A CLASS LIBRARY APPROACH TO CONCURRENT OBJECT-ORIENTED PROGRAMMING WITH APPLICATIONS TO VLSI CAD

BY

STEVEN MICHAEL PARKES

B.S., University of California, Davis, 1982
M.S., University of California, Davis, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

A CLASS LIBRARY APPROACH TO
CONCURRENT OBJECT-ORIENTED PROGRAMMING
WITH APPLICATIONS TO VLSI CAD

Steven Michael Parkes, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1994
Prithviraj Banerjee, Advisor

Despite increasing availability, the use of parallel platforms in the solution of significant computing problems remains largely restricted to a set of well-structured, numeric applications. This is due in part to the difficulty of parallel application development, which is itself largely the result of a lack of high-level development environments applicable to the majority of extant parallel architectures. This thesis addresses the issue of facilitating the application of parallel platforms to unstructured problems through the use of object-oriented design techniques and the actor model of concurrent computation. We present a multilevel approach to expressing parallelism for unstructured applications: a high-level interface based on the actor and aggregate models of concurrent object-oriented programming, and a low-level interface which provides an object-oriented interface to system services across a wide range of diverse parallel architectures. The interfaces are manifested in the ProperCAD II library, a C++ object library supporting actor concurrency on microprocessor-based parallel architectures and appropriate for applications exhibiting medium-grain parallelism. The interface supports uniprocessors, shared memory multiprocessors, distributed memory multicomputers, and hybrid architectures comprising network-connected clusters of uni- and multiprocessors. The library currently supports workstations from Sun, shared memory multiprocessors from Sun and Encore, distributed memory multicomputers from Intel and Thinking Machines, and hybrid architectures comprising IP network-connected clusters of Sun uni- and multiprocessors. We demonstrate our approach through an examination of the parallelization process for two existing unstructured serial applications drawn from the field of VLSI computer-aided design. We compare and contrast the library-based actor approach to other methods for expressing parallelism in C++.

A CLASS LIBRARY APPROACH TO
CONCURRENT OBJECT-ORIENTED PROGRAMMING
WITH APPLICATIONS TO VLSI CAD

Steven Michael Parkes, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1994
Prithviraj Banerjee, Advisor

Despite increasing availability, the use of parallel platforms in the solution of significant computing problems remains largely restricted to a set of well-structured, numeric applications. This is due in part to the difficulty of parallel application development, which is itself largely the result of a lack of high-level development environments applicable to the majority of extant parallel architectures. This thesis addresses the issue of facilitating the application of parallel platforms to unstructured problems through the use of object-oriented design techniques and the actor model of concurrent computation. We present a multilevel approach to expressing parallelism for unstructured applications: a high-level interface based on the actor and aggregate models of concurrent object-oriented programming, and a low-level interface which provides an object-oriented interface to system services across a wide range of diverse parallel architectures. The interfaces are manifested in the ProperCAD II library, a C++ object library supporting actor concurrency on microprocessor-based parallel architectures and appropriate for applications exhibiting medium-grain parallelism. The interface supports uniprocessors, shared memory multiprocessors, distributed memory multicomputers, and hybrid architectures comprising network-connected clusters of uni- and multiprocessors. The library currently supports workstations from Sun, shared memory multiprocessors from Sun and Encore, distributed memory multicomputers from Intel and Thinking Machines, and hybrid architectures comprising IP network-connected clusters of Sun uni- and multiprocessors. We demonstrate our approach through an examination of the parallelization process for two existing unstructured serial applications drawn from the field of VLSI computer-aided design. We compare and contrast the library-based actor approach to other methods for expressing parallelism in C++.

DEDICATION

To my parents, Dayle and Genie

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Prithviraj Banerjee, for supporting and directing this work. I would like to thank the members of my committee, Professors Agha, Chien, Hwu, Patel, and Polychronopoulos, for their efforts in reviewing and critiquing my progress and this thesis.

I would like to express my thanks to the members of the ProperCAD and Paradigm projects for both their technical help in issues of parallel computing and their camaraderie. I would especially like to thank John Chandy, my officemate, who not only served as a source of insight and a sounding board for ideas but also developed several large applications on the ProperCAD II library with virtually no documentation.

I would like to thank the members, both present and former, of the Center for Reliable and High Performance Computing. Special thanks are due to Ken Kubiak for technical insight, encouragement, and empathy.

I owe a great debt to my parents, Dayle and Genie, and my siblings, Cheryl, Chris, and Debbie, for their invaluable love, encouragement, and understanding during my doctoral degree program.

This work was supported in part by the Semiconductor Research Foundation. The Argonne National Laboratory, the San Diego Supercomputer Center, and the National Center for Supercomputer Applications provided support by providing access to their computing resources.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

LIST OF INTERFACES

Chapter 1

INTRODUCTION

The desire to utilize multiple processors to solve significant computing problems has, to date, been largely unattainable for all but a set of restricted problems, namely the numerical problems found in scientific applications and the database problems found in transaction-processing applications. While substantial computing problems exist in other fields, the techniques and implementations used in the parallelization of scientific and transaction-processing applications have not proven similarly effective on unstructured problems. The lack of structure in these classes of problems de-emphasizes floating point vector operations while it emphasizes operations that comprise a mixture of integer and floating point instructions on pointer-based data structures. Existing parallelization methods, both manual and automatic, often fail on this class of application to achieve results comparable to those on numeric and structured applications.

With increasing interest in the parallelization of a larger set of applications comes a shift in the way in which parallelization is approached. For example, many classical parallel applications have been developed to solve specific research problems; these efforts were often targeted toward specific architectures—those available to the researchers. Because the result of the research was the knowledge gained by solving a particular problem and not a parallel application, the dependence on a particular architecture was not considered a significant drawback.

In contrast, as parallel machines have proliferated, a broader range of application designers has been attracted. For these designers, the lack of a dominant architecture or particular machine engenders the need to pursue an architecture-independent solution to archive a cost-effective solution. Moreover, a significant degree of new interest in scalable platforms is coming from vendors of existing serial applications. As a result, there is need for parallelization methods that can be incrementally applied to existing code. Finally, because the cost of developing parallel software is inherently greater than that of developing serial software, the recent interest in code reuse is at least as strong in parallel processing as it is in serial processing.

Technologies from concurrent object-oriented programming can be used to address each of these issues. Abstract high-level models can be used to provide a degree of insulation from architectural details. Encapsulation via well-defined interfaces can be used to facilitate modular development and code reuse. Inheritance and dynamic binding can be used to facilitate incremental parallelization of existing serial object-oriented applications.

While these technologies exist in the field of concurrent object-oriented programming, they take numerous diverse forms. Generating a cohesive interface that meets all application needs is probably infeasible; not only do the requirements of different fields vary widely, the evaluation of a 'good' interface often varies among individuals, even within the same field.

In this work, we present an interface for concurrent object-oriented programming that is applicable to computer-aided design (CAD) applications for VLSI. To the extent that VLSI CAD applications are characteristic of large C++ applications, the interface should also find application in other domains. The interface is defined in terms of C++ classes and implemented in the ProperCAD II class library. In the remainder of this chapter, we consider the characteristics of VLSI CAD applications, briefly comment on how a class library can be used to parallelize CAD applications, summarize the contributions of this research, and give an overview of the chapters that follow.

1.1 Computer-Aided Design for VLSI

To design increasingly complex VLSI systems, continued—and in some cases radical—progress is required in design technologies, especially the algorithms and applications for VLSI CAD. CAD applications differ substantially from the scientific applications which have traditionally formed the bulk of supercomputing workloads. CAD applications are characterized by:

- long execution times, sometimes more than a week for individual runs on contemporary uniprocessor platforms
- qualities of result which are directly dependent on the magnitude of computing resources applied
- a direct correlation between application turnaround time and design cycle length

- multimegabyte data sets
- irregular, unstructured data organizations
- resistance to well-known parallelization techniques

Examples of VLSI CAD problems are test pattern generation, logic synthesis, circuit extraction, and cell placement and routing.

Automatic test pattern generation (ATPG) for VLSI circuits is the process of generating *test patterns*, sets of inputs to integrated circuits that are applied to fabricated devices to determine if any defects occurred during manufacturing. Although the complexity of ATPG is daunting—it is an NP-complete search problem—it is nonetheless considered indispensable for maintaining the manufacturing quality of ever larger VLSI devices.

Logic synthesis comprises the creation and optimization of digital circuits represented as netlists of logic and state elements. Because of increasing circuit densities, the last decade has seen a considerable increase in interest in algorithms for the automatic synthesis of VLSI circuits. There is industry consensus that only through synthesis will it be possible to manage the design complexity of the current and future generations of VLSI chips. Most synthesis algorithms are both memory and processor intensive and display a quality of results tightly coupled to the resources applied.

Circuit extraction is the process of taking a VLSI mask-level layout and extracting circuit connectivity and parametric values. The results of extraction are used to verify both design correctness and performance requirements, usually after automatic placement and routing. Extraction is typically performed on a circuit description provided in terms of rectangles on various mask layers. The number of rectangles is approaching 100 million in contemporary microprocessor designs; few platforms available in industry have the resources to handle these designs efficiently. Given the frequency of use—extraction is iterated with design changes to verify changes and to update extracted parametric information—techniques to take advantage of all available resources are invaluable.

When the logic design for a VLSI circuit has been completed, cell placement and routing are performed. With chips approaching tens of millions of gates, the time required for this process on large chips often exceeds days and is quickly approaching weeks on state-of-the-art workstations. As in other CAD tasks, the quantity of resources applied to the problem has a direct impact on the quality of results.

Even with the preponderance of evidence indicating that virtually any method of managing the application development process and any technique for improving quality through additional resources would appear promising, it is still the case that neither parallel processing nor object-oriented techniques are well represented in the CAD development community. This situation is not without justification for a number of reasons:

- For more than a decade, most CAD development has been performed in the C programming language, and until the advent of C++, use of an object-oriented language implied the sacrifice of existing code, an unacceptable alternative. Even with the availability of C++, adoption is slow; C++ is significantly more complicated than C and is still undergoing rapid development. Development tools are only now attaining the degree of stability required for even the most aggressive commercial development.

- Due to a lack of widely available libraries, much of the promise of code reuse associated with object-oriented programming remains to be realized. Given the ability in C++ to trade flexibility for efficiency, the process of generating reusable code is complicated by the fact that the set of design choices, in terms of flexibility versus overhead, for one application may not be acceptable for another; CAD problems, with their inherent complexity and size, are known to be sensitive to overheads in area or space.

- Until recently, the widespread availability to CAD users of parallel platforms has been severely limited. Supercomputers have in general been limited to the restricted application domains mentioned previously. The techniques developed for these platforms have had no place for application in the CAD community. With little availability of parallel platforms, interest in parallel solutions to CAD problems has been relatively low. With only limited development of parallel applications, little impetus exists for CAD users to explore the cost benefit trade-off of parallel platforms.

- The generation of efficient parallel algorithms has been impeded by concurrent rapid improvements in serial algorithms. Often by the time a parallel CAD algorithm is completed, it lags significantly behind the quality, and sometimes even the performance, of contemporary serial algorithms. Given the high cost of developing new applications, support for a parallel track of separate parallel tools which mirrors a set of serial tools is prohibitive.

1.2 A Class Library for Concurrent Object-Oriented Programming

Based on this past experience, a strong argument can be made that in order for a parallelization environment to be applicable to VLSI CAD it must:

1. be architecture-independent, that is, portable across the majority of parallel machines
2. facilitate incremental parallelization, that is, enable parallel development that may be viewed as a customization of a serial application rather than as an independent effort
3. be tightly integrated with the target language in order to minimize difficulties due to a paradigm shift

These constraints are addressed in the approach taken in the ProperCAD project [1], under which the work presented here was performed (Figure 1.1). The goal of the ProperCAD project is the development of a library which provides a seamless, high-level programming model across a variety of parallel architectures, along with a set of applications based on that model that addresses the most significant tasks in electronic design automation [2, 3, 4].

In this research, we have developed an interface for concurrent object-oriented programming defined in terms of C++ classes and implemented in the ProperCAD II library. The interface consists of two major parts: a high-level interface representing a synthesis of the

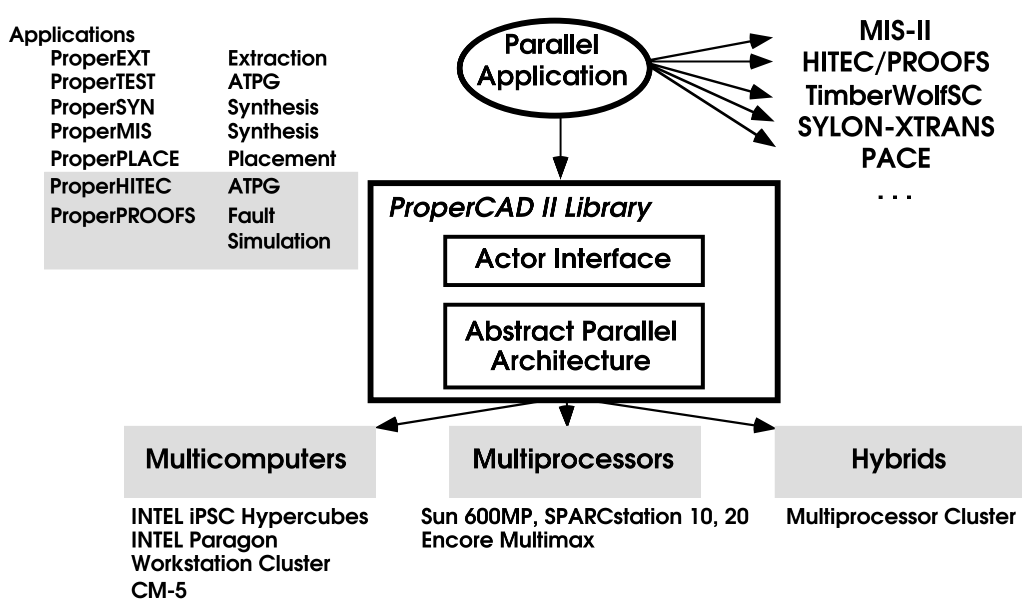

Figure 1.1 An overview of the ProperCAD project

actor model of concurrent object-oriented programming with a statically typed imperative language, C++, and a low-level abstract machine interface that can be used to parametrically describe a wide variety of concrete architectures.

The development of the interface has been driven by and evaluated against two new parallel applications, parallel test pattern generation and parallel fault simulation, each incrementally developed from an existing state-of-the-art serial application. In addition to providing a basis for evaluating the usability of the interface, these applications embody new approaches to their target problems.

1.3 Summary of Contributions

The primary contributions of this thesis are:

1. A class library interface for actor-based parallelism in a statically typed language and based on built-in type mechanisms.

2. An implementation of aggregates providing the functionality described in [5] with additional meta-programmability features.

3. An interface supporting composable meta-programming of an actor system on contemporary microprocessor-based machines.

4. An open implementation supporting application-specific customization of the run time support system.

5. An abstract parallel architectural model with a class library interface, capable of parametrically describing the majority of contemporary parallel architectures.

6. New parallel algorithms for test pattern generation and fault simulation incrementally derived from state-of-the-art serial algorithms.

1.4 Overview

In Chapter 2, we review recent work in the field of concurrent object-oriented programming and consider application of object-oriented programming techniques to unstructured problems. Chapter 3 presents the primitives in the actor interface. In Chapter 4, we consider a platform for implementation of the actor primitives, the abstract parallel architecture. Chapter 5 presents extensions of the actor interface to support for meta-programmability.

Chapter 6 introduces test generation, an unstructured application drawn from the area of VLSI CAD, and demonstrates how the interfaces developed in this work are used to parallelize an existing serial test application. Chapter 7 presents the parallelization of fault simulation by using the actor and aggregate models and reports an implementation based on the ProperCAD II library. Chapter 8 summarizes our experiences parallelizing serial applications using the new interface, presents observations on the most significant success of the interface, and proposes features that would extend the usability of the interface and implementation.

Chapter 2

CONCURRENT OBJECT-ORIENTED PROGRAMMING

Concurrent computing is the use of multiple processors to solve a single problem. Concurrent machines have existed almost as long as computers themselves, yet the use of concurrency to improve run times and results is still limited to a few specialized areas. This is in large part due to the difficulty in producing concurrent programs that are both effective in improving performance and manageable from a development perspective.

When creating concurrent programs, the programmer must deal with two competing issues: expression of concurrency and control of the state interference caused by concurrency. Expression of concurrency is the task of breaking a problem into multiple subproblems that can be then processed concurrently. In this respect, a serial program is the degenerate case of a concurrent program. If insufficient concurrency is expressed in a program, the effectiveness of the program will be limited. A program with limited concurrency is not scalable, i.e., there is a bound on the number of processors which can be applied to the problem beyond which processor utilization falls dramatically. While limits in scalability argue for highly concurrent programs, programs of this type have drawbacks as well. First, if a highly concurrent program has a low ratio of computation to communication, it may fail to achieve sufficient processor utilization. Second, when additional concurrency is expressed, the issue of state interference arises. If two concurrent tasks reference data whose values change as computation progresses, the likelihood of race conditions due to intertask interference rises. Thus, when a programmer expresses additional concurrency, he must at the same time ensure that state interference does not invalidate the result of the program.

In concurrent object-oriented programming [6], the object model and object-oriented programming primitives are used to address issues of expression of concurrency and management of interference. In this chapter, we broadly review hardware architectures, programming models, and programming model implementations. We conclude the chapter with a broad overview of the library interface developed in this work and presented in greater detail in following chapters.

2.1 Hardware Architectures

Parallel architectures cover a broad range of implementations, from low concurrency systems composed of as few as two processors to highly-concurrent systems with thousands of processors. Concurrent architectures also vary in both the manner and efficiency of communication, from systems with interprocessor bandwidths of hundreds of megabytes per second and communication frequency on the order of a few instructions to systems with a few megabytes per second bandwidth and communication latencies in milliseconds.

Traditionally, concurrent architectures could be cleanly divided into two classes, *shared memory* architectures and *distributed memory* architectures. In recent years, *distributed shared memory* (logically shared, physically distributed) architectures such as the Kendall Square KSR-1 [7] have been introduced. Additionally, with the rapid increase in interconnectivity via local- and wide-area networks, virtually all machines can now be considered concurrent in as much as they support some form of message passing interconnection.

This section considers a few key characteristics of each of these architecture classes. It should be noted that these are *hardware* architectures and do not necessarily reflect the programming model as viewed by the application programmer. It is possible, via compilers, run time libraries, and operating systems, to implement any of the application programming models of the next section on any of the hardware architectures presented below. We use the term *low-level programming model* to describe the model supported directly by the hardware and *high-level programming model* to describe the model seen by applications. High-level programming models are considered in Section 2.2.

2.1.1 Shared memory

Shared memory architectures are constructed from a number of processing and memory modules which are connected via an interconnection network (Figure 2.1). In the first generation of shared memory machines, processing modules generally lacked memory other than that represented by the registers in each CPU and the interconnection network was generally a bus. Both of these characteristics led to scaling problems. To improve performance,

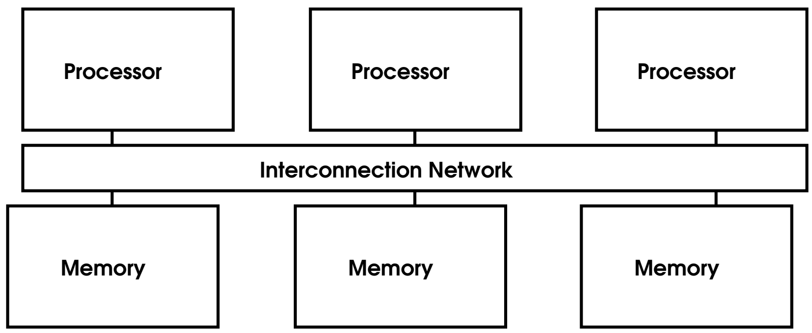

Figure 2.1 Shared memory multiprocessor

local caches were added to each processor. While this addition drastically cuts the latency of memory references it introduces problems of cache coherency. More advanced interconnection networks have also been developed, including crossbar and multistage networks. Even though the addition of cache mechanisms has enabled the scaling of shared memory machines to larger sizes, the difficulty of maintaining cache coherence typically limits the feasibility of this approach to tens of processors. It is the cache coherence of these models that distinguishes them from other uniform address space models.

2.1.2 Distributed memory

In distributed memory architectures, each processing module consists of both a processor and local memory. Processor modules are interconnected by a network (Figure 2.2). The most significant characteristics of distributed memory systems are the individual address space of each processor and explicit access to the network via primitives such as `send` and `receive`. The bandwidth of the network in these architectures has traditionally been less than that of shared memory machines—often an order of magnitude less than those of bus-based shared memory machines—and often varies among different pairs of processors. However, as a result, these architectures are considered more scalable than shared mem-

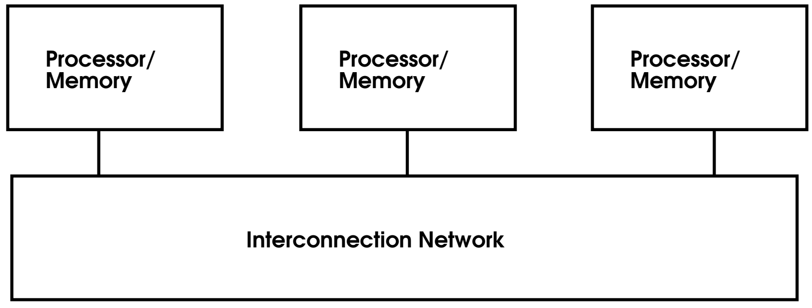

Figure 2.2 Distributed memory multiprocessor

ory architectures. Machines of this type have been built with thousands of processors. The lower bandwidth of the network means that algorithms that perform well on shared memory machines may perform poorly on distributed memory architectures, if a naïve mapping of shared memory reference to distributed memory communication primitive is used.

2.1.3 Distributed shared memory

In the last decade, a new architecture has been developed using techniques borrowed from both the pure shared memory and pure distributed memory architectures. This architecture uses hardware components similar to those developed for purely distributed memory to implement a low-level programming model that mirrors shared memory. Though an interconnection network is used, the hardware does not support explicit `send` and `receive` primitives. Instead, the low-level programming model is a uniform address space, and the hardware detects accesses to nonlocal memory, sending the appropriate messages to gain access to the necessary data. Consistency in these systems is usually maintained via a combination of hardware and software in a mechanism called a directory [8]. Contemporary machines in this class are nonuniform memory architecture (NUMA) machines such as the Stanford DASH [8] and cache-only memory architecture (COMA) machines such as the KSR-1 [7].

2.1.4 Hybrid shared and distributed memory

Hybrid architectures combine shared and distributed memory architectures without adopting a completely shared or distributed model. Some processors will share a uniform address space; otherwise, communication requires explicit `sends` and `receives` (Figure 2.3). With the recent concurrent growth in workstation clusters and desktop multiprocessors, hybrid machines are becoming ubiquitous. Furthermore, massively parallel processor (MPP) manufactures are beginning to consider hybrid architectures for their machines; the Intel Paragon supports configurations which have shared memory multiprocessors at each node within the mesh interconnect [9].

2.2 High-level Programming Models

In this context, "high-level" indicates the programming model to which an application is written; this may differ from the model used by the underlying operating system. The programming model can be broken into two components, a *communication model* and a *thread*

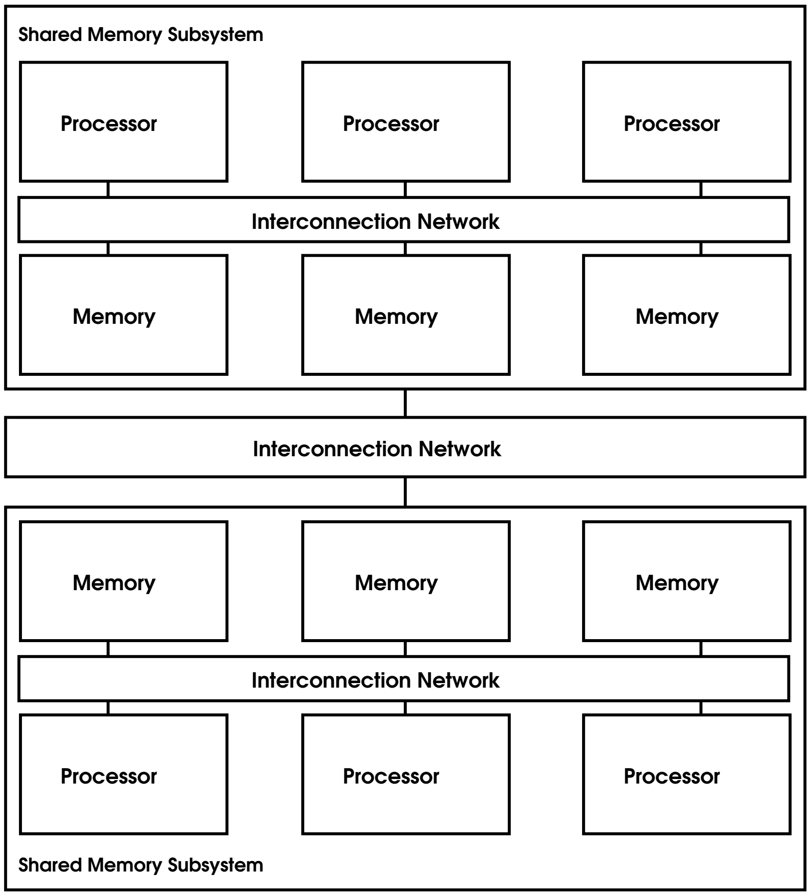

Figure 2.3 Hybrid multiprocessor

model. Though a complete model requires aspects of both components, separate consideration of the components helps clarify the issues while illustrating the space of complete models. While only a few combinations are currently in use, almost any combination of communication and thread models can form a new complete programming model. These models are high-level and thus with proper software support could be implemented on any of the hardware architectures described in the previous section, albeit possibly at higher cost if the programming and hardware models are dissimilar.

2.2.1 Communication model

The communication model defines how the "threads" of a program coordinate data interchange amongst themselves. Though we use the term 'thread' in this section, we defer definition to Subsection 2.2.2. For the purposes of this subsection, threads may be considered an active execution environment (stack) that vies for processor time with other threads.

2.2.1.1 Shared memory

In the shared memory programming model, the application sees a single flat address space. Communication is implicit, through access to shared variables. To fully support such a model, a method of interprocessor synchronization is necessary, usually implemented at the lowest level via atomic operations such as `test-and-set` or through higher-level abstractions such as barriers. The exact semantics of synchronization, i.e., busy-wait versus rescheduling, cannot be defined precisely without reference to a thread model.

Specifying the exact semantics of shared memory machines is complicated by the existence of different *consistency models*. With hardware support for caching and load/store reordering, the most conservative model of shared data consistency, *sequential consistency* [10], is prohibitively expensive. Thus, in addition to the uniform address space, a shared memory model must explicitly define the aspects, both deterministic and nondeterministic, of access to shared memory.

An example of shared memory communication is shown in Figure 2.4. In the figure, two threads access two shared integer variables, `a` and `b`. Because the model specifies a uniform address space, the variables exist at the same addresses in both threads. The figure demonstrates the consistency problem. While the first thread stores five into `a` followed by storing ten into `b`, it is possible, under existing consistency models, for the second thread to see the change to `b` before seeing the change to `a`.

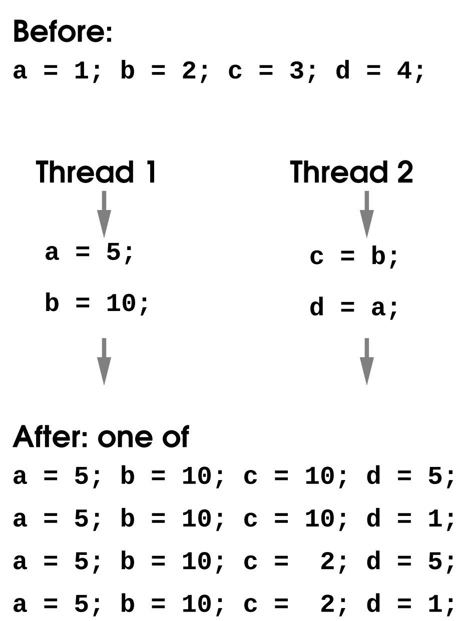

Figure 2.4 Communication and consistency in shared memory

2.2.1.2 Distributed memory

In the distributed memory model, a `send` primitive is used to send data from one thread to another thread. An explicit `receive` operation must be executed before the data is available to the receiving thread. Figure 2.5 shows two possible ways communication can occur in a distributed memory model. In the blocking case, the `send` operation does not complete until the corresponding `receive` operation has begun. This style of communication is one of the types supported by the Thinking Machines CM-5 CMMD library [11]. In the non-blocking case, the `send` operation completes when the necessary data are copied out of the application buffer; it is not necessary that the corresponding `receive` be executed. This type of communication is common on Intel multicomputers [12]. Variations on communications primitives provide for broadcasting, multicasting, synchronous and asynchronous communication, and typed messages.

2.2.2 Thread models

There are two principal thread models, *physical* and *virtual.*

2.2.2.1 Physical threads

In a physical thread model, the thread, as seen by the application, is a processing element and is generally available for “exclusive” use by the application; if the code running on a thread executes a blocking call, the entire thread is blocked for the duration of the call. The physical model includes those systems which provide more threads than processors but for which the application cannot rely on the ability of the underlying run time library to efficiently handle a number of threads vastly greater than the number of available processors.

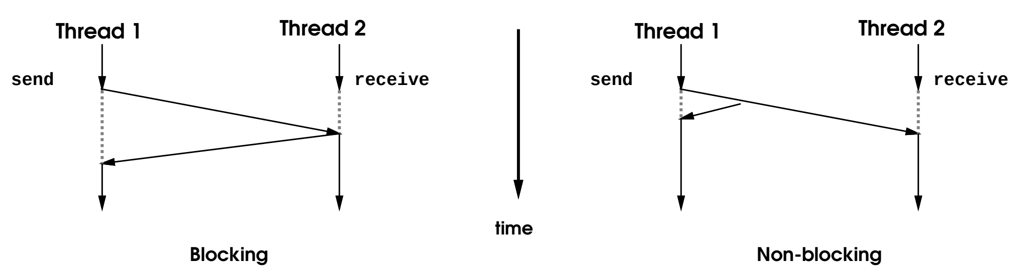

Figure 2.5 Communication on distributed memory architectures

2.2.2.2 Virtual threads

In a virtual thread model, the number of threads visible to the program is typically not related to the number of processing elements. Applications developers think in terms of a number of threads convenient for the application at hand. It is the responsibility of the underlying language and run time support to map these virtual threads to physical processors. When a virtual thread blocks, it is expected that the underlying run time support will find another virtual thread to schedule. This model is often referred to as *light-weight threads.*

2.2.3 Complete models

A complete model is created by combining a communication model with a thread model and then specifying the semantics of the interaction between the two components. We consider three complete high-level models; many more are possible, but the examples serve to demonstrate the issues involved.

2.2.3.1 Unix shared memory multiprocessor

The model generally found on Unix shared memory machines combines shared memory communication and physical thread components. The underlying consistency model is that of the underlying hardware and varies from architecture to architecture. The model is considered physically threaded, because Unix operating systems—in particular, the scheduler—typically do not perform well when the number of kernel-scheduled threads is far greater than the number of processors, e.g., 100 times or more.

2.2.3.2 MPI

The Message Passing Interface (MPI) is a send-receive model recently standardized by a consortium of manufactures and users [13]. The model is similar to the programming interfaces on the Intel iPSC and Paragon multicomputers and those supplied by libraries such as PVM [14] and Express [15].

The basic model provides a send-receive interface atop a physically threaded model. Programs written for this model are written for a fixed number of processors. While they can be parameterized by the size of the machine, the model provides little or no support for context switching of a physical thread over multiple virtual threads.

In MPI, the basic send-receive model is augmented with such concepts as process groups and environments, which makes the interface more applicable to virtual threads. Experience with this advanced virtual thread interface on distributed memory systems is limited.

2.2.3.3 Actors

The actor model is based on continuation passing [16] using virtual threads. From the application viewpoint, an actor is an object with its own thread of control that at most times is blocked waiting for a message—a continuation execution of one of its methods.

Continuation passing resembles message passing but omits the explicit receive operation. Instead, each message sent contains enough information to determine the action to be invoked by the receiver. In this way, the model is similar to *active messages* [17].

The actor model combines continuation passing with a virtual thread model. Most of the time, the thread associated with an actor is blocked awaiting the continuation execution of one of its methods. When a continuation destined for the actor is executed, the appropriate member function is executed. As part of the execution, the actor may execute other continuations, create new actors, or perform computations, possibly with side effects.

In a parallel context, continuations express parallelism; the call to a continuation returns after scheduling the future execution of the continuation body rather than synchronizing with the actual execution. Since every actor has a thread and an actor is simply a concurrent object, actor programs typically have thousands, if not hundreds of thousands, of virtual threads. Thus, the virtual thread mechanism used by an actor implementation must be efficient.

As the actor model is the model adopted in this work, further discussion and examples are deferred until discussion of the Actor Interface in Chapter 3.

2.3 Implementation Architectures

Many of the choices made in the development of the characteristics of the interface in this work involved consideration of what developers of the applications in the targeted fields would need and like. A study of these issues is presented by Pancake et al. [18], [19]; while their work was centered on scientific problems, with a few exceptions their observations and conclusions are equally applicable to other problem domains. We borrow from their discussion, as we consider the characteristics desirable in VLSI CAD and similar domains.

2.3.1 Algorithm classifications

2.3.1.1 Regular versus irregular

Regular applications are those whose computation demonstrates a regular pattern, i.e., the iteration of a large number of operations on large contiguous regions of memory with little change in control flow or memory access patterns. Examples are signal processing applications which compute fast Fourier transforms and other transforms on dense arrays of raw data.

In contrast, irregular problems do not deal with dense contiguous data structures. This class includes not only algorithms defined on completely arbitrary data structures but also algorithms which have a high degree of conceptual regularity but a low degree of regularity in implementation. Among this later class are sparse matrix operations; while the multiplication of two sparse matrices is at a high level a regular operation, in terms of low-level operations, control and data access patterns differ significantly from that of the analogous dense operation. Typically, the metric used to differentiate regular from irregular is the applicability of vector operations.

2.3.1.2 Structured versus unstructured

For our purposes, we define unstructured problems as those that are not naturally expressible as operations on (possibly sparse) vectors and matrices or through iterative solution of a number of conceptually identical problems. Thus, while a sparse matrix operation is not a regular operation, it is highly structured; the operation can be succinctly described as a simple iteration over two graphs.

In contrast, unstructured problems are those for which a number of heterogeneous tasks must be performed. For example, in a CAD application, it may be necessary to concurrently process cost estimation tasks, database update tasks, and global status coherence tasks. The number of unstructured problems implemented only on uniprocessor is large; the area of CAD for VLSI circuits is dominated by such applications.

2.3.1.3 Numeric versus non-numeric

Traditionally concurrent processing has been applied to predominately numeric applications. For example, physical modeling tasks are well represented in current parallel processing applications. These applications tend to be dominated by floating point operations on data representing physical quantities.

While there are a number of problems involving physical modeling in the domain of VLSI CAD, the area is dominated by applications for which floating point operations do not predominate. These applications, such as logic synthesis, test generation, and logic simulation, are tightly tied to the logic model of digital circuits and tend to be dominated by integer and logic instructions.

2.3.1.4 Fine-grain versus medium-grain versus coarse-grain

The granularity of a parallel algorithm refers to the relative ratio of computation to commutation. A fine-grain application expresses far more concurrency and utilizes far more communication than a medium-grain application. It is difficult to affix a label to any particular algorithm, since these terms are largely relative. We choose to describe our approach as applicable to medium-grain concurrency and to describe the most useful distinctions between fine- and coarse-grain without reference to instruction counts.

Fine-grain applications are those for which virtually every operation is concurrent. This concurrency may be implied by data distribution as in a data parallel context or by implicit concurrency in all operations as in the actor model.

We consider coarse-grain parallel algorithms to be those which either limit the number of effectively processed concurrent tasks to the number of physical processors in the machine or for which the communications protocol implies crossing a protection boundary. Thus, an algorithm that is coarse-grained generally is written explicitly in terms of the size of the machine, i.e., it is physically threaded, or expects intervention in communication as is the case for distributed processing via remote procedure calls (RPCs) [20].

Essentially, applications which do not fit either of the previous two categories may be considered medium-grain, and thus many systems can be considered medium grain. The predominant characteristics of medium grain would then be:

- existence of a mixture of concurrent and nonconcurrent objects with concern for efficiency as the number of concurrent objects grows.
- an assumption of efficiency in communication, i.e., when possible, communication among concurrent tasks should be on the order of a function call in a serial language.

