these types required to implement a strongly typed actor system.

5.4.4.1 The Get Behavior Class

The `Get` behavior class is defined in Figure 5.11. The `guard` and `lambda` methods are virtual methods inherited from the `MetaLambda` base class. The `guard` method implements the synchronization predicate for the behavior while the `lambda` method implements the algorithm for removing an element from the buffer and binding the element to the future returned to the caller of the `get` behavior constructor, as shown above. The `resultis` macro effects the future result binding by invoking the assignment operator of the `Future<int>` object bound to the `result` variable defined by the inherited `Lambda<Future<int>>` class.

To elaborate, sending a get message by invoking:

```
Future<int> f = buf.get();
```

results in an inline expansion semantically equivalent to the following:

```
Buffer::Get* obj = new Buffer::Get(&buf);
obj->self = &buf;
buf.send(new Message(obj));
Future<int> f = obj->operator Future<int>();
```

The `Get` constructor passes the address of the actor `buf` to its parent `Behavior` constructor, which binds the address of the `buf` actor as `self`, thereby effecting early binding to the environment (acquaintances) of the `buf` actor. A new `Message` object is then constructed and sent to the buffer actor. The `Future<int>` type converter is generated automatically as part of the template instantiation of a `Lambda<Future<int>>` type. The `Lambda<Future<int>>`

```
class Buffer : public Actor {
   ...

public:

   ...

   friend class Get : public Behavior<Buffer, Future<int>> {

        bool guard()     { return !self->empty(); }
        void lambda()    { self->n--; resultis (self->buf[self->n]); }

   public:

        inline Get(Buffer* b) : Behavior<Buffer, Future<int>>(b) {}
   };


   friend class Put : public Behavior<Buffer, void> {

        int x;

        bool guard()     { return !self->full(); }
        void lambda()    { self->buf[self->n] = x; self->n++; }

   public:

        inline Put(Buffer* b, int x) : Behavior<Buffer, void>(b), x(x) {}
   };

   ...
};
```

Figure 5.11: Nested Behavior Friend Classes.

base type automatically instantiates a `result` object of type `Future<int>` as part of construction, thereby binding a future to which the eventual result value will be sent. The constructed future object is returned as a result of the `Future<int>` type converter applied implicitly to `obj` as part of the return from the `get` constructor. Hence, the caller of the `get` method and the newly constructed `Get` object share the same future, into which the `Get` object `lambda` method will eventually deposit the result of removing an element from the buffer.[5]

5.4.4.2 The Put Behavior Class

The `Put` class depicted in Figure 5.11 differs from the `Get` class in two principal ways:

1. the result type is `void`.

2. an argument value is provided to the constructor.

Since a `put` operation does not produce a value, no future result object is required; hence, the `Behavior<Buffer, void>` class inherits from the specially defined `Lambda<void>` class. The `Lambda<void>` class has no `result` instance variable and no type converter operation since no result value is required. The `Put` constructor takes an integer argument as the value to eventually be placed into the buffer. As with previous examples described in Chapter 2, argument values to lambda objects are bound as instance variables on construction as part of closure formation, and are thus accessible to the `lambda` method when eventually executed.

5.5 A New Foundation for ACT++

The Actor type abstractions provide a foundation for implementing a new version of the ACT++ prototype language originally proposed as a class-based extension to C++ by Kafura and Lee, and re-implemented with PRESTO threads by Kafura and Mukherji. An advantage to basing a new ACT++ on lambda and futures are that the behavior sets can be

[5] After depositing a result value and "falling off" the end of the `LambdaThread` associated with the `eval` method, all `Behavior` objects are automatically garbage collected. The collection mechanism is triggered by thread death and results in the invocation of the `delete` method on a `MetaLambda` typed pointer, which initiates the virtual destructor chain of the behavior object.

combined with guards defined over lambda objects to provide more flexible synchronization schemes.

Additionally, the choice of a threads mechanism can be made independent of the lambda-based execution model. Lambda objects are the fundamental units of execution and any suitable threads package can be used. The independence from a specific threads package provides a simpler execution model based on the simple semantics of `before-lambda-after` method execution sequence defined by the default `eval` method. Since each lambda object represents a complete closure, there is no hidden state internalized by some threads mechanism. The advantage is that debugging actor programs derived from lambda objects is more rational since it is not necessary to be cognizant of what the threads mechanism is doing. Most useful information is contained in the lambda object and is open to examination using normal C++ debuggers.

Many threads packages restrict certain language features because the threads mechanism is primitive and has no type information available to it. For example, in PRESTO, virtual and overloaded member functions may not be scheduled as threads. Furthermore, the thread creation mechanism requires arguments to procedures to be either integer types or pointers of type `void*`, and procedures must be type cast to a canonical procedure type so that a generic "call state" can be constructed. The canonical procedure type is typically a pointer to a procedure taking no arguments and returning a `void` type. A call state is created by capturing the stack frame at the point of invocation.[6] The result is that all useful type information known in the application is lost when using the threads mechanism. This feature of "modern" thread mechanisms is less than satisfying.

Alternatively, the object-based execution model offered by the lambda and future class abstractions preserve type information and allow the full use of virtual and overloaded member functions *in conjunction* with a primitive threads mechanism. The `Lambda<T>` class abstraction provides a first-class closure so that the "call state" created by a thread mechanism for executing a lambda object need contain only two primitive components:

- the address of the `eval` method, which is a procedure of the canonical type required by most thread mechanisms.

[6] Usually, special machine *dependent* assembler code that "understands" the semantics of procedure call and performs the appropriate magic to collect the call state is required; hence, portability is compromised.

- an object pointer of type `MetaLambda`, which is used to invoke the `eval` method so that the entire virtual method dispatch table of the lambda object is accessible.

These two components are sufficient for constructing independent threads for effecting the concurrent execution of arbitrary lambda objects. Furthermore, given the minimal requirements made on a threads package, almost any one will do (e.g., POSIX threads [85]).

5.6 Future Work

The order and time at which bindings are performed to complete a closure for a behavior is not fully explored here. The choice of C++ implies that most bindings are determined as early as possible. The use of the lambda type allows early binding of bound variables (message arguments), a delayed and changeable binding of free variables (environment) through the `self` pointer, late binding of an execution thread, and late binding of a result using a future. There are of course other alternatives. A complete formalization of the semantics of binding order and time in the Actor model should be undertaken. It is believed that once the alternatives are clearly defined, a more general binding mechanism can be constructed that will allow a choice in the "style" of actors one chooses to program with.

A practical enhancement to the lambda and future types is to define a more suitable syntax that can be preprocessed into the C++ class structure defined previously. For example, Figure 5.12 depicts a suitable syntax. The `actor` keyword is used in place of the `class` keyword to indicate that the `Actor` class is inherited. The `behavior` keyword is used as a method attribute to indicate that the method should construct a `Behavior` class definition and replace the body of the method to construct a behavior object that is subsequently enqueued in the mail queue.

A disadvantage of preprocessing is that debugging requires knowledge of the transformation performed by the preprocessor; however, the simple transformations specified here do not require a significant semantic leap on the part of the programmer and would reduce the syntactic complexity of using the Actor types presented previously.

Finally, other types of concurrency paradigms other than actors can be represented using the lambda and future types. These types are fundamental building blocks that capture the notion of a closure for a procedure and the future result of a computation; hence, they can be used to construct various concurrency models.

```
actor Buffer {
private:

   ...

public:

   Buffer() { ... }

   behavior Future<int> get() { ... }
   behavior put(int x) { ... }
};
```

Figure 5.12: Simple Actor Class Syntax.

Chapter 6

Type Structures for Layered Communication Protocols

A new approach to structuring layered protocols using polymorphic service access points and type inheritance between protocol machines is presented. Polymorphic service access points facilitate the flexible instantiation of protocol machines containing the minimal functionality required by an application. The type inheritance structure is induced by a *vertical partitioning* of the upper layers of the ISO Reference Model, resulting in a *non-monolithic* protocol stack that supports concurrent object-oriented applications, such as Actor systems. The results presented represent a modest contribution to the research and practice of advanced protocol development. The type structures introduced demonstrate to software engineers, particularly those developing layered systems software, that object-oriented programming techniques facilitate and enhance the implementation of layered architectures. In addition, the results presented provide justification for object-oriented language designers that polymorphic and extensible type structures are useful for efficiently implementing layered communication protocol architectures.

A pragmatic conclusion to be drawn from this work is that appropriate use of object-oriented programming mechanisms in the implementation of communication architectures results in a type structure that reflects the architectural model, preserves layer protection, and enhances the run-time performance of concurrent applications using the protocol stack. This result is important since it contradicts the assertion by some that layered protocol architectures, such as the ISO Reference Model, necessarily suffer in performance because of layering, thus forcing implementors to violate the layering principle in search of efficiency. The result substantiates the experience of others, notably Clark, that sound network programming methodology leads to efficient implementations that are also understandable.

6.1 Introduction

Communication is a central component of a concurrent object-oriented system, particularly when objects are distributed. The message passing metaphor of the object-oriented paradigm is a natural match with a peer-to-peer distributed computing model because of the inherent asynchrony. The Actor model offers a message passing concurrency model that can be combined with a traditional peer-to-peer communication model to enable distributed Actor programming for *open systems*. The term "open system" has different denotations in different domains of discourse; the following section clarifies this concept.

6.1.1 Open Systems

In computer communications, an open system is narrowly defined as a system that adheres to a set of standards and conventions for the open *interconnection* of heterogeneous computer systems. From this perspective, the primary purpose of an open system is to enable the reliable communication of data among computers via an internetwork of LANs and WANs. The important issues are the structure and rules of interaction for layered protocol machines. Perhaps the term *interoperable system* is more descriptive since the primary objective is to achieve reliable interoperability among heterogeneous components. In semantic modeling and artificial intelligence, an open system denotes any collection of subsystems that are open-ended, incremental, and evolutionary [73]. A primary feature of this type of open system is the ability of its subsystems to communicate and interoperate using standard protocols. As defined by Hewitt [71], an open information system is a particular kind of evolutionary open system based on the Actor model, that depends on standardized communication protocols. Merging both viewpoints leads to the following definition.

Definition 6.1 *An open system is a potentially open-ended, incremental, and evolutionary system relying on standard interconnection protocols for intercommunication among distributed, possibly heterogeneous, computational objects.*

Agha [4] recently proposed the concept of an ActorSpace as a paradigm for distributed groups of actors. Conceptually, the ActorSpace has a purpose similar to the Linda Tuple Space model [62] for cooperative work in parallel and distributed computing environments. Tomlinson [172] implemented a TreeSpace model in the Rosette language that is

semantically a combination of the ActorSpace and Tuple Space models. Each of the group interaction models share the need for a feature rich, flexible, and efficient communications infrastructure that provides the abstractions required to effect the types of communication required for basic peer-to-peer interaction.

The primary focus of this chapter is on the *type structure* of a communications infrastructure that provides the basic peer-to-peer communications required by a variety of interaction models. The intent is not to introduce a group communication model such as the virtual synchrony model described by Birman [18] or the distributed process group model described by Cheriton [33]; but rather, to illustrate the application of type polymorphism and type inheritance to the structuring problems associated with layered peer-to-peer communication. The work described here is already being adopted as the communications infrastructure for peer-to-peer communication in the Rosette virtual machine, which requires a flexible communication infrastructure in order to implement more abstract peer-to-peer interaction models, such as TreeSpaces, in a heterogeneous computing environment.

6.1.2 Outline

The remainder of the chapter begins with a concise review of layered peer-to-peer communication using the OSI Reference Model as a foundation. The primary purpose of Section 6.2 is to introduce key terminology and concepts assumed in later sections. Those familiar with the details of peer-to-peer interaction and the specific functions of upper layer protocols (above transport) may wish to skip this section. Section 6.3 defines the basic problems associated with implementing an upper layer protocol architecture. This section draws its inspiration primarily from the work of Clark. The principal feature of this section is a discussion of the problems of layering, synchronization, data manipulation, and flexible composition with respect to protocol implementations. Section 6.4 contrasts the traditional monolithic approach to structuring protocol machines with a new approach based on non-monolithic protocol machines. The new approach relies on a vertically partitioned protocol stack whose components are representable using polymorphic types and type inheritance between protocol machines. Section 6.5 presents the essential aspects of an object-oriented implementation of the upper layer protocols that is based on the new approach to structuring protocol implementations. Related work is described in Section 6.6, while Section 6.7

concludes the chapter by outlining future extensions.

6.2 Layered Peer-to-Peer Communication

Layering is a common technique for modeling a complex system as a hierarchical collection of services arranged in successively dependent layers. Dijkstra is widely credited with an early application of hierarchical structuring to computer systems [54]. A layered partitioning of services provides a functional separation of concerns by defining a service at a layer n in terms of the services available at layer $n-1$. That is, each layer defines a communication service in terms of the services available at the next lower layer, down to the physical layer. In particular, the well-known ISO Reference Model for Open Systems Interconnection (OSI) defines a set of services and protocols for the interconnection of computer systems [92].

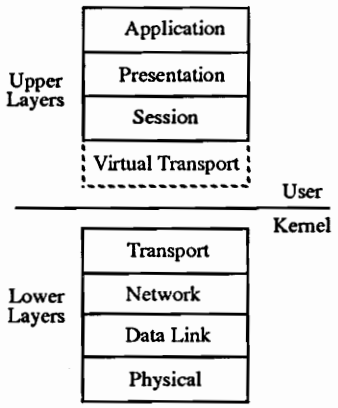

Figure 6.1: Partitioning of the ISO Reference Model.

As depicted in Figure 6.1, the layered communication model is divided into the *upper layers* and the *lower layers*. The upper layers consist of all layers above the transport layer. The collective function of the upper layers is to impose *structure* onto the otherwise unstructured communication service offered by a particular transport/network protocol combination. Each upper layer protocol offers an incrementally richer set of services that enable a two-way structured dialogue between peer application layer entities. From a conceptual

point of view, the transport layer offers a logical separation point. The purpose of modern transport protocols is to provide either a data-stream service or a data-gram service. The data-stream service is typically a reliable connection-oriented transport service while the data-gram service is usually an unreliable connectionless transport service. A transport protocol in conjunction with an appropriate network protocol is able to offer one of these services to transport users. A transport protocol engineered for *end-to-end* reliability is the service of choice on packet-switched networks because of out-of-order or duplicate packet delivery, packet loss due to congestion, or packet checksum errors. From the perspective of the application layer, communication issues such as flow control, bit error detection, and physical data transfer mechanisms are not typically of concern. As noted by Salzer, Reed, and Clark, end-to-end reliability issues permeate all layers [152]; however, modern transport protocols provide sufficient end-to-end reliability mechanisms for basic data transfer.

From an implementation point of view, the reasons for the separation at the transport layer are as follows:

- Transport and network protocols typically reside in the kernel space of an operating system while the upper layer protocols reside in user space.[1]

- Multi-protocol systems require the capability to flexibly instantiate diverse transport/network protocol combinations to meet various quality of service requirements. A multi-protocol architecture requires that the traditional layered model be augmented with a "virtual" transport layer that offers both a *transport switching* and *protocol conversion* service mapping the upper layers onto one of possibly many transport protocols.[2]

The familiar layered diagram conveys little information beyond illustrating the separation of services. A critical observation about a layered model is that while a particular layer has knowledge of the services provided by the adjacent lower layer, a lower layer has little knowledge, and can make few assumptions, about the operation of the layer(s) above.

[1]Note that transport and network protocols can reside in user space as in Mach [129]. However, the Mach NetIPC implementation has better performance when kernel resident [137].

[2]The transport switching and conversion service is technically a session layer service, but it is useful to think of the switching and conversion functions as belonging to a separate "virtual" transport layer. Alternatively, one might call this layer the *transport convergence layer*.

That is, the *downward* relationship between layers is well-defined in terms of the service primitives offered, but the *upward* relationship is ill-defined. A layered model of this form dictates a bias in the way information is exchanged between adjacent layers. As a practical matter, this bias often manifests itself as a breaking of layer encapsulation in an implementation since a lower layer must have some structured manner of interacting with the layer above. The structuring of upward layer interaction is a practical problem, or "local matter", not in the specifications defining the services and protocols of the ISO Reference Model.

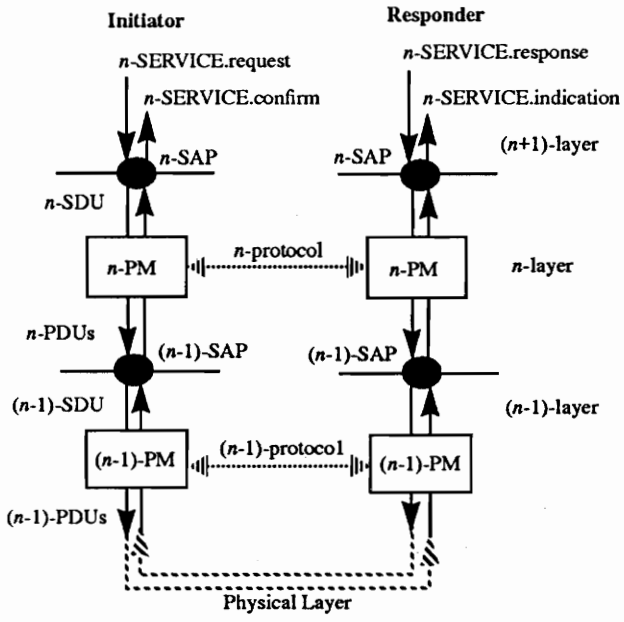

Figure 6.2: Peer-to-Peer Interaction.

A sufficient number of abstractions are provided at the application layer such that the application designer need not be specifically aware of how lower layer services are implemented. The diagram in Figure 6.2 is a depiction of the model of interaction between two communicating peer entities.[3] From the perspective of the network application designer, the goal is to define and develop a network **application process**. From the perspective of

[3]Adapted in part from Svobodova [166].

an operating system (e.g., UNIX) a network application process is like any other large-grain, schedulable computation. Internally, an application process is a collection of **application entities**, each representing some communication aspect of the application process. From an object-oriented perspective, application entities might be objects, or collections of objects.