2.3.2 Interface levels

2.3.2.1 Low-level

A low-level model essentially reflects the underlying hardware with little or no abstraction. Such a model is clearly not architecture-independent.

2.3.2.2 High-level

A high-level model provides a degree of distance or abstraction from the underlying hardware. A high-level model should be implementable across a range of architectures, although with varying degrees of efficiency. While high-level models are conceptually very attractive, they may suffer from greater overheads and may inhibit some kinds of optimizations.

2.3.2.3 Mixed-level

Because high-level interfaces either incur too much overhead or because a degree of architectural tuning is required, many have been been augmented with low-level features, which leads to a mixed-level interface. While mitigating the disadvantages of a purely high-level model, a mixed model can become difficult to understand and manage [18].

2.3.2.4 Multiple levels

Rather than mixing high- and low-level features, two models can be provided, one high and one low, with a well-defined interface. This combination of models should provide greater flexibility than a purely high-level model while helping minimize the confusion resulting from a single interface with arbitrarily mixed abstractions.

2.3.3 Expression of parallelism

One can view the gamut of parallelism as a one-dimensional space with extremes representing completely serial and completely parallel programs. This situation is portrayed graphically in Figure 2.6, where arrows indicate increasing programmer effort.

If we consider, for example, the MPI and actor programming styles, we see that they lie on opposite ends of the parallelism spectrum. If we consider a send-receive code that sends two messages from a thread A to a thread B, in the most straightforward, synchronous case, parallelism is decreased by the imperative `receive` primitive:

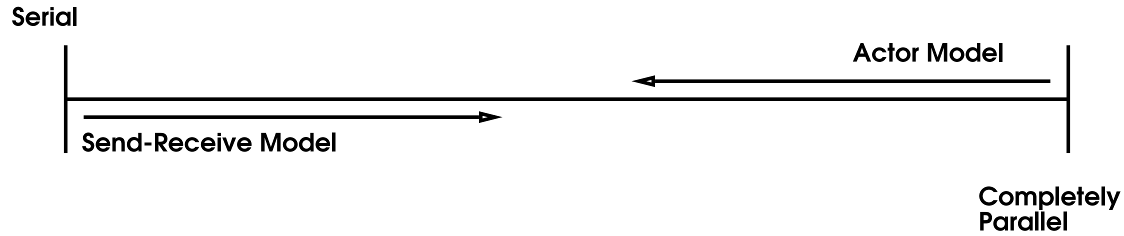

Figure 2.6 Spectrum of parallelism

```
// Thread 1                         // Thread 2
send( 2, type1, data1 );            receive( type1, data1 );
send( 2, type2, data2 );            receive( type2, data2 );
```

Even if the semantics of the application may allow the messages to be processed in any order, the underlying model requires that the messages be received in the order sent, and the serial nature of code execution implies that one receive must precede the other. Though it is possible to express unordered reception via asynchronous `send` and `receive` primitives, such code is significantly more difficult to write. Thus, the basic MPI interface is represented at the left of the parallelism spectrum; the developer must apply more effort to express greater parallelism.

In the actor model, message reception may occur as long as the actor is not processing a prior message. In contrast to the MPI case above, the actor will receive whichever message arrives at the actor thread first. There is, however, a dual of the MPI case; if there is a dependence constraint on message processing, the actor must handle the case in which message reception order is reversed. In this case, the actor must delay processing of the second message until the first is received and processed. Thus, the actor model starts at the completely parallel end of the spectrum and requires that the programmer apply more effort to express less parallelism. The extreme amount of parallelism in an actor program can sometimes lead to difficulty in programming. To remedy this problem, most actor languages include constructs for shifting order maintenance from the programmer to the run time system. As will be seen in Chapter 3, combing the actor model with an imperative language such as C++ also serves to simplify the expression of parallelism without putting an unacceptably large burden on the developer.

2.3.4 Concurrency expression

There are a number of methods for expressing concurrency in wide use; among the most popular are data parallelism, task parallelism, and actor parallelism.

2.3.4.1 Data parallelism

In data parallelism, parallelism is implicit in certain data structures, usually in arrays. Operations on arrays implicitly imply parallelism; it is the responsibility of the compiler and run time support to implement the parallelism. Data parallelism is most often applied to dense arrays but also may be applied to sparse data structures.

2.3.4.2 Task parallelism

In task parallelism, parallelism is explicit, usually in the form of a parallel `parfor` or `dowhile` construct or an imperative `spawn`. In these cases, the body of the loop or the target of the spawn is performed concurrently with other parts of the computation. These types of parallel expression imply a degree of linkage between concurrency and the lexical structure of the program.

2.3.4.3 Actor parallelism

In actor parallelism, the unit of concurrency is the actor, which is also the unit of interference control. Concurrency in an actor program is implicit in the semantics of the actor model. Interprocess communication is expressed via *continuation execution,* an extension of the member function execution mechanism of serial object-oriented languages. There are no imperative constructs for parallelization or synchronization similar to the `parfor` and `barrier` primitives typically found in task parallelism.

2.3.5 Method of implementation

Pancake and Bergmark classify interfaces as concurrent languages, language extensions, and run time libraries [18].

2.3.5.1 Languages

Language implementations provide the greatest flexibility but have the drawback of requiring more effort in development and maintenance of implementation, higher costs in training efforts, sacrifice of existing code, and inability to overcome inertia of existing usage. In the case of concurrent languages, a significant amount of effort can be required to specify the syntax and semantics of serial portions of the language, which detracts from the effort to address issues of concurrency.

2.3.5.2 Language extensions

Language extensions mitigate, but do not eliminate, the difficulties of language implementations. An extension of an existing language represents less flexibility in expression but gains greater ability to utilize existing technologies in implementation. Training and adoption difficulties are fewer than in the language cases, but they are still significant.

2.3.5.3 Libraries

Libraries provide the lowest startup costs, but they do so at the cost of least flexibility in expression and often the most effort in expressing parallelism. On the other hand, libraries provide the greatest ability to coexist with existing libraries and development tools.

2.4 Composability

Of particular interest in this work is the ease of *composability*, the process of taking two existing modules for solving subproblems of a computation and combining them into a single application to solve a more complicated problem. For example, an application might require the use of a linear system solver and a matrix multiplication package. In this example, the implementation and component packages are composable if it is possible to take existing modules for the solver and multiplication tasks and use them, without modification, to solve the appropriate subproblems.

In composability, physical thread-based models tend to fall short of the virtual techniques because partitioning, usually static, of available processors is required. Designers of distributed memory interfaces such as MPI are implementing ways of solving this problem, but as yet there are few examples of such features.

Implementations of the actor model, being based on a virtual thread model, implicitly perform load balancing when a new actor is created. Thus, if an actor type exists for each of the solver and multiplication problems, an actor can be created for each problem without interfering with the other. It is the task of the run time system to load balance and schedule the actors to take advantage of available processing elements.

Figure 2.7 shows the difference between typical message passing and actor implementations of the linear system solver and matrix multiplication problems. In the message passing case, we assume that a procedure exists for each subproblem and that each procedure was designed assuming full use of the parallel machine. Each of the two subproblems is solved internally in parallel, but the two subproblems are sequentially ordered with barriers. In the

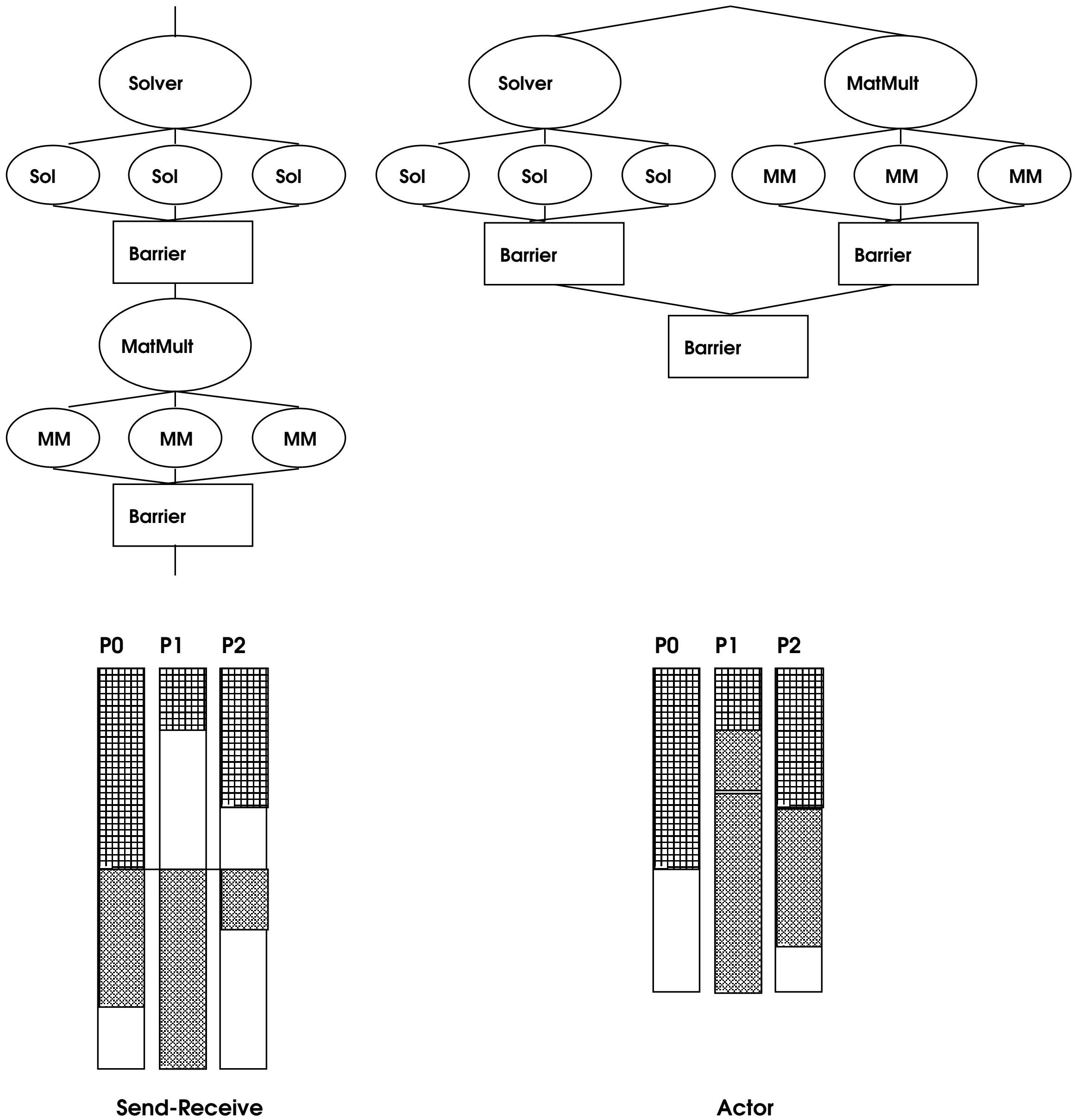

Figure 2.7 Composability in send-receive and actor models

actor case, since each subproblem is represented by an actor, the two actors, and by extension, the subproblem actors, are created concurrently and are synchronized only when the two operations are completed[1] which may result is higher processor efficiency.

[1]The barriers shown in the actor example are abstract; though an actor model does not have blocking primitives, the sequence of continuation executions can effectively creates barriers.

2.5 A Class Library Approach

In this work, we present an interface and implementation for scalable concurrent object-oriented programming which is applicable to the irregular and unstructured problems of which VLSI CAD problems are representative. This section presents the major characteristics of the interface.

2.5.1 Scalability

The interface and implementation are applicable to a wide range of machines, from small shared memory workstations, through workstation clusters, to massively parallel processors (MPPs). By supporting small machines and workstation clusters as well as MPP architectures, we are able to facilitate near-term use on existing workstation clusters while at the same time enabling the exploration of the application of MPP architectures to CAD problems.

2.5.2 Seamlessness

The interface is uniform across all supported architectures, from workstations to MPPs. This uniformity enables the exploration of concurrency using small machines and clusters while at the same time providing a continuous migration path to MPP machines. Development of CAD applications based on the environment allows experimentation across a wider range of architectures than has previously been possible.

2.5.3 Multi-level abstraction

We have developed a multi-level abstraction based on two models. The high-level model, the actor interface, is capable of expressing actor operations that are easily implemented on virtually any machine, though at costs that will vary from architecture to architecture. The actor interface itself is implemented via a well-defined low-level model, the abstract parallel architecture, which captures details such as address space distribution while still unifying other architecture and vendor-specific details. Use of the actor interface does not preclude direct access to the abstract architecture interface. Furthermore, the actor interface has been carefully augmented to express several of the most common architecture-dependent features commonly used for data distribution.

2.5.4 Medium-grain parallelism

Because the high-level actor interface augments the native C++ interface, an application designed has the ability to express serial computation using only native C++ constructs and parallel computation via the library interface.

2.5.5 Class library interface and implementation

The interface, implemented as a library in the C++ programming language, enables the continued use of existing development tools, e.g., compilers and debuggers. The use of the C++ language enables migration from existing C and C++ codes that predominate in VLSI CAD. The library approach has had the added benefit of forcing a tighter integration with the serial language, a goal which is decidedly difficult to attain [18]. It also facilitates incremental parallelism in two ways. First, the library approach described implies that a serial program with no calls to the library is a degenerate library client. Incrementally greater parallelism is expressed by adding additional library primitives. Second, we implement parallelism via derivation; in a very real sense, the parallel application is *derived* from the serial application. By applying parallelism via derivation—the dynamic-binding mechanism in C++—it is possible to share object code and to limit perturbation of the original serial code.

2.6 Other Models and Implementations

In this section, we survey and briefly compare the approaches represented in this work to nonactor models and implementations. Other actor approaches are considered in Section 3.6.

2.6.1 pC++

Gannon and Lee [21], [22], [23], [24] have developed pC++, an extension of C++ with support for distributed data structures similar to FORTRAN D. The pC++ language provides support for distributed collections, both array-based and tree-based, of arbitrary types and with full support for the C++ mechanisms of derivation and dynamic-binding [21]. In their data parallel model, a data structure, usually an array, is distributed across processors in a regular manner. This type of expression is natural in scientific computing where many algorithms are described as operations on arrays. It is difficult to hypothesize how one would implement the unstructured applications in VLSI CAD on top of a data parallel model. The difficulty of expressing these tasks in terms of data parallelism reinforces the necessity to choose the correct tool for the correct problem.

2.6.2 CC++

Compositional C++, or CC++, proposed by Chandy and Kesselman [25] takes a task parallelism approach to concurrency. Where in pC++, processor control is implicit in the parallel data structures, in CC++, parallelism is achieved though `par` and `parfor` primitives which cause code blocks to be performed concurrently in different processing threads. CC++ also provides a number of synchronization primitives necessary for a thread-oriented programming interface.

To express VLSI CAD applications in terms of task parallelism, applications are broken into a number of similarly sized tasks. The greatest difficulty in applying this technique is the necessity to guide the ordering of task execution; often parallel CAD algorithms are sensitive to ordering of execution. Task parallelism generally does not provide a priority mechanism. Furthermore, many task parallelism implementations are tuned for cases where the degree of parallelism expressed is on the order of the number of threads in the machine. In this situation, a master-slave model would often be required to maintain an ordered list of tasks. This lightweight scheduling is implicit in the actor interface.

As is the case for data parallelism, there are problems for which task parallelism and CC++ more closely match the most intuitive solution. In particular, while features such as implicit barriers and futures can be expressed with continuations, such expression may be cumbersome. Applications that rely heavily on these operations could be tedious to translate to continuation passing style.

2.6.3 IC-C++

Recently, a variant of C++, Illinois Concert C++ or IC-C++, has been proposed as a method for the expression of fine-grain concurrency in C++ [26]. IC-C++ is targeted at massively parallel machines and applications with a high degree of concurrency. IC-C++ defines a concurrent semantics for most C++ primitives and operations, which enables the easy expression of a high degree of concurrency. The work on IC-C++ borrows from the compiler analysis technologies in the Concert system [27].

2.6.4 ES-Kit

The Experimental Systems Kernel, or ES-kit, of Leddy and Smith [28] is implemented via modifications to an existing C++ compiler and as such tries to stay true to the spirit and syntax of C++. In ES-kit, pointers are extended to represent a global namespace, and remote execution is represented by the execution of a method call though a pointer to a nonlocal address. Object distribution is either automatic or under program control via the C++ placement syntax [29]. Parallelism in ES-kit is specified through the use of remote function calls and futures [30]. The original target of the ES-Kit was special purpose hardware.

2.6.5 Amber

The Amber system [31], derived from Presto [32], is an extension of C++, via a preprocessor and a supporting run time library, targeted specifically toward workstation clusters running the Topaz operating system [33]. In Amber, the approach is to explicitly locate a shared datum on a particular node and then to cluster Topaz threads on that node. When access is made to a remote node, the run time system traps to the Amber kernel and the thread of control is transferred to the processor on which the data value resides.

2.6.6 COOL

COOL, developed by Chandra, Gupta, and Hennessy [34], is also based on thread-explicit extensions to C++, in this case targeted toward shared memory architectures in general and toward the DASH architecture in particular. COOL provides a full range of classical synchronization constructs. Of particular interest in COOL is the ability to represent the affinity of different objects, which is necessary to achieve high processor utilization in DASH's distributed shared memory architecture.

2.6.7 Linda

Linda, developed by Carriero and Gelernter [35], [36], represents a high-level approach to concurrent processing. In Linda, shared data are represented by a shared *tuple-space* to which all functions have access. Elements in the tuple-space are lists of values. Access to the space is via pattern matching on one or more elements of a tuple. This formulation has particular benefit in logic-programming and artificial intelligence and was one of the motivations for the development of ActorSpaces [37], [38], an extension to the actor model.

Linda has been applied to coarse-grain VLSI applications [39] and implemented on a number of architectures [39], [40], including an implementation that runs atop PVM [14] and thus runs on any cluster of machines supported by PVM [41].

Linda was specifically designed as a *coordination language* to be added to a *computation language* [35], which is both an advantage and a disadvantage. While it allows easy separation of control flow from data flow, this separation violates the object model and thus limits application in object-oriented environments.

2.6.8 Concurrent C++

Concurrent C++, developed by Gehani and Roome [42], [43], is another extension of C++. In Concurrent C++, processes communicate via transactions that can be either blocking or nonblocking. Process bodies in Concurrent C++ are represented by functions, generally with an infinite loop representing the way the process handles events. Concurrent C++ has been implemented on uniprocessors, workstation clusters, and shared memory multiprocessors [43].

2.6.9 Jade

Jade is a high-level language for coarse-grain concurrency [44]. Jade uses a single address space model for communication with explicit declaration of data dependencies; the overall model is one of task parallelism. Jade has been implemented as an extension of the C programming language on shared and distributed memory machines and on heterogeneous workstation clusters.

2.6.10 Mentat

Mentat, developed at the University of Virginia [45], [46], is an environment for coarse-grain parallel application development. The Mentat language is derived from C++ and includes support for explicitly identified parallel class types. One of the more interesting aspects of Mentat is the run time tracking and enforcement of data dependencies. The coarse-grain nature of Mentat stems from the fact that all active classes in Mentat have separate address spaces; the operating system overhead required to implement separate address spaces generally precludes a large number of objects.

2.6.11 μC++

Buhr et al. [47] developed μC++ by adding four classical concurrency abstractions to C++: coroutines, monitors, coroutine-monitor, and task. The communication model in μC++ is a single address space, and implementations support only shared memory multiprocessors. Threads are represented in μC++ by special, explicitly declared coroutine class types. In addition to mutex and other synchronization types, μC++ allows for conditional acceptance of communication, all of which occurs via member function calls. Thread context switches in the current μC++ implementation occur at the user level without the necessity for operating system calls.

2.6.12 Paragon

The Paragon project of Chase, Cheung, Reeves and Smith [48] is implemented via a C++ library for the support of distributed data structures. Support is provided for distributed arrays, both through partitioning and replication. Paragon also provides permutation and reduction operators as well as a number of interesting conditional structures for expressing spatial and temporal distribution.

2.6.13 CA/C++

CA/C++, for Concurrency Annotations in C++, is a method for representing concurrency in C++ programs through the addition of a set of annotations [49]. Annotations, similar in structure to classes, are written to identify the concurrency characteristics of normal classes, for example the set of pre- and postconditions necessary for a method call to be available for scheduling. Annotation objects can have their own states separate from the object they specify. CA/C++ is applicable to shared memory architectures and has been implemented on Sun multiprocessors using Solaris 2 threads.

2.6.14 POOL-T

POOL-T, developed by America [50], is based on message passing but uses synchronous message passing rather than the asynchronous message passing of actors. POOL-T is targeted particularly toward large system development on medium-parallelism architectures (4 to 1000 processing elements) and was developed with an interest in applying formal methods for proving program correctness.

Chapter 3

THE ACTOR INTERFACE

The Actor Interface (AIF) provides a high-level interface for expressing concurrency. The purpose of the high-level interface is to insulate the application—and by extension, the developer—from the details of the underlying hardware without reducing parallelism. The AIF is based closely on the Actor model [51] and supports such extensions as aggregates [5] and meta-programmability. In this chapter, we consider the basic actor interface; in Chapter 5, we consider advanced meta-programmability features which facilitate tuning the operation and performance of the run time support. To exemplify use of the Actor Interface, we will use examples from `ProperHITEC`, a parallel version of the `HITEC` application for sequential test generation [52]. Complete details of the `ProperHITEC` implementation are deferred to Chapter 6.

3.1 Actors and Continuation Passing Style

The fundamental object in the Actor model [51], [53] is the actor, an object that communicates with other actors by sending messages. Message delivery in an Actor system is *reliable, unordered,* and *fair.* All actions an actor performs are in response to messages; when a message is received, the receiving actor may send messages to other actors, create new actors, and change its local state (Figure 3.1). There is a close relationship between sending messages in the Actor model and calling a remote procedure in the RPC model of distributed programming [20]. Because messages automatically invoke a method of the target actor when received, the actor *send* operation more closely resembles a remote procedure call than it does a *send* operation in a send-receive programming model.

The actor model lacks explicit sequencing primitives. Synchronization is implicit and derives from the single-threaded nature of individual actors. The return executed at the com-

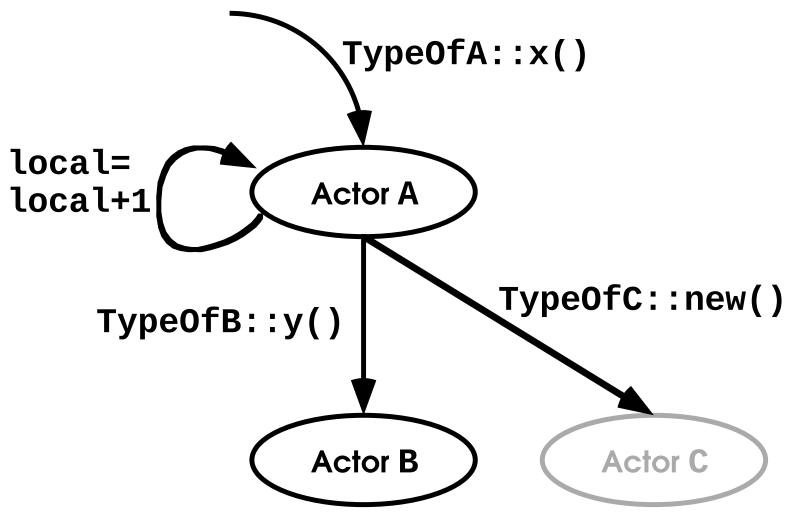

Figure 3.1 Actor operations

pletion of an actor method is an implicit wait; the actor automatically becomes available for any pending method invocations. There is no order on method invocations. Since an actor cannot suspend execution implicitly in the middle of a computation, *continuation passing style* (CPS) [16] is used to express control-flow and data dependencies that in other models would be expressed using imperative *wait* constructs.

Figure 3.2 illustrates the major differences between the RPC and Actor/CPS models. In the RPC implementation [20], a function `f` invokes a function `g` which executes on a remote thread. When the call to `g` occurs, the run time on the local system marshals the arguments required by `g`, sends them to the run time on the remote node, and blocks waiting for the return value from `g`. On the remote node, the run time system receives the request, determines that it is a request for a call to `g`, unmarshals the arguments, and invokes `g`. When `g`

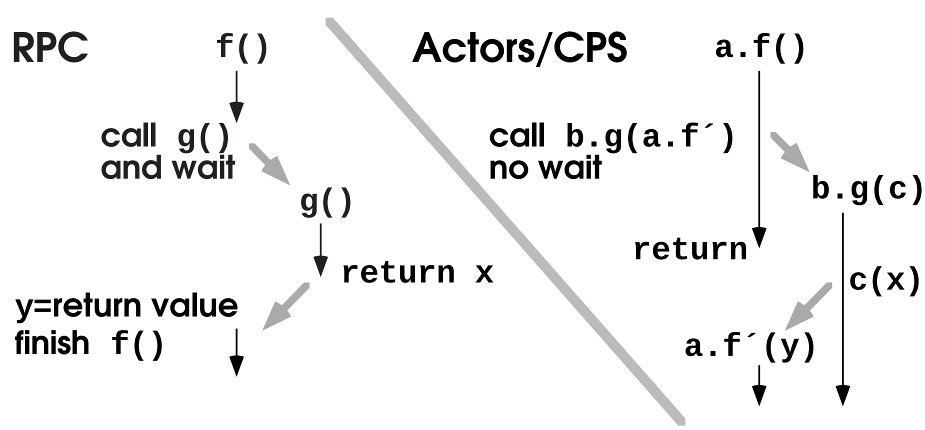

Figure 3.2 RPC and Actors/CPS

executes a `return`, the run time on the remote node marshals the return value and sends it to the local node which unmarshals it and continues `f`. Because RPC calls block, RPC represents *distributed programming*, programming with a single thread of control in multiple address spaces; the RPC model itself does not express parallelism. The marshaling of arguments implies that RPC is call-by-value.

The Actor/CPS case extends the RPC model to support parallelism and object-oriented programming. Because the model is object-oriented, all functions are member functions and must be invoked with respect to an actor; the calls to `f` and `g` are made with respect to objects `a` and `b`, respectively. The actor becomes the unit of synchronization: each actor has its own (light-weight) thread; methods of different actors may execute concurrently, but only one method of an individual actor may be executed at a time.

Like RPC, actor methods have call-by-value semantics; unlike RPC, the call of an actor method is not a blocking operation. As a result, an actor method does not return a value. In the figure, the `f` member of actor `a` calls the `g` member of `b` but does not wait. Instead, a *continuation* is passed as an argument to `g`. The continuation that `f` passes, `a.f'`, specifies that the object `a` and the code `f'` will process the value returned by `g`. The function `g` treats the continuation `c` as a function pointer; when the return value `x` is computed, `g` calls the continuation `c` with that value. Continuation variables are the parallel, object-oriented extrapolation of serial, procedural function pointers. The figure also illustrates several ways the Actor model facilitates the overlap of computation and communication. Not only is the local processor free to process other actor method invocations when `f` completes, but `f` may invoke `b.g` as soon as the necessary arguments are available. If `f` has significant computation to perform that does not depend on the value computed by `g`, the execution of that computation may occur concurrently with the computation of `g`.

3.2 Concurrent Objects

3.2.1 Actor types

The library supports user-specified actor types derived from a common class, `Actor`, provided by the library. The protected and public interface to the `Actor` class are shown in Interface 3.1.[1] Implementation of actor types can be performed in two ways; either via a change to an existing class or via inheritance. For example, the main test generator object

[1] In this and other interfaces resented in this work, the C++ code has been simplified for clarity. Private interfaces are always omitted and often protected interfaces are omitted as well. Implementation details that do not contribute to understanding are omitted.

Interface 3.1 `class Actor`

```
class Actor
{
  protected:
                  Actor();
          virtual ~Actor();

    virtual void terminate();
    ...
};
```

in `HITEC` is the `Window` class. To create a parallel test generator, we could modify `HITEC` to make the `Window` object an actor:

```
class Window : public Actor {
    ...
};
```

This, however, requires an incompatible modification of the serial application. Instead, we choose to define a new type, which is both an actor type and a `Window`:

```
class TestGenerator : public Actor,
                      public Window {
    ...
};
```

Adding the `Actor` base to a class enables the creation of actor methods, actor names, and continuations, described below. Parallelization through derivation was developed in response to one of the major goals of this work: to provide a method of parallelization which does not require backwardly incompatible changes to the serial code. Because C++ specifies static binding for nonvirtual functions [29], it may be necessary to modify the original serial code to enable effective use of derivation. However, such use of dynamic binding is at the center of object-oriented programming and is generally not objectionable. In some cases, care may be necessary in adding dynamic binding to existing functions; dynamic binding incurs a fixed amount of overhead at function dispatch time, and in the core of some algorithms this overhead can be significant.

There are no restrictions on the structure of actor classes; they may have public, protected, and private members, may be derived from other types including other actor types, and may be used as class members. Parallelism and synchronization are expressed only through the use `ActorNames`, `ActorMethods`, and `Continuations`, described in the sequel; member function calls and access to data members expressed via C++ pointers occur as they do for any C++ object. This approach—the addition of methods for expressing parallelism without changing the meaning of native C++ constructs—is taken throughout the interface design and implementation. Most uses of native C++ constructs are safe in a medium-grain parallel environment; rather than preclude many of these constructs because they are potentially unsafe, a design style is supported in which parallel constructs are added when semantics in a parallel environment are undefined.

The alternative approach, precluding the use of C++ constructs for which it is impossible to assign parallel semantics, was never considered for a number of reasons. First, a library-based implementation has no way of enforcing the use or omission of nonlibrary constructs. A particular a set of rules could have been specified, but such idiomatic usage would have been unsafe and difficult to debug. Furthermore, each added restriction would result in additional impediments to reuse of existing serial code. Finally, such enforcement is rarely successful; either the programmer can get around it or unsafe holes remain.

Because a parallel semantics is only supplied for interface objects, the exact semantics of some C++ statements is not defined. For example, if an application uses a global variable, the interface does not define how many copies of that variable will exist.

3.2.2 Actor names

Actor names serve the role of pointers for instances of actor classes; unlike C++ pointers, actor names are valid in the global namespace of a running program, independent of the number, type, and interconnection of threads executing the application. Actor names are manifest as a library-provided template class, `ActorName`. `ActorNames` are combined with `ActorMethods` to create `Continuations`. The `ActorName` interface is shown in Interface 3.2.

In many parallel interfaces, global names are implemented as opaque objects, with little or no public interface other than that used in expressing communication. By contrast, in the AIF, actor names are *first class* values with a rich interface which facilitates their use in parallel programming on contemporary architectures.

As one might expect, actor names may be created from a pointer to an actor object:

```
TestGenerator* p = ...;            // p points to an actor in this
                                   // address space.  p is not
                                   // necessarily valid on all
                                   // threads: machine-dependent

ActorName<TestGenerator> name = p; // name is valid on any thread:
                                   // machine-independent
```

Interface 3.2 `class ActorName`

```
template <class Type>
class ActorName : public ActorName<Generic>
{
  public:

                            ActorName();
                            ActorName( class Type& );
                            ActorName( const ActorName<Type>& );
                            ~ActorName();

           ActorName<Type>& operator = ( class Type& );
           ActorName<Type>& operator = ( const ActorName<Type>& );

                            operator Type* () const;
                      Type* operator -> () const;

     static ActorName<Type> newName();
    ...
};
```

However, the inverse operation is also defined:

```
ActorName<TestGenerator> name = ...; // a global name

TestGenerator* p = name;             // p will point to the actor
                                     // represented by name if the
                                     // actor is within the address
                                     // space of the executing thread;
                                     // else p = 0
```

This style of name/pointer coercion is similar in spirit to the pointer conversion operators in the C++ run time type facility [54]. `ActorNames` support most of the operations of C++ pointers and with much the same syntax. Other features of `ActorNames` are considered in the discussion of aggregates (Subsection 3.3.2) and advanced meta-programability features (Chapter 5).

Actor names have a `newName()` member which allocates a new name in the global namespace that is unique and is not bound to any actor. This name can then be used when a new actor is created to bind the new actor to the preallocated name. In this way, any actor that creates a new actor may learn the name of the new actor. Moreover, by preallocating names, tightly coupled sets of actors can be created which know each others' names at construction:[2]

[2]Because the syntax for calling methods and creating actors has not yet been presented, this example is presented in pseudocode rather than in legal library code.

```
ActorName a = newName(), b = newName();
create actor a ( b );
create actor b ( a );
```

In the absence of the separation of the operation of creating names and actors, it becomes necessary to create the first actor, to create the second with the name of the first, and then to communicate the name of the second to the first:

```
ActorName a, b;
a = new actor;
b = new actor ( a );
send b to a;
```

Not only is this latter method more clumsy, it violates an invariant popular in C++ in which instances are constructed only when all necessary arguments are available [29].

When names are created separately from actors, communication becomes uncoupled in both time and space. A name may be created, exchanged, and used as the destination of communication all without regard for where or when the actor will be bound to the name. Name creation is a distributed process that requires no global communication.

Aside from the `newName()` member, only actors themselves can directly learn their names. The ability to restrict indiscriminate distribution of actor names is useful in reasoning about system behavior [51]; when it can be proven a name does not escape a set of actors, it can be guaranteed that the actor of that name cannot be influenced directly by actors outside the set.

3.2.3 Actor methods and continuations

`ActorMethods` are member functions which may be invoked asynchronously and remotely. `ActorMethods` are executed via `Continuations`, the object-oriented, parallel extension of member function pointers. An actor method is a member function of an actor class that has been declared to be callable through a continuation via the addition of a reference to a library-provided `ActorMethod` class. For example, in `ProperHITEC`, a `Vectors` actor type is created through derivation from the `HITEC` `VectorStates` class and the library `Actor` class. The new `Vectors` inherits a member function, `test`. The `test` member of the vector database class `Vectors` is called by `TestGenerator` actors. To enable calling the member function, a nested class of the same name is defined:

```
class Vectors : public Actor,
                public VectorStates {
    ...
    class test : public ActorMethod<VectorList>
               { ActorMethodOf(VectorList); };
    void test( const VectorList& );
    ...
};
```

The nested class with the same name as the normal C++ member enables the creation of continuations that take a value of type `VectorList` as a parameter. When such a continuation is executed, the `test` member function will be scheduled for execution. The binding type of the C++ member function—dynamic for `virtual` members, static otherwise—is maintained.

The templated base class `ActorMethod` and the macro `ActorMethodOf` are provided by the library; together they define nested `Continuation` classes. Each actor method class defines a nested `Continuation` type, the interface of which is shown in Interface 3.3. The symbols `ActorType` and `MethodName` are replaced for each invocation, e.g., in the case above, they would be `VectorStates` and `test`, respectively. The `Type` parameter is indicated by the template parameter of the `ActorMethod` type, in the example `VectorList`. The `Continuation<Type>` base class is provided by the library and is considered in greater detail in Subsection 3.2.4.

A continuation is created by specifying an actor method and an actor name. The `test` member may be called as:

```
Vectors::test::Continuation cont ( vectorDB );
cont( vectors );
```

The first line defines a continuation `cont`, which when executed will call the member function `test` for the object identified by the name `vectorDB`. The second line calls the continuation with the vector list `vectors` as an argument. Execution via continuations differs from normal member function execution:

- Execution is asynchronous with respect to the caller. The actual execution of the member function occurs at some unspecified time in the future.

Interface 3.3 `class Continuation`

```
class ActorType::MethodName::Continuation : public Continuation<Type>
{
  public:
                        Continuation( ActorType& );
                        Continuation( const ActorName<ActorType>& );
                        Continuation( const Continuation& );
                virtual ~Continuation();

    const Continuation& operator = ( const Continuation& );
                   void operator () ( const Type& ) const;
    ...
};
```

- Due to their asynchronous nature, continuation calls do not return a value.

To ensure type-safety, actor methods take only a single argument, indicated by the template parameter of the `ActorMethod` class. For cases in which multiple arguments are required, they are wrapped in a structure. Continuation execution is the sole method of expressing concurrency in an actor program.