An **association** is an application layer binding between two peer application entities. The entity requesting an association is called the **initiator** and the entity accepting the association is called the **responder**. An association does not necessarily imply client-server semantics following the initial binding of an initiator and a responder. Once an association has been initiated and accepted, a binding of application entities occurs and a true peer relationship exists. Of course, the specific application protocol used between the bound entities may choose to enforce client-server semantics—such a decision is left to the application process designer. The local functions of an application entity are concerned with requesting remote operations from a peer application entity residing on another node, or implementing local operations that are made available for execution by a requesting peer.

There are three phases to an application association. In the **connection establishment** phase, an initiator performs a bind operation to establish an association with a responder. Connection establishment typically implies *negotiation* of a set of functional subsets that will be supported during the association, as well as initial assignment of any tokens used to control the two-way dialogue. If the bind operation is successful, then the **data transfer** phase is entered during which either the initiator or the responder, depending on the semantics of the application protocol, may request specific operations from one another. At some point in time, either the initiator or responder initiates the **connection release** phase by performing an unbind operation, thereby terminating the association.

Peer-to-peer interactions are *indirect* and result from *direct* interactions between vertically adjacent layers. Indirect interaction occurs between peer entities residing at the same service layer that are *behaviorally* related by a common set of rules defining a specific **protocol**. With one exception, indirect interaction occurs only as a consequence of direct interaction.[4] Direct interaction occurs between a service **user** and a service **provider**.

Service users and providers are related by the existence of a functional dependency for

[4]The one exception (stopping short of quantum electrodynamics) being the electrical interaction at the physical layer.

service. The **service access point** (SAP) is the abstraction used by a user to issue a **service request** to a service provider. An n-SAP serves as an interface point for invoking the n^{th} request in a sequence of requests that satisfy some need for service. A service request typically consists of some number of mandatory and/or optional arguments as well as user data in the form of a **service data unit** (SDU). A provider satisfies a service request by indirectly interacting with its peer using a **protocol machine** (PM) that implements the requested service using a subset of its behavior specification. A protocol machine communicates the behavior associated with a service request to its peer using an *a priori* protocol definition. This indirect communication is effected by issuing another service request to the adjacent lower layer, and so on down to the physical layer. As part of a service request, a PM *synthesizes* a **protocol data unit** containing the **protocol control information** (PCI) representing the behavior change required at the peer entity, along with any user data provided as part of the service data unit. This process continues down to the physical layer at which point some physical manifestation of the data is transmitted, via an appropriate signaling channel, from the machine on which the initiator resides to the machine on which the responder resides.[5]

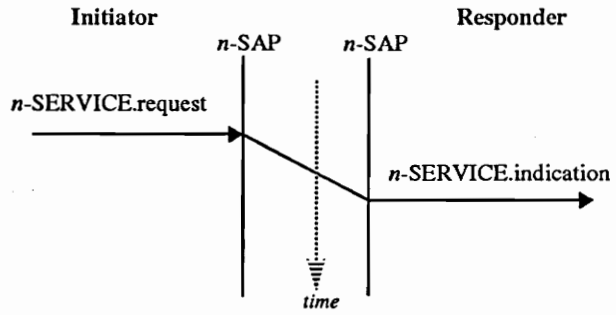

Figure 6.3: Unconfirmed Service Event-Time Diagram.

There are two types of service request interactions between initiators and responders: *unconfirmed* and *confirmed*. The unconfirmed, or unacknowledged, service interaction is depicted by the event-time diagram of Figure 6.3. The confirmed, or acknowledged, service

[5]Shannon [154] defined the basic theory, which is beyond the narrative scope of this work.

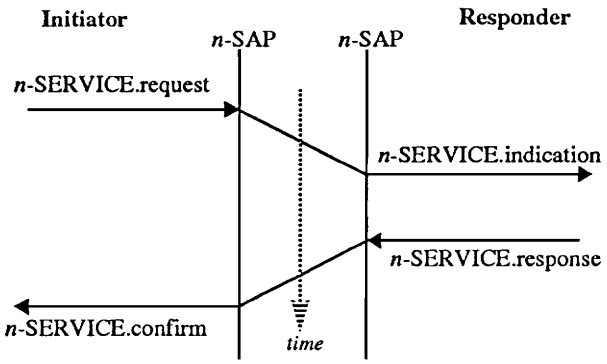

Figure 6.4: Confirmed Service Event-Time Diagram.

interaction is depicted in Figure 6.4. The dual of a service request is an ordered sequence of **service indications** terminating at the peer of the entity that initiated a service request. An $(n+1)$-layer responder receives an n-service request in the form of an asynchronous indication event. An n-service indication is handed up to the n-SAP from the n-PM as a result of the n-PM itself having received an $(n-1)$-service indication from the $(n-1)$-layer. Each layer *analyzes* the SDU provided as part of the indication, extracts the PCI from the PDU, and passed the remainder of the SDU to the next higher layer in the form of another indication. Each indication in the sequence initiates behavior modification of the PM at each layer as prescribed by the information contained in the PCI. If a service request is for a confirmed service, the responder invokes a **service response** operation that is eventually received asynchronously by the initiator as a **service confirm** event. Confirm events effect layer state changes in the same manner as indication events.

The layered model presents a desirable set of abstractions; however, the implementation of the protocol layers presents a challenge since protocols must be efficiently realized. The remainder of the chapter presents a modest contribution towards satisfying the structural and performance challenges posed by the upper layer OSI protocols. The principal focus is on a new view of the organization of the protocol stack. The following section presents the implementation structuring problems presented by the upper layers, reviews current research efforts at solving the structuring problem, and then introduces a new structuring approach based on polymorphic types and type inheritance resulting from a non-monolithic view of the protocol layers.

6.3 The Problem with Layered Protocols

Protocol specifications define the service model and specify the details of the protocol for each layer, but leave most implementation issues undefined. Formal implementation agreements, or profiles, augment the basic specifications by providing implementation guidelines for those areas that are vague or left to interpretation. The general problems facing an implementor of an OSI protocol stack, such as multiplexing, PDU processing, etc., are outlined by Svobodova [166]. Rose gives a detailed treatment of the pragmatics of OSI using the ISO Development Environment (ISODE) as a reference implementation [149].

Research on the implementation and performance tuning of the Internet protocols has been done at the transport layer and below with regard to the tradeoff between structure and efficiency [37, 42, 178]. The experiences gained with the transport/network layer implementations in the Internet translate well to both the OSI lower layers and upper layers. In particular, protocol engineers know that excessive copying of data severely affects performance, as does intra-layer multiplexing [170]. In general, there has been little attention paid to the tradeoff between structure and efficiency in upper layer protocol implementations. In part this is due to the fact that development of new types of OSI-based applications is lagging and in part because a lot of energy has been spent on just making OSI work, in some cases minimally, with traditional applications.

Clark makes the following comment regarding the conflict between structure and efficiency in protocol implementations [35, p. 1]:

> Modularity is one of the chief villains in attempting to obtain good performance, so that the designer is faced with a delicate and inevitable tradeoff between good structure and good performance.

A similar conclusion is echoed independently by Scherlis [153] with respect to structuring software for reuse using abstract data types. Considering the results reported by Clark and Scherlis, how does one go about structuring layered protocol implementations so that good performance is obtained without casting aside good structuring principles and mechanisms, such as information hiding and ADTs? The structure-efficiency problem is further complicated by today's multi-protocol platforms that require *flexible* composition of protocols in order to meet the diverse needs of applications [39, 175].

The new thinking in protocol engineering is to be less myopic about the structure and

performance of the lower layers in isolation and refocus on the structure and performance of the upper layers, with specific concern for the end-to-end needs of applications. This change in focus is the result of the gradual recognition that the *dialogue-oriented* services offered by the upper layers will dominate protocol performance in future applications. Clark and Tennenhouse outline a set of requirements for this "new generation" of protocols with an emphasis on upper layer issues so that better *overall* protocol performance can be engineered [38].

As previously mentioned, the upper layers allow an application process to add structure to an otherwise unstructured communication channel. The structure is principally provided by presentation services that are used to represent and exchange application data using abstract types defined by a suitable representation language, such as Abstract Syntax Notation One (ASN.1) [93]. Other representation languages are common; for example, the Interface Definition Language (IDL) [106] and the External Data Representation language (XDR) [124].[6] An application will typically encode the data sent over a particular association into an appropriate transfer syntax. A transfer syntax commonly preserves type information and represents the data in a machine independent form. In general, the encoding and decoding of application data as part of normal data transfer is a memory intensive operation, which is exacerbated by bit stuffing and variable length encoding schemes, such as the Basic Encoding Rules (BER) [94]. An obvious assault on this problem is to define a more efficient encoding scheme, as was done by Huitema and Doghri [82, 83]. A more fundamental approach is proposed by Clark and Tennenhouse as part of the "new generation" thinking. The new generation approach acknowledges that data encoding/decoding is a dominate cost in the upper layers, despite efficient encodings, and proposes that the upper layers, including the presentation conversion function, be structured to facilitate pipelining of data from the network to the application. The goal is to provide better end-to-end performance from the perspective of an application using presentation layer services. Presentation pipelining is difficult to achieve because of the way transport protocols are currently engineered to handle the possibility for out-of-order delivery in packet-switched networks. Out-of-order

[6] It is beyond the narrative scope of this work to contrast the representational power of the different abstract syntax notations and the performance of their encodings. The relevant issue is that data translation services must be provided in the upper layers so that arbitrary user-defined types may be constructed and exchanged as part of an association between *heterogeneous* peers.

packet delivery results primarily from latency due to alternate routes and/or retransmission of lost packets. Stream-oriented transport protocols buffer all packets until they can be reassembled into the proper order before handing them to the upper layers; hence, the presentation pipeline may be idle when in fact there is data that could be processed were the transport layer able to hand up partial, potentially misordered, pieces. Layering complicates the problem because a generalized transport layer has no knowledge of the semantics of application layer data with respect to logical boundaries, such as distinct arguments to a remote procedure. The fact that the data as seen by the transport layer is probably encoded in one of several possible transfer syntaxes, such as the BER, means that the transport mechanism cannot even begin to deduce application layer data boundaries.

Clark and Tennenhouse advocate two architectural principles that can facilitate pipelining presentation data: Application Level Framing and Integrated Layer Processing. The Application Level Framing principle suggests structuring the upper layers so that data transmitted in application specific aggregates, called Application Data Units (ADUs), can be presented to the upper layers by the lower layers without regard to the order in which the ADUs were sent. The objective is to structure the upper layers so that presentation processing (e.g., decoding) can be pipelined. The Integrated Layer Processing principle suggests structuring protocol layers so that both inter-layer and intra-layer processing is tightly coupled, while maintaining ordering constraints imposed by the layers. The ultimate goal is to implement an upper layer protocol structure that "preserves the benefits of [layer] isolation, while increasing the range of implementation options available to end system designers" [38, p. 206].

Several perspectives exist on the problem of how to implement layered protocols in order to simultaneously gain modularity, efficiency, and compositional flexibility. At one extreme, the *formalists* might argue that correct and efficient protocol implementations should be automatically or semi-automatically generated from structured specifications. At the other extreme, the *pragmatists*, hardened by practical experience, might argue that there is no silver bullet—efficient protocol implementations are hand-crafted by crafty network programmers. Somewhere in-between are the newly emerging *structuralists*, who believe that it is possible to realize modular, efficient, and flexible protocol implementations by applying sophisticated structuring mechanisms (such as polymorphism and inheritance) combined

with pragmatically-based engineering principles (such as Integrated Layer Processing).[7]

Regardless of the approach taken, the overall objective is to engineer a balance between structure and performance. The following performance-related characteristics are required of a protocol implementation:

- efficient implementation of protection boundaries implied by layer abstractions.
- efficient synchronization of bidirectional, asynchronous, inter-layer control flow.
- minimization of inter/intra-layer data manipulation.
- flexible composition of functional subsets.

The following subsections elaborate on each requirement. Subsequently, competing upper layer protocol structures are analyzed with respect to how each requirement is realized.

6.3.1 Implementation of Protection Boundaries

Ideally, an implementation should reflect the mental model one has of the architecture that is being implemented. The obvious software engineering benefits are understandability and maintainability. However, as noted previously, layering may result in adverse performance costs if the layer abstraction is naïvely implemented. The purpose of layering is to define a set of abstractions that are implemented by a set of protection boundaries. For example, a service access point (SAP) might be implemented using an abstract data type that encapsulates layer specific data. In the case of a SAP, language scope rules associated with the ADT mechanism guarantee the necessary level of protection and are often efficiently realized. An important consideration is that if an abstraction representing an interface such as a SAP is subjected to *over-generalization*, then performance suffers [81]. Over-generalization occurs when algorithms or structures are generalized to the point at which the control logic for distinguishing specific cases obviates the benefit of the generalization. In particular, over-generalization of the concept of a "layer" without regard to the specific functions performed by that layer may lead to an ill-conceived implementation that

[7] Brooks [26] outlines the essence of general software construction, which suggests a structuralists approach.

performs excessive copying of data and requires multiplexing of SAPs, both of which are known to have a highly negative impact on protocol machine performance [37].

The principal purpose of the SAP abstraction is to provide a point of entry for a specific application entity requesting a specific function offered by a service layer. The actions that are likely to be performed by a SAP consist of basic error checking of arguments to a request, such as SDU size, and enforcement of layer synchronization control, such as mutual exclusion. In a language supporting efficient abstract data types, the SAP might be represented by an ADT consisting of a collection of inlined procedures. For example, in an object-oriented language a SAP could be implemented as a class with a method interface defined inline. An inlined interface for a SAP maintains abstraction and protection boundaries without incurring the cost of a procedure call, but with a potential for increased code size due to inline expansion.

6.3.2 Synchronization of Bidirectional Control Flow

Bidirectional control flow through a protocol stack must be synchronized so that service requests initiated by a user do not interfere with asynchronously occurring service indication or service confirmation events generated by peer activity. A *downcall* is a downward flow of control initiated by a service request invoked by a user at a SAP. An *upcall* is a upward flow of control initiated as the result of asynchronous network events. The inherently asynchronous nature of peer-to-peer communication dictates that downcalls and upcalls have mutually exclusive access to the protocol stack since each protocol machine is a critical section. The structure of the synchronization control mechanism is dependent on the manner in which the protocol stack is implemented. If the layers are not tightly integrated, then it will be necessary to request mutual exclusion upon entry to each layer since a lower layer, lacking specific knowledge about its user, can not assume that mutual exclusion has already been acquired. If the Integrated Layer Processing principle is employed, meaning all layers are tightly integrated, then mutual exclusion once acquired should apply to the entire protocol stack for the duration of an upcall/downcall flow of control.

A common technique for implementing a critical section in UNIX is to disable selected process-level interrupts (e.g., I/O signals) from occurring, thereby disabling process notification of asynchronous indication or response events. Disabling and enabling interrupts

using this technique to effect mutual exclusion requires, at a minimum, two calls to the kernel.[8] The disadvantage of this approach is that control flow through the protocol layers is effectively serialized. A synchronization policy based on disabling/enabling process-level I/O interrupts implies that a kernel resident transport protocol is further restricted from concurrently handing up multiple application data units to the same application entity to facilitate presentation pipelining.

Alternatively, in a multi-threaded process model it should only be necessary to block a thread associated with a downcall/upcall on a user-level semaphore or condition variable rather than completely disabling process-level interrupts.[9] The advantage of the multi-threaded approach is that the synchronization policy of the protocol stack need not be dependent on the process signaling and scheduling discipline of the operating system. Furthermore, the potential for a high degree of concurrency is maximized rather than artificially limited by a single-thread process model. A protocol structure facilitating multi-threaded upcalls/downcalls has the potential to support concurrent processing of application data units, thereby facilitating the pipelining of the presentation layer data conversion function.

6.3.2.1 Downcalls

A downcall corresponds to an ordered sequence of procedure invocations through the protocol layers. A downcall eventually initiates a system call (syscall) at the virtual transport layer and returns to the continuation point associated with the procedure invocation that initiated the downcall. A downcall may be either blocking or non-blocking. In UNIX there are three primary situations that cause a downcall to block:

- a connect syscall blocks waiting for a positive/negative response.
- a write/send syscall blocks due to congestion (e.g., flow-control).
- a read/receive syscall blocks when no data is available.

In all cases, a downcall that blocks as a result of a blocking syscall will block the entire process. An exception is if the downcall is implemented using a threads capability that only

[8] Assuming Berkeley UNIX signal semantics [112], `sigblock`/`sigsetmask` to disable/enable interrupts.

[9] It is important to distinguish between a real multi-threaded process model, as in Mach's *task* model and Solaris' *lightweight process* model, and a simulated one using a suitable user-level threads package such as POSIX threads [85].

blocks the thread that initiated the blocking system call.[10] For this reason, non-blocking downcalls are of primary interest because they facilitate concurrent applications. If blocking downcalls are used and the kernel does not support a multi-threaded process, then all user-level threads are unable to make progress while a network operation is in progress.

If downcalls are non-blocking they must be supported by a user-level mechanism that facilitates the initiation of a confirmed service request that does not complete immediately. The result of a non-blocking downcall operation is a handle to a *future indication*, either a confirmation or an abort indication. The polymorphic future type introduced in Chapter 4 is an ideal mechanism for representing the result of a confirmed service request, such as the presentation connect request primitive (P-CONNECT.request). For example, a user might initiate a non-blocking connect request at a presentation SAP (PSAP) and receive as the return value a future indication object representing the outstanding result to be supplied by a connect confirmation. As explained in Section 4.3, the user thread may continue execution by binding the future result to a future object as follows:

```
Psap psap;
Future<PSapIndication> future = psap.PConnectRequest(...);
  ⋮
PSapIndication result = future;      /* obtain result, possibly blocking */
```

At some later point in time, the future is satisfied by either an asynchronous abort indication or an asynchronous connect confirmation. In the case of an abort (e.g., connect timed out), the future contains a PSAP abort indication object indicating the reason for the abort. In the case of a connect confirmation, the future contains a PSAP connect confirm object containing peer information confirming the requested association. A blocking connect request can be effected by forcing future evaluation at the time of the call, i.e.:

```
PSapIndication result = psap.PConnectRequest(...);      /* blocks */
```

The invocation implicitly blocks as part of the type conversion of the future returned by the non-blocking connect request.

[10] In at least one POSIX threads implementation [130], a thread will block but the I/O system call is made in non-blocking mode so that the entire process is not blocked. The blocked thread eventually awakens and is rescheduled when the non-blocking I/O operation completes.

In the case of an unconfirmed service request, the result of a downcall sequence is simply an indication of the success or failure of initiating the service request. Typically, an unconfirmed data transfer request initiates a non-blocking network write operation; if all of the data cannot be written because of congestion, the virtual transport layer has the responsibility of queuing the data and then returning immediately to the caller. At some later point in time, when the condition(s) causing congestion have abated, an attempt can be made by the virtual transport layer to write the remaining data independent of the thread of control that initiated the original downcall.

6.3.2.2 Upcalls

Asynchronous network events must be handled by a mechanism that can initiate an independent control flow resulting in an ordered sequence of procedure invocations upward through the protocol stack. The idea for structuring upward control flow in protocol implementations using upcalls originated with Clark [36]. From a programming methodology perspective, the upcall is perhaps the most important contribution to network programming to date. The utility of structuring protocol implementations using upcalls has already been demonstrated [12, 31].

Ideally, an asynchronous network event is scheduled as an independent thread of control that is used by the upcall to initiate an ordered sequence of operations upward through a protocol stack instance, starting at the virtual transport layer. The upcall terminates at a point determined by the user of a service access point. In the case of an upcall initiated as the result of a confirmation event for a confirmed service request, the termination point is the future indication object returned by the service request that is being confirmed, as in the non-blocking connect request downcall example.

The upcall programming methodology requires a language mechanism for allowing a higher layer to bind the local procedures that are components of upcall sequences to the adjacent lower layer. The typical way of achieving this binding relies on language support for defining higher-order functions. A lower layer protocol machine exports one or more binding operations taking procedures, or more commonly procedure addresses, as arguments to be bound into the upcall chain. The higher-order function technique is dynamic since the upcall function bindings are made a run-time. An alternative is to statically define the

upcall bindings via a dispatch table that is constructed at compile time. For example, the method dispatch table of a class created at compile-time by a compiler can be used as an implicit dispatch table for the upcall function chain.[11] The advantage of a method dispatch table approach is that the compiler does the work of creating the upcall chain, and it is therefore able to do better type checking. In the case of higher-order procedures, it generally is not the case that arguments to an upcall procedure can be checked for type consistency since the invocation is accomplished through run-time dereferencing of a untyped or weakly-typed (e.g., `void*`) pointer.

6.3.3 Data Manipulation

Excessive manipulation of data is perhaps the most significant contributor to poor performance in protocol machine implementations [37]. A upper layer protocol stack must be implemented in a manner that eliminates redundant manipulation of SDUs, PDUs, and the state information maintained by SAPs and PMs.

Excessive data copying often arises in an implementation that over-generalizes the abstractions representing the protocol layers. If each layer is generalized as a general "object" (e.g, a process) that receives service requests in the form of asynchronous "messages" (e.g., IPC/RPC), then it is often the case that data will be buffered as part of each inter-layer service request. The ideal case is that all data to be written to the network as part of a downcall is referred to by a vector of descriptors that are incrementally constructed as control passes through the layers. The descriptor vector is subsequently passed as an argument to a system call that *gathers* the data from user space and writes it to a network device. Similarly, a network read operation *scatters* data read from a network device to a set of memory locations identified by a descriptor vector. Scatter/gather input/output operations significantly reduce the amount of extraneous copying of data.[12]

A less obvious aspect of data manipulation performance is the impact of *type rediscovery*. Type rediscovery occurs each time it is necessary to recompute type-related information that was previously known, but which was subsequently "lost" as the result of crossing an abstraction boundary, such as a procedure. Weak type abstractions imply that type in-

[11] For example, the `vtable` in C++ [58].

[12] For example, the `readv`/`writev` system calls effect scatter/gather I/O in Berkeley UNIX.

formation that was once known is not "carried" across abstraction boundaries and must therefore be recomputed. The most obvious occurrence of information loss in protocol machines is in handling PDUs. When a PDU is received, its type is determined by examining protocol control information (PCI) in the header. In many protocol implementations, PDUs are represented by data structures that are part of a containing discriminated union. The type of a PDU is determined by an identity field in the PCI and the id is used to discriminate which of the enclosed PDU data structures is valid. The purpose for using the discriminated union is to allow "overlaying" a structure onto a sequence of octets read from the network. Overlaying eliminates copying the information from the octet sequence representing a PDU into a more structured representation. Within a protocol machine, it is common to find that each site that performs PDU processing must discriminate the type of a PDU using case analysis or table lookup based on the id extracted from the PCI. If PDU processing is not totally centralized, then it is often the case that the process of "discovering" the type of a PDU is repeated at different processing locales. Rediscovery of type information pertaining to a PDU can be avoided if a *type-subtype* PDU structure is implemented. If an implementation were to instantiate an explicit PDU subtype upon initial discovery of the PDU's identity, subsequent non-centralized PDU processing could be type-driven; i.e., organized around polymorphic procedures that are parameterized, and hence dispatched, based on PDU type information. If PDU processing is based on subtypes, then type discrimination need not be explicitly performed since a subtype implicitly "carries" the identity of the PDU, avoiding the need to recompute information that gets "lost" when an abstraction boundary is crossed.

6.3.4 Flexible Composition

Flexible composition refers to the ability of the programmer to control which functional components are bound into a protocol stack instance. Each functional component of a protocol specification may be viewed as a micro-protocol that incrementally extends the basic *kernel* protocol offered by a service layer. For example, the OSI session service is defined as a collection of *dialogue control* functional units, identified in Table 6.1, that extend a basic kernel functional unit offering only connection establishment, normal data transfer, user/provider abort, and orderly release services.

Table 6.1: Session Functional Units.

Functional Unit	Requires	Tokens	Added Service
Kernel	Transport Service	—	session connection normal data transfer orderly release user/provider abort
Duplex	Kernel	—	no additional service
Half-duplex	Kernel	data	give/please data token
Exceptions	Half-duplex	—	user/provider exception reports
Typed Data	Kernel	—	typed (control) data transfer
Expedited Data	Kernel	—	expedited data transfer
Resynchronize	Kernel	—	resync dialogue control
Symmetric Synchronize	Kernel	—	resync dialogue control
Minor Synchronize	Kernel	minor	minor sync dialogue control give/please minor token
Major Synchronize	Kernel	major	major sync dialogue control give/please major token
Activity Mgmt	Kernel	activity	activity start/discard/end activity interrupt/resume give control give/please activity token
Capability Data	Activity Mgmt	—	capability data exchange
Negotiated Release	Kernel	release	negotiated orderly release give/please release token

The OSI presentation service extends the session service by "mirroring" the session functional units as well as adding additional services for conversion of data into a suitable transfer syntax. For a given association, some subset of the functional units will be active. A basic implementation will allow the application designer to request that a functional subset be negotiated for use on a particular association. A sophisticated implementation would also allow the application designer to restrict a protocol stack to only the code and data required to support the negotiated set of functional units, thereby minimizing unnecessary processing and storage space overhead.

There are at least two techniques for configuring a flexible protocol stack: dynamic protocol configuration and static protocol configuration. In the dynamic approach, the functional subset required for an association is determined completely at run-time and dynamically configured. A dynamic configuration implies both either preallocation of all functional units or dynamic binding of code and data. A dynamic configuration suggests that generalized control logic must exist in order to navigate a dynamically created protocol graph. A static configuration implies that the functional requirements of an application can be predetermined at compile-time. If the required functional subsets can not be successfully negotiated at run-time, then the association binding fails. The static approach is well-suited to applications in which the functional requirements for communication are relatively fixed. The advantage of the static approach is that optimizations can be performed since only the code and data of the components required by the application are bound into the application and the control logic for navigating the protocol graph is fixed.

A hybrid configuration approach can be defined that uses a statically determined structure, but which performs late binding to configure negotiated functional subsets at connection establishment time. This hybrid approach facilitates compile-time optimizations without sacrificing the flexibility required for dynamic negotiation at connection establishment time. Although the full generality of a dynamic configuration is not achieved, the absence of extra control logic required for navigation of a dynamic protocol graph suggests less complexity (e.g., less conditional branching) and better performance as the result of stream-lined execution.

One aspect of a flexible upper layer protocol implementation is that the interface to the kernel based transport layer is via a programming library that generalizes the transport

service interface.[13] The service interface generalization is required in order to gain some independence with respect to multiple transport/network protocol combinations that may reside in the kernel. Depending on the desired transport characteristics and the underlying subnetwork technology, the following combinations are prevalent:

- Transmission Control Protocol [145] over the Internet Protocol [144] (TCP/IP).
- User Datagram Protocol [143] over the Internet Protocol (UDP/IP).
- Transport Protocol Class 4 [86, 87] over the Connectionless Network Protocol [88, 89] (TP4/CLNP).
- Transport Protocol Class 0 [86, 87] over the X.25 connection-oriented network protocol [90, 91] (TP0/X.25).
- TP0 over TCP via RFC 1006 [151].[14]

The virtual transport layer introduced previously in Figure 6.1 is specifically defined to deal with the flexible instantiation of multiple transport/network protocol combinations while simultaneously using one or both of the common transport interface libraries. In addition to mapping a session service onto a particular transport layer interface, a principal function of the virtual transport layer is *protocol conversion*, which is a technique for achieving interoperability among disparate network protocols.[15] For example, the mapping of TP0 onto TCP/IP is via the convergence protocol defined by RFC 1006.

Technically, the virtual transport layer is implemented by a hybrid protocol machine, often called a **transport switch** [149], that allows the session service to bind a particular transport/network service combination at connection establishment time based on the quality of service and interoperability requirements of an application. In subsequent sections, the machine at the virtual transport layer will be called a **virtual transport protocol machine** (VTPM) rather than a transport switch because the function of the virtual transport layer includes protocol conversion as well as transport switching.

[13] For example in UNIX, either Berkeley sockets [112] or the System V streams-based transport interface (TLI/XTI) [167, 168].

[14] TCP/IP is used as a connection-oriented network layer service to provide end-to-end reliability for the otherwise unreliable service offered by TP0, thereby enabling the use of the OSI upper layers over IP-based networks, such as the Internet [147].

[15] Lam provides a formal methodology for describing a general protocol conversion process [105].

6.4 Structuring the Upper Layers

The structure of an upper layer implementation should be modular, efficient, and flexible. The traditional structuring approach is based on a *monolithic* protocol stack that closely resembles the layered model, where each layer is implemented by a multiplexing protocol machine. A challenge is to define a new structure that reflects the conceptual layering of the layered architecture, but does not hinder performance while maintaining modularity and flexible composition. A new structuring approach is proposed that is based on a *non-monolithic* protocol stack that syntactically reflects a layered structure, but which is compiled into a tightly integrated protocol machine structure at run-time.

6.4.1 The Traditional Structuring Approach

The monolithic protocol stack depicted in Figure 6.5 directly reflects the upper layers of the ISO Reference model, augmented with the virtual transport layer. Each layer consists of a monolithic protocol machine that either multiplexes or serializes service requests made at a collection of service access points. Each request in turn is mapped into another service request at a corresponding SAP at the next lower layer, and so on. For example, at the presentation layer, multiple PSAPs are bound to a monolithic PPM that implements the presentation protocol by mapping service requests made at a particular PSAP onto a corresponding SSAP.

A choice can be made on how to implement each protocol layer. If each layer is implemented as a separate heavyweight process, a significant performance penalty is incurred due to the cost of context switching and data copying required for inter-process communication. An IPC is performed per layer for each request, indication, response, and confirmation event. Each IPC requires a context switch and copying of request arguments and a possible result from one address space to another. In general, the "process-per-layer" approach is counter to the Integrated Layer Processing principle advocated by Clark since inter-layer communication costs using IPC for each layer service transaction is relatively expensive. For this reason, the multiple process protocol stack approach is not a particularly good implementation technique for the monolithic protocol stack if high performance is required. One can argue, as Bershad does [15], that inter-process communication performance is becoming less of an issue in new micro-kernel operating systems that support high-performance

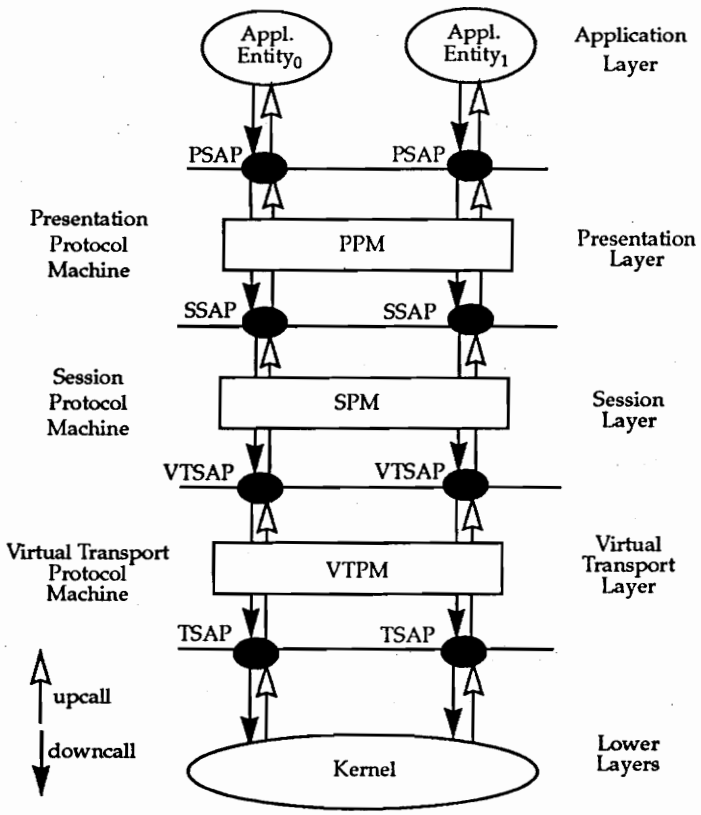

Figure 6.5: Traditional Monolithic Protocol Stack.

IPC/RPC. For example, Mach IPC is efficiently implemented using ports, memory objects, and continuations [57]. Other distributed operating systems, such as the V kernel [32], the x-kernel [84], and Amoeba [132, 177] also provide high-performance IPC/RPC. The only significant advantage of a process-per-layer structuring approach is the degree of real concurrency that can be realized on a multi-processor architecture. However, the sum cost of multi-layer IPC interaction is prohibitive, especially when one considers a good single process implementation augmented with kernel support for lightweight threads. It is generally more sensible to implement the entire upper layer structure in a single process and restrict IPC communication to only the user-kernel boundary at the virtual transport layer.

The ISODE is a widely-used research implementation of the upper layer protocols [150]. Architecturally, ISODE is a single process, single threaded, monolithic protocol stack. In an ISODE application, only one application layer SAP is active at a time, so guaranteeing mutual exclusion simply requires prohibiting any upcalls from starting as the result of asynchronous network activity. A particular service primitive is implemented as a pair of procedures, one corresponding to a SAP operation and the other to a PM operation. Layer protection is achieved by using file scope to restrict access to a PM procedure to only those SAP procedures defined within the same file scope. Within a specific source file, a SAP procedure is defined with global scope and a PM procedure is defined with local scope.[16] The collection of all globally defined SAP primitives for a layer are visible to applications. The locally defined PM primitives are not visible. The combination of a SAP procedure and one or more PM procedures implements a specific layer service. A typical service request in ISODE performs the following steps as part of a downcall:

- SAP procedure:
    1. verify preconditions.
    2. disable I/O interrupts.
    3. lookup the PM control block corresponding to this association.
    4. invoke the corresponding PM procedure.
    5. enable I/O interrupts.

[16] Global/local scope is accomplished using paired `extern/static` procedure declarations within the same file scope.

6. return result of PM operation.

- PM procedure:

  1. verify that the correct state exists for this operation.
  2. construct the necessary PDU(s).
  3. invoke a lower layer service request.
  4. update the local state, returning success or failure.

Precondition verification consists primarily of validating that mandatory arguments are present and that SDU size limitations are not exceeded. Mutual exclusion is guaranteed by disabling I/O interrupts on downcalls, while relying on the UNIX kernel to buffer network I/O events that occur while interrupts are disabled during an upcall. This *global* synchronization policy effectively serializes access through the monolithic protocol structure. A global synchronization policy presents a severe problem in applications that wish to define their own policy for handling I/O events, since the protocol machinery uses I/O interrupt control as the pillar of its global synchronization policy.[17] In a concurrent application, a global synchronization policy limits the degree of concurrent activity since control flow through the protocol machinery must be serialized. A unique identifier is used as the key to locate the control block representing the PM state for the SAP being serviced.[18] The verification of the correct state condition typically includes determining whether or not the functional units that provide the service implemented by the operation were successfully negotiated during connection establishment, whether the proper tokens are currently held, and whether or not outstanding operations preclude this operation, such as waiting for a confirmation.

An advantage of a global synchronization scheme is that data manipulation is minimized since there is no need to buffer data within the protocol layers since the entire monolithic structure is treated as a critical section. Since process-level interrupts are disabled upon entry to the protocol stack, all protocol layers can be traversed atomically without having to buffer data—simple references can be used. In addition, no other thread of control can

[17] For example, an Actor system consisting of communication actors, window actors, database actors, etc., that do more than just network-centered I/O.