`NewActorMethods` provide similar functionality for constructors. A nested continuation type is declared for each constructor. Returning to the `ProperHITEC` `Vectors` example, the `NewActorMethod` for the `Vectors` class is written as:

```
class Vectors : public Actor,
                public VectorStates {
    ...
    struct Arguments { ... };

    class New : public NewActorMethod<Arguments>
               { NewActorMethodOf(Arguments); };

    void Vectors( const Arguments& );
    ...
};
```

When a new actor is created, no name is passed to the constructor:

```
Vectors::New::Continuation cont;
cont( Vectors::Arguments(...) );
```

In this case, the creating actor does not learn the name of the new actor and can only communicate with it if it learns the name though some other form of communication. For an actor to learn the name of the child it creates, it allocates a new name as described previously and provides it as an optional argument to the new actor continuation:

```
ActorName<Vectors> name = ActorName<Vectors>::newName();
Vectors::New::Continuation cont ( name );
cont( Vectors::Arguments(...) );
```

In this case, the creating actor may now use `name` to communicate with the new actor. As with the nonconstructor counterpart, calling new actor continuations schedules creation of the actor; the actor is not created immediately. Any messages sent to the actor before it is created become pending, waiting for the construction. It is not required that the same actor create the name and the actor. Once an actor name is created with `newName()`, it can be used in creating normal method continuations. Again, these continuations, when called, become pending until an actor is created with the allocated name as an argument to the new actor continuation. New actor continuations take a number of other optional parameters which allow the application to influence where the actor will be constructed. This type of control falls under the topic of meta-programming—programming the underlying run time system—and is consider in Chapter 5.

3.2.4 Continuation passing

In the example described in Subsection 3.2.3, the continuation `cont` is created and called directly; thus, the declaration of the `Vectors` class must have been seen prior to that point. This is an example of tight-coupling, in which the method caller is directly dependent on the type of callee. While in the case of the test generator this level of dependency is not of concern, in the more general case it can be a significant impediment. Code that calls a continuation must be dependent only on the type of argument expected by the continuation; it should be independent of the actor type and individual method which were used to create the continuation.

We illustrate with a common example that is used in the CAD applications, a barrier. When a number of actions have to be completed before a subsequent operation is started, a barrier is often used. A barrier actor would require two arguments to perform its function: an integer indicating the number of events on which to wait, and a continuation that it would call when the necessary events occurred. A client actor could then use the barrier:

```
Client::Method::Continuation cont ( *this );
Barrier::New::Continuation create;
create( Barrier::Arguments( 4, cont ) );
```

The first line of the example creates the continuation that the client code wants called, the second line creates a continuation which will create the barrier actor, and the third line calls the continuation with the number of events, four, and the continuation to call when those events have occurred, `cont`. As mentioned previously, continuations take a single argument; thus, a trivial structure continuing the arguments is constructed and passed.

The incompleteness of the interface as presented is illustrated if we now attempt to design the implementation of the barrier class. As illustrated, the `Barrier` constructor expects an argument of type `Client::Method::Continuation`. If we were to design the barrier with this expected argument, the class could not be used by any other type of client; a new barrier class would be required for each type of client. This level of dependence effectively precludes composition.

To support composition, the library defines another, more generic templated continuation type, `Continuation<Type>` (Interface 3.4). Referring back to Interface 3.3, we see that `Continuation<Type>` is the base class of the nested continuation types. The `Continuation<Type>` class has an interface very similar to that of the nested continuation type. The only difference is the lack of public constructors except a copy constructor which means that the only way a `Continuation<Type>` instance may be created is from an existing `Continuation<Type>` instance. Although this definition appears to

Interface 3.4 `class Continuation<Type>`

```
template <class Type>
class Continuation : public Continuation<Generic>
{
  public:
                              Continuation( const Continuation<Type>& );
                              ~Continuation();

    const Continuation<Type>& operator = ( const Continuation<Type>& );

                         void operator ()  ( const Type& ) const;
                         void operator ()  ( Value<Type>& ) const;
};
```

be circular, because nested continuation types are derived from the templated types, they can be used in those cases where a reference to the base class is specified. Thus, `Continuation<Type>` instances can be created by copying instances of nested continuations; the run time support ensures that no type information is lost in this operation.

In the example, a `Barrier` actor can now be defined:

```
class Barrier : public Actor {
  public:
    struct Arguments { Arguments( int, const Continuation<Void>& );
                       int events; Continuation<Void> cont; };

    class New : NewActorMethod<Arguments>
              { NewActorMethodOf(Arguments); };

    Barrier( const Arguments& arguments );
    ...
};
```

With this definition, the example client code works as shown. The client continuation type `Client::Method::Continuation` is created and then implicitly converted at the call point to the base type, `Continuation<Void>`. Thus, the call is type safe: passing a nested continuation of an `ActorMethod<int>` class would result in a syntax error since `Continuation<Void>` is only a base class of `ActorMethod<Void>` nested continuations. Since continuations of either type take an argument of only the specified type, type safety at the point of the continuation call is guaranteed. A call of a continuation using the wrong operand type is not possible. Finally, actor and method compatibility is also type checked at the point where the nested continuation instance is constructed. It is not possible to create a continuation that calls a method on the wrong type of actor.

The value of type safety is a hotly debated topic. Such arguments notwithstanding, C++ is statically type safe and is considered an integral part of the language [29]. Furthermore, anecdotal evidence indicates that weakening the type system causes significant difficulty for C++ programmers who come to rely on the compiler to find trivial type errors [54]. While static type safety leads to the most inelegant aspects of the actor interface, i.e., the necessity to resort to preprocessor macros, such inelegance is considered acceptable in the face of the alternative, lack of type safety.

3.3 Concurrent Collections

Individual actors express neither internal parallelism nor data distribution. Collection types, based on aggregates with explicit distributions, facilitate both object-internal parallelism and data distribution.

3.3.1 Aggregates

An aggregate is a collection of actors that share a common aggregate name [5]. When a continuation which was created from an aggregate name is executed, the appropriate actor method is executed by one or more representative actors. An example of the use of aggregates to represent a concurrent array is shown in Figure 3.3. In the concurrent array, non-overlapping ranges of elements are located in different actors. The figure shows two possible representations; an actor representation and an aggregate representation. In the actor implementation, a *gateway actor* is required—because client code knows only one

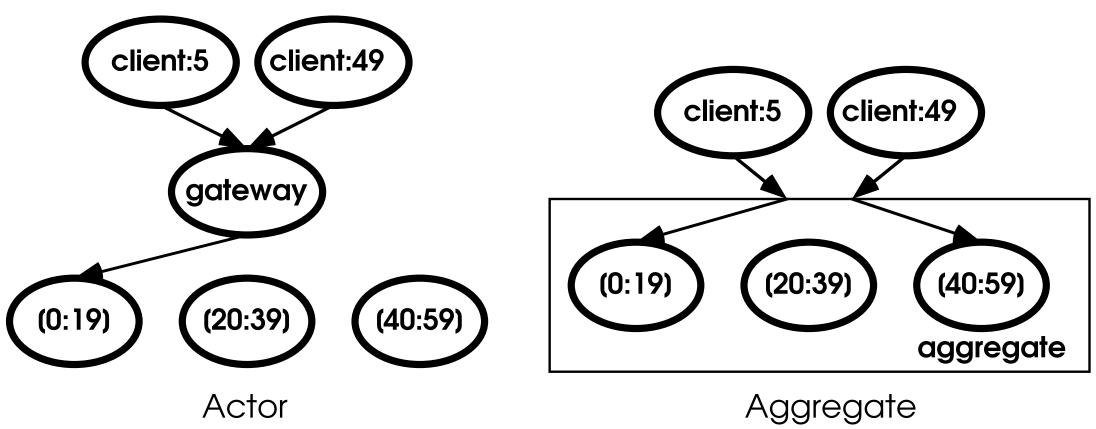

Figure 3.3 Implementations of a concurrent array

name—which must redirect requests to the appropriate subrange actor. This serialization may lead to bottlenecks; an extra indirection is required and the indirection process is serialized. In the aggregate implementation, since all representatives in the aggregate share the same name, there is no necessity to send all requests to a single actor. Instead, a request may be sent directly to the appropriate representative. Serialization will occur only if two clients attempt simultaneously to access an element stored in the same representative.

Aggregate types are derived from the library class `Aggregate` (Interface 3.5). The `Aggregate` interface is similar to the `Actor` interface. The added routines allow the determination of the index of individual representatives as well as the number of representatives in the aggregate.

The `HITEC` fault database class `Fault` is made an aggregate in the parallel application:

```
class FaultDataBase : public Aggregate,
                      public Fault {
    ...
};
```

Again, derivation is used to allow sharing of object code with the serial application. `ActorMethods` and `Continuations` have the same syntax for aggregates as they do for `Actors`. However, when a continuation is called on an aggregate, a representative is automatically selected by the run time system. Distribution of representatives and customization of the representative selection mechanism are considered in Chapter 5, though it is worth noting here that resolution to broadcast is an available choice. In this case, the method will be invoked once for each representative. Because broadcasting is handled through representative selection, there is no syntactical differentiation between a unicast continuation call and a broadcast continuation call.

Interface 3.5 `class Aggregate`

```
class Aggregate : public Actor
{
  public:
    unsigned int representativeIndexOf() const;
    unsigned int numberOfRepresentativesOf() const;

  protected:
                 Aggregate( Director& );
    virtual      ~Aggregate();

    virtual void terminate();
    ...
};
```

3.3.2 Aggregate names

Just as `ActorNames` are used for `Actors`, `AggregateNames` are used for aggregate types. The `AggregateName` interface is given in Interface 3.6.

The operation of the pointer coercion functions for aggregates is an extrapolation of the functionality for actors. In the aggregate case, a pointer to a representative will be returned if any representative is within the address space of the executing thread. Furthermore, if multiple representatives are reachable, the 'nearest' representative will be returned, i.e., if a representative was created on every thread, the coercion function will return a pointer to the representative created on the executing thread.

In addition to the functions available on `ActorNames`, the `AggregateName` interface includes several extensions: collection operations and coercion to `ActorNames`.

3.3.2.1 Intraaggregate operations

Aggregate names provide two collection operations, `representative()` and `allRepresentatives()`. Both operations return a new `AggregateName` instance. In

Interface 3.6 `class AggregateName`

```
template <class Type>
class AggregateName : public AggregateName<Generic>
{
  public:
                         AggregateName();
                         AggregateName( class Type& );
                         AggregateName( const AggregateName<Type>& );
                         ~AggregateName();

                         operator ActorName<Type>& ();

    AggregateName<Type>& operator = ( class Type& );
    AggregateName<Type>& operator = ( const AggregateName<Type>& );

     static
     AggregateName<Type> newName();

                         operator Type* () const;
                   Type* operator -> () const;

     AggregateName<Type> representative( unsigned int ) const;
     AggregateName<Type> allRepresentatives() const;
    ...
};
```

the case of `representative()`, the new name will be resolved to the name of the representative indicated by the argument. While the static interface of the new instance is not changed, if the new instance is used in the creation of a continuation, when called the continuation will cause execution of the appropriate method on the selected representative, bypassing the representative selection mechanism. Similarly, if the name returned by `all-Representatives()` is used, the applicable method will be run once for each representative.

3.3.2.2 Actor name coercion functions

`AggregateNames` can be coerced into `ActorNames` of the same template type without loss of generality, i.e., representative selection semantics. The purpose of allowing such coercion is to facilitate incremental parallelization. Consider the situation in which an actor class is designed, implemented, and used by a number of clients. If it is determined that the actor type is becoming a bottleneck, it is natural to create an aggregate implementation that uses the multiaccess interface of aggregates to express a higher degree of concurrency. If the process of upgrading an actor to an aggregate invalidates a large quantity of existing code, there will be a high cost for the change. However, because `ActorNames` and `AggregateNames` for the same object can coexist, existing code need not be changed. Internally, the two types have the identical representations. While only objects that manipulate `AggregateNames` can perform intraaggregate operations on the names, code that manipulates aggregate names via `ActorName` instances still use representative selection features.

The implicit coercion facilitates the writing of libraries for which client code is independent of the choice of an actor or aggregate implementation. Because the unicast/broadcast nature of a continuation is inherited from the name used to create the continuation, the choice of which mechanism will be used can be made by the creator of the name. By contrast, in the original model, the choice was always made at the point at which the continuation is executed [5]. The ability to encapsulate this information in continuations which are then passed to independent packages should find use in the design of reusable library modules.

3.4 Performance

There are a number of operations in the actor interface whose cost can become the the limiting factor in the applicability of the library. The greater the cost of these operations becomes, the higher becomes the minimum granularity of computation required in order that communication costs do not dominate and cause performance degradation.

The cost of several basic operations in the actor interface is summarized in Table 3.1. Each of the costs is generated by `Quantify` from Pure Software [55] on a Sun 4/690MP, a 45MHz SuperSPARC based multiprocessor[3]. It should be noted that these values represent the current implementation which has been tuned for the performance of the CAD applications being developed in the ProperCAD project, but not otherwise. In particular, it has been not tuned specifically for the operations described in this section.

3.4.1 Actor creation

Actor creation time is measured as the time to execute the task which creates an actor, including all function call overhead. The sample actor used in this test has no member state or initialization other than those inherited from the `Actor` base class. 200z Two values are shown, the first for actors that are not given prebound names and the second for those that are. The difference illustrates one of the optimizations in the name protocol of the AIF: if an actor name is not assigned a name via a parameter to the new continuation call, binding information cannot escape and thus does not have to be maintained by the name server (Chapter 5). Of the 421 cycles required to create the unnamed actor, 79 cycles are memory allocation.

[3]While useful for optimization purposes, `Quantify` results show some inaccuracies when they are compared against hand calculation and thus should be interpreted with care.

Table 3.1 Costs of actor primitives

Operation	Cost	
	Cycles	μsec
Actor Creation *unnamed*	421	10
Actor Creation *named*	3671	91
Actor Name Allocation	166	4
Actor Name Pointer Coercion *direct*	48	1
Actor Name Pointer Coercion *indirect*	255	6
Continuation Call	551	13
Continuation Execution	1130	28

3.4.2 Actor name allocation

Allocation of an unbound name is a local operation. Much of the overhead in this case is attributable to several layers of function calls that have not been inlined; the function call overhead has no appreciable impact on the current suite of applications.

3.4.3 Actor name pointer coercions

Actor name coercion is the process of turning an actor name into a pointer to an actor instance. The weight of this operation is dependent on the internal binding state of the name. When an actor name is created directly from a pointer to an actor, the information is maintained directly in the name, and thus *direct* coercion is fast.

If an actor is created with an optional name argument, pointer coercion of the unbound name after the actor has been created results in the correct pointer value but requires a level of indirection, though the name database, to resolve the name. Once the name has been resolved one time, the binding state of the name is updated in place and subsequent coercion is direct.

3.4.4 Continuation call

The cost of calling a continuation is the cost of creating the task that will execute that continuation plus the cost of inserting the task into the task queue maintained by the run time. Chapter 5 discusses task queues.

3.4.5 Continuation execution

The cost labeled *continuation execution* is not the cost of a primitive operation; it represents the interval of time starting when a continuation is called and ending when the first line of the associated method occurs. In general, this period of time can be arbitrarily long if other tasks are waiting. The values in the table represent the minimum possible duration, when no intervening tasks are executed. It includes the cost of calling the continuation, given earlier, and adds to it the cost of dequeuing the task and executing it.

The values of these various parameters provide some indication of the level of communication the current AIF implementation can efficiently support. Some of these values can be improved, some dramatically, such as the cost of creating an actor with a prebound name. However, there will always be limits to the fineness of granularity supportable, even on shared memory platforms, if not simply because of the separation of concurrent and serial constructs in the library; a truly fine-grain approach requires that virtually all structures be concurrent. The current expression of concurrency in the AIF is too inconvenient to express truly fine-grain concurrency.

3.5 Evaluation

In this section, we summarize our experiences with Actor Interface. We consider both those aspects of the interface that achieved our goals and those that are either inelegant or in need of extension.

3.5.1 Static type safety

Static type safety is completely guaranteed by the interface and, as shown in the barrier example in this chapter, has already found use in existing CAD applications. Static type safety was considered a key goal of the interface design and is one of the characteristics that makes this interface unique among actor approaches to concurrent C++.

3.5.2 First class names and continuations

The ability to create, operate on, and interchange actor and aggregate names has proven to be vital to the construction of an interface that has a high degree of expressibility but that supports static type safety and separate compilation. We believe that the expressibility of names linked with statically typed continuations will be key to implementing application libraries that can be effectively composed to create new applications.

3.5.3 Derivation-based parallelization

Our experience with parallelization through derivation leads us to believe that it will be key to parallelizing existing codes without the doubling of development and support costs. For many medium-grain applications, the added cost of dynamic binding will be acceptable, and the impact on the expressibility and readability of the serial code will be nominal. To attempt to further qualify the applicability of incremental parallelism, we must separate parallelization efforts into two classes: those that are performed within the same development organization as the serial code and those that are not.

Where serial and parallel development occur in tandem, incremental parallelism holds the most potential. The work in this thesis shows that via derivation, serial and parallel applications can be developed from the same source and object codes using dynamic binding to select the appropriate mechanism. Because dynamic binding is not the default in C++, parallelization may require the specification of more dynamically bound functions in the serial code than might otherwise be used. However, since the specification of dynamic binding has no impact on the semantics of the serial code, little extra development and support costs are incurred.

Beyond the addition of dynamic binding, to support concurrency certain styles of programming—for example, passing values among procedures via static variables—must be precluded. While this prohibition potentially reflects a more significant impact on the cost of serial development, the constraints required for parallelization overlap those posed by software engineering quality standards and thus are generally not objectionable. Thus, the costs of supporting parallelism can be brought down to level of the actual cost of expressing the parallel algorithm, specifically eliminating the redundant costs usually implied by concurrent but independent serial and parallel development.

In contrast, when a serial code is developed by an outside party, it is difficult to quantify or bound the effort required to incorporate revisions of the serial code into a parallel application. The most obvious reason for this difficultly is the inability to quantify or bound the degree of change that can occur in the serial code. Beyond the difficulties implied by arbitrary changes in the serial code, if the programming style restrictions required for concurrency are not present in the serial development, the changes required to satisfy these restrictions must be reintegrated in every new serial release. While these constraints are generally the same as those required by software engineering quality standards, in their absence, the techniques developed and used in this thesis may be inapplicable.

3.5.4 The preprocessor

The preprocessor [56] is used to implement the nested classes in `ActorMethods` and to provide some of the type safe interface in actor and aggregate classes. Use of the preprocessor in C++ is generally considered inelegant and to be avoided [54]. Yet at the same time, the name lookup and inheritance mechanisms in C++ are in some respects very inflexible. While it is simple to specify classes with flexible interfaces and functionalities, it is not possible to specify characteristics of *classes of classes,* that is, properties that hold for sets of classes. Classes of classes are called *metaclasses* [57]. Recently, significant interest has arisen concerning *metaobject protocols*, functions that operate on metaclass objects, or metaobjects [58]. Most of this work has been in CLOS, the Common Lisp Object System, and while some of the concepts are not applicable to the static environments usually found in C++, when some of the concepts from a metaobject protocol are used, the expressibility of C++ can be better understood. In this light, both templates and the preprocessor can be seen to provide some level of metaclass programmability. It was in this light that use of preprocessor macros was accepted, since all indications were that no better solution would be found. Experience gained in using the preprocessor to implement metaobject protocols may lead to possible extensions or to a new language that could maintain compatibility with C++ and yet extend the expressibility. Such a language or extension is as applicable to serial programming as it is to concurrent programming.

3.5.5 Multiple inheritance of actor types

Currently, actor types cannot be virtually derived from the actor base class. While multiple inheritance can be used with an actor type and one or more nonactor classes, multiple inheritance of actor types requires that the `Actor` base be a virtual base in all classes derived from `Actor`. This is a result of the type model in C++: it is fundamental in C++ that a member function may not be called for an object unless the type of the object is sufficiently well specified, i.e., to within a base class defining the indicated member. To schedule actor method invocations, the AIF run time casts an `Actor` reference to the target type and then calls the appropriate member. Because the actor interface is type safe, this downcast is always valid. However, in the current language, downcasts from virtual bases are not legal [56]. This restriction has been lifted by the C++ standardization committee [54]. When compiled with compilers that support the new features, the AIF will support virtual bases. The prohibition against virtual bases is the only limitation on actor class organization.

3.5.6 Actor names

While `ActorNames` can be created from actors of derived types, actors names of related types do not support the trivial conversions that pointers and references do. For example, given two actors:

```
class A : public Actor { ... };
class B : public A { ... } b;
```

References and `ActorNames` of class `A` can be created from the `b` object:

```
A& ref = b;
ActorName<A> name = b;
```

The conversion to a base class type is automatic. However, `ActorName` instances have no relationship even if their template argument types do:

```
ActorName<A> x = a;
ActorName<B> y = b;
x = y;                  // error
```

`ActorNames` are among the class of *smart pointers* that is not currently well-supported in C++ [59]. As in the case for virtual bases, a method for full functionality has been adopted by the C++ standardization group and awaits implementation [54].

3.5.7 Applicability

As noted in Section 3.4, the applicability of the interface to concurrent programming is limited by a combination of implementation efficiency and method of expression.

3.5.7.1 Efficiency Constraints

Efficiency constraints take two forms: those that are inherited from the characteristics of the underlying parallel platform and those implicit in the interface and thus present in all implementations. In the former class are issues such as network latency and bandwidth. There is no way to incorporate hardware limitations into a description of the applicability of the AIF, because the AIF is designed to run on a variety of architectures. Therefore, an application based on the AIF may show performance dependence on the underlying platform. While the AIF can be used to express granularities from medium to coarse, efficiency of the overall application will be a function of the combination of application and platform granularities. A goal of the design of the AIF was that the interface and implementation provide coarse-grain concurrency across a range of implementations and medium-grain concurrency on those platforms providing sufficient processor and communication resources.

The second class of constraints relates to the model itself; this class limits applicability regardless of target platform. These limitations are manifest by the operation costs in the interface, reported in Section 3.4. While the values reported may be improved significantly, the programming model—in particular, the meta-programmability described in Chapter 5—places a lower bound on the cost of these operations. In effect, because the application designer has the ability to express low-level dynamics such as scheduling, the run time system is constrained in the set of implementation and optimization techniques. For example, while it is conceptually possible to inline a method call from one actor to another actor when the actors are in the same address space, the run time library is precluded from doing this because it may violate the task prioritization defined by the designer. The ability of the application to influence run time limits the range of possible run time implementations. The result of this choice is that the interface is applicable to situations where optimizations at the level of continuation passing and scheduling will be done only by the developer, not by the run time. Since as concurrency increases, optimization becomes more important yet more difficult to perform manually, the interface is not appropriate for truly fine-grain applications.

3.5.7.2 Expression Constraints

In addition to issues of execution efficiency, another constraint on applicability is the form that concurrency expression takes. The approach taken in this work is that the default expression, i.e., code written in C++ without using types from the library, is serial and that only through manual addition of concurrent types is parallelism expressed. For highly concurrent applications, this would be tedious: in every case where concurrency was desired, generally both a serial and a concurrent construct would be required. In contrast, in a language like IC-C++ [26], all operations are given concurrent semantics and thus only a single construct is required. In cases in which fine-grain concurrency is required, the use of `ActorMethod` classes and explicit continuations would significantly increase the amount of code necessary.

While the limits of applicability are understood qualitatively, quantitative understanding will require more experience. Of particular value would be a measure of the lower bound on the support for medium-grain concurrency, i.e., a measure of the greatest degree of concurrency for which the interface is an effective implementation tool. The applications developed to date exhibit grain sizes that do not push the limits of the interface and implementation and thus provide little insight into a quantitative measure of concurrency which could be used to describe the boundary between effectiveness and ineffectiveness. A clearer understanding of this boundary is a goal of future work.

3.6 Other Actor Models and Implementations

Many implementations of the actor model or actor-like models exist. We survey a number of these implementations and contrast the features of those models and implementations with those of the AIF.

3.6.1 Extensions to the actor model

The actor model in its simplest form is fairly low-level. Several extensions have been made to the model to extend expressibility. Many of these extensions have provided new ways of expressing constraints on when and where tasks may be executed and new ways of coordinating communications among groups of actors [60], [61].

3.6.2 Pure actor languages

A number of pure actor languages have been implemented since the proposal of the original model [62]. Following the original model, most actor languages are functional, i.e., side-effect free, and untyped. Implementations of the majority of these approaches have targeted fine-grain concurrency on massively parallel architectures, and many were never actually implemented on distributed machines. The earliest languages were Act [51], [63], Act1 [64], Act2 [65], Act3 [51], Sal [51], and ABCL/1 [66]. A more recent contribution, with an emphasis on reflection, is MERING IV [67].

3.6.3 HAL

Among pure actor languages, of particular note is the HAL language of Houck and Agha [68], which was implemented on top of the Charm programming system [69]. HAL represents one of the few truly distributed implementations of a pure actor language and is notable for having formed the basis for further exploration into meta-programmability extensions of the actor model.

3.6.4 Concert

Chien [5] recognized the need for a multiaccess interface to augment the serial interface of Actors. He proposed aggregates as collections of actors that present a unified, yet multiaccess, interface to client code. Key to aggregates is support for efficient intraaggregate addressing. Chien and Dally [70] proposed a pure actor language, Concurrent Aggregates (CA), which in addition to having the features of actors, aggregates, and intraaggregate addressing, provides support for first class continuations and messages.

The Concert system of Chien, Karamcheti and Plevyak [27], [71], is a compiler and run time support system for a version of Concurrent Aggregates on stock hardware, i.e., such contemporary parallel machines as Thinking Machines' CM-5 and Intel's Paragon. The objective of the Concert system is to take a fine-grain concurrent language, CA, and through extensive data flow and type analysis on the part of the compiler, produce code that runs efficiently on medium- and coarse-grain machines. The Concert system compiler compiles CA to C++, creating an executable that provides run times competitive with native C and C++ [27].

3.6.5 Charm and Charm++

Charm [69], [72] and Charm++ [73] are parallel programming environments, derived from C and C++ respectively, targeted at medium-grain architectures and applications. The fundamental object of parallelism in Charm is the *chare,* an object with behavior similar to that of actors. Charm supports collection types via *branch-office chares,* which are aggregate-like objects with implicit per-thread distribution. In addition to a parallel interface, branch office chares have a serial interface that can be used to make blocking calls on local representative chares.

Charm was developed specifically to address the requirement for a portable, parallel interface and for implementation across a wide variety of both shared memory and distributed memory architectures. Charm differs from other actor languages in that it is targeted to medium-grain parallelism architectures rather than to the massively parallel processors of many other implementations. Among the most significant and unique features of Charm are:

1. imperative-language interface. Charm is an extension of the C programming language with extensions to support actor- and aggregate-like characteristics. With a few exceptions, the imperative constructs of C are retained.

2. meta-programming features. Charm provides developer-visible models for the operation of the underlying run time system in the areas of load-balancing and prioritized message delivery. Strategies in both areas can be selected from a set of alternatives at link-time. Of particular note is work with lexicographically ordered message priorities applied to search problems [74].

3. library types. Charm provides a library of object types with parallel semantics such as read-only variables, distributed computation types such as accumulators, and distributed mappings [75].

Charm and Charm++ have also been used to implement several other abstractions. As mentioned, HAL [68] is a pure actor language that uses Charm as the underlying infrastructure. Dagger [76] is a coordination language that adds to Charm a method of expressing a partial order on Charm messages, which eliminates some of the difficulties in expressing data dependencies in a low-level actor language. Other extensions include the ability to succinctly represent specific parallel idioms such as divide-and-conquer parallelism [77].

The first phase of the ProperCAD project [1] used Charm as the parallel programming environment, and many of the extensions to the actor model that exist in the AIF are extensions of features of Charm. The priority mechanism of Charm was determined in earlier work [1] to be crucial for efficient execution of CAD applications and was extended in the AIF.

Features of the Actor Interface that differ most significantly from Charm are support for static typing, as represented by first class continuations, and composability, as represented by per-class meta-programmability. In Charm, names are created as a side effect of creating chares. The ability to separate object creation from name creation is necessary to express some types of computation. As in the original aggregate model, different functions are used to specify unicast and broadcast; therefore, client code is dependent on the type of the object to which a message is being sent.

Because Charm does not have first class continuations, two techniques are used to represent the same functionality. A user-defined continuation pair, `<EntryPointID,ChareID>`, can be used. However, because the pair is not a system object, static type checking cannot be done to ensure that the elements of the pair are consistent. Derivation has also been proposed to express a type-safe alternative [73]. A chare that will receive a message from another chare inherits from a receiver type defined by the sender. For example, the user interface in `ProperHITEC` which uses barriers to synchronize aggregate creation could be defined in Charm++ as

```
chare class UserInterface : public BarrierReceiver {
    ...
    virtual void receiveBarrier();
    ...
};
```

However, this use of derivation precludes the use of more than one barrier function in a class. The `UserInterface` actor, which creates three barriers, would have to keep an extra state variable. By contrast, when first class continuations are used, a barrier actor is created with a continuation representing the method to be called in the computation.

3.6.6 ACT++

ACT++ is a C++ library-based implementation of the actor paradigm developed by Kafura and Lee [78], [79]. ACT++ implements an `Actor` base class that supports all the actor model primitives of *new*, *send*, and *become*. Additionally, ACT++ supports an RPC-style of actor method invocation [51] via blocking within the library. ACT++ allows the use of normal C++ objects, but only as private, nonshared acquaintances of an actor. ACT++ is targeted to distributed processing, but distributed implementation details have not yet been reported.

3.6.7 CLAP

Recently, CLAP, C++ Libraries for Actor Programming, has been presented by Desbiens et al. [80]. CLAP uses the actor model and a C++ class library to express actor concurrency on distributed memory machines. The interface to CLAP is similar to that of ACT++ and uses the dynamic type model of the original actor model. Current implementation efforts are aimed at a network of transputers.

Chapter 4

ABSTRACT PARALLEL ARCHITECTURE

The Abstract Parallel Architecture (APA) comprises a model of a parallel computer and a set of objects that represents a reification of that model. The APA provides abstractions for thread, resource, and configuration management and has been designed to provide a standard interface across all platforms with no extraneous overhead. The APA interface provides multiple access points, which allows client code to use generic calls for noncritical performance clients and more precise control when performance is critical.

A unique aspect of the APA design is that, from inception, it was targeted toward parallel architectures that mix shared and distributed memory characteristics, i.e., when some but not all threads share some part of an address space. This integration removes the necessity for separate shared and distributed memory implementations of APA clients, such as the AIF, and also facilitates specializing communication patterns for different configurations. For example, actor applications that are run on a workstation cluster that includes multiprocessors pass messages via shared memory within a multiprocessor and via message passing otherwise.

The APA currently supports Sun and Encore shared memory multiprocessors and Intel and Thinking Machines distributed memory multicomputers. The APA supports IP-connected clusters of a single architectures, i.e., when the same program executable is run on all nodes. The APA is self-sufficient and may be used apart from the actor interface.

4.1 Thread Management

The APA thread subsystem manages a set of virtual processing elements, represented by instances of the `Thread` class. Thread objects are collected into sets characterized by the means by which they may communicate. These sets are represented by the library classes

Process, ProcessGroup, and Cluster. The relationships between the sets are depicted in Figure 4.1.

4.1.1 Thread

The fundamental unit of computation is the thread. A thread may represent an individual processor or a single task under an operating system supporting multitasking. The interface for the Thread class is shown in Interface 4.1.

A reference to the current thread is returned by a call to the thisThread() member. The container instance, in this case a Process object, containing the thread is returned via

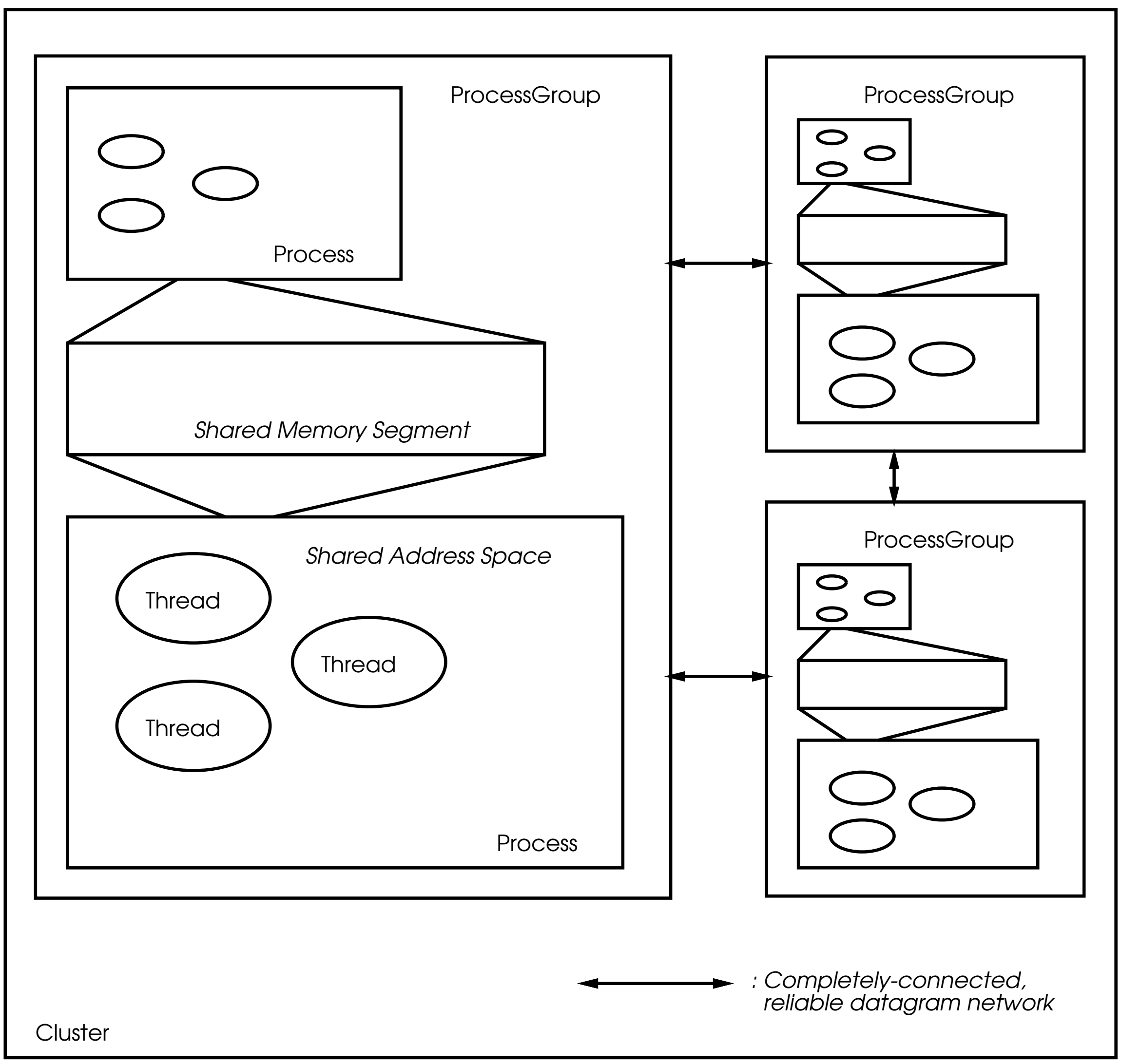

Figure 4.1 APA thread management classes

Interface 4.1 `class Thread`

```
class Thread
{
  public:

    static Thread& thisThread();              // access and
           Process& processOf();              // identification
           unsigned int indexOf() const;

              File& in() const;               // stdio
              File& out() const;
              File& err() const;

         Reservoir& privateStoreReservoirOf(); // memory management
         Reservoir& sharedStoreReservoirOf();  // interface
    ...
};
```

the `processOf()` member. All threads in a machine are given a unique index, returned by the `indexOf()` member. Each thread manages a standard I/O interface similar to that provided by `stdio` in the standard C library and by the `stream` class in the standard C++ library.

Among the most significant responsibilities of the Thread management classes is coherent management of the free store. Each `Thread` instance manages one or more `Reservoirs` which handle efficient, i.e., with minimal critical sections, thread safe allocation and deallocation of memory. The memory management system is described in detail in Section 4.2. It is worth noting here that while `Thread` instances have one or more `Reservoirs` for managing memory in private and shared address spaces, they do not actually 'own' any memory regions. Address space management is relegated to instances of the container classes in a manner analogous to Mach [81]. Also, while the `Thread` class interface provides shared and private free stores in all configurations, on some machines—those that support no form of shared memory—the accessor functions return references to the same object. Whether or not shared memory is supported is a compile time option; the choice does not incur run time overhead on machines which do not support shared memory.

4.1.2 Process

`Threads` that share a complete address space are collected into a container called a `Process`. Because all `Threads` in a `Process` share identical address spaces, all pointers are valid across `Thread` boundaries within a `Process`. This model is applicable, for example, to Mach threads. The interface to the `Process` class is shown in Interface 4.2.