[18] A UNIX file descriptor (an integer) is used as the key in ISODE.

inadvertently discard the user data being sent to the network. At the virtual transport layer, an attempt is made to perform a gather-style write. If congestion prohibits all data from all gather locations being written simultaneously, the unwritten data can be copied into an outgoing write queue and another write attempt made when the congestion condition abates. PDU processing however relies on discriminated unions that are used to overlay memory; hence, extensive case analysis code is required to explicitly discriminate PDU subtypes. To minimize type rediscovery, PDU processing is highly centralized, resulting in large monolithic sections of code.[19]

The ISODE supports flexible composition at the virtual transport layer by determining which protocol switch component to bind for a particular association at connection establishment time. An explicit method dispatch table is constructed as part of the VTPM state so that subsequent operations invoke the real transport interface operations. Subsequent operations are invoked by deferencing pointers to functions. In this respect, the ISODE implementation is partly dynamic and partly static since the transport bindings are determined at run time, but all possible bindings must be established statically at compile time for the later bindings to occur.

The following summarizes how the performance-related requirements, outlined previously on page 141, are implemented in the ISODE:

layer protection is implemented using the `extern/static` procedure attributes in conjunction with C language file scope rules to export SAP primitives and hide PM primitives.

bidirectional synchronization is effected by a global synchronization policy that serializes all upcalls and downcalls through explicit control of the I/O interrupt mechanism using at least two system calls.

data manipulation is minimized because atomicity is guaranteed while traversing the protocol stack as part of an upcall or downcall. Type rediscovery occurs frequently since PDUs are represented by discriminated unions requiring case analysis code at PDU processing locales.

[19]For example, the session layer PDU processing consists of a monolithic loop containing nested case analysis using gotos that is generally unreadable without detailed knowledge of the structure of the PCI portion of all session PDUs and the protocol rules.

flexible composition is limited to the transport switch mechanism which is able to simultaneously support different transport/network protocol combinations within the same application process (i.e., TP4/CLNP, TP0/X.25, and TP0/TCP).

6.4.2 A New Structuring Approach

This section introduces a new approach to constructing a modular, efficient, and flexible upper layer protocol stack. The new approach is based on a *non-monolithic* view of the protocol stack that is derived from a *vertical partitioning* of protocol machines, as depicted in Figure 6.6. Vertical partitioning has been shown to have a positive influence on performance [80, 81] and complements the principle of Integrated Layer Processing.

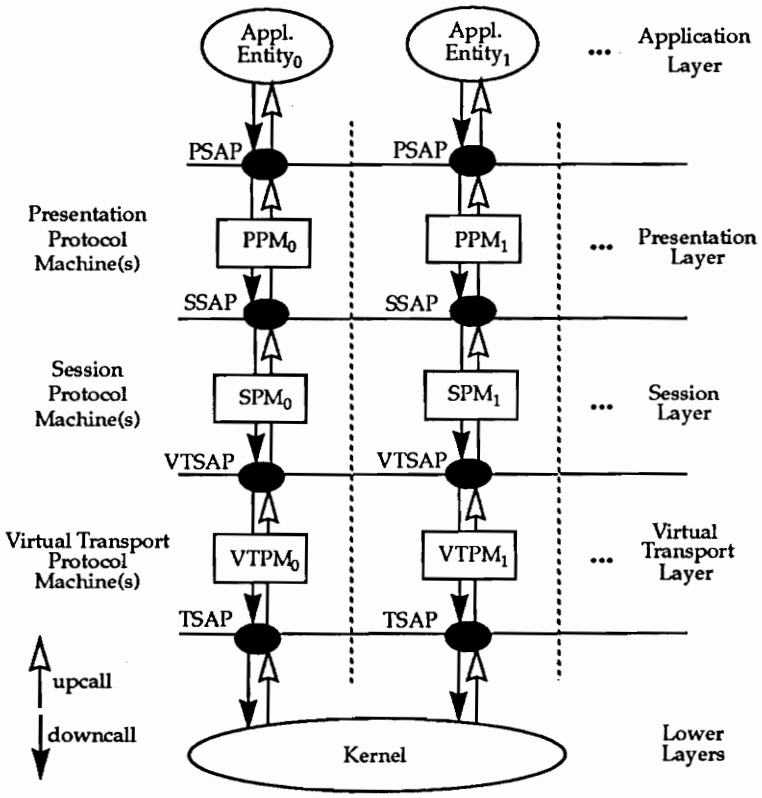

Figure 6.6: Non-Monolithic Protocol Stack

A primary advantage of a vertically partitioned protocol stack is that it encourages a *local* synchronization policy, making it well suited to concurrent applications. The vertical partitioning of the upper layers creates independent collections of vertically integrated protocol machines that are bound above by a single application entity and bound below by a single transport service access point into a kernel. The one-to-one binding enables a simplified synchronization structure that need not perform intra-layer multiplexing nor global serialization. Furthermore, the service access points *between* machines in the protocol stack can be effectively eliminated since their function is greatly reduced. The advantages derived from the local synchronization policy are that:

- mutual exclusion can be implemented using a simple binary semaphore, and
- adjacent protocol machines may be tightly integrated.

In the monolithic protocol stack, a particular SAP is uniquely identified so that the monolithic protocol machine can multiplex among its users. The non-monolithic approach eliminates SAP multiplexing since a complete protocol stack instance is allocated on a per association basis. The protocol stack is implicitly identified by the "upper most" application entity to which it is bound. Eliminating the requirement to multiplex SAPs is a result of the localization of the synchronization control policy. Simplification of the synchronization control policy arises from the fact that each slice is in one-to-one correspondence with an association. The fact that each protocol machine never has more than one upper binding and one lower binding to contend with implies that a simple semaphore-based mutual exclusion mechanism is sufficient to guarantee consistency of the protocol layers. Mutual exclusion can be enforced using a semaphore that is local to each protocol stack instance, in contrast to the global interrupt-based mutual exclusion required in the monolithic structure. Since control flow through a particular slice is independent of control flow throw any other slice, multiple threads of control can execute concurrently without interference, in contrast to the serialization of control flow in the monolithic structure.

The synchronization scheme works as follows. When a downcall is issued by an application entity, say at a PSAP, an attempt is made to acquire mutual exclusion for the given protocol stack instance bound to the PSAP by requesting a lock on the semaphore. If mutual exclusion is granted, the downcall proceeds through all protocol machines without

requiring any additional synchronization, returns immediately, and releases the lock. When a network event occurs at the virtual transport layer as the result of peer activity, an I/O signal is issued by the kernel to the process in which a thread executing in the protocol stack is but one of possibly many application threads. A global I/O handler receives the interrupt and schedules an independent upcall thread of control for the protocol stack instance, starting with the virtual transport protocol machine. The upcall thread proceeds only if the corresponding VTPM instance is able to acquire mutual exclusion by locking the unique semaphore bound to the corresponding protocol stack instance. Mutual exclusion is acquired only if no downcall or upcall for the same protocol stack is in progress; otherwise, the upcall thread blocks on the semaphore associated with the protocol stack instance. The upcall/downcall activity occurring in other protocol stack instances, or other threads elsewhere within the same application are totally independent of one another. Hence, the degree of concurrency of an application is not artificially limited by reliance on process-level synchronization using a global semaphore, as in the case in the monolithic approach of ISODE. An additional efficiency advantage is that I/O interrupts can be handled by a very short code sequence for scheduling a new thread; hence, the time window within which the kernel must buffer additional I/O interrupts is minimized.

The assumption that two adjacent protocol machines can be vertically integrated depends on the existence of a mutex-enforcing SAP at the point at which a downcall sequence is initiated. That is, two protocol machines can be considered atomically composed only if some SAP at a higher layer guarantees the enforcement of mutual exclusion for the first procedure in a downcall chain. For example, each application entity in Figure 6.6 interfaces to the protocol stack at a PSAP, hence the PSAP must be able to access the semaphore associated with the protocol stack to enforce mutual exclusion for the entire duration of a downcall service request. Similarly, the VTPM must enforce mutual exclusion for upcalls that result from asynchronous I/O events occurring as a result of network or peer activity.

As with the global synchronization policy in ISODE, the local synchronization policy ensures that a descriptor vector may be used to pass data between layers. Copying and buffering of data is required only in the event of network congestion at the virtual transport layer. To minimize type rediscover, PDUs are arranged in a type hierarchy that allows all PDUs to be defined as subtypes of a generalized PDU. The use of subtypes allows

PDU processing to be type-driven; hence, the code that processes a particular PDU may be distributed rather than centralized since once a PDU is instantiated as a particular subtype, the implicit type information maintained by the compiler guarantees that type specific data manipulation operations are correctly applied.

The vertical structure also supports flexible composition of the protocol layers since each protocol stack instance can be uniquely tailored for the particular needs of the application entity that has instantiated the stack. For example, if only the session kernel functional unit is required for a particular association, the session protocol machine instantiated as part of the corresponding stack instance can be configured to contain only the kernel functional unit. If properly structured, the "mirroring" relationship between the presentation and session layers implies by transitivity that the presentation protocol machine is similarly limited in functionality to only those components required by the presentation layer kernel. The advantage of the vertical structure is that inter-layer control flow can be simplified as a result of exploiting the implicit transitivity between protocol machines in one-to-one correspondence. Similarly, the structuring of the virtual transport protocol machine can be driven by the knowledge that a one-to-one relationship exists with the protocols above and a single transport service access point at the user-kernel boundary, configured for one of the transport/network protocol combination listed previously.

A particularly novel benefit of the vertical structuring is that it is straightforward to establish multiple stack instance within the same process, with different components, over distinct or even the same transport endpoint. Multiple stack instances, if used properly by the application designer can effect presentation pipelining in support of the Application Level Framing principle. Figure 6.7 depicts two vertical stack instances sharing the same virtual transport machine

The application layer has the responsibility for creating the necessary stack instances and identifying which application data units are sent over each stack instance. For example, an application sending two large application data units could send both simultaneously to different stack instances bound to the same VTPM. A "sub-association" identifier could be transmitted with the transport data so that the receiving end is able to hand the application data to the appropriate stack instance. Alternatively, the VTPM could simply initiate a thread on the first available stack instance and let the application layer deal with

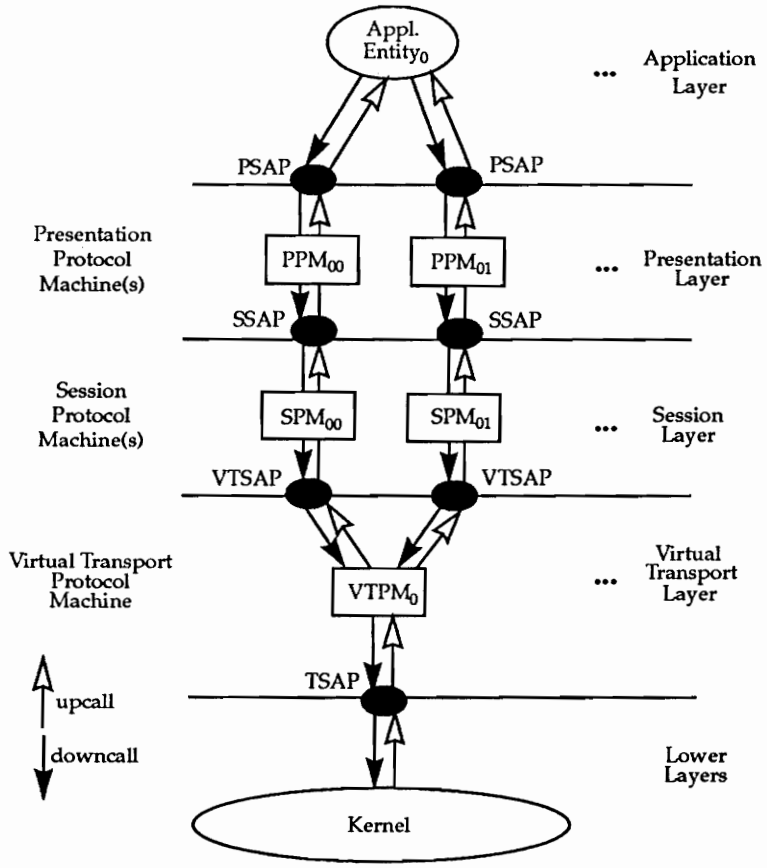

Figure 6.7: Pipelining Using Multiple Stacks.

identification. If the former choice is made, then a minor modification can be made to the RFC 1006 header to encode an identifier to allow the VTPM to multiplex incoming data.[20] The later case is perhaps more interesting since no modification is required to existing protocol definitions. The VTPM simply becomes a pipeline scheduler for incoming application data, handing the data to any available protocol stack to which it is bound. The choice of approach depends on the sophistication of the dialogue control required by the application. In general, an application can instantiate n stack instances, limited only by the number of process-level threads, but must be prepared to cope with non-deterministic arrival ordering of application data units.

In the following section, the essential features of a non-monolithic protocol stack implementation using object-oriented techniques to structure vertically integrated protocol machines is presented. The following summarizes how the performance-related requirements are implemented in this new structure:

layer protection is implemented using class constructs that use C++ class scope rules to provide public SAP primitives and private PM primitives.

bidirectional synchronization is effected by a local synchronization policy that uses a semaphore private to each protocol stack instance to serialize the upcalls and downcalls particular to that stack instance. The vertical separation of protocol stack instances permits multiple threads of control within a single process.

data manipulation is minimized because atomicity is guaranteed while traversing the protocol stack as part of an upcall or downcall. Type rediscovery is minimized by structuring protocol data units as subtypes rather than discriminated unions.

flexible composition is effected by defining functional unit subtypes arranged in a type hierarchy to support flexible instantiation of presentation and session protocol machines. Similar to ISODE, the virtual transport machine is able to simultaneously support multiple transport/network protocol combinations.

[20] The unused `reserved` field in the RFC 1006 header could be used [151].

6.5 The OOSI Implementation

This section presents the features of OOSI (*ooo-zi*), an application of object-oriented structuring techniques to a vertically partitioned protocol stack.[21] The non-monolithic, vertical structure depicted in Figure 6.6 suggests that object-oriented structuring techniques can be applied in a manner that will achieve modularity, efficiency, and flexibility in an implementation of the upper layers. Modularity is achieved by using the class construct to represent layer abstractions, such as SAPs and PMs. A class mechanism allows the representation of abstraction boundaries and the implementation of protection boundaries using language scoping mechanisms. Obvious candidates for inlining are the methods defined in the class representing SAPs. To minimize excessive code growth and enhance performance, it is necessary to be selective in choosing procedures to be inlined. A selective approach to method inlining leads to a reduction in the number of procedure calls at run-time without code size explosion. Restricting inlining to procedures with no local variables further reduces the possibility of performance degradation due to excessive register manipulation at run-time. To minimize over-generalization the following objects are created at run-time as part of a protocol stack instance:

- a polymorphic mutex-enforcing SAP object.
- a flexibly instantiated presentation/session protocol machine (PPM/SPM) object.
- a flexibly instantiated virtual transport protocol machine (VTPM) object.

A polymorphic mutex-enforcing SAP object represents the top-level interface to a presentation/session PM object. A polymorphic SAP permits instantiation of a PM object with only the functional subsets required. Hence, the presentation/session abstractions should be implemented in a manner permitting flexible composition, but which also represents the "mirroring" relationship between their respective functional units. Inheritance between layer functional units can be used to capture this relationship. A flexible VTPM object is required to permit flexible instantiation of the proper transport protocol conversion and switching functions.

[21] OOSI is an acronym for *objectified* OSI.

6.5.1 A Polymorphic SAP Type

A key feature of OOSI is that layer service access points are represented by polymorphic types parameterized by a protocol machine subtype. A protocol machine subtype represents some combination of functional units arranged in a type hierarchy. The entire type hierarchy implements a service layer. The type inheritance structure implicitly guarantees that all functional dependencies are met. The advantage of a polymorphic service access point is that a service layer may be tailored to contain only those functional units that meet the requirements of a specific peer-to-peer association. The ability to construct a customized service results in the minimal amount of code and data required to implement the desired service. This feature is critical to real-world applications since the user of a service has a choice about which features are provided by a service without violating the layering principle. Empowering the service user with the ability to control the configuration of the protocol machines minimizes the negative impact of baroque protocols, such as the ISO session protocol, since either the minimal functionality can be selected for a particular association or the entire service can be omitted altogether.[22] In contrast, the monolithic approach typically results in a service layer containing all protocol functionality whether the user needs it or not.

A polymorphic SAP type is parameterized with the type of the protocol machine that is to be instantiated for an association. In addition, a SAP must enforce synchronization control on a per operation basis to ensure that a downcall has exclusive access to the protocol stack. Figure 6.8 illustrates the definition of a generic Sap object. A Sap represents the "head" of a protocol stack instance. A user instantiates a Sap object by declaration and uses the object as a service interface object for requesting operations. For example, the following fragment illustrates the instantiation of a generic Sap object that in turn instantiates a protocol machine that contains only a kernel functional unit:

```
Sap<PM::Kernel> sap;
```

SAPs representing other protocol machines composed of different functional subsets are similarly instantiated by providing the appropriate type parameter to the Sap<T> type template.

[22] For example, the *lightweight presentation service*, defined in RFC 1085 [148], which bypasses the session service completely by mapping the presentation service directly onto a transport service.

```
template<class T> class Sap {
private:

   T*    pm;       /* protocol machine subtype */

public:

   inline Sap()  { pm = new T(this); }
   inline ~Sap() { delete pm; }

   inline Future<SapIndication> ConnectRequest(...) {
         Downcall<T> mutex(pm);
         return pm->ConnectRequest(...);
   }

   ...
};
```

Figure 6.8: A Polymorphic Service Access Point.

Each `Sap<T>` method is defined inline and enforces downcall mutual exclusion by automatic construction of a `Downcall<T>` type on entering scope. For example, in the definition of the `ConnectRequest` method the statement:

```
Downcall<T> mutex(pm);
```

instantiates a `mutex` object that either acquires mutual exclusion for the protocol stack represented by `pm`, or blocks on the local semaphore associated with the protocol stack. As discussed in Chapter 3 and Chapter 4, implicit construction of a mutual exclusion type on entering scope is equivalent to a P operation (`acquire`) on a semaphore object and implicit destruction is equivalent to a V operation (`release`). The structure of the `Downcall<T>` type is depicted in Figure 6.9. The `Downcall<T>` object is inlined into the `Sap<T>` method, hence there is no associated procedure call overhead. Similarly, an `upcall<T>` type is defined for use when initiating upcalls at the virtual virtual transport layer as the result of an asynchronous I/O event.

If a `Sap<T>` method represents a confirmed service request, a `Future<SapIndication>` object is returned as the handle to the future confirmation event. For example, the following fragment illustrates the instantiation of a `Sap<T>` object and the invocation of one of its methods:

```
template<class T> class Downcall {
private:

   Semaphore* sem;        /* system dependent semaphore type */

public:

   /* construction = P(sem), destruction = V(sem) */

   inline Downcall  (T* pm)        { sem = pm->semaphore(); sem->acquire(); }
   inline ~Downcall ()             { sem->release(); }
};
```

Figure 6.9: A Polymorphic Downcall Mutex Type.

```
Sap<PM::Kernel> sap;
Future<SapIndication> future = sap.ConnectRequest(...);
```

The following inline expansion is semantically equivalent:

```
Downcall<PM::Kernel> mutex(sap.pm);
mutex.sem = sap.pm->semaphore();
mutex.sem->P();
Future<SapIndication> future = sap.pm->ConnectRequest(...);
mutex.sem->V();
```

The primary benefit of the Sap<T> type is that it presents a service access point abstraction that enforces layer protection without additional procedure call overhead. Modularity in this case is simply syntactic modularity that is compiled away at run-time. The polymorphic nature of the Sap<T> template type allows the user flexibility in the choice of which functional subset of a protocol machine is instantiated.

6.5.2 Using Inheritance to Integrate Protocol Machines

The polymorphic SAP abstraction represents a mechanism that an application uses to instantiate protocol stack instances, each with potentially different functional capabilities determined by the type parameter.

Figure 6.10 illustrates the relationship between polymorphic SAP objects and the corresponding protocol stack that is instantiated. Initiation of a downcall at a SAP or an

upcall initiated by the I/O signal handler results in an attempt to acquire exclusive access to the protocol stack via a semaphore object private to each stack. In general, a protocol stack consists of two objects: VTPM object and a composite PPM/SPM/VTSap object providing some functional subset of the session and presentation services. Two separate objects are within the protocol stack because the determination of which type of transport component to instantiate is made at connection establishment time based on the quality of service requirement of the initiating application layer entity and a reachability requirement based on the network address of the peer entity. The VTSap sub-object is a lightweight (inlined), virtual transport SAP used to bind either a TSap or an SPM::T object to a VTPM object at run-time. A VTPM object represents an instance of one of the transport protocol services previously listed; for example, a TP4 transport service or a TP0 service over TCP using RFC 1006.

The diagram in part (a) illustrates a protocol stack consisting of a single VTPM object that is bound to a virtual TSAP (VTSap) object through which a user accesses the services of some particular kernel-based transport service. The VTSap object is not polymorphic, but it does implement downcall mutual exclusion. This type of configuration is typically used to instantiate a transport listener object that listens for asynchronous connection requests.

The diagram in part (b) illustrates a polymorphic session SAP (SSap<T>), parameterized by the type of the session protocol machine to instantiate. The type parameter results in the instantiation of an SPM::T object containing only the functional units designated by the type parameter to the SSap<T>. For example, a session protocol machine that provides only the session kernel functional unit on an association is instantiated as:

```
SSap<SPM::Kernel> ssap;
```

which results in the creation of an SPM::Kernel object. The attached VTPM object is instantiated dynamically at run-time as part of connection establishment, when a determination can be made as to the type of transport service for the association.

The diagram in part (c) illustrates the protocol stack constructed by instantiating a PSap<T> object. Like the polymorphic SSap<T> object, the PSap<T> object causes a PPM subtype corresponding to the type parameter T to be instantiated. The type parameter used to instantiate the PPM::T type is, by transitivity of the mirroring relationship, used to instantiate a corresponding SPM::T type.

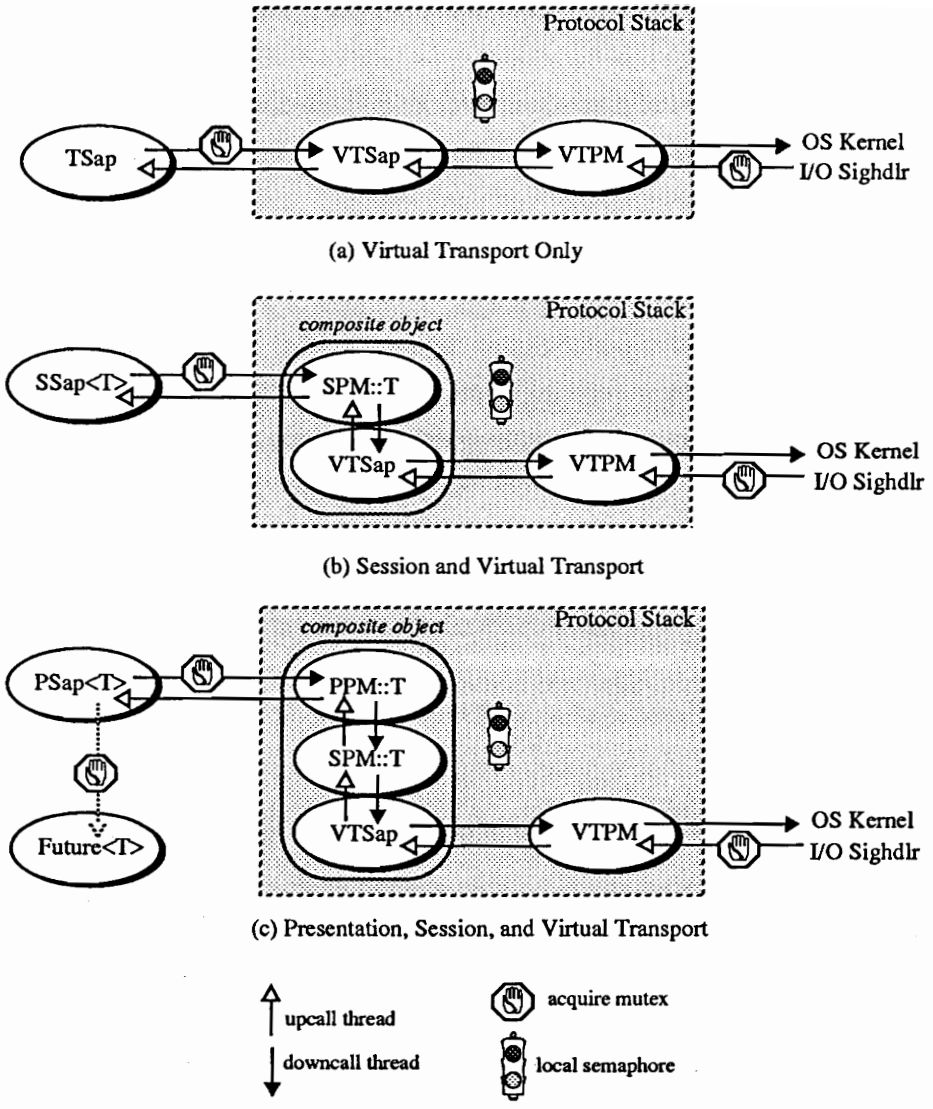

Figure 6.10: SAP and PM Object Structure.

As depicted in parts (b) and (c), the type parameter T determines the protocol machine subtype to instantiate. Type inheritance is used to allow a single composite PPM/SPM object to be instantiated based on the type parameter used to instantiate the corresponding PSap<T>. An important observation is that the composite PPM/SPM/VTSap object structure results from an inheritance structure that is *inverted* with respect to the layered model. That is, the PPM inherits from the SPM, which in turn inherits from the VTSap. The inheritance structure is inverted because a PPM type has a functional dependency on an SPM type for a service which the PPM specializes; that is, a PPM service request eventually calls upon a corresponding SPM service as part of a downcall. As previously explained, the dual of a service request is a service indication, which a lower layer must eventually hand to the upper layer. Hence, the SPM also has a functional dependency on the PPM for handling service indication and confirmation events as part of an upcall.

A principal result of this structure is that downcalls are implemented as *subclass-to-superclass* method invocations while upcalls are implemented as *superclass-to-subclass* method invocations, with the exception of the calls to/from the VTPM object. Downcalls through the protocol objects are denoted by a sequence of method invocations, denoted by $\downarrow$. For example, the following downcall sequence corresponds to the flow of control for initiating an operation at a PSap that terminates ($\perp$) at a syscall that writes data to the network.

$$\text{PSap<T>}_{\downarrow}, \text{PPM::T}_{\downarrow}, \text{SPM::T}_{\downarrow}, \text{VTPM}_{\downarrow}, \text{Kernel}_{\perp}$$

Note that neither the VTSap nor an SSap<T> is present in the downcall sequence. The VTSap is totally inlined into the SPM::T object that uses it to bind to the VTPM instance. The non-monolithic nature of the protocol machines make the use of an intermediate SSap<T> superfluous. The call from the PPM::T object to the SPM::T object takes the form of a subclass-to-superclass method invocation, some of which can be inlined. Similarly, an upcall is denoted by $\uparrow$. For example, the following upcall sequence corresponds to the flow of control that terminates by depositing a result in a future object.

$$\text{Sighdlr}^{\uparrow}, \text{VTPM}^{\uparrow}, \text{SPM::T}^{\uparrow}, \text{PPM::T}^{\uparrow}, \text{PSap<T>}^{\uparrow}, \text{Future<T>}^{\top}$$

In an upcall sequence, the call from the SPM::T to the PPM::T takes the form of a superclass-to-subclass method invocation. A superclass-to-subclass invocation is effected using a pure

virtual method declared by the SPM::T type and given an implementation by the PPM::T object.

The following subsections elaborate on the details of the type inheritance structure of the protocol stack configurations depicted in Figure 6.10.

6.5.3 The Virtual Transport Layer

The virtual transport layer offers an interesting opportunity to utilize subtyping. The purpose of the VTPM is to present a transport user with a single consistent transport abstraction irrespective of the *type* of the transport mechanism and the *implementation* of the mechanism. The virtual transport layer is represented by the subtype hierarchy depicted in Figure 6.11. The top-most class in the hierarchy is an abstract VTPM class. The VTPM class defines a set of pure virtual functions that each specialization must implement. There are two subclasses representing the two principal types of OSI connection-oriented transport service:

- Transport Class 0 (TP0), and
- Transport Class 4 (TP4).

The TP0_TPM class is a concrete class that implements the TP0 protocol, but is unique in that it is only a partial implementation. The TP4_TPM class is abstract and is further specialized into either a TP4_BSD or TP4_SYS5 subclass. The TP0_TPM class implements the basic TP0 protocol but defines an additional set of pure virtual functions that the TP0_TCP and TP0_X25 subclasses must implement for the TP0 protocol machine to be complete. The TP0_TCP subclass implements the RFC 1006 convergence protocol mapping TP0 onto TCP. Similarly, the TP0_X25 class implements a mapping of TP0 onto the X.25 connection-oriented network service.

The different combinations that may be instantiated are illustrated in Figure 6.12. Although instances of the VTPM hierarchy are treated uniformly, they offer distinct services. For example, TP4 offers an expedited data transfer service, while TP0 does not. However, a TP0_TCP instance augments the TP0 service by providing expedited data transfer using the out-of-band data channel provided by the TCP. The TP4_BSD class maps service requests

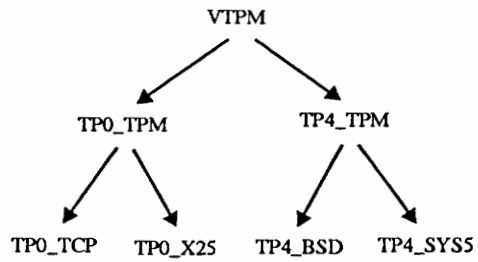

Figure 6.11: Virtual Transport Layer Inheritance Hierarchy.

VTPM	VTPM	VTPM	VTPM
TP0_TPM	TP0_TPM	TP4_TPM	TP4_TPM
TP0_TCP	TP0_X25	TP4_BSD	TP4_SYS5

Figure 6.12: Alternative Virtual Transport Machines.

onto the 4.4BSD UNIX TP4 kernel module via sockets. The TP4_SYS5 class maps onto the System V UNIX TP4 kernel module via the System V Transport Layer Interface.

6.5.4 A Rational Session Layer

The session layer defines a service that is composed of the set of functional units previously depicted in Table 6.1. Each functional unit can be viewed as a micro-protocol that incrementally extends a basic kernel protocol. The protocol definition is seemingly irrational, resulting in unnecessary technical complexity [149]. This complexity is primarily due to an attempt to keep the session protocol definition consistent with a pre-existing protocol definition rather than approaching the session layer problem from first principles. In this section, a rational approach to structuring the session protocol machine using inheritance is introduced. The SPM is organized in an inheritance structure composed of types derived from the base type representing the kernel protocol machine. Each subtype incrementally extends the session service by extending the kernel protocol machine.

Many basic networking applications do not require the full functionality of the session service. Fortunately, the specific functional capability for a session association can be

negotiated between peer communicating entities. During session connection establishment, the initiator and responder negotiate session functional requirements. A subset of functional units is proposed by an initiator and the responder indicates which proposed units it will support. Following connection establishment, only those protocol operations corresponding to the negotiated subset of functional units are allowed.

The monolithic approach to protocol machine construction typically means that the protocol machine must support all session service operations and conditional control logic must be in place to detect when a session user attempts to use a non-negotiated service. The non-monolithic approach adopted in OOSI is ideally suited to applications that can determine their session requirements at compile time and instantiate the correct session protocol machine type. In practice, a large set of applications fall into this category; however, OOSI does not inhibit the construction of more sophisticated applications that would require a more dynamic approach. Simply stated, OOSI optimizes for the common case.

Structuring functional units in a type inheritance hierarchy results in the ability to instantiate protocol machine instances that have the minimal functionality required. Furthermore, multiple protocol machines with different functionality can be instantiated within a single application. Every SPM type instance has the kernel machine functionality by default. Any additional functional units required result in an incrementally extended protocol machine. An advantage of this type structure is that only the code and data required to support the functionality corresponding to the type is bound into an application since no references are made to operations and data not defined by the type structure. The result is a protocol machine object that has:

- minimal state information,
- minimal code, and
- a simplified control flow.

The control flow is simplified since the relationship between subtypes is functionally well-defined and it is often possible to optimize the protocol machine based on the assumption that if a particular machine type is instantiated, then all functional unit types on which it depends are also instantiated.

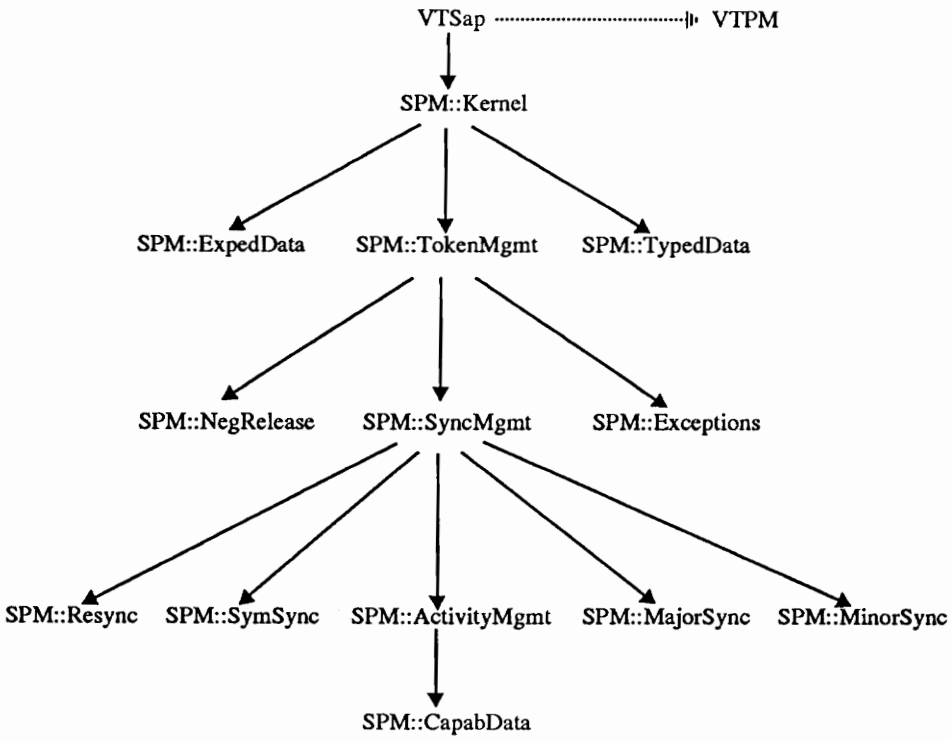

Figure 6.13: Session Protocol Machine Type Hierarchy.

```
class SPM {

    class Kernel : public virtual VTSap {
    public:
                                                                        5
        virtual SSapAbort* SConnectRequest(...);
        virtual SSapAbort* SConnectIndication(...);
        ...
        virtual SSapAbort* SGiveTokenRequest(...) { return Error(...); }
        ...                                                             10
    };

    class TokenMgmt : public virtual Kernel {
    public:
                                                                        15
        SSapAbort* SGiveTokenRequest(...);
        ...
    }

    class SyncMgmt : public virtual TokenMgmt {                         20
    public:
        ...
    }

    ...                                                                 25
};
```

Figure 6.14: SPM Functional Unit Classes.

The functional units defined in Table 6.1 are represented by types arranged in the inheritance structure depicted in Figure 6.13. Each functional unit is defined by a nested class of the `SPM` class depicted in Figure 6.14. For example, the kernel functional unit is implemented by the `SPM::Kernel` type, which is the base type in the inheritance structure. The `SPM::TokenMgmt` and `SPM::SyncMgmt` types along the main trunk are *pseudo* functional units introduced into the inheritance hierarchy to factor common data and operations used by various functional units. Almost all session functional units require some form of token management to provide dialogue control. Similarly, several functional units are concerned with dialogue synchronization.