Like `Threads`, `Processes` are indexed within the running machine. Furthermore, `Process` instances provide access to the threads which they contain. As shown in Figure 4.1, `Process` instances own the free store segment that is used by the `Thread` instances they contain. The interfaces of the `FreeStore` and `PageTable` classes are given in Section 4.2. The memory managed by this free store is the same as that managed by `malloc` in the standard library, i.e., it is managed via the `brk()` system call. For use in existing libraries and applications, together the `Thread` and `Process` classes provide implementations of `malloc`, `free`, and `realloc`.

Many existing functions in standard system libraries are not thread safe yet they must be used for various functionalities. To facilitate this, on machines supporting multiple threads per process, a semaphore (Section 4.3) is available that can be used, cooperatively, to protect calls to functions that potentially conflict. An example of this type of function is the standard library call `fopen`. The `fopen` function scans a list of file descriptors, searching for a free descriptor, which it returns. If two threads execute an `fopen` call concurrently, race conditions may occur. Therefore, each thread is required to lock the semaphore returned by `addressSpaceLockOf()` before entering `fopen`. While a better solution would be to provide a thread-safe version of `fopen`, rewriting all libraries that are not thread-safe is not

Interface 4.2 `class Process`

```
class Process
{
  public:

    static Process& thisProcess();        // access and
      ProcessGroup& processGroupOf();     // identification
       unsigned int indexOf() const;
       unsigned int numberOfThreadsOf() const;
       unsigned int indexOfFirstThread() const;
            Thread& thread( unsigned int );

         PageTable& pageTableOf();        // memory management
         FreeStore& freeStoreOf();        // interface

#if THREADED
         Semaphore& addressSpaceLockOf(); // catch-all lock
#endif // THREADED
    ...
};
```

a feasible solution. The address space lock provides a reasonable if conservative solution for current usage. The APA does not actually use the address space lock; all locking within the APA is done at a finer level.

4.1.3 Process group

Threads that share some portion of their address space are collected into a container called a `ProcessGroup`. Since all `Threads` in a `ProcessGroup` do not necessarily share identical address spaces, pointers may not be valid across `Thread` boundaries; addresses are guaranteed to be valid if and only if they point within a shared memory segment. This model is applicable, for example, to systems that use `mmap()` [82] and Unix System V IPC. The interface of the `ProcessGroup` class is shown in Interface 4.3.

`ProcessGroups` are indexed within the running machine and provide access to the `Process` instances they contain. `Process` instances own the free store segment and page table which is shared among `Thread` instances they contain (Figure 4.1). The free store in a `ProcessGroup` is mapped at the same address in all contained `Processes`. For use in existing libraries and applications, together the `Thread` `Process` and `ProcessGroup` classes provide implementations of `shmalloc`, `shfree`, and `shrealloc`.

Interface 4.3 `class ProcessGroup`

```
class ProcessGroup
{
  public:

    static ProcessGroup& thisProcessGroup(); // access and
                Cluster& clusterOf();        // identification
            unsigned int indexOf() const;
            unsigned int indexOfFirstProcess() const;
            unsigned int numberOfProcessesOf() const;
                Process& process( unsigned int );

              PageTable& pageTableOf();      // memory management
              FreeStore& freeStoreOf();      // interface
    ...
};
```

4.1.4 Cluster

All threads in a program are collected into a container type called a `Cluster`. `ProcessGroups` are interconnected by a completely-connected, reliable, unordered, datagram network (Interface 4.4). There is conceptually one cluster in every program. Physically, there is an instance of the `Cluster` class in every `ProcessGroup`; therefore every running `Thread` has access to a `Cluster` instance. In addition to determining the size of the virtual machine, the cluster provides routines for mapping between `Thread` indices and `ProcessGroup` indices.

If a machine has more than one `ProcessGroup`, `ProcessGroups` are conceptually connected via an ordered, reliable datagram network. The `send` and `receive` members of the `Cluster` class are used to exchange datagrams. Since message passing occurs among `ProcessGroups`, the unicast `send` member takes a datagram and a `ProcessGroup`

Interface 4.4 `class Cluster`

```
class Cluster
{
  public:
    static Cluster& theCluster();        // access and
                                         // identification

        unsigned int numberOfProcessGroupsOf() const;
        unsigned int numberOfProcessesOf() const;
        unsigned int numberOfThreadsOf() const;

        unsigned int threadToProcessGroup( unsigned int ) const;
        unsigned int processGroupToThread( unsigned int ) const;

#if DISTRIBUTED                              // message passing

                void send( Datagram&,        // unicast
                           unsigned int toProcessGroup,
                           unsigned int toThread = UINT_MAX );

                void send( Datagram& );     // broadcast

                 int receiveReady() const;
            Datagram receive();

#endif // DISTRIBUTED

    ...
};
```

index as parameters. Optionally, if a message is to be processed by a particular `Thread` of a `ProcessGroup`, the `Thread` index may be provided. The APA does not necessarily route messages to individual threads; a datagram may be received by any thread within the destination process group. The APA does not implement message passing via shared memory; this functionality, if desired, would be part of a higher level library, such as the AIF.

Instances of the thread management classes described above are all instantiated automatically by the library. In the degenerate case in which an application is being run on a single processor, exactly one instance of each class is created, and that instance is statically allocated in the APA library. For machines on which a particular model is not implemented, for example, multithreading on an Intel hypercube, accesses to data in the `Thread`, `Process`, and `ProcessGroup` instances are resolved at link time and incur no run time overhead.

4.1.5 Thread manager

The classes described in Subsections 4.1.1 through 4.1.4 reflect the state of the running system, but they do not include functionality for starting and stopping multiple threads. The `ThreadManager` class, whose interface is shown in Interface 4.5, provides for spawning and reaping threads.

The constructor for the thread manager object takes a pointer to a function with a signature identical to `main` and arranges for that function to be called once on each thread.

Interface 4.5 `class ThreadManager`

```
class ThreadManager
{
  public:
              typedef int (*TypeOfMain)( int, char*[] );

                          ThreadManager( TypeOfMain ) = 0 );

                  virtual ~ThreadManager();

    static ThreadManager& managerOf( Thread& );
    static ThreadManager& thisManager();

                      int isFirst() const;
    ...
};
```

An example of the use of `ThreadManager` instances is shown in Figure 4.2. The actual call of the target function, `parallel` occurs in the destructor of the `ThreadManager` object; therefore, code that occurs after the definition of the manager but before it goes out of scope can be used for initialization. The `ThreadManager` destructor completes when the target function completes on all threads.

Depending on the application, it may be necessary to protect the initialization code with a condition as shown in the figure. The number of times `main` is executed is not defined by the APA. Code that should run on all processors should be be placed within the function passed to the `ThreadManager`. Code that should run on only one thread may be placed either within the parallel function or within `main` but in either case should be executed only if the `ThreadManager.isFirst()` predicate is true.

Spawning threads within the destructor appears somewhat awkward at first glance but was chosen as a compromise between portability and utility. While it would be more aesthetic to simply execute `main` once for each thread, this functionality cannot be implemented on existing systems for which a function pointer is used in a spawn call. On these systems, the indicated function runs in parallel on each thread, followed by an implicit reap as the function returns on each thread. Alternately, a procedural approach could have been adopted. In this case an imperative function provided by the library would be called to perform the spawn. This type of approach would be strictly procedural and would not allow customization via derivation. The middle ground, the use of a member function, was chosen. Use of the destructor as the selected member function is a variation on the *resource acquisition is initialization* paradigm popular in C++ libraries and applications [54].

```
extern "C" int parallel( int, char*[] );

int
main( int, char*[] )
{
    ThreadManager manager ( parallel );

    // initialization code
    if ( manager.isFirst() ) {
        ...
    }

    // fork occurs at destruction of manager when
    // it goes out of scope here
}
```

Figure 4.2 Use of `ThreadManager` class

The code given in Figure 4.2, without initialization, is present in the library, which enables the use of existing code with only a small change, renaming `main` to `parallel`. For example, the parallel "hello, world" C program for the APA is

```
parallel()
{
    printf("hello, world\n");
}
```

4.2 Resource Management

Free store (`malloc`) management is an area of common difficultly and little standardization. Many machines that supply a shared memory interface provide either little support for allocation of memory within shared memory segments or do so in a vendor-dependent manner. Implementations of `malloc` are also well known for widely varying performance characteristics. The APA free store management classes were designed to address both issues as well as to provide extended functionality.

Figure 4.3 illustrates the relationship among the component classes. The operating system interface to the free store hierarchy is contained within the `FreeStore` class, which manages a contiguous segment of memory in units of pages. A `PageTable` is allocated for each `FreeStore`, which provides allocation and deallocation of arbitrary numbers of pages and manages concurrency on those architectures for which it is available. `Reservoirs` provide a per-thread interface which efficiently manages large numbers of small and medium size objects. Reservoir operations do not usually require the use of a critical section; therefore synchronization overhead is minimized.

4.2.1 Free store

The operating system interface to the free store hierarchy is contained completely within the `FreeStore` class. A `FreeStore` object manages a contiguous segment of memory in units of pages. `FreeStores` allocate, on demand, new pages of memory at the end of the current segment and can return free pages at the end of the segment to the underlying source. `FreeStores` neither handle noncontiguous segments nor maintain lists of free pages. The `FreeStore` class interface is shown in Interface 4.6.

In addition to functions for allocating and free pages, free stores support a `contains` predicate that returns a Boolean value indicating whether the address passed as an argument is within the range of the managed segments. This predicate allows higher levels in the free

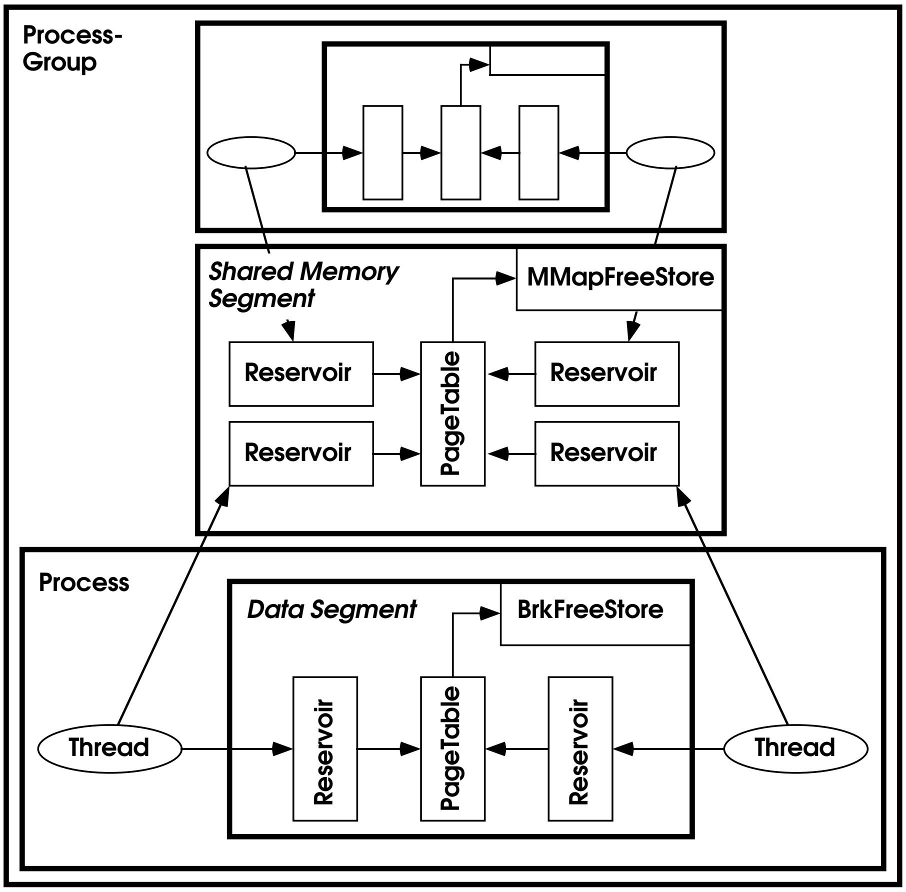

Figure 4.3 APA free store management classes

Interface 4.6 `class FreeStore`

```
class FreeStore
{
  public:
        virtual ~FreeStore();

    virtual void* allocate( unsigned int lengthInPages ) = 0;
     virtual void free( void* ) = 0;

           size_t pageSizeOf() const;

              int contains( const void* ) const;
    ...
};
```

store hierarchy to free objects back into the correct region when the application does not provide this information.

FreeStore is an abstract class; concrete classes derived from FreeStore provide services for a particular operating system interface. The APA currently supports free stores for memory allocated using the Unix brk() and sbrk() system calls (BrkFreeStore), for memory allocated using the Unix mmap() system call (MMapFreeStore), and for memory allocated using the Unix brk() call and shared using the Encore Multimax share() call (ShareFreeStore).

4.2.2 Page table

Because FreeStores do not support arbitrary deallocation or provide a thread-safe interface, another interface layer is necessary to manage lists of pages and which can be safely used in a multi-threaded applications. The PageTable object provides this functionality. The PageTable interface is shown in Interface 4.7.

One PageTable is allocated for each FreeStore. A PageTable keeps an array of page descriptors that encode information on the state of each page: whether it is free or allocated and, if allocated, how it was allocated. This last feature is used by Reservoirs, which are described in Subsection 4.2.3. Although the PageTable has a few critical regions to prevent chaotic behavior, it has been optimized for performance. Since most accesses to the PageTable do not modify the state of the table, the PageTable object dif-

Interface 4.7 class PageTable

```
class PageTable
{
  public:

            PageTable( FreeStore&,
                       size_t initialFreeStoreSize = 512 * 1024,
                       unsigned char minimumGrowth = 20,
                       unsigned char maximumOverSuppy = 50,
                       unsigned int minimumGiveBack = 8 );
    virtual ~PageTable();

      void* pageAllocate( unsigned int numberOfPages, Pool* = 0 );
       void pageDeallocate( void* );

      Pool* poolOf( const void* ) const;
    ...
};
```

ferentiates between read and write access and allows concurrent access to multiple readers. Since access to the underlying free store to request pages generally requires a system call, the page table overallocates pages in an attempt to have pages on hand to satisfy subsequent requests. The characteristics of this algorithm can be customized via parameters to the `PageTable` constructor.

4.2.3 Reservoir

`PageTables` support only allocations in multiples of the page size, and a critical section is required for most operations. Therefore, an interface is required to efficiently handle large numbers of small and medium size objects. The `Reservoir` class fills this requirement. A `Reservoir` contains a number of pools of equal size objects. Since most of the

Interface 4.8 `class Reservoir`

```
class Reservoir
{
  public:
          Reservoir( PageTable& pageTable,
                     int linStep, size_t linTop, size_t expTop=-1,
                     int linGrowthRate=10, int linInitCapacity=128,
                     int expGrowthRate=0, int expInitCapacity=16 );
          ~Reservoir();

    void* allocate( size_t );

     void free( void* );                        // deallocation and
     void free( void*, size_t );                // reallocation: safe
    void* reallocate( void*, size_t );

     void privateFree( void*, size_t );         // deallocation and
    void* privateReallocate( void*, size_t );  // reallocation: private
                                                // (local data only)

      int manages( const void* ) const;

      int disableTracking();                    // turn on/off object
      int enableTracking();                     // counting: use around
                                                // calls that allocate
                                                // memory but don't
                                                // free it
    ....
};
```

time there are unused objects in a pool, allocation for most cases is very fast. The interface of the `Reservoir` class is shown in Interface 4.8.

Although reservoirs handle pools of equal size objects, a pool is not created for every unique object size represented in an application. Instead, requests for objects of a size not pooled are satisfied by an object of the next larger size. `Reservoirs` uses a three-segment scheme to map requested sizes to allocated sizes, as shown in Figure 4.4. For small objects, the size of the returned segment is the next larger multiple of a small value. In the exponential region, the size returned is the smallest power of two that will hold the object. In the last region, the request is converted into the equivalent number of pages and is filled exactly. The characteristics of the pool allocator were designed based on the patterns of allocation common in VLSI CAD problems but should be equally applicable to other large problems.

Unlike `FreeStore` and `PageTables` which have only once instance for each unique address space, every `Thread` has a `Reservoir` instance for every free store. Most `malloc`-like operations are handled within a thread's own `Reservoir` instances, and this access does not require a critical section. Only when a free store operation involves the `FreeStore`, `PageTable`, or another thread's `Reservoir` is the use of a critical section required. Since allocation always occurs from a thread's own reservoir, only deallocation is an issue. Memory is always returned to the reservoir from which it was allocated; therefore, if an object is freed from a processor other than the one on which it was allocated, a critical section must be invoked. Moreover, an application may not know the source of

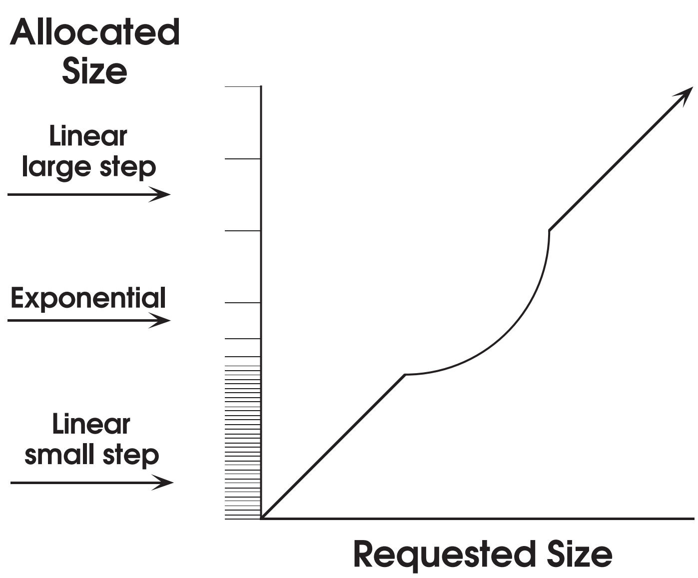

Figure 4.4 Reservoir size mapping

a particular object to be freed. In these cases, the normal `free()` and `reallocate()` members are used. These operations query the page table to determine the thread that owns the page within which the object is allocated and then invoke a critical section if the object was remote. If application semantics ensure that the object to be freed was allocated on the local processor, `privateFree` or `privateReallocate` may be called. These routines neither check the source of the object (unless debugging is turned on) nor invoke a critical region.

4.3 Communication Management

The final portion of the APA provides the interface necessary to support interprocess communication. Abstractions are provided `Semaphores`, `Networks`, and `Datagrams`. For network protocols in which delivery is unreliable, the APA implements reliability.

4.3.1 Datagram

Datagrams are objects that manipulate contiguous sequences of bytes during network operations. Figure 4.5 shows the conceptual structure of datagram objects. The `Datagram` class interface is shown in Interface 4.9. The datagram is actually an object that points to a contents buffer that may have unused space at the beginning. As the datagram is passed down through levels of protocols, headers are added to the beginning as required by the protocol. Space for these headers is allocated from the unused portion at the beginning of the contents buffer. If there is insufficient space to service a request to add header bytes, a new contents buffer is allocated with unused header bytes, and the contents of the old buffer are copied into the new larger buffer.

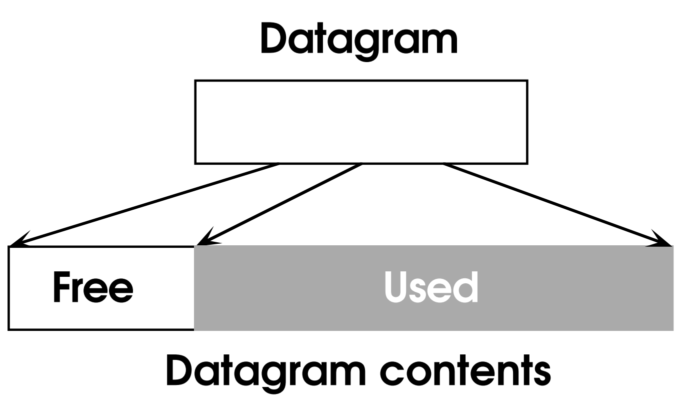

Figure 4.5 Datagram layout

Interface 4.9 `class Datagram`

```
class Datagram
{
  public:
              Datagram( void* pData, size_t length, void* pPacket = 0 );
              Datagram( Datagram& );
              ~Datagram();

    Datagram& operator = ( Datagram& );

        void* disownContents();

        void* addHeader( size_t );
        void* removeHeader( size_t );
    ....
};
```

To construct a datagram, two values are required: a pointer to the beginning of data and a data length. If the creator has preallocated free space at the beginning of the buffer in expectation of added headers, the start of the contents buffer can be passed as a third argument. The datagram 'adopts' the object pointed to when the data is constructed; the datagram will free the memory space when it is deleted. The `disownContents` member can be used to retrieve the contents of the datagram while at the same time transferring responsibility for freeing the contents to the caller.

By preallocating buffers with header space, applications can reduce or eliminate copies required when messages are sent. The APA preallocates a small amount of header space in `Value` types (Section 5.1.3) and thus does not have to copy message contents when marshaling is not required.

4.3.2 Semaphore

The `Semaphore` class provides a uniform interface to spin-lock style semaphores on machines supporting some form of shared memory. The `Semaphore` class interface is shown in Interface 4.10. Similarly, `ReadWriteSemaphores` provide separate read and write locks; multiple reader locks are allowed at one time, while a write lock ensures exclusive access.

In addition to the normal `lock` and `unlock` operations, `Semaphores` provide a state-based interface based on the `Guard` class. Instances of a `Guard` class guarantee exclusive access for exactly their lifetimes. For example, typically a `PageTable` member function

Interface 4.10 `class Semaphore`

```
class Semaphore
{
  public:
         Semaphore();
         ~Semaphore();

    void lock();
    void unlock();

    class Guard {        // use to lock/unlock critical sections
      public:
        Guard( Semaphore& );
        ~Guard();
        ...
    };
    ...
};
```

locks the page table for the duration of a member call. Rather than execute separate lock and unlock functions, a guard object is used:

```
PageTable::function(...) const
{
    ReadWriteSemaphore::ReadGuard guard ( accessSemaphore );

    // Since guard is live during the call to unsafeFunction, the state
    // of the PageTable will not be changed (or examined) by other
    // threads for the duration of the call.

    return unsafeFunction(...);
}
```

The guard object locks `accessSemaphore` in its constructor and unlocks it in its destructor. Unlocking within the destructor eliminates the chance that a semaphore will be left locked. This idiom is a direct application of *resource acquisition is initialization* [54].

For performance reasons, the necessary indivisible read-modify-write operations for `Semaphores` are implemented in assembly code on those machines for which they are necessary. Moreover, the `Semaphore` class is *not* an abstract class; a single implementation is chosen when the library is compiled. In this case, the overhead of dynamic binding is of particular concern, and the benefits of polymorphism are questionable.

4.4 Configuration Management

The `Machine` and `Network` classes provide abstract interfaces to underlying hardware. They have interfaces sufficiently flexible and recursive to describe almost any interconnection of machines. Because all machine classes have the same interface, client code is not dependent on the particular machine architecture. While `Machine` instances are visible to application code, they are primarily used to implement configuration of the thread management classes.

4.4.1 Machine

The primary responsibility of the `Machine` class is the management of machine configuration, initialization, and destruction. The interface of the abstract `Machine` class is shown in Interface 4.11. Most of the functions in the machine interface mirror those in the logical `Cluster` class. However, where in the `Cluster` class these functions simply return the configuration as it exists—possibly with reference to the `Machine` object—the same functions in the `Machine` objects must determine the necessary information without resorting to APA information. Thus, the derived `Machine` classes are the focal points for machine-specific data and are the focus of most consideration when a new port of the APA is generated.

Interface 4.11 `class Machine`

```
class Machine
{
  public:

        unsigned int numberOfProcessGroupsOf() const;
        unsigned int numberOfProcessesOf() const;
        unsigned int numberOfThreadsOf() const;

        unsigned int threadToProcessGroup( unsigned int ) const;
        unsigned int processGroupToThread( unsigned int ) const;

     static Machine* newObject();

#if DISTRIBUTED
    virtual Network& thisNetwork();
#endif // DISTRIBUTED

};
```

The Machine object to be created is chosen by the static newObject() member. Currently, this function must be selected at compile time, though the function itself may do run time configuration.

Current machine classes are the Machine abstract base class, IPSCMachine for Intel iPSC and Paragon computers, CM5Machine for Thinking Machines CM-5, UnixMPMachine for shared memory Unix machines where sharing occurs in only a single segment, and UnixMTMachine for shared memory Unix machines where entire address spaces are shared.

4.4.2 Composite machines

One machine class, CompositeMachine, reflects no particular physical machine but instead reflects a composite machine made up of separate physical machines joined by an external network. When a composite machine is constructed, the CompositeMachine initialization code selects the proper leaf machine and network types and creates them. Currently, the CompositeMachine class supports UnixMPMachines connected via a UDPNetwork. This configuration supports clusters of workstations in which individual workstations may be multiprocessors.

Aside from support for networks of shared memory machines, the most significant difference between the composite machine interface and other existing systems supporting workstation clusters is the meta-machine model used to design both the program and user interfaces. In the meta-machine model, when multiple physical machines are aggregated to create an abstract machine, a meta-machine is created. The meta-machine must be booted like a physical machine, can run multiple processes, and has a single console that is similar to serial machines. The meta-machine is booted via a single command, metaboot, which instantiates meta-machine processes on each physical machine and waits for applications to make requests to be run on the meta-machine. When an application with CompositeMachine support is run, it first attempts to contact the meta-machine server to obtain configuration information. If this attempt fails, it falls back to using the physical machine interface, currently UnixMPMachine.

The goal of the meta-machine interface is to perturb the existing serial model as little as possible. For example, in the common serial model, every program starts with two output streams, stdout and stderr. If we wish to maintain our experience with serial programs, these streams should be maintained. The composite machine attempts to support the most useful extrapolation of serial abstractions. In the case of the standard streams, they are reflected back to the process originally started at the command line and are, by default, line buffered; while output from different processes is intermixed, this intermixing happens only at line boundaries; therefore, the output remains relatively coherent. Optionally, every line may prefaced by the processor index that caused the output.

Finally, initialization and termination of composite machines are important, both in terms of usability and reliability. Processes must start quickly and must halt quickly on exit and error conditions. The meta-machine provides the former by maintaining meta-machine processes on each processor which can quickly fork and load an application when a parallel application is run. The meta-machine itself uses a more expensive protocol such as `rsh` to gain access to the remote nodes, but this is required only when booting the meta-machine, which is again analogous to physical machines.

Error conditions are detected quickly via an interconnection of TCP sockets that will immediately signal an error if a process dies unexpectedly. Thus, if the application user interrupts a running program via a keyboard interrupt, all clients are immediately terminated. Furthermore, if any meta-machine process terminates, all other meta-machine processes and application processes are immediately terminated, as they would be if a physical machine crashed.

4.4.3 Network

The `Network` classes implement a reliable, unordered, complete, datagram interconnection. The interface to the `Network` class, shown in Interface 4.12, is again similar to

Interface 4.12 `class Network`

```
class Network
{
  public:
              Network( unsigned int numberOfProcessGroups );
     virtual ~Network();

        void send( Datagram&, unsigned int toProcessGroup,
                              unsigned int toThread );
        void send( Datagram& );

         int receiveReady() const;

    Datagram receive();
    Datagram receiveIfReady();
};
```

the network operations provided by the `Cluster` class. Current network classes are the `Network` abstract base class, the `IPSCNetwork` for Intel iPSC and Paragon computers, and the `CM5Network` for Thinking Machines CM-5. The `Network` class interface has been designed to be hierarchical, as would be required for two Ethernet-connected Paragon machines, but support for such configurations has not yet been implemented.

4.4.4 IP networks

The APA provides the `UDPNetwork` class to support meta-machines comprising machines connected by a network supporting the User Datagram Protocol (UDP) [83], [84], [85]. UDP is a user-level interface to the underlying Internet Protocol (IP) [86], an unreliable datagram protocol. The `UDPNetwork` class layers on top of the UDP protocol a reliability and fragmentation protocol, both required because the UDP protocol is unreliable and has a packet length limitation of 64,000 bytes. In contrast, the Transmission Control Protocol (TCP) [87] layers on top of IP both reliability and a byte-stream model including support for flow control. UDP was chosen as the implementation for several reasons. First, TCP is a point-to-point protocol; thus, the number of active connections grows as the square of the number of nodes in the machine. Further, the flow control protocol in TCP was optimized for two types of applications: interactive terminal sessions and bulk transfer of of data, both over very large, lossy networks [88]. The characteristics of this flow control are not optimal in the APA model for which distances are small, bandwidths large, and latency critical. The stream orientation of TCP means that it does not recognize logical packet boundaries and thus cannot be optimized for the case when sending the last part of a packet will enable further computation on the destination node which cannot occur until a whole packet is transmitted. Finally, because TCP will not deliver bytes out of order, though several packets may have been received, the TCP protocol will not deliver those packets until all earlier data has been received, even if retransmission is required. Since the APA interface does not require ordered delivery, effort expended on ordering packets may lead to a loss of performance.

Though TCP is not optimal for datagram processing, much of the flow control and recovery theory can be applied with a few adaptations. In particular, TCP uses two window protocols to deal simultaneously with congestion in the network and receiver buffer limitations. Both sender and receiver maintain windows into the data stream which indicate the active range of data; the transmitter will not send data that does not fit within both of these windows. The transmitter maintains the transmitter window information; it receives updated receiver window information with every data or acknowledgment packet. The transmit and receive window sizes are updated by carefully tuned heuristics [89]. While the whole flow control system functions as a unit, the primary responsibility for congestion control, i.e., not overrunning the available network bandwidth, is held by the transmission window update algorithm, while responsibility for not overflowing the receiver is held by the receiver window algorithm.

The primary difference between the APA communication model and TCP is the lack of point-to-point connections in the APA model. Where in TCP the receiver always has a buffer dedicated to the TCP connection, in the APA the receiver may receive data from many sources simultaneously. Two alternatives are possible: dedicating fixed buffers to every potential sender, or maintaining a shared pool from which buffers are allocated as needed. The latter is more flexible in responding to unbalanced communication but makes it impossible to advertise accurate receiver window information, since any subsequent data from another source may immediately invalidate previous receiver window estimates.

The window protocol used in the current UDP implementation essentially follows the TCP protocol [88], with the exception that window sizes are fixed. The receiver window size is fixed to a large value and the transmit value to a low value, which is calculated based on the maximum packet size the network can support and the amount of buffer space available in the kernel socket interface. The retransmission protocol is identical to that of TCP [90], including delayed and forced acknowledgements. Similarly, the TCP shutdown protocol is used to ensure graceful shutdown.

4.5 Performance

Much of the originality in the APA lies in expression as opposed to performance. The most significant low level performance issue in VLSI CAD applications is network communication.

Network communication in the APA takes two forms: a thin veneer above existing services provided by MPPs, i.e., the NX library on the Intel Paragon [12], and the UDP network which provides a full set of fragmentation and reliability features above the underlying UDP protocol. The network interface incurs no measurable overhead over existing vendor protocols; nor does it improve on those protocols, although this is possible if high-level vendor protocols do not provide the performance characteristics required by the target application characteristics. Improvements in vendor protocols have not yet been investigated. In the rest of this section, we consider message passing on the UDP protocol.

Tables 4.1 and 4.2 show communication round-trip latency and bandwidth on an IP network for the APA and several workstation cluster message packages [91]. TCGMSG [92] uses static TCP sockets which created at the time the program is started. In the configuration tested, p4 [93] and PVM [14] both use dynamic TCP sockets which are created on demand at first communication. The application under test is a simple program that sends a packet to a single other node which immediately sends the packet back. This procedure was repeated 100 times, and the mean, variance, minimum, and maximum elapsed times were calculated. The configurations for testing were slightly different between the APA and the other systems. The APA experiments were done on two SPARCstations 2s connected to a lightly loaded Ethernet; the workstations had normal system background tasks running. The

Table 4.1 Round-trip latency for IP message passing

Message Size (bytes)	Latency (ms)						
	APA				TCGMSG	p4	PVM
	mean	variance	min	max			
100	2.9	1.1	2.5	13.2	3.6	4.9	5.7
400	3.3	0.4	3.0	6.2	4.9	5.2	6.7
1000	4.5	0.5	4.1	8.6	5.9	6.3	9.2
4000	11.3	3.2	10.4	41.3	12.6	15.4	17.7
10000	26.7	14.1	22.8	118.2	23.4	32.8	42.5
40000	85.3	22.1	76.2	174.0	79.9	123.4	147.3
100000	412	589	184.7	2258	201.6	308.1	356.3
400000	1700	1296	739.6	6683	794.2	1213.4	1383.3
1000000	4378	2174	2097	9972	1978.0	3030.5	3479.3

Table 4.2 Bandwidth for IP message passing

Message Size (bytes)	Bandwidth (Kb/sec)			
	APA	TCGMSG	p4	PVM
100	67.9	55.6	40.8	35.0
400	233.8	163.2	153.8	119.4
1000	432.2	339.0	317.4	217.4
4000	689.0	635.0	519.4	452.0
10000	724.9	854.8	609.8	470.6
40000	916.0	1001	648.2	543.2
100000	475.8	992.0	649.2	561.4
400000	460.4	1007	659.4	578.4
1000000	446.1	1011	660.0	574.8

other systems shown were tested on SPARCstation 1 workstations on an isolated Ethernet. The tables show that the APA performance results are comparable to those of the other systems up to the socket buffer size in the kernel, limited to 50,000 bytes under SunOS 4.1.3. Beyond this point, the transmitter overruns the receiver, and packets are dropped leading to a drop in performance. The effect of packet dropping is clearly observable in the statistics; while the minimum latency values are competitive with the other systems, the maximum value jumps to over two seconds, the APA retransmit time (adopted from TCP). The dropped packets result in a significant increase in both the mean and variance of the transmission time.

Packet dropping is caused by the lack of receiver windows and asynchronous acknowledgements in the APA. Furthermore, because TCP is integrated into the kernel, service latency for network traffic is much lower than in the user-level APA case. Only a limited amount of tuning has been performed on UDP network interface; it may be possible to improve performance by adjusting transmitter window sizes and acknowledgement and retransmission timeouts. Other characteristics of the APA algorithm could be varied as well. For example, under SunOS when a new packet comes in and the kernel buffer for the socket is full, the new packet overwrites the older packet, effectively dropping the older packet in favor of the newer. The protocol in the APA—adopted as it is from TCP—is not tuned for this case. Therefore, it is possible to tune the APA algorithm to detect the case in which overrun is occurring and to take action to reduce the flow of data. Another potential method for improving performance is the use of interrupt-driven I/O to decrease service latency. It may not be possible to achieve latency comparable to kernel-service, which might require resorting to a kernel protocol such as TCP.

We conclude this chapter with an evaluation of the design and implementation of the APA, followed by a review of other systems with similar functionalities.

4.6 Evaluation

4.6.1 Expressibility

The parametric expressibility of the APA thread management model can be demonstrated by considering the mapping of contemporary architectures to the model. Table 4.3 shows the number of each type of thread management object for a number of common parallel architectures. The first number in each triple represents the number of `ProcessGroups` per `Cluster`, the second represent the number of `Processes` per `ProcessGroup`, and

Machine	APA Configuration
Figure 4.1 architecture	3/2/3
16 node Intel iPSC	16/1/1
4-processor Sun 4/600MP	1/4/1
Two 4-processor Sun 4/600MPs connected via Ethernet	2/4/1
Intel 64 fat node Paragon	64/1/4

Table 4.3 APA triples for various machines

the last represents the number of `Threads` per `Process`; the total number of threads in the abstract machine is the product of the triple elements. Of particular interest are the last two rows in the table, an IP-connected pair of Sun multiprocessors and an Intel Paragon with fat nodes; the Paragon is a mesh-connected multicomputer; when configured with fat nodes, each node contains four processors sharing a single memory module. In contrast to most previous work, the APA represents machines that are hybrids of shared and distributed memory architectures. The APA is not restricted to architectures that can be expressed by triples of the form above, e.g., uniform numbers of `Processes` per `ProcessGroup`.

4.6.2 Factorization

The design of the APA interface has enabled aggressive factorization of the implementation code. Table 4.4 gives the total number of lines of code in the APA and the number of lines of machine-specific code for several configurations. The goal of factorization is to share code to the greatest extent possible without violating the constraint that the factoring process not contribute significant overhead. Factoring common code facilitates portability by minimizing the amount of code that must be changed to support new operating system interfaces. It also facilitates maintenance; since code is not duplicated, when errors are found

Table 4.4 Lines of code in APA

Machine	Lines of Code
Encore Multimax	357
Intel iPSC/2, iPSC/860, & Paragon	610
Sun SPARC	117
APA Total	13 800

it is not necessary to search for the same error in every copy of the code. Finally, factoring facilitates optimization, since the optimization of a segment of shared code benefits all clients.