The `SPM::Kernel` class implements a set of virtual methods that correspond to the complete set of session primitives. Those that implement specific kernel protocol functions, such as connection establishment, are completely defined in the `SPM::Kernel` class definit-

ion. The methods that are defined but not part of the kernel protocol immediately return an error unless overridden by a derived class. For example, the `SPM::TokenMgmt` class overrides the session request primitive used to pass a token used for dialogue control. The `virtual` attribute used in an inheritance clause in C++ means that only one copy of a base class is inherited when the same name occurs in the inheritance clause of two classes and both classes are multiply inherited by a third class.[23] The objective of virtual inheritance in OOSI is to force the creation of a single composite protocol machine object when extending the inheritance structure.

The subsets defined in Table 6.2 represent the functional unit combinations commonly found in current practice. The type inheritance structure in Figure 6.13 is implemented such that a type corresponding to each functional subset can be defined that multiply inherits the correct components, or a class representing a specific user combination can be defined. For example, if an application requires the kernel protocol extended with expedited data and negotiated release, the following empty class definition suffices:

```
class MySubset :  public SPM::ExpedData, public SPM::NegRelease {
        /* empty class def'n */
};
  ⋮
SSap<MySubset> ssap;
```

The `SSap<MySubset>` type causes the instantiation of an SPM object containing the kernel, expedited data, token management, and negotiated release functional units.

6.5.5 The Presentation Layer

As stated previously, the presentation service is a mirroring of the session service, augmented with data conversion functions. Stated another way, the functional units of the presentation service are type-based extensions of the functional units of the session service. It is natural then to treat the presentation service as an extension to the SPM type hierarchy presented in Figure 6.13 where each presentation functional units inherits from the corresponding session functional unit.

[23] A complete description of the syntax and semantics of virtual inheritance is found in [58].

Table 6.2: Session Functional Subsets.

Subset	Includes
Basic Combined	Kernel Duplex Half-duplex
Basic Synchronization	Kernel Half-duplex Typed Data Capability Data Minor Synchronize Exceptions Activity Mgmt
Basic Activity	Kernel Half-duplex Exceptions Typed Data Resynchronize Minor Synchronize Major Synchronize Activity Mgmt Negotiated Release

The resulting type hierarchy is illustrated in Figure 6.15. A noteworthy observation is that no SAP abstraction is used to compose a PPM instance and an SPM instance—the two protocol machine are *vertically integrated* using type inheritance between types representing functional units. The result is a composite PPM/SPM protocol object, as depicted earlier in Figure 6.10. Layer protection is guaranteed by the class boundary between inherited types and mutual exclusion is guaranteed by the local semaphore used by the polymorphic SAP type and the I/O signal handler to acquire exclusive access to the protocol stack. Modularity is thus maintained, but at run-time a single vertically integrated protocol machine object is instantiated.

Like the `SPM` class, the `PPM` class is defined as a set of nested classes, as illustrated in Figure 6.16. Like the `SPM::Kernel` class, the `PPM::Kernel` class implements a set of virtual methods that are selectively overridden by subtypes to effect the required service. PPM functional unit classes are defined using multiple inheritance in order to effect the instantiation of the necessary PPM and SPM functional units. For example, the `PPM::TokenMgmt` protocol extends both the `PPM::Kernel` protocol and the `SPM::TokenMgmt` protocol; hence, the `PPM::TokenMgmt` class multiply inherits from both. The use of multiple inheritance is not superfluous, a PPM object instantiated by a `Psap<T>` that requires token management

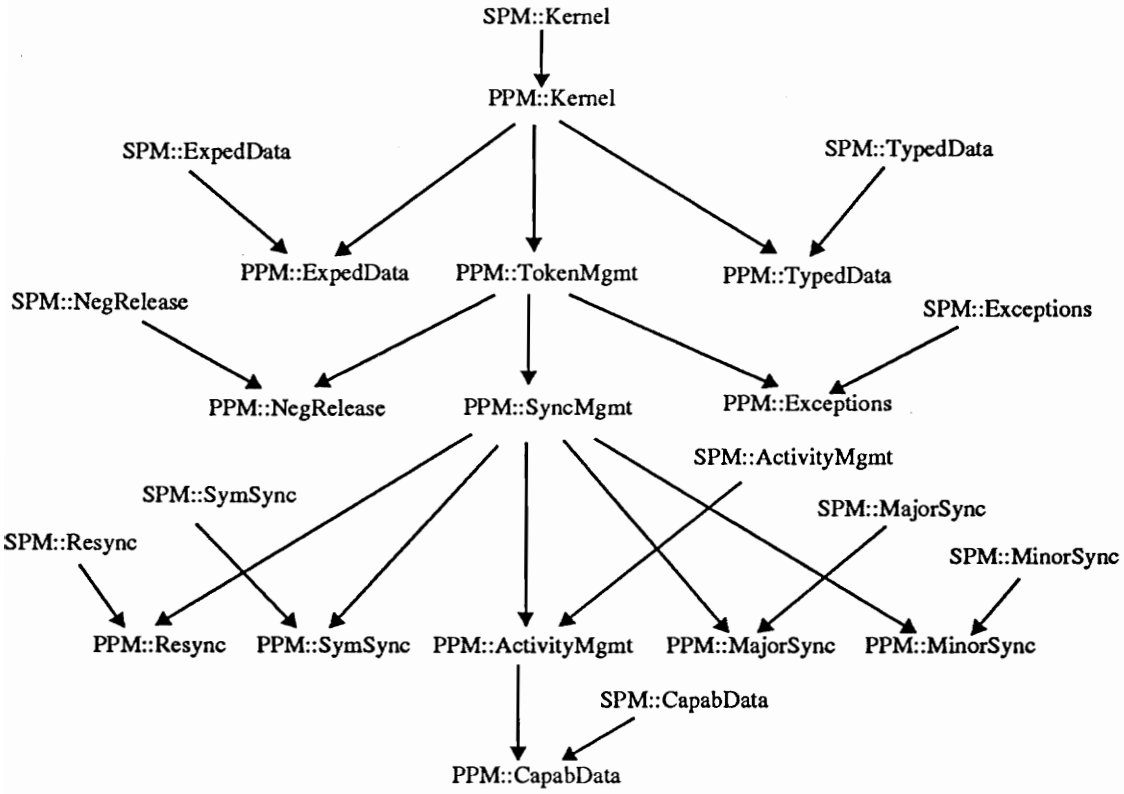

Figure 6.15: Presentation Protocol Machine Type Hierarchy.

```
class PPM {
public:

    /* Functional Units defined as nested classes */

    class Kernel : private virtual SPM::Kernel {
    public:

        virtual PSapAbort* PConnectRequest(...);
        virtual PsapAbort* PConnectIndication(...);
        ...
        virtual PsapAbort* PGiveTokenRequest(...) { return Error(...); }
        ...
    };

    class TokenMgmt : public virtual Kernel, private virtual SPM::TokenMgmt {
    public:

        PsapAbort* PGiveTokenRequest(...);
        ...
    };

    class SyncMgmt : public virtual TokenMgmt, private virtual SPM::SyncMgmt {
    public:
        ...
    };
};
```

Figure 6.16: PPM Functional Unit Classes.

services must ensure that the session token management service is also enabled. Multiple inheritance (using the `virtual` attribute) ensures that the correct session functional units are instantiated.

6.6 Related Work

Object-oriented languages are being used by others to implement layered communication protocol architectures because of the ability to define generalized constructions that can be specialized using class inheritance. The Morpheus protocol development framework [1] and the OTSO protocol development framework [103, 104, 169] are both object-oriented approaches to structuring layered protocols. Like OOSI, these object-oriented protocol development frameworks consist of a set of classes that define abstractions for common

elements used in constructing a protocol layer; however, there are distinct differences.

The Morpheus framework is being done in conjunction with the *x*-kernel, a network-based operating system [84, 141]. Morpheus is an extension of previous *x*-kernel research on dynamic protocol machine configuration [135, 136]. Dynamic protocol machine configuration is one approach to providing a flexible protocol stack. The primary advantage of a flexible protocol stack is that extraneous control machinery that is not needed for the service required by the connection is omitted at run-time, thereby minimizing wasted CPU cycles. The original *x*-kernel approach separates condition control logic from the actual protocol machine structure and incorporates it into a *virtual* protocol that provides dynamic protocol graph navigation at run-time. This approach is more flexible than the static approach adopted in OOSI; however, the *x*-kernel approach is applied to the lower layers where dynamic protocol instantiation is most often required because of the variability in the lower layers. OOSI is focused on the upper layers where it is often known in advance what protocol services are required; hence, a static protocol graph instantiated using polymorphic SAPs to obtain flexible composition is sufficient for a wide class of applications. Note however that OOSI can operate over a transport/network protocol combination that is implemented using the dynamic approach. The virtual transport layer provide the necessary abstraction to separate the OOSI implementation from the lower layer structure.

Unlike the original *x*-kernel approach, the Morpheus work is an attempt to provide a more object-oriented structure by identifying those protocol components, or *shapes*, that are common to lower layer components. Common shapes are "over" and "under" SAPs, over and under sessions, multiplexors, workers, and routers. A specific protocol is implemented by specializing abstract classes representing these shapes. The SAP shape is an abstract type, but does not support parameterization as does the polymorphic SAP in OOSI. The differences between OOSI and Morpheus arise primarily because of the different requirements of the upper and lower layers. Lower layers are structured as multiplexors or routers, whereas the upper layers, if vertically partitioned, do not require this functionality.

The OTSO framework consists of tools for automatically generating protocol objects, in the form of classes, from protocol descriptions in Abstract Syntax Notation One (ASN.1). A class representing a protocol object is automatically generated to inherit from multiple predefined *abstract* classes that provide both user and provider service layer interfaces.

The user and provider classes are conceptually similar to the "over" and "under" SAPs in Morpheus in that they provide the glue for the layer above and the layer below. In OOSI, a SAP is not glue, it is the top-most interface point at which the protocol stack is entered. Vertical integration using inheritance between protocol machine types means that the over/under SAP concepts are not required. The generalization of a SAP as the glue between layers is superfluous in a non-monolithic protocol structure.

An interesting feature of the OTSO run-time system is the support for concurrent "runner" objects. Runners permit the initiation of network operations as asynchronous threads of control. To cope with the eventual result, OTSO uses a *promise* to return a handle to the eventual result value of an asynchronous invocation. The promise in OTSO provides the same service as the future in OOSI; however, the promise in OTSO is defined as an integer type only since OTSO protocol service primitives always return integer results. As demonstrated in the present chapter, as well as Chapter 5, the polymorphic future type is a very general mechanism that is useful for general concurrent programming.

6.7 Future Work

OOSI represents a protocol infrastructure that can be extended by defining application layer protocols that inherit from the presentation layer. The implementation of application service elements such as remote operations, association control, and reliable transfer are already in progress. This work is dependent upon a corresponding C++-based ASN.1 tool set that facilitates the implementation of application protocol specifications written in ASN.1. The basic ASN.1 structure has already been defined and implemented as a separate piece of work that originated from the requirements of OOSI; however, an ASN.1-to-C++ translator needs to be implemented.

The implementation of OOSI for multi-threaded applications could be complemented by structuring the kernel-resident transport layer using the same vertical partitioning methods. The transport layer can be vertically partitioned since a unique identifier is used to relate the upper layer stack with the transport layer. Ports in TCP and transport selectors (TSELs) in TP0/4 are unique transport endpoint identifiers. A vertical partitioning of the network protocol is highly problematic since a network protocol must multiplex diverse transport protocols onto a single network device, which is perhaps shared by other distinct

network protocols in the same kernel (e.g., IP and CLNP over a single 802.3 device). A practical obstacle is that the separation between the transport and network layers is not what one would think. Most TCP/IP implementations are closely integrated [43]; however, the streams-based approach to protocol implementation in UNIX System V provides a model of separation that could support a vertically partitioned transport protocol structure. In general, the methods for lower layer protocol implementation in the x-kernel can be combined with the OOSI upper layer structure resulting in a complete seven layer communications infrastructure.

A thorough quantitative analysis of the performance of OOSI is required. The following chapter provides a list of criteria for performing a comparison with the ISODE implementation. Data is provided for the virtual transport layer. Further work is required before complete performance data of the entire protocol stack can be obtained.

Chapter 7

OOSI Performance Issues

The non-monolithic approach to structuring protocol machines is fundamentally different from the traditional protocol structuring approach. This difference arises primarily from the fundamental assumption in OOSI that applications using the protocol stack are inherently concurrent. A primary goal of OOSI is that the degree of concurrency in an application is not artificially limited by the communications infrastructure. In other implementations, the ISODE among them, the protocol stack in an obstacle to concurrency. In ISODE, the primary obstacle is the monolithic structure of the protocol stack, which requires a global synchronization policy in order to guarantee consistency of the protocol machines. The global synchronization policy serializes use of the entire protocol structure, artificially limiting the degree of concurrency. Other implementation constraints, such as non-reentrant procedures and blocking I/O, also inhibit multi-threaded applications.

The structural differences and objectives of OOSI and ISODE make it difficult to perform a detailed and completely accurate comparison of the two implementations. Subsequent sections enumerate the major structural differences, define a simplistic but informative comparison metric, present data resulting from a preliminary quantitative analysis of the virtual transport layer, and outline the direction of future work in this area.

7.1 Comparison Criteria

A complete and thorough quantitative analysis of all of the issues is a difficult and complex task because of the lack of good instrumentation tools for comparing object-oriented programs with traditional procedural programs, and the difficulty of isolating behavior in an inherently asynchronous environment. The following list summarizes the important issues that underly a complete and thorough analysis of the efficiency of the OOSI protocol stack structure:

- non-blocking versus blocking downcalls.
- asynchronous indication event delivery using interrupts instead of polling (e.g., blocking reads).
- thread creation, initiation, and management for an upcall.
- a local synchronization policy using thread-based semaphores versus a global synchronization policy based on blocking interrupts.
- an inheritance-based class structure using inlining and virtual methods versus a traditional structure using non-inlined procedures and hand-coded procedure dispatch tables for structuring downcall/upcalls.
- type-driven PDU processing based on a PDU type hierarchy versus hand coded case analysis of a PDU structure defined as a discriminated union.

The tool problem can be overcome to a degree by using a compiler whose front-end parses both C and C++ syntax, using the same back-end algorithms for optimization and code generation. The existence of this type of compiler eliminates potential bias introduced by compilation effects. For example, the GNU GCC compiler generates native machine code for both C and C++.[1] To eliminate bias introduced by compilation, both OOSI and ISODE are compiled using GCC. For comparison purposes, both implementations are compiled with the highest level of optimization (i.e., `O2`). Profiling can also be done, but generally provides only information about where execution time is spent and does not provide general high-level structural information.

Non-deterministic behavior due to the occurrence of asynchronous events presents problems for debugging as well as performance measurement. A reasonable approach is to force predictable behavior and compare performance of only deterministic activity. Although not completely satisfactory, this approach makes it possible to obtain informative quantitative measurements of *overall* end-to-end throughput of user data, which is the bottom line when dealing with real-world communications.

[1] In contrast, the AT&T C++ translator (`cfront`) generates C code that must then processed by a C compiler.

7.2 Restriction of OOSI Behavior

To make a fair comparison of the structural aspects of OOSI (e.g., virtual method calls and SDU segmentation/reassembly), it is necessary to force OOSI to behave in manner consistent with the behavior of ISODE; otherwise, it is difficult to interpret performance data because of other system factors. Since ISODE is inherently sequential, OOSI is configured to perform sequential operations as well. In addition, the ISODE test program implements a blocking write; hence, OOSI should also force a blocking write. One difference that is not normalized is the handling of indications. OOSI handles indications in an asynchronous interrupt-driven manner; in contrast, ISODE adopts a synchronous polling philosophy. When an indication is received, the OOSI I/O signal handler initiates a upcall. If the received data completes a pending SDU, then the SDU is handed up to the application as a sequence of virtual function calls; otherwise, the data is appended to the SDU and the upcall immediately returns. This interrupt driven style is fundamental to the OOSI protocol stack structure. In the ISODE test program, a synchronous read is issued immediately after data is sent if a response is pending. In contrast, the OOSI test program enters a busy wait. The major difference between the two modes of interaction under UNIX is that an interrupt requires extra kernel-level processing to setup and manage the initiation and return of a signal handler while a synchronous read does not. While the fully asynchronous mode of indication processing is more expensive in the sequential case, it is a critical component in the concurrent case.

To summarize, OOSI is purposely restricted to deterministic behavior that is similar to the behavior in ISODE, with the exception of indication handling. The purpose of the restriction is so that the data obtained from a comparison of average throughput can be used to make reasonable inferences about the object-oriented structure of OOSI in contrast to the non-object-oriented structure of ISODE. In the following comparison, one must bear in mind that in a fully concurrent setting, OOSI is designed to operate in a non-blocking, asynchronous fashion with multi-threaded protocol stacks. The current ISODE implementation is incapable of supporting multiple threads of control within the protocol stack.

7.3 Virtual Transport Layer Performance

The virtual transport layer is responsible for segmenting a service data unit from the upper layers and writing the data to the real transport module residing in the operating system kernel. Performance is dominated by the following factors:

- the maximum Transport Service Data Unit (TSDU) size, which may cause the VTPM to segment user data prior to transmission and force reassembly of received user data prior to issuing an indication to the session layer.
- whether or not non-blocking gather/scatter read/write syscalls are offered by the kernel, eliminating the need to coalesce data sources prior to writing.
- flow control exercised by the TCP module during a write syscall due to kernel/network congestion.