4.6.3 Optimization

A primary goal of the APA design was an interface which provided multiple access points, which allows client code to trade generality for performance. For example, the APA provides generic memory allocation service via the C `malloc()` and `free()` functions, or the equivalent C++ `::operator new()` and `::operator delete()` functions. These operations must do a number of tests to determine the size of the object being allocated or deallocated and whether the the objects are or should be in shared or private memory. In cases in which the semantics of the program statically define what these values must be, `Reservoir` member functions can be used directly, bypassing those checks that are no longer necessary.

4.6.4 Limitations and extensions

In the remainder of this section, we consider limitations of the current interfaces. We do not consider issues that are particular to the current implementation and thus are of limited value to others.

4.6.4.1 General

The most significant abstraction missing from the APA is a representation of processor performance and load. For homogeneous machines running in a batch mode, where a program has exclusive use of all resources, measuring performance and load is generally noncritical. However, in environments in which neither homogeneity nor exclusivity exist, it is difficult to balance load to achieve good processor utilization. With a growing number of heterogeneous workstation clusters and specialized machine architectures, e.g., the Cray 3/SSS, a method for representing load and performance would find immediate use.

The issue of performance has been considered during the course of this research, not so much to propose a solution, which is beyond the scope of this work, but to ensure that the solutions developed would not restrict or preclude addition of load information in the future. Based on the success of the factorization of the APA into separate logical components, the thread management classes, and physical components, the machine classes, we envision extending the APA to three major dimensions; logical, physical, and performance (Figure 4.6). Each dimension shares information with the others but presents a focused interface. By not complicating the logical dimension with information from the performance dimension, a high degree of encapsulation would be achieved; client code that is dependent only on the logical structure of the machine would not be exposed to extraneous details of performance and load.

4.6.4.2 Datagram processing

The current model of datagrams as contiguous sequences of bytes could be loosened to support one level of indirection, to allow representation of a list of contiguous segments. This indirection would increase flexibility and usability without significantly increasing the number of copies, since in these cases generally only one copy is required. It could also simplify marshalling routines, since the buffer length would not be needed a priori. Finally, many modern systems support a `writev` system call which takes as arguments a list of contiguous segments. Use of a system call of this type could eliminate the necessity for copies that are now required.

4.6.4.3 Memory models

The only support for concurrency in shared memory currently is the `Semaphore` class, which is sufficient for current shared memory systems that use a fairly conservative co-

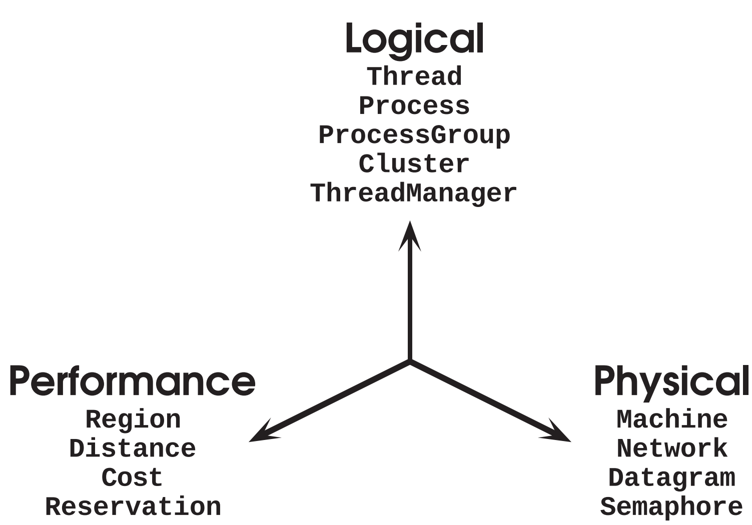

Figure 4.6 APA dimensions

herency model. Under these models, different processors generally will not see different orders of memory operations; while the relative timing of loads and stores may change, all processors see loads and stores originating from a particular processor in a consistent order. Newer machines have implemented weaker memory models for which processors may see changes in the order of operations generated by an individual processor. These models also incorporate new instructions which allow the program to signal to the hardware that the machine must be brought to a consistent state before progress can continue. The APA represents none of these low-level issues, and with the lack of commonality in the various coherency models being developed, it is unclear whether a single interface could be developed.

4.6.4.4 UDP Support

The most significant divergence between the TCP protocol and the current implementation is the lack of interrupt-based packet processing, which has proven to be a critical issue. In the current implementation, if the application is not ready to receive a packet, the UDP network will not receive packets from the network interface. Originally, it was anticipated that higher-level protocols would poll the network interface sufficiently often. However, the acknowledgment and retransmission scheme in TCP was developed under the assumption that the TCP driver would receive and respond to packets within a time much shorter than any of the protocol timeouts. This difference is exacerbated by the fact that the receiver window was enlarged to reflect shared resources. In effect, only a few thousand bytes of data can be delivered if the transmitter and receiver are not attempting to write and read at the same time. The result is that transmitters often overrun receivers, especially if the application is compute-intensive. The solution to this problem, the use of interrupt based I/O, is relatively straightforward.

4.7 Other Models and Implementations

A significant number of models and implementations have been developed to provide facility similar to that of the APA. Most previous work has been either for multiprocessors or multicomputers, but some recent work has addressed the type of hybrids included in this work.

There has been a significant amount of work done on message passing interfaces. MPP manufacturers such as Intel and Thinking Machines support propriety interfaces on their machines. Recently, ARPA and NSF sponsored the development of a new standard for message passing, MPI. This interface is similar to the MPP interfaces and includes extensions to improve flexibility.

4.7.1 PVM

Probably the most widely used software for parallel processing is PVM, the Parallel Virtual Machine [94], [14]. PVM enables parallel programs to be run on IP-connected workstations and on massively parallel machines by using vendor-supplied communication primitives. The library provides C and FORTRAN interfaces for synchronous and asynchronous message passing and provides support for heterogeneity of processing elements, both in data representation and processing power. Included tools provide particularly strong support for machine configuration, debugging, and performance analysis.

A program running on a PVM machine has a number of processes on each CPU and one `pvmd` daemon process on each node. In the default communication mode, interclient communication occurs via TCP sockets to the daemon on the local node, via UDP between daemons on the source and destination nodes, and again via TCP between the daemon and client on the destination node. The user may elect to route messages directly in some implementations. When this option is chosen, point-to-point TCP connections are made between clients that wish to communicate.

In addition to raw communication, PVM provides support for groups of tasks, and some implementations now support message passing via shared memory segments, though these segments are not exported through the interface to client code. Recent work has reported experiences adding user-level threads to PVM [95]. PVM does not provide direct support for shared memory, synchronization, or memory management.

4.7.2 p4

Developed at the Argonne National Laboratory, p4 [93] provides a send-receive model on workstation clusters via TCP sockets. The p4 library is a thin layer above the socket interface, which generally reduces problems of initialization and shutdown. The p4 interface also provides shared memory primitives on machines that provide them but does not provide support for memory management within those primitives. The library p4 includes a set of reduction primitives similar to PVM.

4.7.3 TCGMSG

TCGMSG [92] provides a send-receive model on various MPP and IP-connection workstations. On IP-connected machines, TCP is used as the transport protocol. On MPPs, the native operating system calls are used. On multiprocessors supporting shared memory, message passing is done via shared memory.

4.7.4 MPI

MPI [13] is a standard interface with message passing functionality. It is an inclusive standard that includes virtually all styles of send-receive communication, support for group protocols, reductions, and noncontiguous data structures. The APA does not provide the complete set of features available in an interface such as MPI. However, since the APA provides a standard interface to both shared memory and message passing, it would be moderately easy to implement an MPI interface on top of the APA. One of the more interesting aspects of such an implementation would be support for message passing primitives on shared memory architectures which is straightforward when the APA is used but rare in other implementations. Moreover, support for hybrid machines would be automatic.

Chapter 5

META-PROGRAMMABILITY

The Actor Interface has been designed to provide a high degree of meta-programmability by providing an open implementation, also called a reflective interface [96]. Meta-programmability refers to the ability to 'program' the actor model, i.e., to be able to specify aspects of the operation of the run time library. For example, the actor model specifies that continuations are unordered, that they may be scheduled in an arbitrary order. Through the meta-programming interface of the AIF, an application can influence the order in which continuations are executed.

In addition to local influences such as scheduling, an application may require a global control of the operation of the system. For example, in [97] a reliability protocol is developed for providing fault tolerance in actor systems. The AIF supports global meta-programmability via a meta-circular, open implementation. Run time support is implemented via a number of library-supplied aggregates that perform operations such as continuation scheduling and name resolution. By deriving and instantiating an application-specific type from a system aggregate type, the run time can be extended to support new protocols and features.

5.1 Local Meta-programmability

Local meta-programmability encompasses per-class meta-programmability, i.e., the ability to influence the operation of the actor system on a class-by-class basis. Local meta-programmability is important when viewed from the criteria of composability. If all meta-programmability features are implemented as global, aggregation of program modules is restricted to the case in which those modules utilize only compatible features.

The local meta-programmability features of the AIF are prioritized scheduling, first class continuations, first class values, aggregate distributions, and aggregate representative selection.

Before discussing the meta-programmability features, we require a new concept, generally one not explicitly manipulated in an actor system. The new object is a *task* [51], an object representing an executable context. A task is created when a continuation is called; it is a combination of a continuation and the arguments to the continuation, which is analogous to a function pointer in the serial, procedural case. Just as a function pointer requires a set of arguments before a call can occur, a continuation requires an argument before it is considered an executable entity. In the AIF, calling a continuation with an argument creates a task, represented by the AIF class `Task`, which the the run time system then schedules for execution.

5.1.1 Prioritized scheduling

The pure actor model does not specify an ordering on the execution of tasks; it guarantees only fairness [51]. This characteristic implies one of two choices: the application must implement a scheduling protocol that guarantees that tasks are created in such an order that progress is made on the computation, or that the run time system is responsible for determining an ordering of tasks which results in acceptable performance. Both of these alternatives suffer from significant drawbacks. In the former case, each application must include a scheduler, which implies both a significant degree of duplication of effort and a loss of composability, since the scheduling is being done on an application basis rather than on a per-class basis. The latter case, in which the run time system determines the order, is infeasible in general without communication between the application and the run time concerning the relative importance of tasks.

Our approach to solving this problem is to address the latter aspect, the lack of communication between the application objects and the run time scheduler. The run time system defines a priority interface from which the programmer can define new priority types. Priorities can have total or partial orderings among them. Each priority also may specify task queue types that the run time uses to store and schedule tasks. The foremost goal of the priority system design was a system that supports composable priorities, i.e., an interface that can be used in modules in any manner required by the application without sacrificing the ability to compose modules together. The run time system uses the priority interface internally to ensure that information required to support the general model is communicated with the appropriate precedence.

5.1.1.1 Task queues

Tasks are managed by the run time in a priority-defined hierarchical set of queues (Figure 5.1). The task queue interface is shown in Interface 5.1. The `add` member is called to add new tasks to the queue, while the `remove` member is called to dequeue and return the highest priority task in the queue. The library currently provides two predefined queues of tasks, `Deque<Task>` and `PriorityQueue<Task>`, and one queue of queues, `OrderedSet<TaskQueue>`. The latter uses the `priorityOf` member of the task queue interface to implement hierarchical queues of queues. Queues of queues and the priority value protocol, described next, are used to implement an extensible, composable, interface.

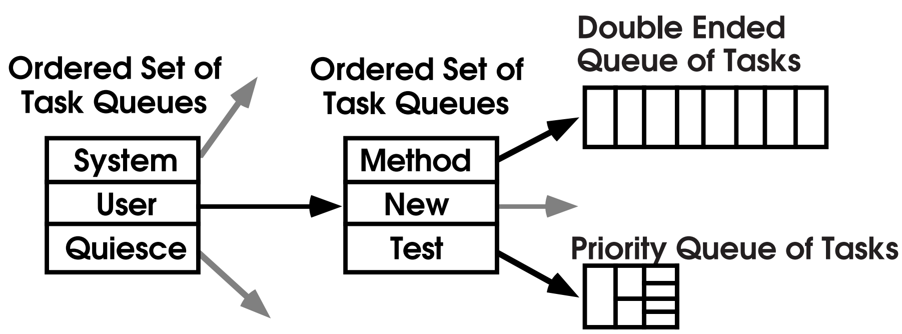

Figure 5.1 Task queues

Interface 5.1 `class TaskQueue`

```
class TaskQueue
{
  public:
                            TaskQueue( const Priority& );
                    virtual ~TaskQueue();

               virtual void add( Task& ) = 0;
               virtual Task remove() = 0;

                virtual int isEmpty() const = 0;

    virtual const Priority& priorityOf() const;
    ...
};
```

5.1.1.2 Priority values and comparators

Priority values, used to order the elements within a task queue, are defined using derivation and a class-by-class total or partial order. Figure 5.2 shows the priority classes provided by the actor interface with an additional priority used by the test generator described in Chapter 6. The '¿' signs show the relative precedence of priorities. The priority value system is in essence a lexicographic scheme with a heterogeneous set of symbol types. Figure 5.3 shows an example of how comparison occurs between two priorities of different classes. Priorities comprise a sequence of symbols, and all priorities with a common prefix must be comparable. Thus, in the figure the longest common prefix is the single symbol `UserPriority`, and there must be a partial or total ordering in the next symbol in the sequence. In this case, the next symbol differs and therefore the partial order is complete: task A has higher priority than task B. Because no further comparison occurs, the fact that the task B has additional priority information is of no consequence. If task A had also had

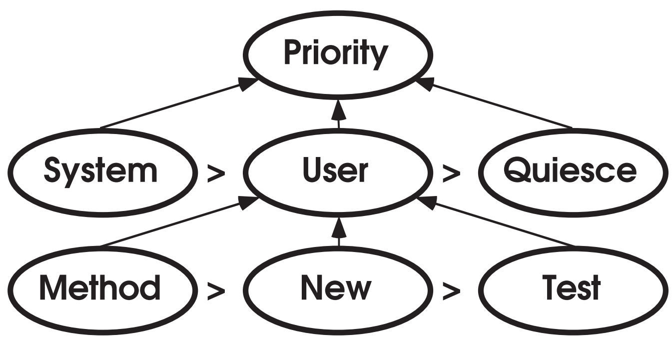

Figure 5.2 Priority class hierarchy for ATPG

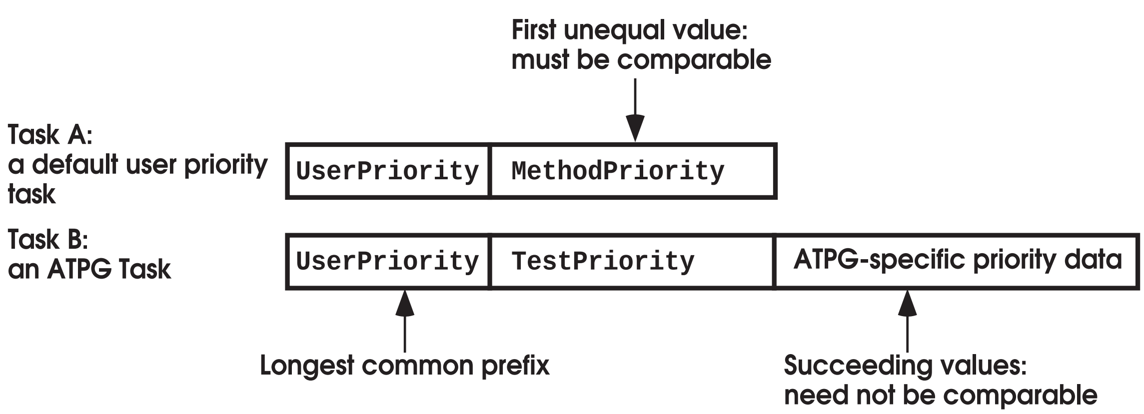

Figure 5.3 Heterogeneous lexicographic priorities

`TestPriority` as its second symbol, the comparison would have continued, based on data in the ATPG-specific portion of the priority value.

Heterogeneous lexicographic priorities are implemented via a pair of coupled interfaces, `Priority` and `PriorityComparator`, presented in Interfaces 5.2, and 5.3 respectively. Priority comparator classes determine two properties: where the priority classes fit within the derivation tree, and the ordering of objects lower in the tree.

The unary `compare` function is used to determine the queue within which to enqueue a new task. For example, the unary `compare` for the `TestPriorityComparator` class, is defined as

```
int
TestPriorityComparator::compare
                       ( const PriorityComparator& queueComparator )
{
    if ( dynamic_cast<TestPriorityComparator*>( &level ) ) {
        // The queue uses a TestPriorityComparator so this is
        // the correct queue.
        return 0;
    } else {
        // Let the class above in the derivation tree make
        // the decision.
        return UserPriorityComparator::compare( queueComparator );
    }
}
```

Interface 5.2 `class Priority`

```
class Priority
{
  public:
                        virtual ~Priority() = 0;
    virtual PriorityComparator& comparatorOf() const;
            virtual TaskQueue* newTaskQueue( const Priority& ) const;
    ...
};
```

Interface 5.3 `class PriorityComparator`

```
class PriorityComparator
{
  public:
    virtual int compare( const PriorityComparator& ) const;
    virtual int compare( const Priority&,
                         const Priority& ) const;
};
```

The binary `compare` function is used to determine the relative order of tasks and queues within a queue. For the test queue, this function simply returns the result of comparing the bit vectors in the priorities (Chapter 6).

The ability to prioritize tasks has proven instrumental in the parallelization of CAD algorithms [1] and other applications [74] even when restricted to homogeneous priority representations [69]. It is anticipated that support for heterogeneous priorities will facilitate aggregation of modules that require conflicting representations, as is the case for `ProperHITEC` in Chapter 6 and `ProperPROOFS` in Chapter 7.

5.1.2 First class continuations

Because continuations are first class, i.e., can be copied and manipulated by application code, a wealth of concurrent control structures can be represented [5]. The continuation interfaces were considered in detail in Chapter 3 and are not repeated here.

5.1.3 First class values

The actor model is *call-by-value*, i.e., parameters to method invocations are copied, as shown in Figure 5.4. In CAD and other medium-grain applications, the cost of copying an argument can be considerable. In these cases, it is desirable to be able to access the actual value—the actual storage location—to be used in a continuation call. For this functionality, the AIF provides first class value types. First class values or very similar abstractions are called *messages* in both Concurrent Aggregates [5] and Charm [69]. We will prefer the term *value* to *message* as it helps distinguish the values and tasks in the actor model from the messages of send-receive models.

First class values are represented by the library class `Value` (Interface 5.4). Each value type is a container of a single instance of the template type parameter. Values are manipulated by the application by defining actor methods which take a `Value<Type>` as an argument. Using first class values, the actor method declaration

```
void ActorType::MethodName( ArgumentType );
```

would be replaced with

```
void ActorType::MethodName( Value<ArgumentType> );
```

In general, value types can be used anywhere an object of the template parameter type can be used.

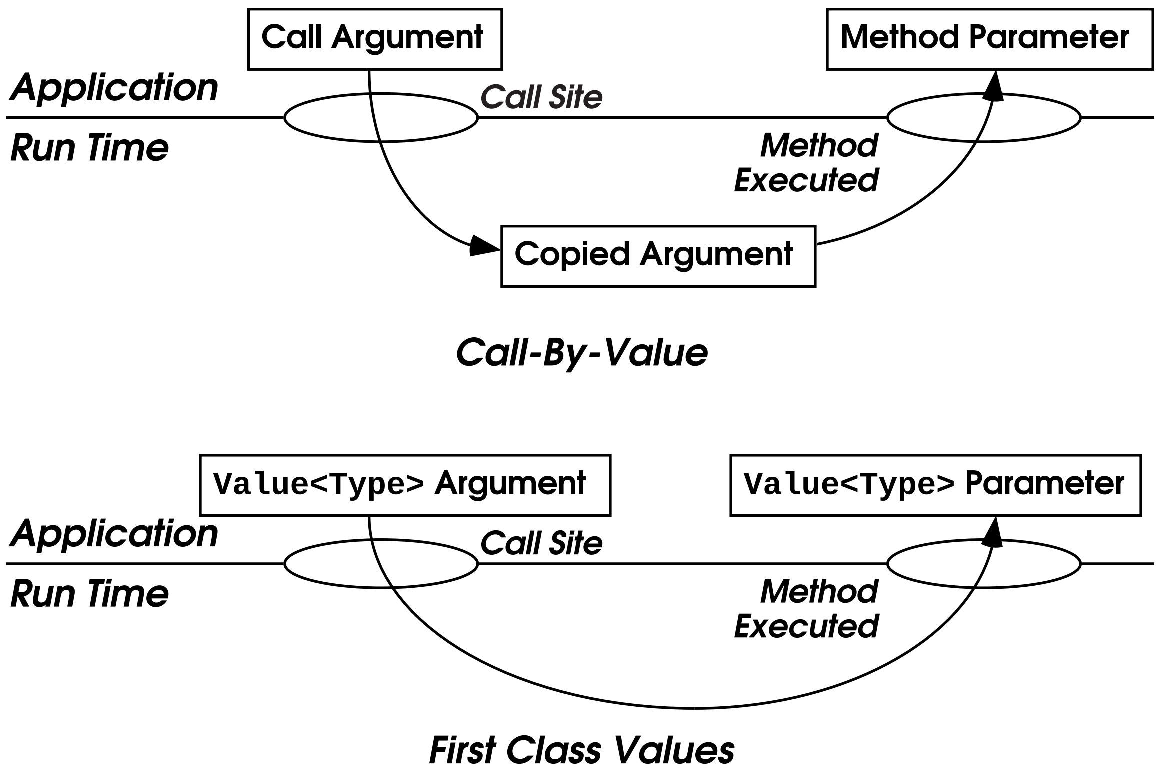

Figure 5.4 Call by value and first class values

Interface 5.4 `class Value`

```
template <class Type>
class Value : public Value<Generic>
{
  public:
                        Value();
                        Value( const Type& );
                        Value( Value<Type>& );
                        ~Value();

           Value<Type>& operator = ( Value<Type>& );

    virtual Value<Type> copy() const;

                  void* at();

                        operator Type& ();
                        operator const Type& () const;
    ...
};
```

By using first class types, an application gains control over the scope of the object. For example, if an object calls a continuation with an object

```
Argument arg ( ... /* Argument constructor parameters */ );
ActorType::MethodName::Continuation( name )( arg );
```

a copy of the argument `arg` occurs. Because the value sent to the method invocation is constructed within the run time, the application must construct an object, copy it via the continuation call, and then destruct the original. When first class values are used, the copy can be eliminated by constructing the value used by the run time

```
Value<Argument> value;
new ( value.at() ) Argument ( ... );
ActorType::MethodName::Continuation( anActorName )( value );
```

The example calls the value `at()` member to extract the address of the buffer of the unconstructed object within `value` and then uses the C++ *placement new* syntax to construct the argument in place. Finally, the value is passed to the continuation.

In a similar manner, the scope of the object can be lengthened at method invocation time. When an actor method is defined with native types, e.g.,

```
void
ActorType::MethodName( const Argument& arg )
{
    ...
}
```

the scope of the object referred to by `arg` is the duration of the function call. Any pointer or reference to `arg` stored becomes dangling after `MethodName` returns. If, instead, the method is defined to take a `Value` instance:

```
void
ActorType::MethodName( Value<Argument> arg )
{
    ...
}
```

the scope of the argument can be lengthened by copy constructing or assigning the parameter to another value object

```
Value<Argument> save;
void
ActorType::MethodName( Value<Argument> arg )
{
    save = arg;
}
```

The scope of the value now becomes the scope of the assigned value, in this case, file scope. Copy construction, assignment, and continuation calls all consume `Values`; trying to access the contents of a `Value` object after it has been consumed results in an exception.

First class values are usually handled completely within the library, but for reasons of efficiency, flexibility, and resource management, the may be manipulated by the application explicitly. First class values are most often used in CAD applications when data structures are distributed, i.e., the data representing the circuit. Copying of this data takes a significant amount of time and may limit the size of problems that can be addressed, since an extra copy implies that there must be memory available to hold two copies of the circuit.

5.1.4 Aggregate distributions

The original aggregate model allows the specification of the number but not the placement of the representatives aggregate [5]. In medium-grain applications, placement of representatives can be crucial to achieving acceptable performance. In addition to performance issues, the ability to conditionally coerce aggregate names to object pointers exposes aggregate distribution to the application in a way not present in the original model that uses a uniform address space model.

The AIF supports descriptions of aggregates via the library class `Distribution`, shown in Interface 5.5. Distributions specify the number and location of representatives of an aggregate. The interface of the abstract `Distribution` class is general enough to express both enumerated and algorithmically computed distributions. Distributions are bound to aggregates via an optional argument supplied when `NewActorMethod` continuations for

Interface 5.5 `class Distribution`

```
class Distribution
{
  public:
                Distribution();
                Distribution( const Distribution& );
        virtual ~Distribution();

    virtual int numberOfRepresentativesOf() const = 0;

    virtual int numberOfRepresentativesOnThisThread() const = 0;
    virtual int indexOfThreadRepresentative( int ) const = 0;

    virtual int numberOfRepresentativesInThisProcess() const = 0;
    virtual int indexOfProcessRepresentative( int ) const = 0;

    virtual int numberOfRepresentativesInThisProcessGroup() const = 0;
    virtual int indexOfProcessGroupRepresentative( int ) const = 0;
};
```

aggregate types are created. If no distribution is supplied, one representative is created on every thread.

The AIF provides three standard distributions based on the APA, `PerThread`, `PerProcess`, and `PerProcessGroup`. Aggregates created with these distributions will have one representative for each of the related APA objects. Since the APA configures itself automatically when an application is invoked, no further user code is necessary. `PerThread` distributions may be used to represent data private to individual threads; member functions and data of representatives in aggregates created with `PerThread` distributions can be accessed by other actors allocated to the same thread without the possibility of race conditions. In this way, `PerThread` aggregates find their most common use in task and data parallel models. `PerProcessGroup` distributions are often used to represent read-only or, in concert with the use of `Semaphores`, write-rarely data. 200z The availability of APA-based distributions is an example of the trade-off between expressibility and generality. In some respects, the ability to specify location of aggregate representatives and the ability to conditionally resolve actor and aggregate names to object pointers represent a weakening of the high-level actor model. However, many medium-grain applications require or benefit from some amount of architecture-specific customization, even if this customization consists of simply copying data into all address spaces in a distributed memory machine. This functionality is so commonly required that making it difficult would serve no purpose except to limit the usability of the interface, which would result in applications bypassing the actor interface and dealing directly with the APA. While the complete APA interface is available to actor applications, it is desirable to regularize common architecture-specific optimizations, thus reducing the need to reinvent these algorithms numerous times.

5.1.5 Actor and aggregate placement

The location of an actor or an aggregate may be selected by the application by an option argument to the new method continuation constructor. In the absence of the argument, placement is random. Aggregates may have meaningful placements if the size of the aggregate does not encompass all processors in the system.

5.1.6 Representative selection

The original aggregate model specifies random representative selection [5]. In a fine-grain environment in which aggregates are used solely for their nonserialized interface, this selection may be sufficient. In medium-grain applications in which aggregates are used for data distribution as well, random selection is often not desirable.

The AIF supports representative selection on a per-method basis. By adding a resolution function to the nested actor method class, the default resolution, 'closest', can be overridden. For example, in `ProperPROOFS`, the `Simulate` method is called by the user interface to begin fault simulation within the fault simulator aggregate. Since the `Simulate` method should be called for each representative, the `Simulate` method is declared as

```
class Simulate : protected Method<Void>
{
    ActorMethodOf(Void);
    static AggregateName<FaultSimulatorAggregate>
        resolve( const AggregateName<FaultSimulatorAggregate>& name ) {
        return name.allRepresentatives();
    };
};
```

The `resolve` function takes as an argument an aggregate name and returns an aggregate name that is bound to a representative or to all representatives, created by `representative()` and `allRepresentatives()`, respectively.

One of the interesting aspects of representative selection as implemented in the AIF is the handling of broadcasts. Virtually all systems that handle broadcasts do so by using an imperative variant of a send operation, which implies that the decision of unicast versus broadcast is made at the time of continuation execution and is made by the sender. In contrast, in the AIF the client is not required to decide whether to use unicast or broadcast, and often the client is unaware of which mechanism is being used. In our experience with CAD applications, the construction of the algorithm is generally such that it is the object on which the continuation is being called that logically holds the responsibility for the unicast/broadcast choice. The lossless coercion of aggregate names to actor names, i.e., without loss of representative selection, should also facilitate incremental parallelization. When it is determined that an actor has become the bottleneck in a computation, an aggregate with the same interface can be substituted in its place. Client code that uses actor names still functions as before with the aggregate implementation.

5.2 Global Meta-programmability

Three library classes work together to support the operations of actor creation and task routing and scheduling. The `Director` aggregate controls the thread on each processor, passing control to individual actors to execute tasks by using the policy described in Subsection 5.1.1. The `NameServer` aggregate maintains actor and aggregate name binding information, resolving names as continuations are called. The `QuiescenceDetector` aggregate monitors the state of the abstract actor machine and detects idle conditions.

Global meta-programmability can be achieved by deriving new, application-specific classes from the library-provided types. These changes are global; they affect all modules linked with the new class. Such changes are also generally not composable; although it is possible to use multiple inheritance to combine two application-specific system aggregate types, such aggregation must be done manually, and the potential complexity of the problem raises significant doubt as to the feasibility. Because system aggregate customization is global, the set of conditions under which this type of meta-programmability is applicable is more specialized. Thus, global meta-programmability generally finds less application than does local meta-programmability. Thus, while priorities are used by classes to solve modular problems, derivation of a new `Director` type is more appropriate to change global system properties such as reliability [97], [98], [99]. The `Director` aggregate also provides an object-level interface which allows limited customization without the use of derivation. All system aggregates have `PerThread` distributions.

5.2.1 Director

The `Director` aggregate, derived from both the AIF `Aggregate` and APA `ThreadManager` classes, controls the thread on each processor, maintains task queues, and passes control to individual actors to execute continuations by using the policy described in Subsection 5.1.1. The `Director` aggregate creates and communicates with the `NameServer` and `QuiescenceDetector` aggregates. It informs the `NameServer` when actors are created and destroyed in order that name binding information can be updated.

The public interface of the `Director` class is shown in Interface 5.6. There is no director instance in the library; it must be manually declared by the application. Generally, most actor applications have a simple `main()` routine that simply instantiates a director object and then, on one thread, creates a user interface object that drives the rest of the computation

```
int
main( int, char*[] )
{
    Director director;

    if ( director.isFirst() ) {
        UserInterface::New::Continuation()();
    }
}
```

Interface 5.6 `class Director`

```
class Director : public ThreadManager,
                 public Aggregate
{
  public:
                         Director();
                 virtual ~Director();

    static Director& thisDirector();

                  void shutdown();
                  void notifyAtQuiescence( const Continuation<Void>& );
    ...
};
```

As the APA `ThreadManager` object from which it is derived, the `Director` controls the thread from within its destructor. The body of `main` functions as an anonymous actor method and is generally used to call a continuation responsible for performing the body of the application. As mentioned in Subsection 4.1.5, the number of times `main` is executed is not defined by the APA; code usually should be conditioned by the `isFirst()` predicate inherited from the `ThreadManager` base class. Actors can gain access to the director on which they are running by the static `thisDirector()` call, analogous to the `thisThread()` call in the APA.

The director mediates the interface between applications and the `QuiescenceDetector` aggregate via the `notifyAtQuiescence()` method. Operation of quiescence detection is described in Subsection 5.2.3. If the `QuiescenceDetector` detects quiescence with no continuations on the notify list, it informs the director, which performs an orderly shutdown. The `shutdown()` method may be called by application code to effect normal shutdown without regard to quiescence.

5.2.2 Name service

The `NameServer` aggregate is responsible for coordinating the allocation, distribution, and resolution of actor and aggregate names within the system. This coordination includes allocating names that are unique across all processors, routing actor method calls to the appropriate processor for execution by the `Director`, and maintaining binding information as new actors and aggregates are created.

The `Director` aggregate creates the `NameServer` aggregate and informs the `NameServer` when actors are created and destroyed in order that naming information can be updated. When continuations are called on actor names that are not completely resolved, the actor name requests that the `NameServer` aggregate forward the name by using the binding information it maintains. The `NameServer` consults its internal name database and takes action, depending on the state of the binding of the name. If a complete binding exists in the database, the forwarded name is updated in place and is re-sent.

If the name is unbound, the task being forwarded is temporarily enqueued within the `NameServer`, pending the arrival of binding information. Binding information is added when a new actor is created; for every actor created or destroyed, the director informs the `NameServer` representative. The `NameServer` protocol never broadcasts names; sufficient binding information is communicated in all requests for the `NameServer` to maintain all bindings via unicast continuations. Tasks resulting from continuations called on unresolved names are never forwarded to intermediate nodes. Instead, the task is temporarily enqueued within the `NameServer`, and, if necessary, a query for binding information is made to the appropriate `NameServer` representative. When the binding information is available, it is returned to the requesting representative which uses it to route the task directly to the destination thread. Binding information is maintained in order that once a request has been made, it is not repeated.

The binding information maintained by the `NameServer` is also used to perform actor and aggregate name-to-pointer coercion.

5.2.3 Quiescence

The `Director` aggregate coordinates its action with a quiescence detection aggregate to determine when the system is idle; i.e., there are no pending tasks. Because the actor model is event driven, an actor system that has become idle will not change state unless an outside event occurs. Any actor may pass to the `QuiescenceDetector`, via the `Director`, a `Continuation<Void>` instance. This continuation will be executed when quiescence is detected.

While in many cases convenient, the usefulness of quiescence is debatable in a modular environment. Quiescence is not a composable property and modules that rely on quiescence detection are inherently noncomposable. Since quiescence is defined as no pending tasks in the system, any activity, no matter how innocuous, will keep the overall actor system from achieving quiescence. If a module is written to progress in phases separated by quiescent states, outside activity can inhibit the module from making progress indefinitely.

A good example of the limits of quiescence detection is the implementation of the quiescence detection algorithm itself. The quiescence detector object must hide the tasks implementing the detection process from the process itself, or quiescence would never be achieved.

While quiescence is not composable, it can be used with care on an application level. The quiescence protocol in the AIF is an extension of that implemented in Charm [100],[101],[102]. However, the AIF implementation of quiescence is open; the ability to hide tasks from the quiescence detector is provided to applications as well as being used within the detector itself. As in the case of representative selection, selective quiescence hiding is provided by a method function. For example, `ProperPROOFS`, described in Chapter 7, does fault redistribution via a method `Split` which is hidden from the quiescence detector:

```
class Split : protected Method<int>
{
    ActorMethodOf( int );
    static int representsWork() { return 0; }
};
```

5.3 Evaluation

5.3.1 Prioritized execution

The priority system is general enough to allow composition of modules like the `TestGenerator` class from `ProperHITEC` and the `FaultSimulatorAggregate` from `ProperPROOFS`. A potential problem is the cost of adding tasks to the multilevel queues which requires a number of operations which scales logarithmically in the number of levels. This number of operations directly affects the minimum useful grain size on some machines. While the value of generality for the target applications justifies the cost, techniques for optimizing the addition of new tasks would be beneficial.

5.3.2 First class tasks

During implementation of the quiescence detector, the benefit of having first class tasks available to the application became apparent. Furthermore, since the run time system manipulates tasks explicitly, an open implementation should provide an application interface. Currently, with first class continuations, one level of indirection is achieved: an actor can create a continuation and pass it to client code to execute when a certain set of events occurs. The client code is independent of the actor and the method which will be called, and a degree of modularity is attained. However, the client is still dependent on the argument type of continuation and must provide the argument when the continuation is called. An even greater degree of the expression is attained if the the continuation and argument are combined but not executed; the result is a task (or *thunk* [103]). An interface to the task types would improve the expressibility of the system. Some effort has been expended on developing an interface for tasks; the most significant barrier appears to be the generation of an interface that is statically type safe.

5.4 Other Models and Implementations

5.4.1 Concurrent aggregates

Concurrent Aggregates [5] provides support for first class continuations and messages. Continuations in CA take two forms: user, much like those in the AIF, and true continuations in the sense of encapsulating the complete run time context. Messages in CA fill the role of both first class values and first class tasks, since they contain both the method argument list and the name of the target actor.

5.4.2 Charm and Charm++

Charm was one of the first distributed actor models to implement advanced meta-programmability features, and the Actor Interface incorporates extensions of many of the features in Charm. The most significant difference in the area of meta-programmability is the emphasis on composability in the AIF.

Charm provides first values, called messages, but does not provide a call-by-value interface; the application may pass only message types and is always responsible for allocating and deallocating messages. Concurrent collections in Charm, called branch office chares, have a fixed distribution with one representative per thread. Representative selection is specified at the call point by passing an argument which is either a representative index or the pseudoindex `ALL`.

Charm provides several global meta-programming interfaces, selected at the time an application is linked. Prioritized execution is chosen at link time by selecting from a set of priority modules [74]. Since there is only one module per executable, all priorities must be used uniformly across an application. Supported priority values are integer and bit vector. Supported queue types are stack, queue, and a deque that dynamically selects head or tail insertion depending on its length. There is no published interface for adding new priority value types and new queue types.

Charm also provides a globally selectable load balancing module. In addition to random placement, Charm provides three adaptive load balancing modules. Charm also allows the specification of manual placement.

Chapter 6

PARALLEL TEST GENERATION

Despite concerted effort, automatic test pattern generation for sequential circuits remains one of the most time consuming tasks in the integrated circuit design process. Experts in the field of ATPG and integrated circuit manufacturing have expressed concern not only for reducing run times and improving test coverage but also for the feasibility of maintaining current levels of performance and quality on increasingly complex circuits.

The potential for the use of increasingly available parallel platforms makes test a prime candidate for parallelism. Yet, the use of parallelism in solving the ATPG problem remains inconsequential, even in the face of several proposals for parallel ATPG. This lack of use is due in part to the fact that many of the algorithms previously produced are tightly tied to a particular architecture; while parallel platforms are increasing in number, the number of a particular type is still relatively small.

Much of the difficulty of parallelization of test stems in large part from the inherent exponential complexity of ATPG. This complexity makes careful algorithm design and implementation critical; even small variations introduced by the parallelization process can have a dramatic impact on performance and resultant quality. Many parallel implementations of ATPG have been developed from the ground up and have diverged significantly from existing ATPG algorithms, which has produced results that either fail to achieve significant multiprocessor utilization or suffer from a significant loss in quality.

In this chapter, we present `ProperHITEC`, a parallel implementation of the `HITEC` program for sequential ATPG. In the Section 6.1, we briefly review test generation. In the Section 6.2, we discuss the organization of the `HITEC` sequential test generator. In Section 6.3 we summarize existing methods and applications for parallelizing test generation. In Section 6.4, we consider modifications to `HITEC` necessary to support actor-based parallelism. In Section 6.5, we discuss the implementation details of `ProperHITEC`. Section 6.6 summarizes the performance of `ProperHITEC` on a number of circuits in the ISCAS-89 benchmark set [104].

6.1 Test Pattern Generation

Most computationally intensive CAD problems can be viewed as either *search problems* or *optimization problems.* We consider here test pattern generation, a form of search problem. *Test patterns* are sets of inputs to integrated circuits that are applied to fabricated devices to determine if any defects occurred during the manufacturing process. The output patterns produced by a device in response to the test patterns are compared against expected results, and the device is rejected if the patterns do not match. Generating a set of tests that ensure that no device containing a defect is erroneously flagged as good is extremely difficult for contemporary chips that contain millions of transistors.

We illustrate with an example. Test generation for a particular fault on a simple chip might reduce to finding a set of Boolean assignments to inputs a through h, such that a function $t = a(b + c(def + g) + h)d + be$ is true. The simplest way to find such an input is to search for one: we start by assigning an input, for example a, to a one. In the example, the result is $b+c(def+g)+h)d$; therefore, another input is chosen and the process is repeated. If assigning an input results in $t = 1$, that set of inputs is a test. If, however, the result is $t = 0$, this means only that that set of inputs is not a test, not that no test exists, although this is a possibility. Each of the earlier assignments must be retried with a value of zero.

The problem of finding a set of inputs that yield a one is called *satisfiability* [105], a well-known NP-complete problem. Members of the class of NP-complete problems have the characteristic that the number of possible solutions that must be examined grows exponentially with the number of inputs, making naïve enumeration impossible. Instead, *heuristics* are used to try to 'guess well'. In this context, heuristics are mathematical functions that can be evaluated for each potential guess; higher values indicate better guesses. For heuristic functions with certain properties, algorithms exist which guarantee optimal performance [106]. The art of search is the art of finding good heuristics.

Given a set of heuristics, the process of parallelizing search is relatively straightforward; one simply evaluates multiple possible solutions in parallel. When no solution exists, near perfect speedup is observed. However, when a solution does exist, the effectiveness of the parallelization effort depends directly on how much extra search is done beyond that necessary in the serial algorithm. If the first solution is nearly always correct, i.e., the heuristics are usually successful, little speedup will be observed, as most processors do unnecessary work.

6.2 HITEC: A Serial Test Generator

HITEC is a sequential test generation that exhibits performance and quality among the best known in the field [52]. Following the premises of the ProperCAD project [1], it was chosen as the basis for development of an efficient and effective parallel test generator. In this section, we review briefly techniques embodied within HITEC. Through careful examination of HITEC, we can determine where the application of object-oriented and parallel mechanisms will introduce inefficiencies, overheads, or constraints that would render the algorithm ineffective.

Although search is in some respects a regular problem, search as it is manifested in test generation shows little of this regularity. The most successful combinational and sequential ATPG algorithms are based on the PODEM algorithm [107], itself a descendant of the D algorithm [108]. PODEM differs from the D algorithm in the set of potential assumptions that the algorithm will consider. Whereas the D algorithm will assume values on internal nodes, PODEM will assume values only on primary inputs, which reduces the number of backtracks the algorithm may have to execute and accelerates the overall process.

The extension of PODEM to sequential circuits is straightforward. The sequential circuit is modeled as an infinite iterative array of identical copies of the combination portion of the circuit. The latch outputs of each time frame form the latch inputs of the next time frame. The sequential PODEM algorithm starts by considering the 'time 0' frame; a D is assumed at the specified fault and a procedure essentially the same as the combinational case is run. In the sequential case, if a D or $\overline{D}$ is propagated to a latch input, another time frame is created, a D is inserted at the fault site in that frame, and fault propagation is continued. Similarly, if a justification goal reaches a latch output, an earlier time frame is created and justification is continued in that frame.

HITEC is a descendent of and an extension to the PODEM approach. The extensions, roughly in decreasing order of significance, are:

6.2.1 Targeted *D*-frontier

Since the calculation of all implications of an assignment is NP-complete [109], test generation algorithms compute only a subset of all implications. One way of maximizing the size of this subset is to maintain a 'D-frontier', the set of nodes closest to the primary outputs that have a D or $\overline{D}$ value. For each element of the frontier set, the set of *dominators* is found; that is, the set of single assignments required to justify the frontier element. The dominator sets for each element are then intersected; the result of the intersection is a set of implications that is necessary to justify every possible propagation path to a primary output.

`HITEC` does not perform the intersection operation. Instead, the frontier element most likely to propagate to a primary output is identified, and a D or $\overline{D}$ value is *assumed* on that node. The number of implications that can be derived from this assumption is much greater than the implications resulting from the intersection of every possible propagation path. However, since a value is being assumed, this operation is a backtrackable decision; if further processing determines that the assignment cannot be propagated or justified, it must be backtracked and another element of the frontier chosen. If all elements of the frontier are exhausted without generating a test, a prior assignment must be backtracked or, if none exist, the fault is declared untestable because the fault effect is not observable.

6.2.2 State Justification

`HITEC` breaks into separate processes intraframe and interframe justifications. The determination of a set of implications necessary to justify a value always halts at frame boundaries, i.e., the primary inputs and the latch outputs, until all justification objectives for the current frame are met. At this point, if there are justification objectives on latch outputs, state justification is performed. `HITEC` attempts to speed state justification by:

- keeping a list of states, for each fault, which have been proven unjustifiable. State justification fails immediately if the new state objective is covered in the Boolean sense by any previously failed state. This method also prevents infinite extension when justification of a state requires justification of the same state in an earlier time frame, i.e., there is a cycle in the state-transition graph.

- reducing the state assignment. Because it is generated heuristically, the state assignment may be more constrained than is necessary; changing a non-X assignment to an X may still propagate the fault. By reducing the number of constraints, the state justification process is made easier.

- taking advantage of the the state of the latches produced as a result of simulating previous test vectors.

- using information computed by the fault simulator to detect the case in which a state justification objective is covered by a state generated by previous test vectors.

6.2.3 Fault Simulation

HITEC uses the PROOFS fault simulator to fault simulate, for all undetected faults, any test vectors generated or supplied externally. Since many test vector sequences detect more than one fault, this process dramatically reduces the number of faults which must be considered by the test generator. HITEC orders the fault list to attempt to take maximum advantage of these serendipitously covered faults.

6.2.4 Variable Time Frame

HITEC does not fix the length of the iterative array used to model the sequential circuit, but instead starts with a single frame and creates 'past' and 'future' frames as necessary. HITEC bounds the length of the 'window' into the iterative array but allows the window to scroll forwards and backwards as required by the propagation and justification subtasks. Scrolling has the advantage of bounding the space required to store the window without requiring the number of justification and propagation frames be fixed a priori.

The organization of HITEC (including the PROOFS fault simulator) is shown in Figure 6.1. In their original form, HITEC and PROOFS run as separate programs, communicating via a TCP/IP socket. Though the two programs run in parallel, the synchronous

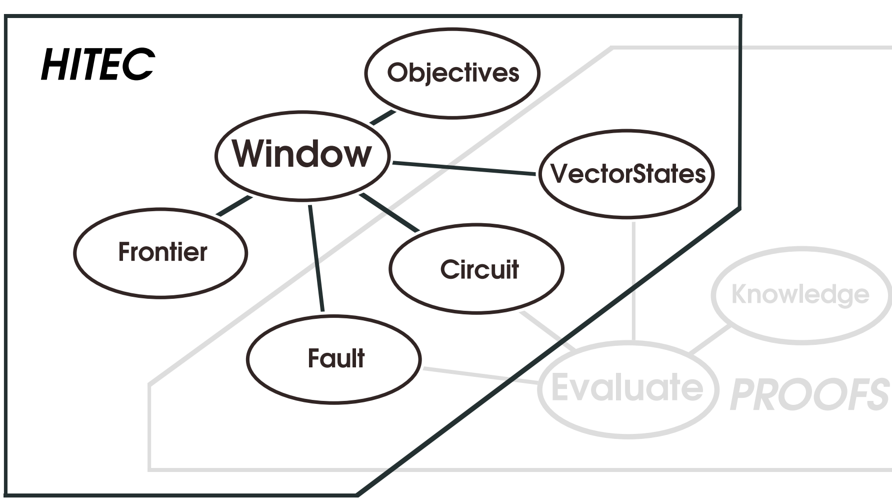

Figure 6.1 HITEC/PROOFS organization

nature of the communication precludes parallelism. The original application is written in C++ using an object-based design style. In particular, the object-oriented concept of encapsulation is only weakly represented and the use of dynamic-binding is omitted. Table 6.1 provides a brief description of the major classes in `HITEC`. Discussion of the operation of the fault simulator, `PROOFS`, is deferred to Chapter 7.

6.3 Approaches to Parallel Test Generation

A number of approaches have been proposed for parallelizing test pattern generation. We review the models and previous implementations.

6.3.1 Fault parallelism

Fault parallelism is the process of targeting a number of the many faults in a design independently. In this method, the fault set is divided—usually equally—among available processors, each processor generating tests for its fault set independently. Implementations based on this method have been proposed by Chandra and Patel [110], Patil and Banerjee [111], [112], Patil, Banerjee and Patel [113], and Agrawal et al. [114]. The main advantage of fault parallelism is low communication overhead; it is possible to achieve linear speedups when the number of processors is small compared to the number of faults. The main disadvantage is that the time required to generate a test for faults that are difficult to detect in the serial algorithm, i.e., faults that require a large number of backtracks in the serial algorithm, is not reduced in the parallel implementation. Additionally, since most test generation systems now use fault simulation to capitalize on the serendipitous detection of multiple faults by the patterns generated for a single fault, when fault parallelism is used,

Table 6.1 Classes in `HITEC`

Class	Description
`Circuit`	circuit connectivity, observability, and controllability
`Window`	time window representing circuit state for all active time frames
`Objectives`	List of justification objectives
`Frontier`	list of D-frontier nodes
`VectorStates`	database of test vectors and resulting state from `PROOFS`
`Fault`	Fault status database

speedups can fall due to one processor expending useless effort to detect a fault which will be serendipitously detected by the vectors generated for another.

6.3.2 Decision parallelism

Decision parallelism refers to the evaluation of the functions associated with several decision alternatives in parallel. In this respect, techniques to parallelize ATPG borrow from the parallelization of pure depth-first search [115], [116], [117], [118]. This technique was proposed for test generation in [119], but the search space allocation strategy did not utilize heuristics to increase the probability of searching in a solution area. A parallel branch and bound algorithm was proposed by Patil and Banerjee [120], [121] that is based on searching different portions of the search space concurrently. A similar parallel algorithm for combinational test generation, suitable for execution on a network of workstations, was proposed by Arvindam et al. [122]. Recently, a parallel algorithm for ATPG on sequential circuits has been proposed by Patil, Banerjee, and Patel [113]. The parallel algorithm, suitable for execution on shared memory multiprocessors, uses a variation of decision parallel functional decomposition. Ramkumar and Banerjee [123] used Charm [69] to create a parallel version of [113] which used both fault parallelism and decision parallelism. The work presented here adopts many of the techniques of their work.

6.3.3 Functional parallelism

As with any program, the test generation process can be partitioned into subtasks based on functional blocks, e.g., the test generator, the fault simulator, the databases, etc. Each of these tasks can be performed in parallel, limited only by data dependencies among them [124]. Motohara et al. [119] present a functional decomposition for test generation of combinational circuits. Patil has proposed a functional decomposition of test generation for sequential circuits suitable for shared memory multiprocessors [125]. The difficulty of handling data dependencies makes implementations of functional parallelism approaches particularly difficult.

6.3.4 Heuristic parallelism

All test generation algorithms use heuristics to guide the search process. Experiments reported in [126], [127] suggest that there is no clear advantage to using one heuristic over another. Parallelism can therefore be exploited by assigning to each processor a different heuristic to guide the search for the same fault. This method is referred to as heuristic parallelism. Chandra and Patel [110] report results on a parallel algorithm for test generation of combinational circuits through the use of heuristic decomposition. The primary limitation of this method is that the parallelism is limited by the number of heuristics available for search, which is generally no more than five or six. Furthermore, when different heuristics are used, there is no guarantee that search spaces are disjoint, which may lead to redundant search. Finally, no improvement is possible if a fault remains undetectable for all the heuristics.

6.3.5 Partition parallelism

Another approach to parallel test generation is based on circuit decomposition or partition parallelism. In the other parallel approaches, each processor has a copy of the complete circuit; for large circuits, the memory of each processor may not be capable of storing the entire circuit. In a partition parallel approach, each processor keeps a partition of the circuit and performs backtracing operations on its own subcircuit to satisfy the various test generation objectives [128]. It has proven to be extremely difficult to achieve effective speedups using this approach due to the high level of communication.

6.4 Parallel Test Generation using Actor Parallelism

As noted previously, any changes to an established and proven ATPG algorithm are more likely to have an adverse effect on performance than they are to have a beneficial effect. Since parallelization potentially changes the operation order of the algorithm, every change must be considered carefully. In this section, we consider the changes required to parallelize `HITEC`. We defer less significant, more mechanical changes and actual implementation details to Section 6.5. Our approach to parallelization is an extension of methods developed in earlier work in parallel ATPG [112], [121], [123], [125]. Parallelism is generated in two ways; through fault parallelism and decision parallelism (Figure 6.2).

6.4.1 Fault parallelism

Fault parallel execution is simply the process of partitioning the set of undetected faults among the available processors and then running the test generation algorithm for each partition in parallel. Implementation of fault parallelism is relatively easy since test generation for different faults is largely independent. In terms of `HITEC`, fault parallelism is achieved

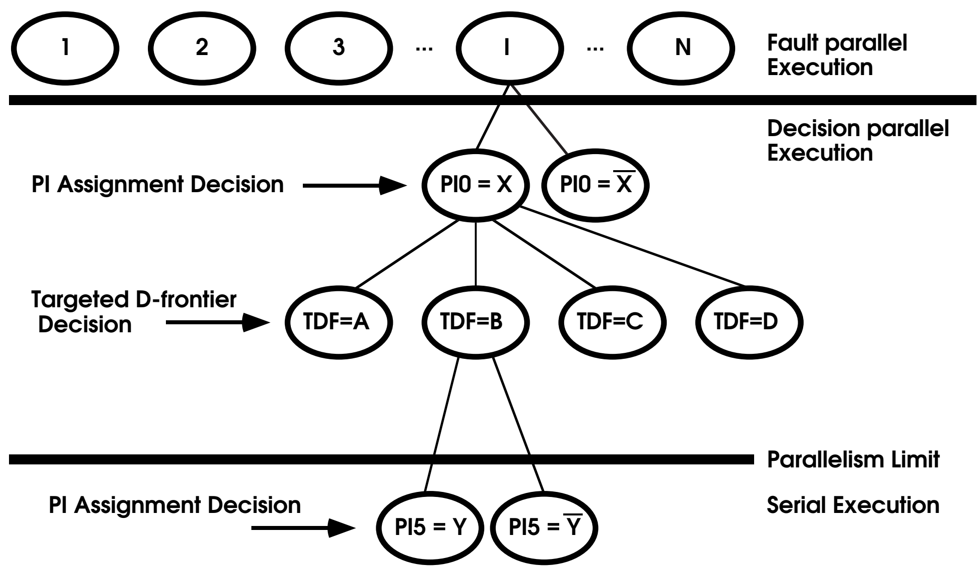

Figure 6.2 Parallelism in `ProperHITEC`

simply by creating the `Window` objects in parallel rather than serially. There is minimal impact on execution of `HITEC` when faults are tested in parallel. Since the only information required to create a `Window` object is the target fault, creation of `Window` objects for all faults in parallel has a negligible impact on performance. Although results of the parallel implementation will vary due to differences in the order of execution, very little of the core of the algorithm is affected.

The only heuristics impacted by fault parallel execution are the state knowledge heuristics. In serial `HITEC`, a `Window` object can capitalize on the fact that the the state of the circuit resulting from the execution of previous test vectors is known. Moreover, the state of the circuit resulting from each previous vector is recorded in the `VectorStates` object. In parallel `HITEC`, the state of the circuit due to the previous test vectors is not known, because it is dependent on the order of evaluation of the individual test generation objects. Similarly, although the states produced by previous test vectors are still available, in fault parallel execution, the information available at any given time will not be identical to that of the serial algorithm. While the issue of nondeterminism is not of particular concern, the inability to model the previous latch state could be cause for concern if the impact were great. The efficacy of this approach as reported in [52] is unclear, and this feature is not enabled by default in the original `HITEC` implementation. We did not try to support this feature and did not attempt to measure the impact.

6.4.2 Decision parallelism

In decision parallel execution, when a backtrackable decision is made, rather than performing a depth-first search of the alternatives, new `Window` objects are created in parallel to explore the alternatives. In PODEM, the only decisions are input assignments, thus the object graph for each fault is a binary tree. In `HITEC`, backtrackable decisions are also made on the targeted D-frontier; therefore, each node in the search tree is either an input assignment with two out-edges representing the alternate assignments or a D assignment with N out-edges, where N is the total number of nodes in the D-frontier when one was targeted.

The use of decision parallelism was deemed necessary to achieve high efficiency on many processors while quality of results is maintained. Quality is higher for decision parallel execution because the evaluation of test objectives more closely matches the serial version when decision and fault parallelism are combined than when fault parallelism alone is used. In strictly fault parallel execution, all but one processor are working on a fault different from the serial algorithm; if additional faults are covered by patterns generated by a previous fault, all work done in generating tests for those faults is wasted. Moreover, since test generators generally spend most of their time generating tests for a relatively small number of hard faults, even with fault parallelism, execution time is bounded on the low end by the time required to test the most difficult fault. Decision parallelism explores different areas of the search space in parallel; therefore for cases in which a large portion of the search space must be explored, significant speedup can be achieved. Note that by casting the ATPG search in such a general framework, it is easy to enable purely fault parallel, purely assignment parallel (i.e., PODEM/`ProperTEST`), or assignment and D-frontier parallel execution (`HITEC`).

Only one significant change to `HITEC` heuristics was required to implement decision parallelism. The change stems from the inability to pass information up the search tree in parallel search. When a backtrack occurs in depth-first search, information gained in detecting the backtrack can be passed up the search tree to influence the backtracking process itself. `HITEC` utilizes this information when a state-justification failure occurs; when this type of condition occurs, a `backtrack-must-change-state` flag is set, which indicates that any decision alternatives that do not change the state-justification target should be backtracked immediately. When a decision alternative results in a different state for justification, normal operation resumes.

When decision parallelism is used, a breadth-first-like search is employed. Since decision alternatives no longer execute strictly in sequence, there is no way to pass the `back-`

track-must-change-state flag among them. Fortunately, this information is closely related to the complete set of unjustifiable states, already maintained by the fault database. For those cases in which the backtrack-must-change-state flag is set, the backtracked state must be in a list of failed states. Checking the failed states list is actually more accurate, because a backtrack that does change the state may simply change it to another failed state, a condition which the original HITEC algorithm does not detect. Since the amount of processing required to check the failed states list is greater than that required to simply test a flag, the choice of which technique to use is dependent on the relative amount of time required to check the list versus that required to perform the useless work of exploring nonsolution alternatives. Testing of HITEC using both heuristics showed the cost of useless searching to be far greater than the cost of checking the failed states list; therefore, HITEC and ProperHITEC have been modified to use the effective heuristic.

Implementation of decision parallelism involves overloading the decision routines store_pi() and store_Dfrontier() to create new Window objects. In essence, the process consists of cloning the current Window object, with the exception of the assignment to be made. Since the Window object contains the state of all nodes in the circuit for all active time frames, the cost of this operation can be significant. Also, since the number of new Window objects produced rises exponentially in the number of decisions performed in parallel, beyond a relatively low limit the benefit of decision parallel execution disappears. When decision parallelism is used, a limit on the depth of decisions performed in parallel is set; after a fixed number of decisions is executed in parallel, operation resumes using the HITEC depth-first algorithm.

6.5 ProperHITEC

ProperHITEC, based on the original HITEC code and the ProperCAD II library, implements the parallelism features described in Section 6.4. To implement parallel search, multiple search processes—Window objects—are created, either from the fault to be tested (fault parallel) or from the fault, the current state of the heuristics, and the set of previously assigned inputs (decision parallel). Furthermore, we have to distribute the data contained in the other objects, e.g., the fault database, the vector/states database, and the fault simulator. Figure 6.3 shows the ProperHITEC objects that have parallel semantics, along with the HITEC and AIF objects from which they are derived.

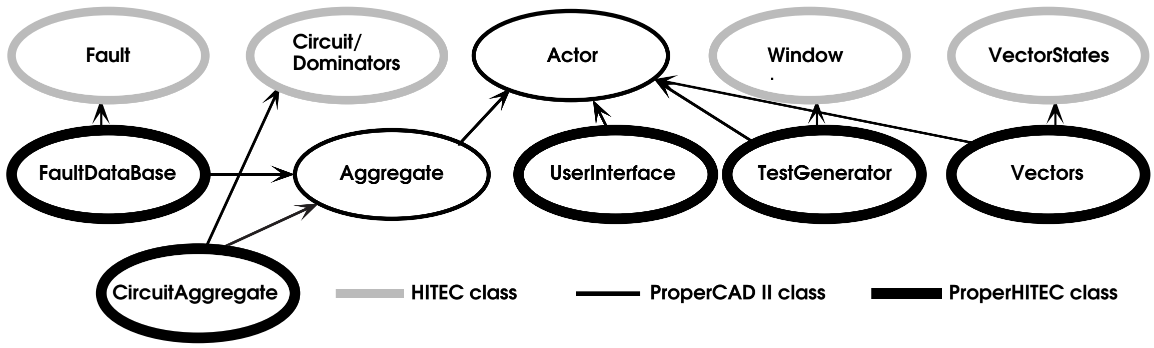

Figure 6.3 `ProperHITEC` organization

6.5.1 TestGenerator

The `TestGenerator` object is an actor that represents a ‘test generator machine.’ `TestGenerator` instances can be created for a specified fault to implement fault parallelism, or they can be cloned from an existing instance and an alternate assignment to implement decision parallelism. Decision parallelism is bounded by a user-specified limit, beyond which decisions are made in a depth-first manner using the serial code.

`ProperHITEC` uses lexicographically ordered bit-string priorities to guide the execution of `ProperHITEC` as closely as possible to the order used in the sequential algorithm. When bit-string priorities are used, the next test generator object to be evaluated, chosen from a group of tasks, is always the one which would have been evaluated first by the sequential algorithm. On a single processor, test objectives are evaluated in the same order as they are in `HITEC`; the only case for which `HITEC` and `ProperHITEC` on one processor generate different results occurs when per-fault time limits are used, in which case small differences in run times can cause a fault to be aborted in one application, where it is detected in the other.

The use of bit-string priorities in `ProperHITEC` is an extension of the method proposed in [123]. The priorities are extended to support the nonbinary nature of the search tree in `HITEC`. We note that in the AIF, the use of bit-string priorities for the test generator objects has no effect on the priority representations used by the other objects in `Proper-HITEC` or in the library in general.

6.5.2 CircuitAggregate

The circuit is implemented as an aggregate with a `PerThread` distribution. Data sharing via a `PerProcessGroup` distribution was implemented but was later removed because the HITEC algorithm uses fault injection through circuit modification; a method of parallel fault injection algorithm that would allow the circuit data to be shared resulted in excessive overhead in the core HITEC algorithm. Switching the distribution of the aggregate is a one-line change.

6.5.3 FaultDataBase

The fault database is implemented as an aggregate with a PerThread distribution. Each representative stores the most recent state of the test generation process for each fault and provides the same interface to the test generator objects as does the serial Fault object. When an update is received from a TestGenerator object or from the vector/states database, the FaultDataBase representative records the information locally, and if this results in a change of the local state, it broadcasts that information to all other representatives. The broadcast operation is implemented via operations on first class names.

6.5.4 Vectors

The Vectors object is essentially the same as the serial object but uses ActorMethods to record new vectors and to send results to the FaultDataBase. Fault simulation is performed using the FaultSimulator object used in HITEC, invoked by actor methods provided by the Vectors actor. Test generators have been observed to spend little time in fault simulation; therefore, parallelization of the fault simulator used for test generation was not investigated. Parallel fault simulation as an independent task is covered in Chapter 7.

6.5.5 UserInterface

The UserInterface object is used to interact with the user during the running of the test application. It creates the system objects and then creates test generator objects for each undetected fault in the circuit. If the progressive time limit feature of HITEC is used, the process of creating test generator objects is iterated with progressively larger time limits.

6.6 Performance

6.6.1 Performance results

Tables 6.2, 6.3, and 6.4 show the results of `ProperHITEC` for a number of circuits drawn from the ISCAS-89 benchmark set [104]. All times, T, are reported in seconds and represent the elapsed wall clock time.[1] Fault efficiency, E, computed as

$$\text{efficiency} = \frac{(\#\text{faults} - \#\text{aborted})}{\#\text{faults}}$$

is presented. V is the number of test vectors generated.

For each circuit, the results of the sequential `HITEC` algorithm and the `ProperHITEC` algorithms on various machine configurations are reported. The `HITEC` numbers presented are for the version of `HITEC` that shares code with `ProperHITEC`. Although the current version of `HITEC` takes greater advantage of dynamic memory allocation, the amount of time spent doing memory management has been carefully analyzed and has been determined to be less than a fraction of a percent.

`ProperHITEC` achieves consistent speedup with only a marginal effect on quality across a range of moderately difficult test problems. `ProperHITEC` achieves consistent speedup with only marginal effect on quality across a range of moderately difficult test problems. The major effect on quality is the addition of a small amount of noise in the results; while `ProperHITEC` does not always achieve results identical to `HITEC`, the number of cases for which it produces higher quality results are comparable to the number of cases for which it produces lower quality results.

[1] The only time excluded from reported times are for initialization and final writing of results, which are required because some of the parallel machines implement I/O poorly.

Table 6.2 `ProperHITEC` results on Sun 4/670MP

Circuit/ Seconds Per Fault	HITEC			ProperHITEC Processors 1			2			3			4		
	T	E	V	T	E	V	T	E	V	T	E	V	T	E	V
s344/20	369.4	95.9	121	374.3	95.9	121	251.9	96.2	110	160.0	96.5	130	156.2	96.2	112
s820/20	435.9	99.3	956	396.8	99.3	956	225.4	99.3	1010	196.3	99.1	1059	140.3	99.1	1013
s953/20	125.8	100	20	134.2	100	20	71.37	100	12	64.17	100	16	47.24	100	12
s1238/2	13.13	100	386	21.64	100	386	15.15	100	390	13.31	100	405	16.18	100	385
s1494/20	722.0	99.1	1058	663.4	98.9	1058	434.1	98.9	1123	350.5	98.9	1153	240.1	99.1	1093

Table 6.3 `ProperHITEC` results on Intel iPSC/860

Circuit/ Seconds Per Fault	HITEC			ProperHITEC Processors 1			2			4			8		
	T	E	V	T	E	V	T	E	V	T	E	V	T	E	V
s344/20	481.4	94.2	89	485.8	94.2	89	215.8	96.8	105	194.5	96.8	112	142.1	96.5	102
s820/20	438.3	99.3	959	440.8	99.3	959	270.4	99.2	958	158.0	99.3	951	108.0	98.9	1034
s953/20	140.2	100	14	147.7	100	14	89.13	100	14	49.00	100	24	28.66	100	14
s1238/1	14.15	100	374	23.29	100	374	14.69	100	383	12.16	100	369	11.12	100	402
s1494/20	819.8	99.0	1079	821.3	98.7	1079	503.3	99.1	1168	310.1	98.6	1113	192.2	98.8	1151

Table 6.4 `ProperHITEC` results on Encore Multimax

Circuit/ Seconds Per Fault	HITEC			ProperHITEC Processors 1			2			4			8		
	T	E	V	T	E	V	T	E	V	T	E	V	T	E	V
s344/20	484.2	93.9	105	493.2	93.9	105	274.3	95.6	95	167.0	95.6	85	131.2	94.7	108
s820/20	1200	97.8	891	1206	98.1	891	761.8	97.6	1008	418.9	97.2	959	255.6	96.9	955
s953/100	572.4	100	20	597.1	100	20	343.3	100	18	252.1	100	14	166.7	100	10
s1238/10	65.41	100	386	97.49	100	386	60.53	100	382	53.83	100	382	55.35	100	406
s1494/10	2615	87.0	492	2920	84.2	402	1654	85.5	460	997.0	83.9	497	540.2	85.5	510

Cases for which the parallel algorithm does not achieve acceptable results are the 'easy' benchmarks, those for which 100% efficiency is achieved by the serial algorithm within a few seconds. Since these are easy problems which finish quickly, optimization for these cases is not of great interest.

6.6.2 Workstation cluster results

Table 6.5 shows the results of `ProperHITEC` on a workstation cluster of Sun SPARCstation 10s and Sun 4/600MPs. The columns labeled 2 and 4 processors were run on the indicated number of individual workstations. The column labeled 2/2 was run on an IP-connected pair of Sun 4/600 multiprocessors. Each machine had four processors, but only two were used in each case. The results for the workstation cluster port show much greater variation than do the previous results. This variation is a function of the performance of the APA UDP implementation (Section 4.5). If the cases in which the network protocol failed are excluded, the results show speedups comparable to those on multiprocessors and mul-

Table 6.5 `ProperHITEC` results on clusters

Circuit	HITEC		ProperHITEC Processors							
			1		2		4		2/2	
	T	E	T	E	T	E	T	E	T	E
s526	6083	12.1	1.01	12.1	2.0	12.3	1.0	12.1	3.1	12.3
s820	468	99.5	0.95	99.5	2.0	99.5	0.4	99.3	1.6	99.5
s5378	2982	71.1	1.03	71.1	0.5	70.3	2.4	69.4	3.0	72.5

ticomputers. It appears that the use of clusters of multiprocessors helps to mitigate some of the UDP artifacts.

6.6.3 Efficiency results

In addition to providing faster turnaround, parallel processing can be used to achieve higher fault efficiency in a fixed amount of time. Table 6.6 shows the results of running test generation on s1494 where the per-fault time limit was scaled with number of processors.

Consistent improvement is observed for all platforms. Of note is the fact that even though the time limit was raised in tandem with the number of processors, run times still decrease on parallel runs. This result is due to the fact that once a fault is detected, raising the per-fault time limit does not increase run time.

Table 6.6 Increased efficiency in `ProperHITEC`

Machine	Seconds Per Fault	# of Processors	T	E
Sun MP	1	1	707.0	82.2
	2	2	230.7	97.1
	3	3	222.0	97.6
	4	4	185.4	98.1
iPSC/860	2	1	777.4	79.5
	3	2	335.0	94.4
	8	4	212.7	97.8
	20	8	192.2	98.8
Multimax	2	1	2069	42.0
	4	2	1465	67.1
	8	4	1047	77.4
	20	8	540.2	85.5

6.6.4 Comparison to ProperTEST

Table 6.7 compares the results of ProperHITEC with those of ProperTEST [123], an earlier work in parallel ATPG, based on similar parallelism concepts. The table shows the results of the two applications run on an iPSC/860 with eight processors. In general, where ProperTEST achieves high fault efficiencies, the results are faster but comparable to ProperHITEC. In cases where ProperTEST does not achieve high efficiency, ProperHITEC does and in a fraction of the time.

6.7 Evaluation

One of the most interesting aspects of the development of ProperHITEC was the use of derivation to extend the serial application to a parallel application. In previous work using Charm [123], similar algorithmic techniques were used, but they required incompatible changes to the original serial code. In the case of ProperHITEC, the only fundamental change was the modification of several functions in the serial code to use dynamic binding in order that those functions could be overridden in the derived actor and aggregate classes. Because changing static dispatch to dynamic dispatch involves the addition of a level of indirection, the performance impact was considered carefully. The affected functions are not called within the most computationally intensive part of the test generation process and no performance degradation was observed.

Aggregates were key to expressing data distribution in a simple way and to keeping status information representing the progress of the test generation process up to date. Heterogeneous priorities were used both to order the evaluation of test generator objects and to ensure that continuations called on the aggregates took priority; delays in distributing updates

Table 6.7 Comparison of ProperHITEC and ProperTEST on iPSC/860

Circuit/ Seconds Per Fault	ProperHITEC		ProperTEST	
	T	E	T	E
s344/20	142.1	96.5	97.5	95.0
s349/20	128.1	94.6	109.9	95.4
s820/20	108.0	98.9	1081.5	60.4
s832/20	118.0	98.7	1053.8	61.1
s1494/10	192.2	98.8	756.6	73.4

to the fault database, for example, result in wasted work, as TestGenerator objects try to generate tests for faults that have already been tested.

It is difficult to quantify the effort required to create ProperHITEC from HITEC. The design and implementation of ProperHITEC were performed in parallel with the developments of the AIF and APA, which makes determination of the duration of the development period impossible. Moreover, a significant amount of work was required to regularize the original serial HITEC/PROOFS code; these applications represent the first use of C++ by the original designer. When the time necessary to clean up the original code is excluded, it is estimated that the preliminary parallelization effort took approximately two man-months, and improvements and the most significant debugging required an additional man-month.

Quantification of changes to the original application is equally difficult, in this case not because statistics are difficult to generate, but because simple measurements—such as the raw number of lines of code—contain limited information without additional contextual information such as coding style. We report several metrics of the HITEC and ProperHITEC implementations in Table 6.8; interpretation should be done with care. In addition to reporting the raw number of lines of code, we report the number of lines containing a semicolon; this measure shows less variation across coding styles. Also reported is the number of classes in the different versions; virtually all the significant classes are shared between the parallel and serial versions. The exceptions are the five actor and aggregate classes shown in Figure 6.3 and the TestPriority class used to specify an order on parallel evaluation of test generation tasks.