Performance data is presented that shows the results of a performance analysis of the OOSI virtual transport layer operating over RFC 1006 [147]. The data reflects the comparison of OOSI with the ISODE RFC 1006 component and with raw TCP. The raw TCP data is provided to illustrate the best that can be done without the effects of the processing overhead introduced in both OOSI and ISODE in implementing RFC 1006. For comparison purposes, the TCP socket is in all cases configured so that data is not buffered prior to sending (i.e., the `TCP_NODELAY` flag is on). Testing is done using an ISODE initiator process and an OOSI initiator process sending data to the ISODE transport responder daemon (`tsapd`).[2] The `tsapd` is dedicated to servicing the initiators one at a time; hence, there is no extraneous interference when an initiator is sending data. The initiators reside on a single SPARC machine separate from the SPARC machine on which the responder resides. The interconnection between the two machines consists of a lightly loaded, physically contiguous 802.3 local area network. The OOSI initiator instantiates a single stack instance like the one depicted in part (a) of Figure 6.10 on page 167.

The maximum TSDU size is set at 2048 bytes. Both ISODE and OOSI segment user data that is greater that the maximum TSDU size minus the overhead for the RFC 1006

[2]The ISODE initiator is the `isoc` client. Both `isoc` and `tsapd` are configured so that `TCP_NODELAY` is in effect. Raw TCP writes perform segmentation based on the maximum TSDU size. In addition, `WRITEV` is enabled so that gather-style writes are used. Queued writes are disabled.

Table 7.1: Transport Sink Performance Data.

SPARC Transport Sink (max TSDU size = 2048 bytes)				
User Data (bytes)	10 × 100 T-DATA.requests Highest Average Kbytes/s			
	TCP	ISODE	OOSI	%±
1024	995.586	970.921	952.126	-1.9
2048	1993.280	659.063	610.336	-7.4
4096	3718.620	523.950	481.102	-8.2
8192	989.296	609.482	563.689	-7.5
16384	769.912	616.733	599.884	-2.7
32768	685.385	634.902	622.167	-2.0
65536	648.207	585.037	585.056	0.0

header (4 bytes) and the minimum size of a TP0 data TPDU PCI (3 bytes). In both cases segmentation is performed for user data greater than or equal to 2041 bytes. With raw TCP, there is no overhead for a header—the data is sent entirely as user data, segmented along 2K boundaries.

7.3.1 Transport Sink

Table 7.1 depicts the data for a transport sink operation. Once a connection is established, n bytes of user data are sent to the responder using a blocking write. The highest average number of kilobytes transmitted per second is computed from 100 iterations of a T-DATA.request operation. The client processes are executed 10 separate times and the highest average is used in the table, which represents a "best effort" comparison over a range of user data sizes.

The data in the TCP column is presented so that a benchmark exists for the highest attainable user data throughput with TSDU segmentation and no header processing. The ISODE and OOSI columns are the data of interest. The %± column shows the percent gain/loss of OOSI in comparison to ISODE. The data sizes in the range 1K–8K provides the most informative data. At the 1K size, no segmentation is being done and OOSI is a modest 1.9% slower than ISODE. This difference is most likely the result of the object-oriented structuring in OOSI. At sizes greater than 2K, segmentation occurs since the maximum user data after computing the RFC 1006 header overhead is 2041 bytes. In the 2K–8K range, OOSI is performance deteriorates by an additional 5.5–6.2%. The obvious

Table 7.2: Transport Echo Performance Data.

SPARC Transport Echo (max TSDU size = 2048 bytes)			
User Data (bytes)	10 x 100 T-DATA.requests Highest Average Kbytes/s		
	ISODE	OOSI	%±
1024	249.040	243.446	-2.2
2048	303.812	279.439	-8.0
4096	397.318	379.268	-4.5
8192	469.055	462.622	-1.4
16384	477.245	475.279	-0.4
32768	500.450	501.365	+0.2
65536	482.331	494.888	+2.5

reason for this drop in performance is that the OOSI segmentation algorithm is less efficient than the one used by ISODE. It is reasonable to expect that additional tuning can be used to eliminate this additional drop. A profiling tool can help in this regard. At sizes greater than 16K, factors such as kernel and network congestion start becoming more apparent and throughput eventually converges to approximately 600 Kbytes/s for TCP, ISODE, and OOSI.

7.3.2 Transport Echo

The transport echo operation sends data and waits for the same data to be echoed from the responder. The `tsapd` does not support raw echo, so no data is shown for TCP. An echo response in OOSI is handled as an asynchronous indication event that is turned into an upcall by the I/O signal handler, the upcall terminates at a handler that computes the round-trip time. In ISODE, a blocking read request issued immediately following each T-DATA.request is satisfied by the echoed data and the round-trip time is then computed. Table 7.2 depicts the data obtained from a comparison of the echo operation. As with the sink operation, OOSI is capable of less throughput than ISODE, but the overall performance in the 4K–8K range is slightly better. A sound hypothesis is that the improvement is a result of the reassembly algorithm. The OOSI reassembly algorithm attempts to minimize the number of reads required to bring in a complete TSDU and then do extra work to separate headers, etc. ISODE on the other hand has more of a piece-meal read strategy which reads only what it needs at the moment. The spike at the 2K range suggests the OOSI

approach is more costly when the user data size matches the TSDU size, but performance improves and begins to exceed ISODE as data sizes become larger. The implication is that any cost associated with the extra structuring in OOSI is overshadowed at higher data sizes by the cost of handling the data.

7.3.3 Conclusion

The principal conclusion drawn from this preliminary performance measurement is that the OOSI virtual transport protocol machine is approximately 2% slower than ISODE, assuming a more highly tuned segmentation algorithm.[3] Overall, the results are encouraging and offer tangible evidence that OOSI is viable implementation. The modest difference can potentially be negated by additional performance gains at the session and presentation layers. However, at this juncture, performance data of the session and presentation layers of OOSI is not available. The following sections enumerate the issues affecting the performance of the session and presentation layers.

7.4 Session Layer Performance

The dominate performance issue at the session layer is the processing of session PDUs (SPDUs) and the handing up of indication and confirmation events to the presentation layer as part of an upcall sequence. Since TSDU segmentation is done in the virtual transport layer, there is no other serious processing at this layer. The processing of SPDUs is a memory intensive operation because of the large number of distinct SPDUs and the amount of variable information stored in the PCI portion. In ISODE, SPDU processing is centralized using case analysis to discriminate and process PCI information. In OOSI, the SPDUs are arranged in a type inheritance structure and processing is distributed amongst procedures that process specific SPDU types. The fundamental difference between the two approaches is that OOSI uses the language type rules and compiler generated type discrimination to effect SPDU processing while the ISODE uses hand-coded case analysis. Subsequent

[3]Note that ISODE is a widely-used implementation, currently at version 8.0, and represents several man-years of development and performance tuning effort. In contrast, OOSI is a highly experimental implementation currently at version 0.9.

performance comparison is warranted to determine the effects of a type-centered approach to SPDU processing.

7.5 Presentation Layer Performance

The dominate factor in the presentation layer is the encoding of user data and presentation PDUs (PPDUs) prior to passing a service data unit to the session layer and decoding PPDUs and data from an indication or response. A separate research problem is the representation and implementation of ASN.1 types as polymorphic types in an object-oriented language. This problem is currently being addressed by a design and implementation of the primitive and constructed ASN.1 types in C++. The design and implementation are being driven by the structure and purpose of OOSI. The current ISODE implementation requires a two-step translation process in converting user data into a transfer syntax sufficient for transmission between heterogeneous systems. The new polymorphic type-based approach reduces this to a single-step process. A secondary issue is the choice of the encoding rules and the flexibility with which different encodings rules can be employed by the presentation layer to encode/decode user data using the rules requested by the presentation user at connection establishment time; for example, the lightweight encoding rules proposed by Huitema and Doghri [83]. The independent protocol stack instances permitted by OOSI are ideally suited to supporting flexible bindings of different encoding rules. The integration of the OOSI presentation layer with this separate ASN.1 work must be completed before realistic and informative end-to-end performance data can be obtained.

Chapter 8

Conclusion

The focus of this work has been the synthesis of programming constructs for concurrent object-oriented programming and layered communication protocols. By combining concurrent programming and communication protocol type structures, flexible and efficient distributed applications can be constructed. The concepts and type structures introduced in this dissertation are significant because they provide:

- a conceptual and programming framework, based on functions as objects, that serves as a primitive model for concurrent object-oriented programming using actors, and
- pragmatically derived programming abstractions for building distributed applications based on flexible protocol stacks.

Figure 8.1 illustrates how the principal components are interrelated. The principal components are the polymorphic lambda objects, polymorphic futures, and protocol layer types. Lambda objects and futures are sufficient programming abstractions for realizing actor concurrency. Protocol layer types in conjunction with futures are used to construct flexible protocol stacks. Combining concurrent objects, such as actors, and flexible protocol stacks yields the basic mechanisms required for constructing concurrent and distributed applications.

8.1 Lambda Objects and Futures

The view of functions as objects introduced in Chapter 2 draws its inspiration from both the functional and object-oriented programming language paradigms. The polymorphic lambda type template for defining a class representing a function is a synthesis of the concept of λ-abstraction, higher-order functions, and object. The result is a flexible generalization for expressing both sequential and concurrent operations as objects, with all the attendant

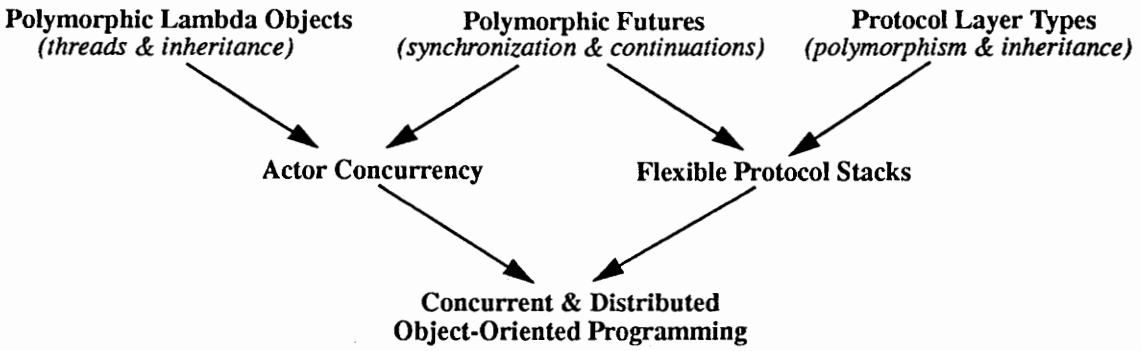

Figure 8.1: Key Concepts and Their Relationships.

benefits. The principal concept is that the structural aspect of procedures, in particular the return type, can be generalized and represented as a class that can then be specialized to effect the behavior of a particular function For the polymorphic lambda type to be useful in a concurrent setting it must be augmented with the polymorphic future type introduced in Chapter 4. Polymorphic futures permit the expression of first-class, user-defined continuations that are combined with a specific lambda object. Lambda object defined with a future as a result type may be asynchronous invoked since the future provides the mechanism for obtaining the eventual result.

The class constitutes a first-class environment on which specific operations can be specialized to effect arbitrary user-defined execution behavior for procedures. The parameterization of the polymorphic lambda type with a polymorphic future type extends the basic abstraction to allow the expression of a concurrent operation. That is, the type of the return value determines whether or not an operation may be executed concurrently. The lambda type and its parameterization by a future type, combined with a primitive threads mechanisms, provide a basic concurrent execution model that can be used to construct more sophisticated concurrency models, such as actors.

The synchronization structuring principles defined in Chapter 3 are critical to an efficient and flexible implementation of the polymorphic future type. In order to effect the implementation, the inheritance anomaly that arises when inheriting synchronization constraints must be avoided. The "principle of separation" derived from eliminating synchronization predicates from methods defined as part of an object requiring synchronization is the key

to avoiding the inheritance anomaly. To inherit structure and operations, predicates representing synchronization constraints must be separated from the methods that use the synchronization predicates. A set of monitor examples demonstrated the application of the principle of separation to traditional synchronization structures. The result is a more flexible way to construct traditional synchronization control using inheritance. The use of inheritance with passive monitor constructs can be extended to generic active objects with the addition of a mechanism like the behavior set. The inheritance anomaly that arises in concurrent object-oriented languages is analyzed using Milner's *CCS* as a descriptive tool. The use of *CCS* to explicate the basic underlying problem suggests a direction for a more formal approach to specifying and reasoning about synchronization constraints in the context of inheritance in concurrent object-oriented languages.

The real power of futures is realized when the polymorphic lambda type is parameterized by a user-defined future type. Since a lambda object is a complete environment, the addition of a mechanism for returning a future result value is all that is needed to enable execution of the `lambda` method as an independent thread of control. The future object represents the continuation that the `lambda` method uses to pass its result value to other parts of a program. To a client, the future object represents a value that can be converted to a result value, subject to synchronization conditions, using an appropriate type conversion operator.

The combination of the polymorphic lambda type and the polymorphic future type provide the foundation for constructing more advanced concurrent computing paradigms. A set of types for representing the components of the Actor model were defined as extensions to the basic lambda types augmented with futures. The principle contribution of this set of Actor types is that it provides a more abstract operational semantics for the ACT++ language. The "functions as objects" view of concurrent computation allows the definition of a new ACT++ computational model based solely on the evaluation of lambda methods. This new model allows the underlying ACT++ execution model to be separated from a particular threads implementation.

8.2 Protocol Layer Types and Futures

The OOSI communication infrastructure is based on a vertical partitioning of the upper layer OSI protocols, resulting in a non-monolithic protocol stack. A non-monolithic pro-

tocol stack structure is particularly well-suited to supporting concurrent applications since protocol stack instances are non-interfering. Class inheritance is used to define upper layer protocol machines as collections of inter-related micro-protocols arranged in an inheritance graph and service access points are represented using a polymorphic type that allows flexible instantiation of protocol machines to suite the diverse needs of an applications.

A "virtual" transport layer, or convergence layer, is defined in order to cope with heterogeneous transport/network protocol combinations (transport switching) and the need to perform a protocol conversion function. A primary advantage of OOSI's inheritance structure is that protocol machines are tightly integrated, thereby minimizing the adverse effects of layering. Furthermore, the ability to instantiate multiple protocol stacks facilitates building an application that supports presentation layer pipelining using multiple stack instances over the same transport endpoint.

Futures combined with polymorphic service access points provide the mechanism necessary to realize flexible protocol stacks supporting fully asynchronous operations. Confirmed network operations incur a time delay between invocation and indication of a result. Polymorphic future types that are parameterized by the type of the indication are ideal for structuring this type of interaction.

Although OOSI represents a specific application of vertical partitioning to the OSI protocols, the OOSI methodology for protocol machine construction and the specific use of polymorphism and inheritance in realizing a vertical structure can be applied to other complex protocol structures that must support concurrent object-oriented applications. The success of applying this methodology to a set of protocols as feature rich and complex as the OSI protocols is encouraging.

The cost of imposing an object-oriented structure on the OSI protocols is a modest 2% processing overhead in the sequential case. That is, although OOSI is designed specifically to support multiple, concurrent protocol stacks within a single application, it performs almost as well as a sequential implementation (ISODE) when made to emulate sequential behavior. In a concurrent application, an overall throughput increase can be expected over a sequential application since limitations to concurrent execution inherent in the monolithic structure have been eliminated by the non-monolithic protocol structure.

References

[1] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. In *SIGCOMM'92 Conference Proceedings*, pages 27–38, 1992.

[2] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, July 1986.

[3] Gul Agha. *A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[4] Gul Agha and Christian J. Callsen. Actorspace: An open distributed programming paradigm. In *Principles and Practice of Parallel Programming Conference Proceedings*, 1993. To appear.

[5] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *ECOOP'87 Conference Proceedings*, pages 234–242. Springer-Verlag, 1987.

[6] Pierre America. POOL-T: A parallel object-oriented language. In *Object-Oriented Concurrent Programming* [184], pages 199–220.

[7] American National Standards Institute. *Programming Language PL/I*, October 1976. ANSI X3.53–1976.

[8] Birger Anderson. Ellie language definition report. Technical Report 91/3, University of Copenhagen, Department of Computer Science, June 1991.

[9] Sten Andler. Predicate path expressions. In *Sixth ACM Symposium on Principles of Programming Languages*, pages 226–236, 1979.

[10] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin-Cummings, 1991.

[11] Gregory R. Andrews and Fred B. Schneider. Concepts and notation for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[12] M. Stella Atkins. Experiments in SR with different upcall program structures. *ACM Transactions on Computer Systems*, 6(4):365–392, November 1988.

[13] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics, Volume 103. North-Holland, 1984. Revised Edition.

[14] Brian N. Bershad. The PRESTO user's manual. Technical Report 88-01-04, University of Washington, Department of Computer Science, 1988.

[15] Brian N. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *Proceedings of the 1992 USENIX Workshop on Microkernels*, 1992.

[16] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[17] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Fifth ACM Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–233, October 1992.

[18] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Eleventh ACM Symposium on Operating System Principles*, pages 123–86. ACM, November 1987.

[19] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[20] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *Simula Begin.* Chartwell-Bratt, 1979.

[21] Toby Bloom. Evaluating synchronization mechanisms. In *Seventh ACM Symposium on Operating System Principles*, pages 24–32, December 1979.

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

[23] Per Brinch Hansen. *Operating System Principles.* Prentice-Hall, 1973.

[24] Per Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice-Hall, 1977.

[25] Jean-Pierre Briot and Akinori Yonezawa. Inheritance and synchronization in object-oriented concurrent programming. In A. Yonezawa, editor, *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.

[26] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, pages 10–19, April 1987.

[27] Alan Burns, Andrew M. Lister, and Andrew J . Wellings. *A Review of Ada Tasking.* Number 262 in Lecture Notes in Computer Science. Springer-Verlag, 1987.

[28] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In *Operating Systems*, pages 89–102. Springer-Verlag, 1974. Lecture Notes in Computer Science 16.

[29] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[30] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[31] David R. Cheriton. Exploiting recursion to simplify RPC communication architectures. In *SIGCOMM'88 Conference Proceedings*, pages 76–87, August 1988.

[32] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[33] David R. Cheriton and Wally Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[34] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941. Annals of Mathematics Studies.

[35] David D. Clark. Modularity and efficiency in protocol implementation. Internet Protocol Specification RFC 817, Network Information Center, SRI International, July 1982.

[36] David D. Clark. The structuring of systems using upcalls. In *Tenth ACM Symposium on Operating System Principles*, pages 171–180, December 1985.

[37] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[38] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM'90 Conference Proceedings*, pages 200–208. ACM, 1990.

[39] Russel J. Clark, Mostafa H. Ammar, and Kenneth L. Calvert. Multi-protocol architectures as a paradigm for achieving inter-operability. Technical Report GIT-CC-92/36, Georgia Institute of Technology, College of Computing, 1992.

[40] William Clinger and Jonathan Rees. *Revised Report on the Algorithmic Language Scheme*, 1992. Version 4.

[41] William Douglas Clinger. *Foundations of Actor Semantics.* Ph.D. dissertation, Massachusetts Institute of Technology, Department of Mathematics, 1981.

[42] Douglas E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture.* Prentice-Hall, 1988.

[43] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP: Design, Implementation, and Internals.* Prentice-Hall, 1991.

[44] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963.

[45] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-54, Carnegie Mellon University, School of Computer Science, September 1990.

[46] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[47] Ole-Johan Dahl and C. A. R. Hoare. Hierarchical program structures. In *Structured Programming*, pages 175–220. Academic Press, 1972.

[48] Ole-Johan Dahl and Kristen Nygaard. SIMULA—an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[49] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[50] Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, SE-18(2):89–102, February 1992.

[51] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[52] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[53] Edsger W. Dijkstra. Go to statment considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968. Letter to the Editor.

[54] Edsger W. Dijkstra. The structure of the 'THE' multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[55] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In Hoare and Perrott [77], pages 72–93. Also in *Acta Informatica* 1(2):115–138, 1971.

[56] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[57] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Thirteenth ACM Symposium on Operating System Principles*, pages 122–136, October 1991.

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

[59] Nissim Francez. *Fairness.* Springer-Verlag, 1986.

[60] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press, 1992.

[61] N. H. Gehani and W. D. Roome. Concurrent C++: Concurrent programming with class(es). *Software—Practice and Experience*, 18(12):1157–1177, December 1988.

[62] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–113, January 1985.

[63] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[64] et al. Gregory R. Andrews. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[65] Carl A. Gunter. *Semantics of Programming Languages—Structures and Techniques.* MIT Press, 1992.

[66] Bruce K. Haddon. Nested monitor calls. *ACM Operating Systems Review*, 11(4):18–23, October 1977.

[67] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[68] Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):583–598, October 1987.

[69] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3/4):143–153, 1986.

[70] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[71] Carl Hewitt. Towards open information systems semantics. Presented at the ECOOP-OOPSLA'90 Workshop on Object-Based Concurrency, 1990.

[72] Carl Hewitt and Russell R. Atkinson. Specification and proof techniques for serializers. *IEEE Transactions on Software Engineering*, SE-5(1):10–23, January 1979.

[73] Carl Hewitt and Peter de Jong. Open systems. In Michael L. Brodie, editor, *On Conceptual Modeling*, pages 147–164. Springer-Verlag, 1984.

[74] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[75] C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrott [77], pages 61–71.

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

[77] C. A. R. Hoare and R. H. Perrott, editors. *Operating System Techniques.* Academic Press, 1972.

[78] John H. Howard. Signalling in monitors. *Second International Conference on Software Engineering*, 19(5):47–52, May 1976.

[79] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language independent garbage collector toolkit. To appear, 1991.

[80] Stephen P. Hufnagel. *Vertically Partitioned Object-Oriented Software Design for Dependability and Good Performance.* Ph.D. dissertation, University of Texas, Austin, January 1987.

[81] Stephen P. Hufnagel and James C. Browne. Performance properties of vertically partitioned object-oriented systems. *IEEE Transactions on Software Engineering*, SE-15(8):935–946, August 1989.

[82] Christian Huitema and Assem Doghri. Defining faster transfer syntaxes for the OSI presentation protocol. *ACM Computer Communication Review*, 19(5):44–55, October 1989.

[83] Christian Huitema and Assem Doghri. A high speed approach for the OSI presentation protocol. In H. Rudin and R. Williamson, editors, *Protocols for High-Speed Networks*, pages 277–287. Elsevier/North-Holland, 1989.

[84] Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley. RPC in the *x*-kernel: Evaluating new design techniques. In *Twelfth ACM Symposium on Operating System Principles*, pages 91–101, December 1989.

[85] IEEE Technical Committee on Operating Systems. *Threads Extension for Portable Operating Systems*, February 1992. POSIX P1003.4a/D6.

[86] International Organization for Standardization. *Information Processing — Open Systems Interconnection — Transport Service Definition*, 1986. ISO 8072 (CCITT X.214).

[87] International Organization for Standardization. *Information Processing — Open Systems Interconnection — Connection-oriented Transport Protocol Specification*, 1986. ISO 8073 (CCITT X.224).

[88] International Organization for Standardization. *Information Processing — Data Communications — Addendum to the Network Service Definition Covering Connectionless-mode Transmission*, 1987. ISO 8348/AD1.

[89] International Organization for Standardization. *Information Processing — Data Communications — Protocol for Providing The Connectionless-mode Network Service*, 1987. ISO 8473.

[90] International Organization for Standardization. *Information Processing — Data Communications — Network Service Definition*, 1987. ISO 8348 (CCITT X.213).

[91] International Organization for Standardization. *Information Processing — Data Communications — Use of X.25 to Provide The Connection-mode Network Service*, 1987. ISO 8878 (CCITT X.223).

[92] International Organization for Standardization. *Information Processing — Open Systems Interconnection — Basic Reference Model*, 1987. ISO 7498.

[93] International Organization for Standardization. *Information Processing — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1)*, 1987. ISO 8824/CCITT X.208.

[94] International Organization for Standardization. *Information Processing — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One*, 1987. ISO 8825/CCITT X.209.

[95] Van Jacobson. Tgrind macros. TEX macros for typesetting program text.

[96] M. Joseph and V. R. Prasad. More on nested monitor calls. *ACM Operating Systems Review*, 12(2):21–25, April 1978.

[97] Dennis Kafura and Keung Hae Lee. ACT++: Building a concurrent C++ with Actors. *Journal of Object-Oriented Programming*, 3(1):25–37, May-June 1990.

[98] Dennis Kafura, Manibrata Mukherji, and Greg Lavender. ACT++ 2.0: A class library for concurrent programming in C++ using Actors. *Journal of Object-Oriented Programming*, 1992. To appear.

[99] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor-based concurrent object-oriented languages. In *ECOOP'89 Conference Proceedings*, pages 131–145. Cambridge University Press, 1989.

[100] Sonya E. Keene. *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[101] J. L. W. Kessels. An alternative to event queues for synchronization in monitors. *Communications of the ACM*, 20(7):500–503, July 1977.

[102] Donald E. Knuth. *The TEX book.* Addison-Wesley, 1986.

[103] Juha Koivisto and Juhani Malka. OTSO—an object-oriented approach to distributed computation. In *USENIX C++ Conference Proceedings*, pages 163–177, April 1991.

[104] Juha Koivisto and James Reilly. Generating object-oriented telecommunications software using ASN.1 descriptions. In *Proceedings of the IFIP TC6/WG6.5 International Conference on Upper Layer Protocols, Architectures, and Applications*, May 1992.

[105] Simon S. Lam. Protocol conversion. *IEEE Transactions on Software Engineering*, 14(3):353–362, March 1988.

[106] David Alex Lamb. Idl: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, July 1987.

[107] Leslie Lamport. LATEX*: A Document Preparation System.* Addison-Wesley, 1986.

[108] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. In *Seventh ACM Symposium on Operating System Principles*, pages 43–44, December 1979.

[109] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[110] Bill Leddy and Kim Smith. The design of the experimental systems kernel. In *Proceedings of the Conference on Hypercube and Concurrent Computer Applications*, Monterey, CA, 1989.

[111] Keung Hae Lee. *Designing a Statically Typed Actor-Based Concurrent Object-Oriented Programming Language.* Ph.D. dissertation, Virginia Polytechnic Institute and State University, June 1990.

[112] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[113] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[114] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Eleventh ACM Symposium on Operating System Principles*, pages 111–122, November 1987.

[115] Barbara Liskov and et al. *CLU Reference Manual.* Springer-Verlag, 1981.

[116] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[117] Barbara Liskov, Maurice Herlihy, and Lucy Gilbert. Limitations of synchronous communication with static process structure in languages for distributed computing. In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 150–159, January 1986.

[118] Barbara Liskov and Luiba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 260–267, June 1988.

[119] Andrew Lister. The problem of nested monitor calls. *ACM Operating Systems Review*, 11(3):5–7, July 1977.

[120] Carl R. Manning. Introduction to programming Actors in Acore. Presented at the ECOOP-OOPSLA'90 Workshop on Object-Based Concurrency, 1990.

[121] Christopher D. Marlin. *Coroutines: A Programming Methodology, A Language Design And Implementation.* Lecture Notes in Computer Science 95. Springer-Verlag, 1980.

[122] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. Analysis of inheritance anomaly in concurrent object-oriented languages. *OOPS Messenger*, 1991.

[123] Norman Meyrowitz, editor. *OOPSLA'87 Conference Proceedings.* ACM, October 1987.

[124] Sun Microsystems. XDR: External data representation standard. Internet Protocol Specification RFC 1014, Network Information Center, SRI International, June 1987.

[125] Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics: Volume A*. Academic Press, 1976.

[126] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[127] Robin Milner. Functions as processes. In *Proceedings of the 1990 Logic in Computer Science Conference (LICS)*, pages 167–180, 1990.

[128] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[129] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Eleventh ACM Symposium on Operating System Principles*, pages 39–51, November 1987.

[130] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the 1993 USENIX Winter Conference*, pages 29–41, January 1993. To appear.

[131] Manibrata Mukherji. The implementation of ACT++ on a shared memory multiprocessor. Masters project report, Virginia Polytechnic Institute and State University, Department of Computer Science, February 1992.

[132] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, and Robbert van Renesse. Amoeba: a distributed operating system for the 1990s. *Computer*, pages 44–53, May 1990.

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

[134] Oscar Nierstrasz. Active objects in Hybrid. In Meyrowitz [123], pages 243–253.

[135] Sean W. O'Malley and Larry L. Peterson. A new methodology for designing network software. Technical Report TR 90-29, University of Arizona, Department of Computer Science, September 1990.

[136] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 2(10):110–143, May 1992.

[137] H. Orman, E. Menze III, S. O'Malley, and L. Peterson. A fast and general implementation of Mach IPC in a network. In *Proceedings of the 1993 USENIX Mach Conference*, 1993. To appear.

[138] Michael Papathomas. Concurrency issues in object-oriented languages. In Denis Tsichritzis, editor, *Object Oriented Development*, pages 207–245. Centre Universitaire D'Informatique, Université De Geneva, 1989.

[139] David L. Parnas. The non-problem of nested monitor calls. *ACM Operating Systems Review*, 12(1):12–14, January 1978.

[140] Oren Patashnik. BIBTEXing. Paper included with the TEX source distribution, January 1988.

[141] Larry Peterson, Norman Hutchinson, Sean O'Malley, and Herman Rao. The x-kernel: A platform for accessing internet resources. *Computer*, pages 23–33, May 1990.

[142] G. Polya. *Inducation and Analagoy in Mathematics.* Princeton University Press, 1954. Volume I of Mathematics and Plausible Reasoning.

[143] John Postel. User datagram protocol. Internet Protocol Specification RFC 768, Network Information Center, SRI International, August 1980.

[144] John Postel. Internet protocol. Internet Protocol Specification RFC 791, Network Information Center, SRI International, September 1981.

[145] John Postel. Transmission control protocol. Internet Protocol Specification RFC 793, Network Information Center, SRI International, September 1981.

[146] Tomas Rokicki. DVIPS: A TEX driver. Version 5.484, 1992.

[147] Marshall T. Rose. OSI transport services on top of the TCP. *Computer Networks and ISDN Systems*, 12(3), 1986.

[148] Marshall T. Rose. ISO presentation services on top of TCP/IP-based internets. Internet Protocol Specification RFC 1085, Network Information Center, SRI International, December 1988.

[149] Marshall T. Rose. *The Open Book: A Practical Perspective on OSI.* Prentice-Hall, 1989.

[150] Marshall T. Rose. *The ISO Development Environment User's Manual, Volumes 1-5.* Performance Systems International, July 1991. Version 7.0.

[151] Marshall T. Rose and Dwight E. Cass. ISO transport services on top of the TCP. Internet Protocol Specification RFC 1006, Network Information Center, SRI International, May 1987. Version 3.

[152] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1986.

[153] William L. Scherlis. Abstract data types, specialization, and program reuse. In *International Workshop on Advanced Programming Environments.* Springer-Verlag, 1986.

[154] Claude E. Shannon and Warren E. Weaver. *The Mathematical Theory of Communication.* University of Illinois Press, 1949.

[155] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, 1992. Version 2.2.

[156] Guy Lewis Steele Jr. LAMBDA: The ultimate declarative. AI Memo 379, MIT Artificial Intelligence Laboratory, November 1976.

[157] Guy Lewis Steele Jr. and Gerald J. Sussman. LAMBDA: The ultimate imperative. AI Memo 353, MIT Artificial Intelligence Laboratory, March 1976.

[158] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[159] Joseph E. Stoy. Some mathematical aspects of functional programming. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and Its Application: An Advanced Course*, pages 217–252. Cambridge University Press, 1982.

[160] C. Strachey and C. P. Wadsworth. Continuations — a mathematical semantics for handling full jumps. Technical Report PRG-11, Programming Research Group, University of Oxford, 1974.

[161] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1986.

[162] Bjarne Stroustrup. A history of C++: 1979–1991. In *Second ACM SIGPLAN History of Programming Languages Conference (HOPL-II)*, pages 271–297, April 1993. Preprinted in SIGPLAN Notices, 28(3), March 1993.

[163] Sun Microsystems. *Sun Release 4.0 Programmer's Reference Manual*, March 1990. Section 3L: Lightweight Processes Library.

[164] Sun Microsystems. *Sun System Overview*, March 1990. Lightweight Processes.

[165] SunSoft. *SunOS 5.2 System Services*, 1993. Section 4: Multithreading.

[166] Liba Svobodova. Implementing OSI systems. *IEEE Journal on Selected Areas in Communications*, 7(7):1115–1130, September 1989.

[167] UNIX Systems Laboratories Inc. UNIX *System V Release 4 Programmer's Guide: Networking Interfaces.* Prentice-Hall, 1990.

[168] UNIX Systems Laboratories Inc. UNIX *System V Release 4 Programmer's Guide: STREAMS.* Prentice-Hall, 1990.

[169] Technical Research Center of Finland. *OTSO User's Guide*, 1992. Version 1.1.5.

[170] David L. Tennenhouse. Layered multiplexing considered harmful. In H. Rudin and R. Williamson, editors, *Protocols for High-Speed Networks*, pages 143–158. Elsevier/North-Holland, 1989.

[171] Michael D. Tiemann. *User's Guide to GNU C++.* Free Software Foundation, 1992.

[172] Chris Tomlinson, Greg Lavender, Greg Meredith, Darrell Woelk, and Phil Cannata. The Carnot extensible services switch (ESS) - support for service execution. In *Enterprise Integration Modeling: Proceedings of the First International Conference.* MIT Press, 1992.

[173] Chris Tomlinson, Mark Scheevel, and Vineet Singh. Report on Rosette 1.1. Technical Report ACT-OODS-275-91, MCC, Austin, Texas, July 1991.

[174] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA'89 Conference Proceedings*, pages 103–112, October 1989.

[175] Christian Tschudin. Flexible protocol stacks. In *SIGCOMM'91 Conference Proceedings*, pages 197–205, September 1991.

[176] David Ungar and Randall B. Smith. Self: The power of simplicity. In Meyrowitz [123], pages 227–242.

[177] Robert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. The performance of the Amoeba distributed operating system. *Software—Practice and Experience*, 19:223–234, March 1989.

[178] Richard W. Watson and Sandy A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.

[179] Peter Wegner and Scott A. Smolka. Processes, tasks, and monitors: A comparative study of concurrent programming primitives. *IEEE Transactions on Software Engineering*, SE-9(4):446–462, July 1983.

[180] Horst Wettstein. The problem of nested monitor calls revisited. *ACM Operating Systems Review*, 12(1):19–23, January 1978.

[181] Yasuhiko Yokote. *The Design and Implementation of Concurrent Smalltalk.* World Scientific, 1990.

[182] Yasuhiko Yokote and Mario Tokoro. Concurrent programming in concurrent smalltalk. In *Object-Oriented Concurrent Programming* [184], pages 129–158.

[183] Akinori Yonezawa, Etsuya Shibayama, Toshiro Takada, and Yasuaki Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In *Object-Oriented Concurrent Programming* [184], pages 55–89.

[184] Akinori Yonezawa and Mario Tokoro. *Object-Oriented Concurrent Programming.* Computer Systems Series. MIT Press, 1987.

Vita

Robert Gregory Lavender was born on 20 May 1960 in the Republic of Panama. He obtained the Bachelor of Science degree in Computer Science, Magna Cum Laude, Phi Beta Kappa, from the University of Georgia, Athens, in June 1983. Leaving a promising, but totally unenlightened 3-year career in the Washington Intelligence Community during the feckless Reagan administration, he returned to school in 1986 and received the Master of Science degree in Computer Science in May 1988, despite suffering from acute sunlight deficiency disorder during the Blacksburg winters. Escaping to MCC and the sunshine of Austin, he completed a Doctorate in May 1993.

Following graduation, Dr. Lavender plans to contemplate his future from the village of Kassiopi on the island of Kérkyra (Corfu) in the Ionian sea, off the north western coast of Greece, where acute sunlight deficiency disorder is practically unheard-of.

Dr. Lavender's research interests include concurrent programming languages, communication protocols, and models for distributed and concurrent computing. His overall interests are researching, applying, and validating theoretical concepts in a real-world domain, and maintaining a sense of humor despite all the adversity in this "modern" world.