While the values for lines of code indicate a significant amount of new code, the bulk of the additional code is declarative, which provides new control flow, via ActorMethods, and aggregation of data, i.e., the structures used to represent the single argument to an actor

Table 6.8 Comparison of software metrics for HITEC and ProperHITEC

Metric	HITEC	ProperHITEC	Shared
Lines of code	336	6287	10685
Lines of code with ‘;’	110	1165	3056
Classes	0	6	17
excluding ActorMethod and argument classes			
ActorMethod classes	0	27	0
ActorMethod argument classes	0	13	0

method. The number of classes used to represent these two sets of operation are shown in the last two lines of Table 6.8.

A breakdown of the number of actor methods by class is shown in Table 6.9. The majority of `ActorMethods` are found in the `FaultDataBase` and `UserInterface` classes. In the `FaultDataBase` class, the methods are used to communicate allocation and status information on the current state and usage of resources for each fault. In the `UserInterface`, the majority of actor methods are used to sequence the creation and destruction of aggregate instances.

Table 6.10 presents the numbers of member functions in the concurrent types and the serial types from which they are derived. These data represent a measure of the parallelization effort and show that most functionality in `ProperHITEC` is inherited from `HITEC`. Most of the functions in `ProperHITEC` are small; they perform most of their computation through calls to members of the serial base class.

In addition to the creation of the concurrent object types, to implement parallelism a number of previously statically bound members of serial `HITEC` classes had to be modified to use dynamic binding; the original serial code contained no use of dynamic binding. The number of virtual members is reported in Table 6.11 along with the total number of mem-

Table 6.9 `ActorMethods` for each class in `ProperHITEC`

`ProperHITEC` class	Number of `ActorMethods`
`TestGenerator`	3
`CircuitAggregate`	4
`FaultDataBase`	8
`Vectors`	4
`UserInterface`	8

Table 6.10 Member functions in `HITEC` and `ProperHITEC`

`HITEC` class	Number of member functions	`ProperHITEC` class	Number of member functions
`Window`	78	`TestGenerator`	8
`Circuit`	47	`CircuitAggregate`	6
`Fault`	35	`FaultDataBase`	21
`VectorsStates`	12	`Vectors`	5

bers. Most classes required few if any new virtual members, the exception being the fault database which contains a number of small status update functions that are overloaded in the parallel application to propagate updated status information to all processors.

Table 6.11 New virtual members in HITEC

HITEC class	Number of member functions	Number of new virtual member functions
Window	78	4
Circuit	47	0
Fault	35	13
VectorsStates	12	0

Chapter 7

PARALLEL FAULT SIMULATION

Fault simulation is used to determine the fraction of faults in a circuit that are covered by a given set of tests. As was shown in Chapter 6, fault simulation is used during automatic test pattern generation to identify those faults detected serendipitously to minimize the number of faults to be targeted by the test generator. Fault simulation is also used to evaluate the efficacy of sets of test vectors. These vectors may be functional vectors generated during the design of a circuit or may be generated randomly prior to test generation to minimize the number of faults to be processed by a deterministic test generator. The results of fault simulation are lists of faults detected and undetected by the set of vectors. From these lists, the *fault coverage,* the ratio of the number of detected faults to the number of total faults, can be calculated. On large complex circuits with large test sets, fault simulation can take from hours to days.

In this chapter, we present `ProperPROOFS`, a parallel implementation of the `PROOFS` fault simulation package. `ProperPROOFS` uses the aggregate model as implemented in the AIF to decompose the fault simulation problem. In addition to the use of derivation to incrementally parallelize an existing state-of-the-art serial fault simulation application, `ProperPROOFS` incorporates a new asynchronous and distributed method of fault redistribution to improve load balance.

After briefly reviewing the area of fault simulation, we summarize the features of the `PROOFS` package for sequential ATPG. In Section 7.4, we consider modifications to the `PROOFS` package necessary to support parallelism. In Section 7.5, we discuss the implementation details of `ProperPROOFS`. Section 7.6 summarizes the performance of `Proper-PROOFS` on a number of circuits in the ISCAS-89 benchmark set [104].

7.1 Fault Simulation

The task of fault simulation is the determination of the set of detected faults in a circuit for a given set of test vectors. For each fault in a specified list of faults, the fault simulator determines whether the results of the fault-free circuit and the circuit containing the specified fault would differ in any circuit output. A naïve fault simulator could be built from a logic simulator by running a logic simulation of the test set against the good circuit, recording the results, modifying the circuit in order that it exhibits the behavior of the faulty circuit, running the test set again, and comparing the results. If the two results differ, the fault is detected; otherwise, it is not detected.

While such a methodology is sufficient and thus provides an upper bound on the computation required by fault simulation, it is exceedingly expensive. With current fault models, the number of faults in a circuit grows linearly with the size of the circuit, and with circuits of hundreds of thousands of gates, significant effort has been expended in optimizing the basic logic simulation concept for fault simulation.

Fault simulation can be viewed as filling the entries of the table in Figure 7.1 [129]. Each row in the table represents a circuit: the first row represents the good error-free circuit, the succeeding rows represent a circuit with one fault, as indicated by the subscript. There is one column in the table for each vector in the test set. The entries of the table represent the state of the indicated circuit after application of the indicated and all preceding vectors. Fault simulation techniques may be classified by the manner in which they fill the table. The most popular approaches in serial fault simulation are concurrent, deductive, differential, and bit-parallel.

	Vector_1	Vector_2	•••	Vector_n
Good	G_1	G_2	•••	G_n
Faulty_1	$F_{1,1}$	$F_{1,2}$	•••	$F_{1,n}$
Faulty_2	$F_{2,1}$	$F_{2,2}$	•••	$F_{2,n}$
•••	•••	•••	•••	•••
Faulty_m	$F_{m,1}$	$F_{m,2}$	•••	$F_{m,n}$

Figure 7.1 Fault simulation table model

In concurrent fault simulation [130], each faulty entry is filled using the good circuit entry for the same vector, computed separately, and the entry to the immediate left, the state of the same faulty circuit during the previous vector (Figure 7.2). Concurrent techniques generally result in fewer numbers of events at the expense of memory necessary to store lists of active faulty machines.

In deductive fault simulation [131], faulty entries are filled in the same order as concurrent fault simulation. Deductive fault simulation differs from concurrent in that set-intersection operations are performed on the set of active faulty machines for each node. While the storage requirements are less than those of concurrent fault simulation, the computational costs are higher.

In differential fault simulation [132], each faulty entry is filled using only the entry immediately above, the state of the previous faulty circuit during the same vector (Figure 7.3). While memory efficient, the dependency between faulty machines prohibits *fault dropping*,

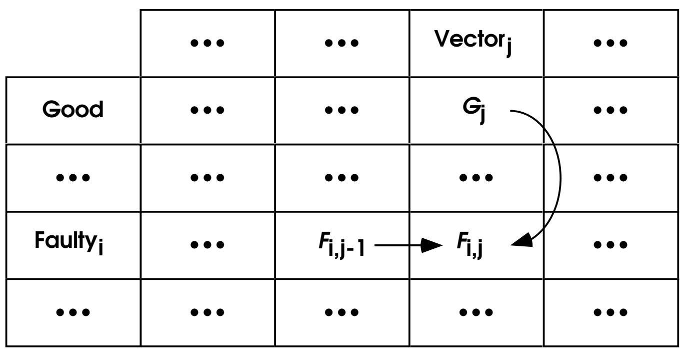

Figure 7.2 Concurrent and deductive fault simulation

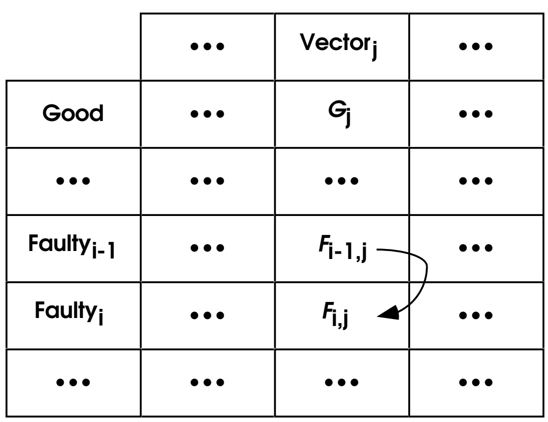

Figure 7.3 Differential fault simulation

the process of removing faults from future consideration once the fault has been detected for the first time.

In bit-parallel fault simulation [133], all bits in the processor word are used and entries are filled in groups of the word size (Figure 7.4). On a k-bit processor, the good circuit and $k-1$ faulty machines are simulated in each operation. This use of all bits in the processor word can lead to as much as a k-fold improvement in performance, though in practice smaller improvements are observed due to an increase in the number of events processed by the simulator.

In the literature, what we are calling bit-parallel is generally called *parallel fault simulation*. Due to the ambiguity of the term 'parallel' when used in the context of parallel processing, we will prefer the phrase bit-parallel to describe this class of techniques.

7.2 PROOFS: A Serial Fault Simulator

`PROOFS`, developed by Niermann et al. [134], combines techniques from bit-parallel, concurrent, and differential fault simulation to achieve performance exceeding the individual methods without incurring large memory requirements. In this section, we review briefly the techniques incorporated within `PROOFS`.

7.2.1 Dynamic fault grouping

`PROOFS` uses a dynamic method of grouping faults into words to perform bit-parallel fault simulation. For each test vector, `PROOFS` iterates over the list of undetected faults,

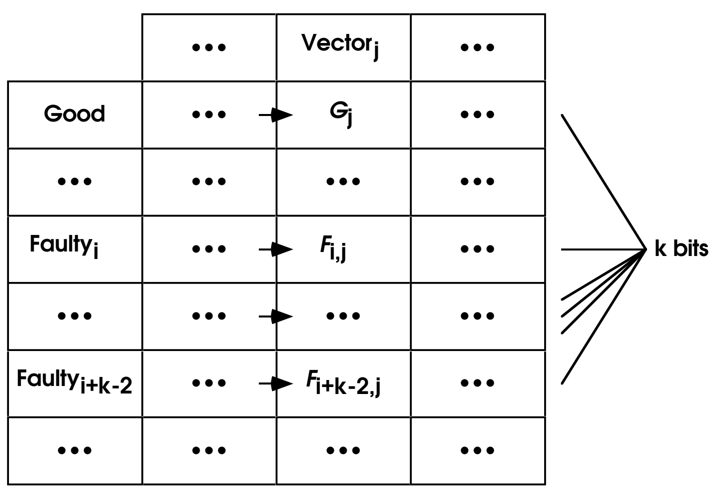

Figure 7.4 Bit-parallel fault simulation

selecting faults until the word length is reached or until there are no additional faults. Good circuit simulation is performed separately; therefore all bits of the words are used. During selection of faults, each candidate is checked to see whether it is active in the current time frame; faults that are not active are excluded, which eliminates useless simulations. The fault list is ordered to assure that there is a high probability that simulation of faults within the same group will result in activity in the same part of the circuit, which will decrease the number of events processed.

7.2.2 Fault injection

`PROOFS` uses the same fault injection scheme used in `HITEC`. By injecting faults via circuit modification, the event evaluation loop, the core of the algorithm, becomes more regular and therefore faster.

7.2.3 Fault dropping

The combination of techniques incorporated into `PROOFS` allows for fault dropping as soon as a fault is detected.

7.2.4 Faulty state storage

`PROOFS` simulates all faults for an individual vector before processing the next vector. Before evaluating any faults, `PROOFS` simulates the good circuit using an event-driven logic simulator, storing the values at every node for reference during faulty simulation. Faults are simulated in groups, again using event-driven simulation. In this case, the event-driven simulation is relative to the values in the good circuit. At the end of each group, only the state values for faulty circuits that are not detected and that are not equal to the good circuit values are saved; the complete state of the circuit for each faulty circuit is discarded.

The organization of `PROOFS` (including the `HITEC` test generator) is shown in Figure 7.5.

7.3 Approaches to Parallel Fault Simulation

A number of approaches have been proposed for the parallelization of fault simulation. We review the models and previous implementations, restricting the discussion to techniques

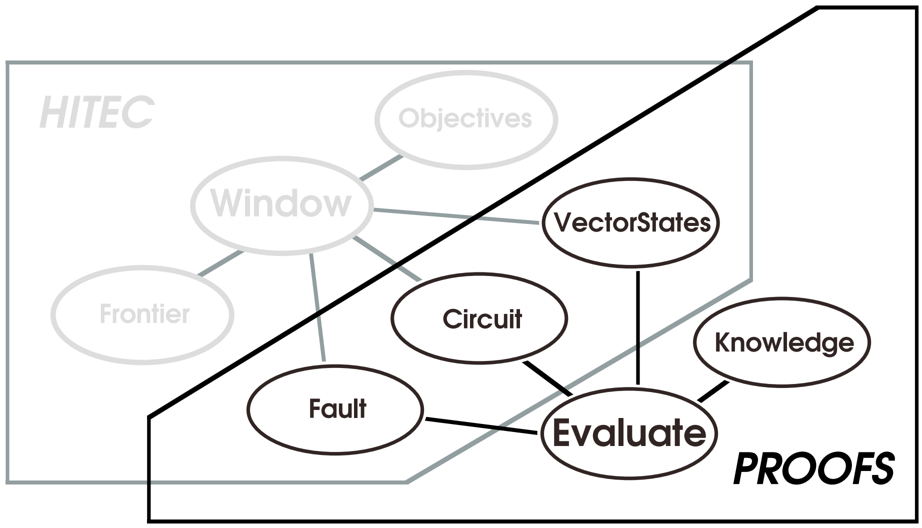

Figure 7.5 PROOFS organization

applicable to sequential circuits. Discussion of techniques applicable only to combinational circuits can be found in [135].

7.3.1 Fault partitioning

Fault partitioning in fault simulation is similar in concept to fault parallelism in test generation. The fault list is divided—again, usually equally—among available processors, each processor simulating all faults in its partition independently. However, more than in test generation, load balance is a critical issue in fault simulation. Due to the wide variation in event activity generated by different faults, static partitioning is generally not successful. Among the parallel fault simulation implementations based on fault partitioning, a number of differences are found in partitioning and load balancing. In addition to losses of efficiency due to uneven load balance or of computation necessary to implement load balancing, partitioning of the fault list gives rise to overheads due to duplicate computation of good circuit values. The only techniques that do not incur this overhead are bit-parallel simulations for which the good circuit values are always calculated with every fault group.

Several implementations based on fault partitioning have been reported. Virtually all are based on a master-slave model in which a single processor is dedicated to maintaining the list of undetected faults. Slaves request fault groups from the master, perform simulation, and return results. Duba et al. [136] report this type of scheme based on the CHIEFS concurrent hierarchical serial fault simulator [137]. The implementation is targeted at workstation clusters and uses remote procedure calls (RPCs) [20] for communication. Speedups from five to six were reported on eight processors. Markas et al. [138] report a distributed fault simulation algorithm on a heterogeneous workstation cluster; speedups ranged from two to six on eight workstations for a small number of examples.

7.3.2 Circuit partitioning

The main alternative to fault partitioning is circuit partitioning, in which the good circuit being simulated is partitioned among available processors. Circuit-partitioned fault simulation is effectively a variant of parallel logic simulation, itself a difficult problem [139]. In fault simulation, the problems of logic simulation are compounded by the link between circuit partitioning and fault list partitioning; the number of faults a processor simulates is fixed by the partitioning. Short of redistributing the circuit, there is no way to redistribute faults if all faults in a partition are dropped.

Fault simulation based on circuit partitioning has been reported by Mueller-Thuns et al. [140] and Nelson [141] for a vector-synchronous implementations on message passing machines. Ghosh [142] presents an asynchronous implementation based on asynchronous logic simulation techniques that, while novel, falls short of achieving high processor efficiency. In [143], Patil et al. present a circuit-partitioned approach applicable to shared memory machines machines that incorporate techniques from parallel logic simulation [139], [144].

7.3.3 Pattern partitioning

For combinational circuits, fault simulation can be trivially parallelized by partitioning the test vector set. The only significant issue is load balance, similar to the fault partitioning case. For sequential circuits, the problem is much more difficult, because simulation of any vector in a sequential circuit requires the state resulting from all previous vectors. Kung and Lin [145] present a novel technique for applying pattern partitioning to sequential circuits. In their algorithm, the good circuit is simulated for all vectors, but faulty circuits are simulated for only a subset of the vectors. Because the faulty state is not available for unsimulated vectors, faulty simulations may be inaccurate. In those cases, the error is detected and multiple simulation passes are performed. The advantage of this method is that it is based on a fault simulation technique that demonstrates good performance but is otherwise inapplicable to sequential circuits. Banerjee [135] has proposed a parallel fault simulation algorithm for sequential circuits based on pattern parallelism; no implementation has yet been reported.

7.4 Parallel Fault Simulation Using Actor Parallelism

In addition to casting fault simulation into an actor framework, a primary goal of this work was the elimination of single points of contention, e.g., the actor or processor that does load balancing in master-slave methodologies.

7.4.1 Fault partitioning

The approach adopted in this work is the fault partitioning method covered in Subsection 7.3.1. A single aggregate, `FaultSimulatorAggregate`, is created to represent the fault simulator; faults are partitioned among representatives. In a fault simulation application, the aggregate is given a `PerThread` distribution. Each representative is responsible for fault simulating its set of faults for all vectors in the test set.

Fault partitioning in the aggregate model is similar to the techniques reviewed in Subsection 7.3.1 with a significant difference: a master-slave model for load balancing is not used. Instead, two methods were investigated.

7.4.1.1 Static partitioning

Static partitioning was implemented, the results for which are shown in Table 7.1 for circuits from the ISCAS-89 benchmark set. For each circuit, the results of the serial application, `PROOFS`, is shown for fault simulation of 1000 random vectors. The results for the parallel application, `ProperPROOFS`, without dynamic redistribution are shown for a single processor and several multiprocessor configurations. The results in the table show that static redistribution led to less than optimal speedup, and as a result dynamic redistribution was considered.

7.4.1.2 Dynamic partitioning

The motivation for the aggregate model was the use of a multiaccess interface; to add a bottleneck in the form of a master representative is a limitation to scalability. Instead, an asynchronous, distributed method of fault redistribution was developed to achieve effective load balance without limiting scalability. In the remainder of this subsection, we look at the asynchronous redistribution of faults and address the issue of termination detection when the fault redistribution process is distributed.

Table 7.1 Run time and speedup for static fault distribution on the iPSC/860

Circuit	PROOFS (sec)	ProperPROOFS Processors (speedup)			
		1	2	4	8
s208	9.1	0.99	1.51	2.60	4.48
s208o	3.6	0.95	1.40	1.79	1.98
s298	4.5	0.94	1.21	1.60	1.62
s344	4.4	0.89	1.01	1.05	1.10
s349	4.5	0.89	1.01	1.04	1.09
s382	10.9	0.94	1.29	1.92	2.38
s386	3.0	0.93	1.38	1.68	2.19
s400	12.2	0.95	1.36	1.91	2.45
s420	23.5	0.97	1.49	2.56	4.43
s444	14.5	0.96	1.34	2.02	2.84
s510	37.8	0.96	1.62	2.55	4.41
s526	17.4	0.96	1.45	1.87	2.52
s526n	16.8	0.96	1.41	1.88	2.54
s641	5.9	0.96	1.12	1.50	1.57
s713	6.7	0.97	1.19	1.58	1.74
s820	11.5	0.96	1.20	1.54	1.83
s832	11.7	0.95	1.22	1.55	1.88
s838	63.2	1.01	1.50	2.55	4.37
s953	101.9	1.03	1.41	2.66	3.96
s1196	9.9	0.99	1.41	1.90	2.28
s1238	11.4	0.99	1.50	2.00	2.51
s1423	46.1	0.98	1.64	2.70	4.14
s1488	21.5	0.98	1.26	1.50	1.65
s1494	21.8	0.96	1.27	1.52	1.67
s5378	150.3	0.99	1.34	2.19	3.74
s9234	458.8	1.00	1.60	2.48	2.90
s13207	1598	0.99	1.38	1.93	3.24
s15850	966.8	0.98	1.75	2.96	4.88
s35932	1124	1.01	1.39	1.90	2.43
s38417	5112	0.97	1.84	3.60	6.49

1000 random vectors

7.4.2 Asynchronous fault redistribution

As a result of targeting a scalable approach, master-slave techniques, for which that master processor or communications network will at some point saturate, were excluded. Instead, an initial constraint was made that all representatives should be equal, that none should perform computation significantly differently or be in any way more authoritative than another. Also, global communication was to be avoided, since all representatives doing global communication could result in large communication requirements.

7.4.2.1 Requesting faults

Each representative fault simulates all vectors for all faults in its current partition. At the completion of the simulation, the representative sends a request to another representative for more faults to be simulated. In the master-slave model, the other representative would be an authoritative representative that would generally be dedicated to serving requests for fault partitions. We adopt the default random representative selection model of aggregates [5] with a constraint: the selected representative cannot be the representative making the request. The call to the other representative has as an argument the representative index of the requester. If faults are available, they are returned via a request to simulate a new set of faults starting from a specified vector (Figure 7.6). The set of faults is created by dividing the set of remaining faults to be simulated by the other representative. After sending a split request, an actor remains idle until new faults are supplied by another representative. Techniques similar to the random representative selection method described here have been used in loop scheduling for FORTRAN programs on shared memory [146] and distributed memory machines [147].

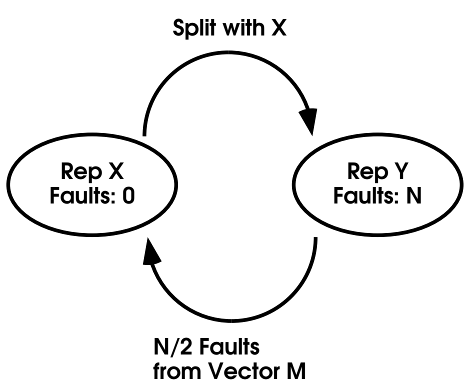

Figure 7.6 Split request in fault redistribution

7.4.2.2 Splitting fault lists

Upon receiving a request to split a fault list, an actor performs a number of tests. First, it determines whether it has any faults remaining to be simulated. If it does not, the actor simply forwards the request to another representative at random, this time excluding both the original requester and itself (Figure 7.7). If the actor has remaining vectors to send, it divides its fault list and returns half to the requesting representative.

7.4.2.3 Rescheduling

Because the actor model does not support preemption, in order to support fault redistribution it is necessary to have representatives reschedule themselves, i.e., send continuations to themselves to continue the current computation. This rescheduling occurs instead of serially processing all vectors. Without rescheduling, an actor that has faults that may be split will never receive a request to split its fault list until it already has no faults to share. Because rescheduling continuations are always sent to the actor itself, they never incur delays due to network latency.

7.4.2.4 Communication characteristics

In the process of fault simulation, an actor will generally either be performing fault simulation in response to a rescheduling request or it will be waiting for more vectors to simulate from another representative. If the actor has faults to simulate, it will periodically send itself a simulate request and return control to the run time system. Thus, for active representatives, there exists one task, always in the local task queue, representing the continuation of the current simulation. For idle representatives, there will be a task somewhere in the machine representing a request for more faults to process. At the beginning of the simulation when the first representative completes, this split message is generally satisfied by the first actor to receive the request. As simulation progresses toward completion, the number of idle processors grows and fewer split messages are satisfied by the first actor to receive the request. While the number of messages in flight in the network grows, the number is bounded by the number of idle processors because an idle processor sends only one request message.

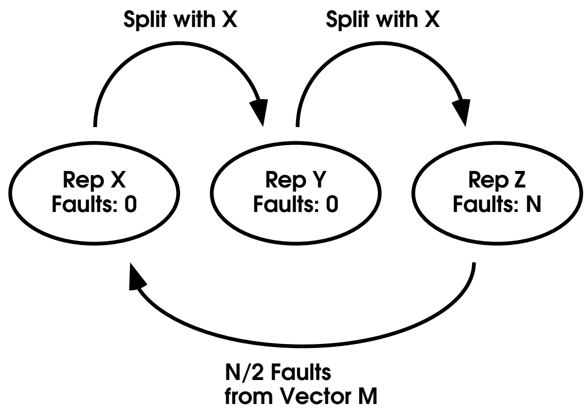

Figure 7.7 Forwarding of split requests

7.4.2.5 Termination detection

One aspect of asynchronous redistribution that causes more difficultly than the master-slave case is the detection of completion of all simulation. In the master-slave case, the master knows immediately when all faults have been simulated. In the distributed case, no representative knows the total state of the fault simulation.

There are two straightforward methods for determining when fault simulation is complete. The first is to centralize the status of the fault simulation progress in an arbitrary representative, for example, representative zero, and to have all other representatives send a status message to that representative every time they complete the simulation of a set of faults for all vectors. The status representative can trivially determine when the operation is done by summing these requests and comparing against the total number of faults in the circuit. While this approach implies a small amount of asymmetry among representatives, it does not suffer from the primary drawback of the master-slave approach which is synchronization, the inability of the slave to perform any computation until a response is received from the master. In the case of status-only data, no reply is expected in response to sending a status message to the coordinating representative; therefore normal processing can continue. Moreover, since if the coordinating representative has any vectors to simulate it is trivially the case that the simulation has not completed, status messages can be given low precedence, thereby not impeding the coordination representative's fault simulation progress.

The second alternative for termination is the use of quiescence detection on outstanding rescheduling requests. When the number of outstanding rescheduling events reaches zero and no representatives are performing fault simulation, the simulation has completed. This technique was implemented and is considered more fully in Subsection 7.7.3.

7.5 ProperPROOFS

The techniques in Section 7.4 are implemented in `ProperPROOFS`, the architecture of which is shown in Figure 7.8. In this section we address implementation issues of the `FaultSimulatorAggregate` class.

7.5.1 Rescheduling and prioritized execution

Rescheduling is implemented by having each representative call a method `Simulate` with no arguments on itself periodically. The most obvious points at which to do rescheduling are after every fault group is executed and after every vector is simulated for all fault groups. To determine which point to choose, serial `PROOFS` was instrumented to measure the time required for each operation. Results for a few sample circuits are shown in Table 7.2. The table shows that the fastest fault groups still take hundreds of microseconds to compute; therefore minimum grain size would not appear to be a problem. Vectors can take

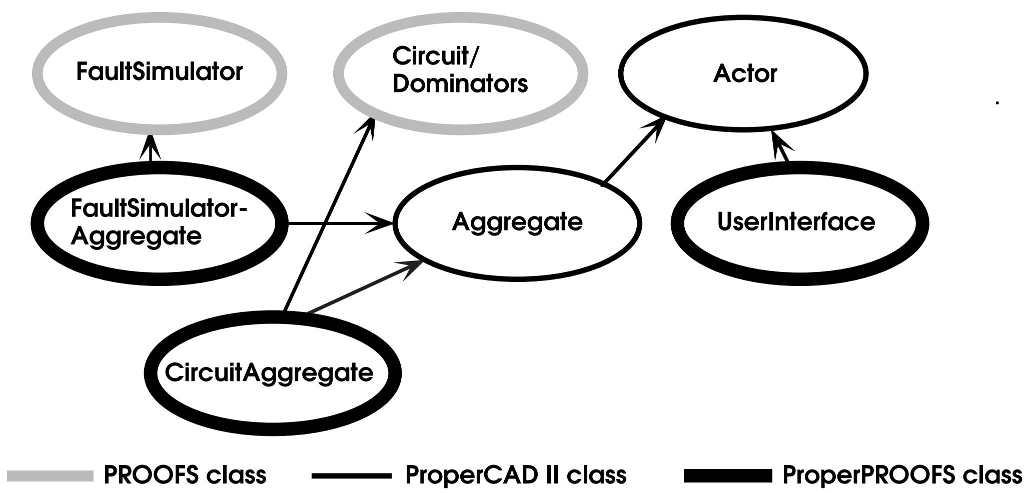

Figure 7.8 `ProperPROOFS` organization

Table 7.2 Time (ms) of fault simulation operations

Circuit	Per Fault Group				Per Vector			
	Mean	Variance	Min	Max	Mean	Variance	Min	Max
s382	3.739	0.958	0.132	18.38	5.457	2.711	3.189	22.90
s5378	8.489	8.297	0.249	180.7	140.3	57.21	92.06	692.9
s9234	3.855	3.366	0.969	39.59	497.6	38.65	457.9	550.0
s35932	9.366	23.10	1.254	389.1	2071.3	1185	518.3	5597

a significant amount of time to process, up to five seconds on large circuits, thus latency of delivery should be considered.

Though small grain size did not appear to be a problem, per-vector rescheduling was implemented. The major reason for this was ease of implementation; the fault-group loop is deeper within `PROOFS`, which requires significant saving of state during rescheduling. In addition, the frequency of fault redistribution is low enough that vector loop latency does not have a significant impact on run time. Prioritized execution is used to ensure that requests to split a fault list have higher precedence than the rescheduling request.

7.5.2 Splitting fault lists

While splitting fault lists is conceptually straightforward, implementation within `PROOFS` is more difficult. When faults are split, in addition to the list of faults, several values are required to restart the simulation on the remote representative:

- the next vector to be simulated.
- the good circuit state. Good circuit state is not saved; therefore it must be transferred in order for the other representative to restart simulation.
- the good circuit events. Information from the previous time period is stored as a list of events until incorporated into the good circuit state.
- the faulty circuit states. The only information required between time steps for the faulty machine are those values of the state elements that differ from the values in the good circuit.

All this data must be gathered and sent to the remote node. While the amount of information appears large, the infrequency of redistribution keeps this from having a significant impact on execution times.

When the number of undetected faults or unsimulated vectors becomes low, the usefulness of splitting faults falls. To restrict communication overhead from growing, a lower bound is placed on the number of vectors and faults that may be split; if either number falls below the bound, splitting does not occur. Moreover, once that bound is reached, the representative performs the rest of the simulation serially without rescheduling. Lacking this serial processing of small sets of faults and vectors, as the number of idle processors increases, so many split requests are received and forwarded that the representative has no time to finish its list of faults.

7.5.3 Termination detection

Termination detection is implemented with quiescence, as mentioned above. For quiescence to occur, the split requests must be hidden from the quiescence detector through the `representsWork()` method described in Subsection 5.2.3. Once this is done, quiescence is detected after the last rescheduling task is processed.

7.6 Performance

Table 7.1 shows the results for static partitioning; Tables 7.3, 7.4, and 7.5 show the results of `ProperPROOFS` with dynamic redistribution on an Intel iPSC/860, an Intel Paragon, and a Sun 4/690MP, respectively. The circuits are drawn from the ISCAS-89 benchmark set [104]. Test vectors were generated randomly.

Comparison of `PROOFS` and `ProperPROOFS` shows that rescheduling and other concurrency issues contribute little overhead to the fault simulation process. Results for the message passing machines show moderate speedup for many of the larger circuits. Limitations on scalability and the lack of speedup for some large circuits are considered in Section 7.7.

The results for the Sun MP show that while speedup is moderate on two and three processors, often little run time improvement is achieved on four processors. Since the message passing architectures achieve significant performance on a number of circuits for which the shared memory version does not, differences must be due to the shared memory message passing mechanism. There are two possibilities for the lack of speedup. The first is the effect of the redistribution messages, which are transmitted much more quickly in shared memory than through a network. Since these requests are continually forwarded when there are no faults to redistribute, they are basically passed at the maximum message passing bandwidth of the interface. The result is an increase in contention for system resources. The second issue is general contention in memory. This contention is difficult to evaluate and is a topic of future interest for the library in general. Alternate fault redistribution schemes that do not drive the library at maximum bandwidth are also a topic of future work.

In addition to random test patterns, ATPG-generated test patterns were considered. Test patterns generated by automatic test pattern generators generally simulate differently than do random test vectors, since ATPG vectors usually result in a much higher number of detections per vector than do random vectors.

Tables 7.6 through 7.8 show the results for `ProperPROOFS` on deterministic test vectors generated by the STG test generator [129]. The results for ATPG vectors are gener-

Table 7.3 `ProperPROOFS` results on Intel iPSC/860: random vectors

Circuit	PROOFS (sec)	ProperPROOFS Processors (speedup)			
		1	2	4	8
s208	91.3	0.99	1.46	2.48	4.09
s208o	33.0	0.95	1.00	1.54	1.90
s298	33.5	0.93	1.01	1.26	1.26
s344	41.3	0.92	0.96	0.99	1.04
s349	41.9	0.92	0.96	0.99	1.03
s382	109.2	0.94	1.41	2.18	2.67
s386	25.4	0.93	0.86	1.29	2.07
s400	121.7	0.95	1.47	2.18	2.71
s420	235.7	0.97	1.64	2.86	4.55
s444	142.4	0.96	1.52	2.41	3.08
s510	377.6	0.96	1.77	3.29	5.75
s526	170.0	0.96	1.58	2.43	3.05
s526n	165.4	0.96	1.57	2.32	2.92
s641	53.3	0.96	0.93	1.34	1.45
s713	60.6	0.96	0.95	1.45	1.60
s820	101.9	0.95	1.17	1.52	1.76
s832	104.4	0.94	1.19	1.52	1.75
s838	642.0	1.01	1.84	3.25	5.08
s953	1023	1.02	1.91	3.41	5.73
s1196	50.5	0.97	1.00	1.30	1.34
s1238	62.0	0.97	1.05	1.33	1.47
s1423	358.8	0.98	1.68	2.70	3.93
s1488	192.0	0.98	1.27	1.46	1.53
s1494	194.9	0.96	1.28	1.48	1.54
s5378	1108	0.99	1.61	2.41	3.26
s9234	4517	1.00	1.94	3.52	4.80
s13207	16588	0.99	1.96	3.82	6.91
s15850	8395	0.98	1.92	3.55	5.61
s35932	1124	1.01	1.68	2.34	2.85
s38417	5112	0.97	1.99	3.87	7.31

s15850–s38417: 1000 random vectors
all others: 10000 random vectors
dynamic redistribution

Table 7.4 `ProperPROOFS` results on Intel Paragon: random vectors

Circuit	`PROOFS` (sec)	`ProperPROOFS` Processors (speedup)						
		1	2	4	8	16	32	64
s208	87.6	0.98	1.52	2.59	4.23	6.27	7.29	8.14
s208o	31.4	0.97	1.14	1.67	2.08	2.97	3.06	3.01
s298	32.5	0.94	1.08	1.30	1.28	1.44	1.42	1.44
s344	38.9	0.97	1.02	1.05	1.11	1.14	1.09	1.16
s349	39.4	0.97	1.01	1.04	1.10	1.14	1.17	1.16
s382	107.0	0.97	1.52	2.31	2.84	3.15	3.43	3.47
s386	25.2	0.94	0.99	1.44	2.37	2.80	3.03	3.02
s400	118.1	0.97	1.53	2.29	2.84	3.15	3.46	3.55
s420	230.7	0.97	1.72	2.97	4.71	7.22	9.70	10.6
s444	138.3	0.98	1.58	2.47	3.13	3.69	3.79	4.19
s510	366.2	0.98	1.77	3.34	5.59	7.85	11.4	15.1
s526n	162.0	1.00	1.67	2.43	3.11	3.51	3.82	4.12
s526	166.7	1.00	1.65	2.52	3.16	3.69	4.14	4.33
s641	50.4	0.95	0.95	1.33	1.41	1.42	1.43	1.44
s713	58.0	0.96	0.99	1.44	1.59	1.60	1.58	1.58
s820	101.3	0.98	1.27	1.65	1.90	1.95	2.00	1.95
s832	104.1	0.98	1.30	1.65	1.92	1.98	2.04	2.02
s838	619.2	0.99	1.81	3.22	5.13	7.47	9.99	11.7
s953	959.4	0.99	1.87	3.39	5.69	7.98	11.7	13.7
s1196	48.8	0.95	1.04	1.30	1.33	1.43	1.61	1.59
s1238	59.8	0.95	1.08	1.35	1.50	1.67	1.76	1.89
s1423	356.6	0.99	1.71	2.74	3.89	5.05	6.02	6.37
s1488	190.9	0.97	1.29	1.49	1.55	1.58	1.62	1.65
s1494	194.0	0.98	1.31	1.51	1.58	1.61	1.65	1.70
s5378	1094	1.01	1.63	2.37	3.17	3.35	3.60	3.53
s9234	4716	1.00	1.92	3.48	5.00	7.47	9.47	12.1
s13207	1685	0.98	1.92	3.68	6.46	9.18	13.1	16.0
s15850	1034	0.98	1.87	3.48	5.59	8.22	11.0	13.6
s35932	1105	0.99	1.65	2.27	2.73	3.04	3.21	3.21
s38417	5401	0.99	2.13	4.15	7.82	13.3	21.0	27.7
s38584	3370	0.97	1.89	3.50	5.80	7.53	12.5	15.3

s13207–s38584: 1000 random vectors
all others: 10000 random vectors
dynamic redistribution

Table 7.5 `ProperPROOFS` results on Sun 4/670MP: random vectors

Circuit	`PROOFS` (sec)	`ProperPROOFS` Processors (speedup)			
		1	2	3	4
s208	83.3	0.97	1.50	2.11	2.28
s208o	31.5	1.03	1.16	1.46	1.66
s298	32.3	0.99	1.09	1.12	1.12
s344	35.5	0.94	1.03	1.05	0.92
s349	36.4	0.94	1.04	1.07	0.99
s382	101.7	1.00	1.41	1.74	1.86
s386	24.1	0.89	0.99	1.14	1.29
s400	106.7	1.04	1.59	1.85	1.80
s420	246.6	1.12	1.76	2.53	2.82
s444	122.9	0.96	1.64	1.83	1.99
s510	324.8	1.06	1.82	2.14	2.57
s526n	161.7	1.13	1.88	1.91	2.24
s526	166.9	1.10	1.87	2.01	2.38
s641	41.4	0.93	0.88	0.91	0.97
s713	46.9	0.82	0.79	1.00	1.05
s820	92.7	0.99	1.24	1.29	1.43
s832	95.1	0.99	1.24	1.43	1.30
s838	571.7	1.01	1.95	2.23	2.27
s953	790.5	1.08	1.86	2.07	2.40
s1196	45.9	0.95	1.02	1.27	1.09
s1238	53.4	0.93	1.04	1.17	1.17
s1423	308.0	0.99	1.43	1.68	1.97
s1488	164.9	0.93	1.25	1.40	1.19
s1494	166.2	0.91	1.26	1.35	1.31
s5378	978.7	0.97	1.41	1.75	1.40
s9234	4353	0.97	1.53	1.81	1.77
s13207	17973	1.01	1.79	2.08	2.18
s15850	8684	0.97	1.62	1.95	1.91
s35932	1068	0.91	1.46	1.48	1.62
s38417	8270	0.97	1.41	1.53	1.48
s38584	3519	0.95	1.47	1.77	1.76

s15850–s38584: 1000 random vectors
all others: 10000 random vectors
dynamic redistribution

Table 7.6 `ProperPROOFS` results on Intel iPSC/860: STG vectors dynamic redistribution

Circuit	`PROOFS` (sec)	`ProperPROOFS` Processors (speedup)			
		1	2	4	8
s208	1.1	0.99	1.33	1.95	3.09
s298	0.7	0.93	0.85	1.29	1.25
s344	0.5	0.91	1.00	1.11	1.16
s349	0.5	0.91	0.98	1.09	1.16
s382	13.2	0.94	1.12	1.30	1.51
s400	10.4	0.94	1.17	1.57	1.88
s420	4.7	0.97	1.64	2.62	3.80
s444	15.4	0.94	1.20	1.48	1.81
s526	9.2	0.95	1.30	1.67	2.03
s526n	6.5	0.94	1.15	1.61	1.70
s641	0.9	0.97	0.99	1.44	1.49
s713	1.0	0.98	1.03	1.65	1.78
s820	3.8	0.94	1.00	1.22	1.37
s832	3.6	0.93	1.00	1.24	1.38
s838	10.9	1.01	1.83	3.13	4.87
s953	1.6	1.01	1.72	2.66	3.36
s1196	3.0	0.99	1.25	1.77	2.16
s1238	3.8	0.98	1.24	1.75	2.13
s1423	3.1	1.00	1.75	2.80	3.34
s1488	11.0	0.99	1.03	1.19	1.25
s1494	9.2	0.98	1.06	1.20	1.29
s5378	47.5	0.99	1.56	2.33	3.23
s9234	1.5	1.01	1.50	2.37	3.50
s35932	154.1	1.00	1.69	2.54	3.43

ally inferior to those of random vectors, which is expected; the amount of fault simulation required is less since faults are dropped quickly. Additionally, where the added level of detection occurs unevenly, more fault redistribution is required, which increases the amount of good circuit simulation required.

Table 7.7 `ProperPROOFS` results on Intel Paragon: STG vectors

Circuit	`PROOFS` (sec)	`ProperPROOFS` Processors (speedup)						
		1	2	4	8	16	32	64
s208	1.1	0.99	1.14	2.10	2.73	4.84	1.81	1.06
s298	0.7	0.95	0.81	1.32	1.35	1.30	1.22	0.55
s344	0.5	0.95	0.85	1.19	1.32	1.27	1.08	0.62
s349	0.5	0.95	0.84	1.20	1.28	1.29	1.13	0.63
s382	12.8	0.96	1.17	1.33	1.55	1.86	1.84	1.71
s400	10.0	0.96	1.22	1.59	1.90	2.41	2.43	2.14
s420	4.6	0.98	1.66	2.60	2.53	5.01	6.37	4.61
s444	14.9	0.97	1.25	1.52	1.89	2.26	2.23	2.18
s526	9.0	0.99	1.33	1.76	2.12	2.75	2.83	2.87
s526n	6.3	0.73	1.24	1.70	1.83	2.24	2.38	2.19
s641	1.3	1.37	1.34	1.87	1.88	1.79	1.52	1.12
s713	0.9	0.97	0.81	1.51	1.63	1.64	1.53	1.40
s820	3.8	0.97	1.09	1.31	1.49	1.52	1.53	1.24
s832	3.6	0.97	1.08	1.32	1.53	1.59	1.61	1.53
s838	10.5	0.98	1.78	3.08	3.55	5.07	5.74	10.4
s953	1.5	1.00	1.59	2.39	1.82	2.01	1.06	2.23
s1196	2.9	0.97	1.23	1.77	1.57	1.77	1.74	2.06
s1238	3.7	0.97	1.28	1.79	2.21	2.53	2.68	2.40
s1423	3.0	1.00	1.69	2.75	3.62	3.56	1.45	2.83
s1488	10.9	0.98	1.06	1.20	1.25	1.27	1.29	1.31
s1494	9.2	0.98	1.08	1.24	1.30	1.35	1.34	1.38
s5378	47.0	1.01	1.57	2.25	3.10	3.19	3.32	3.49
s9234	1.5	1.03	1.49	2.32	3.54	4.97	5.70	2.88
s35932	153.2	1.00	1.68	2.41	3.21	3.69	3.80	4.35

dynamic redistribution

7.7 Evaluation

7.7.1 Scalability

Effort was taken to ensure that the fault redistribution scheme in `ProperPROOFS` would not limit scalability because of the serialization of a single actor or the resource limitations and processing power of a single processor. While this effort was successful, the most significant barrier to scalability in `ProperPROOFS` is the revaluation of good circuit values on

Table 7.8 `ProperPROOFS` results on Sun 4/670MP: STG vectors

Circuit	`PROOFS` (sec)	`ProperPROOFS` Processors (speedup)			
		1	2	3	4
s208	1.0	0.97	1.48	2.04	2.29
s298	0.7	1.02	1.04	1.18	1.15
s344	0.5	0.90	1.03	1.20	1.01
s349	0.5	0.94	1.07	1.04	0.95
s382	12.4	0.95	1.14	1.04	1.34
s400	9.1	0.99	1.23	1.47	1.50
s420	4.9	1.13	1.75	2.60	2.64
s444	13.8	0.91	1.26	1.24	1.38
s526	9.1	1.07	1.49	1.55	1.73
s526n	6.5	1.12	1.43	1.53	1.56
s641	0.8	0.96	0.97	0.97	0.88
s713	0.8	0.84	0.87	1.02	1.11
s820	3.5	0.95	1.07	1.18	1.12
s832	3.3	0.95	1.05	1.11	1.17
s838	9.8	1.02	1.88	2.08	2.59
s953	1.2	1.10	1.70	1.66	1.64
s1196	2.7	0.95	1.19	1.42	1.34
s1238	3.2	0.94	1.18	1.34	1.33
s1423	2.6	0.99	1.54	1.85	1.73
s1488	9.5	0.94	1.03	1.13	1.06
s1494	8.0	0.92	1.09	1.16	1.01
s5378	42.5	0.98	1.38	1.51	1.45
s9234	1.5	0.92	1.16	1.20	1.10
s35932	149.2	0.92	1.40	1.62	1.58

dynamic redistribution

every processor. Because of this extra evaluation—not needed in the serial case—overhead grows linearly with the number of processors. This overhead is compounded in `Proper-PROOFS` by the fact that fault redistribution implies restarting the good circuit simulation; the number of good circuit simulations is not bounded by the number of processors.

To investigate issues of scalability, we studied in detail the run time dynamics of s35932 on the Paragon (Table 7.4). While large, this circuit demonstrated only a small amount of speedup over all configurations. `ProperPROOFS` was instrumented to record the amount of time and number of events required by good and faulty circuit simulation. Results are shown in Table 7.9. A study of the number of events evaluated in each case shows that the the extra effort expended in the parallel version is virtually all due to multiple good circuit evaluations which do not occur in the serial algorithm. Not only must good circuit simulation be performed per processor, it must also be duplicated in part when a set of faults is redistributed. While it was anticipated that communication due to fault redistribution scheme might be a bottleneck, in practice fault redistribution is infrequent and the redistribution process itself contributes little overhead. Virtually all of the overhead on large circuits is incurred by the the extra good circuit simulation on all processors and by the necessity to duplicate good simulation for all unsimulated vectors when a set of faults is received in response to a split request.

The good circuit resimulation overhead caused by fault redistribution effectively limits the advantage of dynamic redistribution over that of static distribution. Table 7.10 compares the results of static and dynamic fault distribution. The data show that while the dynamic redistribution scheme achieves either equivalent or better results than the static case, the incremental speedup is lower than if fault redistribution incurred no overhead.

We have attempted to develop a scheme for mitigating the extra costs implied by fault parallelism. For example, the extra simulations caused by load redistribution can be eliminated by saving the state of the good circuit the first time it is computed on each processor and then by using this saved state in later passes that occur as a result of fault redistribution. Unfortunately, this approach exhibits two flaws. First, it does not address the issue of the first good simulation on each processor which is a significant amount of the duplicated effort on large circuits. Second, the cost of saving the good circuit state for large circuits

Table 7.9 Good and faulty simulation of s35932 on Paragon

Processors	Good		Faulty	
	Millions of Events	Time (sec)	Millions of Events	Time (sec)
1	8.04	77.4	34.6	524.2
2	30.1	290.1	34.7	519.3
4	76.5	738.3	34.8	522.9
8	137	1324	35.1	523.4
16	227	2182	35.2	538.9
32	363	3496	35.3	552.3
64	557	5393	35.5	601.7

1000 random vectors
dynamic redistribution

Table 7.10 Comparison of static and dynamic fault distribution on iPSC/860

Circuit	PROOFS (sec)	Dynamic Distribution Processors (speedup)				Static Distribution Processors (speedup)			
		1	2	4	8	1	2	4	8
s208	9.1	0.99	1.42	2.38	3.86	0.99	1.51	2.60	4.48
s208o	3.6	0.95	1.00	1.57	1.97	0.95	1.40	1.79	1.98
s298	4.5	0.94	1.02	1.51	1.59	0.94	1.21	1.60	1.62
s344	4.4	0.89	0.99	1.05	1.10	0.89	1.01	1.05	1.10
s349	4.5	0.89	0.98	1.04	1.10	0.89	1.01	1.04	1.09
s382	10.9	0.94	1.37	2.11	2.54	0.94	1.29	1.92	2.38
s386	3.0	0.93	0.89	1.30	2.12	0.93	1.38	1.68	2.19
s400	12.2	0.95	1.42	2.12	2.67	0.95	1.36	1.91	2.45
s420	23.5	0.97	1.62	2.66	4.38	0.97	1.49	2.56	4.43
s444	14.5	0.96	1.48	2.32	2.95	0.96	1.34	2.02	2.84
s510	37.8	0.96	1.76	3.24	5.36	0.96	1.62	2.55	4.41
s526	17.4	0.96	1.56	2.39	2.96	0.96	1.45	1.87	2.52
s526n	16.8	0.96	1.53	2.31	2.89	0.96	1.41	1.88	2.54
s641	5.9	0.96	0.94	1.38	1.57	0.96	1.12	1.50	1.57
s713	6.7	0.97	0.99	1.51	1.73	0.97	1.19	1.58	1.74
s820	11.5	0.96	1.19	1.55	1.78	0.96	1.20	1.54	1.83
s832	11.7	0.95	1.20	1.55	1.83	0.95	1.22	1.55	1.88
s838	63.2	1.01	1.82	3.12	4.84	1.01	1.50	2.55	4.37
s953	101.9	1.03	1.90	3.35	5.64	1.03	1.41	2.66	3.96
s1196	9.9	0.99	1.20	1.72	2.08	0.99	1.41	1.90	2.28
s1238	11.4	0.99	1.27	1.78	2.27	0.99	1.50	2.00	2.51
s1423	46.1	0.98	1.72	2.90	4.37	0.98	1.64	2.70	4.14
s1488	21.5	0.98	1.30	1.51	1.63	0.98	1.26	1.50	1.65
s1494	21.8	0.96	1.31	1.56	1.68	0.96	1.27	1.52	1.67
s5378	150.3	0.99	1.73	2.75	4.30	0.99	1.34	2.19	3.74
s9234	458.8	1.00	1.93	3.50	4.79	1.00	1.60	2.48	2.90
s13207	1598	0.99	1.97	3.81	6.82	0.99	1.38	1.93	3.24
s15850	966.8	0.98	1.92	3.58	5.86	0.98	1.75	2.96	4.88
s35932	1124	1.01	1.68	2.34	2.85	1.01	1.39	1.90	2.43
s38417	5112	0.97	1.99	3.87	7.31	0.97	1.84	3.60	6.49

1000 random vectors

and for large numbers of vectors is prohibitive. Other techniques of speeding a very fast event-driven fault simulator like PROOFS are topics for future study.

7.7.2 Rescheduling

The method of rescheduling worked well for balancing load and required essentially no tuning. Because the operation of an actor sending a continuation to itself is always a local operation, rescheduling can be used without regard to the network communication latency of multicomputers. The same type of rescheduling could be used in ProperHITEC to increase the currency of fault data and would improve performance results. However, in the HITEC implementation, the point at which rescheduling would occur—when a backtrack is required—is relatively far down the call stack; therefore, extra effort is required to enable exit and return to the same point within the code during rescheduling. In the case of ProperPROOFS, because the per-vector loop in PROOFS is at a high level no additional state has to be saved.

7.7.3 Termination detection

As mentioned in Chapter 5, quiescence detection is not a composable property; therefore, any use of quiescence in the logic of an algorithm should be examined for limitations. The quiescence-based termination detection algorithm was developed for ProperPROOFS to provide experience and knowledge about the usefulness of the termination interface in the AIF and, in particular, the usefulness of user-level 'hidden' tasks as a meta-programmability feature. Although using quiescence to complete fault simulation is at a high level a simple problem, its practical implementation exposes various subtle issues, each of which must be identified—which sometimes requires significant debugging—and handled. In the case of ProperPROOFS, the utility of termination via quiescence is doubtful. The direct approach described in Subsection 7.4.2.5 is simpler to implement, composable, and potentially more efficient. This experience exemplifies the position stated in Chapter 5 that the complexity and noncomposability of quiescence detection mitigates its utility.

7.7.4 Applicability

The approach taken in this chapter has been the generation of a technique for fault simulation applications; no attempt was made to specifically address the use of the fault simulator within a test generation application, because both the static interface and the dynamic context differ sufficiently to make some of the techniques applicable to an application inapplicable in a module to be composed with a test generator. The serial object is still used within ProperHITEC. Still, though generally the use of a per-thread aggregate would be considered excessive in a test application, it is clear that as test generation is scaled to larger machines sizes, the use of a serial object must at some point become a bottleneck; signs of this limitation were observed in running ProperHITEC. An intermediate approach—between the alternatives of serial and per-thread—would be to use an aggregate whose size was a function of the number of threads in the system. For example, one could use a distribution in which the number of representatives was a fraction of the number of nodes in the machine. This use represents a direct application of the multiaccess principle proposed by Chien [5]. Because the fault simulator is already specified as an aggregate, extension to support other than per-thread distributions would be minor and completely backwardly compatible with the fault simulator application.

7.7.5 Parallelization Effort

As was the case for ProperHITEC, ProperPROOFS was created through the use of derivation to express parallelism. Again, the only fundamental change was the modification of several functions in the serial code to use dynamic binding in order that those functions could be overridden in derived classes. In the case of ProperPROOFS, only a single class, FaultSimulator, was modified in this way. Thirteen of the forty-four member functions where made virtual. Most of these modifications were in the area of gathering the fault simulation results, which at the end of the simulation is distributed among aggregate representatives.

ProperPROOFS was developed later than ProperHITEC, when the library was more stable, which made measurement of the parallelization process easier. Again, when the time necessary to clean up the original code is not included, the parallelization effort took approximately one man-month, with additional improvements and debugging requiring approximately one-half man-month.

Implementation of parallelism of PROOFS required fewer changes than did parallelization of HITEC. Table 7.11 reports several metrics of the PROOFS and ProperPROOFS implementations. As with ProperHITEC, virtually all the significant classes are shared between the parallel and serial versions. Among the four new classes, the CircuitAggregate class is the same as that used in ProperHITEC. A breakdown of the number of actor methods by class is shown in Table 7.12. Table 7.13 presents the numbers of member functions

Table 7.11 Comparison of software metrics for PROOFS and ProperPROOFS

Metric	PROOFS	ProperPROOFS	Shared
Lines of code	455	3938	5659
Lines of code with ';'	140	733	1981
Classes	0	4	12
excluding ActorMethod and argument classes			
ActorMethod classes	0	17	0
ActorMethod argument classes	0	5	0

Table 7.12 ActorMethods for each class in ProperPROOFS

ProperPROOFS class	Number of ActorMethods
FaultSimulatorAggregate	7
CircuitAggregate	4
UserInterface	6

in the concurrent types and the serial types from which they are derived. These data again show that the majority of features in ProperPROOFS are inherited from PROOFS.

Table 7.13 Member functions in PROOFS and ProperPROOFS

PROOFS class	Number of member functions	ProperPROOFS class	Number of member functions
FaultSimulator	44	FaultSimulator-Aggregate	16
Circuit	47	CircuitAggregate	6

Chapter 8

CONCLUSIONS

We have presented an interface and run time library implementation for concurrent object-oriented programming suitable for a statically typed language which implements the actor model of concurrent computation. The interface contains a rich set of interfaces for expressing actor computations with emphasis on support for composable meta-programmability. We demonstrated how a library-based implementation can be used to incrementally parallelize an existing serial code with only incremental increases in development cost.

We have demonstrated the use and efficacy of the library in the parallelization of two significant CAD applications, `ProperHITEC` for test generation and `ProperPROOFS` for fault simulation. We presented results for these applications when they were run on shared memory, distributed memory, and hybrid architectures. We conclude with some observations on the effectiveness and efficiency of the interface and implementation.

The features of the library that have proven to be the most effective in expressing parallelism are first class names, first class statically typed continuations, and derivation-based parallelization. The ability to create, operate on, and interchange actor and aggregate names has turned out to be very effective in implementing an interface that has a high degree of expressibility but that does not make code unacceptably interdependent, thus restricting reuse. We believe that the expressibility of names linked with statically typed continuations will be key to implementing reusable application libraries that can be easily composed to create new applications.

Our experiences using derivation to incrementally parallelize existing serial applications lead us to believe that this feature will be a key to parallelizing existing codes written in object-oriented languages without doubling development and support costs. For many medium-grain applications, the added cost of dynamic binding will not be measurable, and the impact on the expressibility and readability of the serial code will be nominal or positive.

The ability to compose meta-programmability features—to use different representations in modules combined to create a single application—has been useful in the applications addressed so far and has significant potential in the development of modular solutions to common problems in VLSI CAD and other domains. Although global meta-programmability has not yet been used in CAD applications, several ways of capitalizing on the ability to refine the underlying run time system have generated significant speculation.

A significant reduction in porting difficulty has been achieved at the same time providing a richer interface which facilitates architectures tuning. This reduction is primarily due to the implementation of the high-level interface in an open manner and via a well defined, parameterizable, low-level interface.

In our work using the library, we identified a number of cases that either required tedious and possibly error-prone coding techniques or for which expressibility of the current interface is not sufficient. The two current limitations in the current implementation, the inability to derive from the `Actor` class virtually and the inability to cast actor names in a manner similar to casting pointers have been solved by recent work by the C++ standardization committee. However, the necessity to use macros to implement the library does not appear to have a solution short of a language extension or a new language. Experience to date indicates that macros, while inelegant, become acceptable with familiarity.

The basic actor model provides no way of directly expressing an actor method call that blocks or returns a value. Although continuation passing style is sufficient to express anything requiring a blocking value and in general expresses greater parallelism, there are cases in which data dependencies preclude parallelism; the use of CPS in those cases can be tedious. Although implementation can be difficult and semantics may be difficult to specify, experience indicates that blocking calls are of sufficiently great utility to justify significant effort in their design and implementation.

The actor interface provides no direct support for I/O operations, and the APA provides direct support only for the standard streams, `stdout` and `stderr`. A richer interface with a higher-level, i.e., AIF-level, interface would be valuable.

Although the APA provides a model of the logical interconnection of processors, it does not capture the relative power or dynamic load of individual processors. For efficient use of workstation clusters, some method of determining static processing power and dynamic load will be required. These methods would also simplify the composition of modules. Representation of load and power is a significant abstraction problem.

REFERENCES

[1] B. Ramkumar and P. Banerjee, “ProperCAD: A portable object-oriented parallel environment for VLSI CAD,” *IEEE Transactions on Computer Aided Design*, vol. 13, pp. 829–842, July 1994.

[2] K. De, B. Ramkumar, and P. Banerjee, “ProperSYN: A portable parallel algorithm for logic synthesis,” in *Digest of Papers, International Conference on Computer-Aided Design*, pp. 412–416, Nov. 1992.

[3] B. Ramkumar and P. Banerjee, “ProperEXT: A portable parallel algorithm for VLSI circuit extraction,” in *Proceedings, 7th International Parallel Processing Symposium*, pp. 434–438, 1993.

[4] S. Kim, “Improved algorithms for cell placement and their parallel implementations,” Ph.D. dissertation, University of Illinois, Urbana, IL, July 1993.

[5] A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. Cambridge, MA: The MIT Press, 1993.

[6] G. Agha, “Concurrent object-oriented programming,” *Communications of the ACM*, pp. 125–141, Sept. 1990.

[7] Kendall Square Research, *KSR-1 Technical Summary*, Waltham, MA, 1992.

[8] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. L. Hennessy, M. Horowitz, and M. Lam, “The Stanford DASH,” *IEEE Computer*, vol. 25, pp. 63–79, Mar. 1992.

[9] Intel Supercomputing Systems Division, *Paragon XP/S Product Overview*, Beaverton, OR, 1991.

[10] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Transactions on Computers*, pp. 241–248, Sept. 1979.

[11] Thinking Machine Corporation, *The Connection Machine CM-5 Technical Summary*, Cambridge, MA, 1991.

[12] P. Pierce and G. Regnier, “The Paragon™ implementation of the NX message passing interface,” in *Proceedings of the Scalable High-Performance Computing Conference*, pp. 184–190, May 1994.

[13] Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, 1994. Available as `http://www.mcs.anl.gov/mpi/mpi-report.ps`.

[14] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam, "PVM and HeNCE: Tools for heterogeneous network computing," in *Environments and Tools for Parallel Scientific Computing*, vol. 6 of *Advances in Parallel Computing*, Amsterdam: North-Holland, 1993, pp. 139–153.

[15] Parasoft Corporation, *Express Reference Guide for FORTRAN Programmers*, Pasadena, CA, 1992.

[16] A. W. Appel, *Compiling with Continuations*. Cambridge, England: Cambridge University Press, 1992.

[17] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 256–266, May 1992.

[18] C. M. Pancake and D. Bergmark, "Do parallel languages respond to the needs of scientific programmers?," *IEEE Computer*, vol. 23, pp. 13–23, Dec. 1990.

[19] C. M. Pancake and C. Cook, "What users need in parallel tool support: Survey results and analysis," in *Proceedings of the Scalable High-Performance Computing Conference*, pp. 40–47, May 1994.

[20] J. R. Corbin, *The Art of Distributed Applications*, (Sun Technical Reference Library). New York: Springer-Verlag, 1991.

[21] D. Gannon and J. K. Lee, "Object-oriented parallelism: pC++ ideas and experiments," in *Proceedings of the Japan Society for Parallel Processing*, pp. 13–23, 1993.

[22] J. K. Lee and D. Gannon, "Object oriented parallel programming experiments and results," in *Proceedings, Supercomputing '91*, pp. 273–282, Nov. 1991.

[23] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr, "Implementing a parallel C++ runtime system for scalable parallel systems," in *Proceedings, Supercomputing '93*, pp. 588–597, Nov. 1993.

[24] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, , and F. Bodin, "Performance analysis of pC++: A portable data-parallel programming system for scalable parallel computers," in *Proceedings, 8th International Parallel Processing Symposium*, pp. 75–84, Apr. 1994.

[25] K. M. Chandy and C. Kesselman, "Compositional C++: Compositional parallel programming," in *Proceedings of the 5th Workshop on Compilers and Languages for Parallel Computing*, pp. 79–93, 1992.

[26] Concurrent Systems Architecture Group, "Illinois Concert C++ (IC-C++) language report 1.0," Department of Computer Science, University of Illinois, Urbana, IL, Tech. Rep. In preparation.

[27] A. A. Chien, V. Karamcheti, and J. Plevyak, "The Concert system — compiler and runtime support for efficient, fine-grained concurrent object-oriented programs," Department of Computer Science, University of Illinois, Urbana, IL, Tech. Rep. UIUCDCS-R-93-1815, June 1993.

[28] W. J. Leddy and K. S. Smith, "The design of the experimental systems kernel," in *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers and Applications*, pp. 10–17, Mar. 1989.

[29] B. Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley Publishing Company, 2nd ed., 1991.

[30] H. Baker and C. Hewitt, "The incremental garbage collection of objects," in *Conference Record of the Conference on AI and Programming Languages*, pp. 55–59, Aug. 1977.

[31] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield, "The Amber system: Parallel programming on a network of multiprocessors," Department of Computer Science and Engineering, University of Washington, Seattle, WA, Tech. Rep. 89-04-01, Sept. 1989.

[32] B. N. Bershad, E. D. Lazowska, and H. M. Levy, "PRESTO: A system for object-oriented parallel programming," *Software—Practice and Experience*, vol. 18, pp. 713–731, Aug. 1988.

[33] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, pp. 39–59, Feb. 1984.

[34] R. Chandra, A. Gupta, and J. L. Hennessy, "Integrating concurrency and data abstraction in a parallel programming language," Computer Science Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA, Tech. Rep. CSL-TR-92-511, Feb. 1992.

[35] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, pp. 444–458, Apr. 1989.

[36] N. Carriero and D. Gelernter, "How to write parallel programs: A guide to the perplexed," *ACM Computing Surveys*, vol. 21, pp. 323–357, Sept. 1989.

[37] G. Agha and C. Callsen, "ActorSpaces: A model for scalable heterogeneous computing," in *Proceedings of the 26th Hawaii International Conference on System Sciences*, 1993.

[38] G. Agha and C. J. Callsen, "ActorSpaces: An open distributed programming paradigm," in *Proceedings of the 4th ACM Symposium on Principles & Practice of Parallel Processing*, pp. 23–32, May 1993.

[39] L. D. Cagan and A. H. Sherman, "Linda unites network systems," *IEEE Spectrum*, vol. 30, pp. 31–35, Dec. 1993.

[40] A. Deshpande and M. Schultz, "Efficient parallel programming in Linda," in *Proceedings, Supercomputing '92*, pp. 238–244, Nov. 1992.

[41] B. R. Seyfarth, J. L. Bickham, and M. R. Fernandez, "Glenda: An environment for easy parallel programming," in *Proceedings of the Scalable High-Performance Computing Conference*, pp. 637–641, May 1994.

[42] N. H. Gehani and W. D. Roome, "Concurrent C++: Concurrent programming with class(es)," *Software—Practice and Experience*, pp. 1157–1177, Dec. 1988.

[43] N. H. Gehani and W. D. Roome, "Implementing concurrent C," *Software—Practice and Experience*, pp. 266–285, Mar. 1992.

[44] M. C. Rinard, D. J. Scales, and M. S. Lam, "Heterogeneous parallel programming in Jade," in *Proceedings, Supercomputing '92*, pp. 245–256, Nov. 1992.

[45] A. S. Grimshaw, "An introduction to parallel object-oriented programming with Mentat," Department of Computer Science, University of Virginia, Charlottesville, VA, Tech. Rep. TR-91-07, 1991.

[46] The Mentat Research Group, Department of Computer Science, University of Virginia, *Mentat 2.5 Programming Language Reference Manual*, Charlottesville, VA, 1993.

[47] P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, and B. M. Younger, "μC++: Concurrency in the object-oriented language C++," *Software—Practice and Experience*, vol. 22, pp. 137–172, Feb. 1992.

[48] C. M. Chase, A. L. Cheung, A. P. Reeves, and M. R. Smith, "Paragon: A parallel programming environment for scientific applications using communications structures," in *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 211–218, 1991.

[49] C. Baquero and F. Moura, "Concurrency annotations in C++," *SIGPLAN Notices*, vol. 29, pp. 61–67, July 1994.

[50] P. America, "POOL-T: A parallel object-oriented language," in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds., Cambridge, MA: The MIT Press, 1987, pp. 199–220.

[51] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: The MIT Press, 1986.

[52] T. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," in *Proceedings of the European Design Automation Conference*, pp. 214–218, Feb. 1991.

[53] G. Agha, "Semantic considerations in the actor paradigm of concurrent computation," *Library Notes of Computer Science*, vol. 197, pp. 151–179, July 1984.

[54] B. Stroustrup, *The Design and Evolution of C++*. Reading, MA: Addison-Wesley Publishing Company, 1994.

[55] Pure Software Inc., *Quantify User's Guide*, Sunnyvale, CA, 1993.

[56] M. A. Ellis and B. Stroustrup, *The Annotated* C++ *Reference Manual*. Reading, MA: Addison-Wesley Publishing Company, 1990.

[57] A. Goldberg and D. Robson, *SMALLTALK-80: The Language and its Implementation*. Reading, MA: Addison-Wesley Publishing Company, 1983.

[58] G. Kiczales, J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. Cambridge, MA: The MIT Press, 1991.

[59] D. R. Edelson, "Smart pointers: They're smart, but they're not pointers," in *USENIX C++ Technical Conference Proceedings*, pp. 1–20, Aug. 1992.

[60] S. Frølund and G. Agha, "A language framework for multi-object coordination," in *Proceedings of the 1993 European Conference on Object-Oriented Programming*, 1993.

[61] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman, "Abstraction and modularity mechanisms for concurrent computing," *IEEE Parallel and Distributed Technology*, pp. 3–13, May 1993.

[62] G. Agha, "An overview of actor languages," *SIGPLAN Notices*, vol. 21, pp. 58–67, Oct. 1986.

[63] P. de Jong, "Compilation into actors," *SIGPLAN Notices*, vol. 21, pp. 68–77, Oct. 1986.

[64] H. Lieberman, "Concurrent object-oriented programming in Act1," in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds., Cambridge, MA: The MIT Press, 1987, pp. 9–36.

[65] D. Theriault, "Issues in the design and implementation of Act2," MIT Artificial Intelligence Laboratory, Cambridge, MA, Tech. Rep. 728, June 1983.

[66] E. Shibayama and A. Yonezawa, "Distributed computing in ABCL/1," in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds., Cambridge, MA: The MIT Press, 1987, pp. 129–158.

[67] J. Ferber and P. Carle, "Actors and agents as reflective concurrent objects: A MERING IV perspective," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, pp. 1420–1436, Nov. 1991.

[68] C. Houck and G. Agha, "HAL: A high-level actor language and its distributed implementation," in *Proceedings of the International Conference on Parallel Processing*, pp. 158–165, Aug. 1992.

[69] W. Fenton, B. Ramkumar, V. A. Saletore, A. B. Sinha, and L. V. Kalé, "Supporting machine independent programming on diverse parallel architectures," in *Proceedings of the International Conference on Parallel Processing*, Aug. 1991.

[70] A. A. Chien and W. J. Dally, "Concurrent aggregates (CA)," in *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Processing*, pp. 187–196, Mar. 1990.

[71] V. Karamcheti and A. Chien, "Concert — efficient runtime support for concurrent object-oriented programming languages on stock hardware," in *Proceedings, Supercomputing '93*, pp. 33–36, 1993.

[72] L. V. Kalé, "The Chare Kernel parallel programming language and system," in *Proceedings of the International Conference on Parallel Processing*, vol. II, Aug. 1990.

[73] L. V. Kalé and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 91–108, Sept. 1993.

[74] L. V. Kalé, B. Ramkumar, V. Saletore, and A. Sinha, "Prioritization in parallel symbolic computing," *Library Notes of Computer Science*, 1993.

[75] L. V. Kalé and A. B. Sinha, "Information sharing mechanisms in parallel programs," in *Proceedings, 8th International Parallel Processing Symposium*, pp. 461–468, Apr. 1994.

[76] A. Gürsoy and L. V. Kalé, "Dagger: Combining benefits of synchronous and asynchronous communication styles," in *Proceedings, 8th International Parallel Processing Symposium*, pp. 590–596, Apr. 1994.

[77] A. Gürsoy and L. V. Kalé, "High level support for divide-and-conquer parallelism," in *Proceedings, Supercomputing '91*, pp. 283–292, Nov. 1991.

[78] D. Kafura and K. H. Lee, "ACT++: Building a concurrent C++ with actors," *Journal of Object-Oriented Programming*, pp. 25–37, May 1990.

[79] D. G. Kafura and K. H. Lee, "Inheritance in actor based concurrent object-oriented languages," *The Computer Journal*, vol. 32, no. 4, pp. 297–304, 1989.

[80] J. Desbiens, M. Lavoie, S. Pouzyreff, P. Raymond, T. Tamazouzt, and M. Toulouse, "CLAP: An object-oriented programming system for distributed memory parallel machines," *OOPS Messenger*, vol. 5, pp. 44–48, Jan. 1994.

[81] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso, *Programming Under Mach*, (Addison-Wesley UNIX and Open System Series). Reading, MA: Addison-Wesley Publishing Company, 1993.

[82] R. A. Gingell, J. P. Moran, and W. A. Shannon, "Virtual memory architecture in SunOS," in *USENIX Association Conference Proceedings*, pp. 81–94, June 1987.

[83] J. B. Postel, "User datagram protocol," Internet Request For Comments RFC-768, Aug. 1980. Available as `ftp://ds.internic.net/rfc768.txt`.

[84] D. E. Comer, *Internetworking with TCP/IP Vol I: Principles, Protocols, and Architecture*. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1991.

[85] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol II: Design, Implementations, and Internals*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[86] "User datagram protocol," Internet Request For Comments RFC-791, Sept. 1981. Available as `ftp://ds.internic.net/rfc791.txt`.

[87] "Transmission control protocol," Internet Request For Comments RFC-793, Sept. 1981. Available as `ftp://ds.internic.net/rfc793.txt`.

[88] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, (Addison-Wesley Professional Computing Series). Reading, MA: Addison-Wesley Publishing Company, 1994.

[89] V. Jacobson, "Congestion avoidance and control," *Computer Communication Review*, vol. 18, pp. 314–329, Aug. 1988.

[90] D. C. Lynch and M. T. Rose, Eds., *Internet System Handbook*. Reading, MA: Addison-Wesley Publishing Company, 1993.

[91] C. C. Douglas, T. G. Mattson, and M. H. Schultz, "Parallel programming systems for workstation clusters," Yale University Department of Computer Science, New Haven, CT, Tech. Rep. YALEU/DSC/TR-975, Aug. 1993.

[92] F. J. Harrison, "Portable tools and applications for parallel computers," *International Journal of Quantum Chemistry*, pp. 847–863, 1991.

[93] R. Butler and E. Lusk, "User's guide to the p4 parallel programming system," Argonne National Laboratory, Argonne, IL, Tech. Rep. ANL-92/17, June 1992.

[94] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315–339, 1990.

[95] R. Konuru, J. Casas, R. Prouty, S. Otto, and J. Walpole, "A user-level process package for PVM," in *Proceedings of the Scalable High-Performance Computing Conference*, pp. 48–55, May 1994.

[96] J. Ferber, "Conceptual reflection and actor languages," in *Meta-Level Architectures and Reflection*, Amsterdam: Elsevier Science Publishers B. V., 1988, pp. 177–193.

[97] M. P. Peercy, "Reconfiguration and recovery in distributed memory multicomputers," Ph.D. dissertation, University of Illinois, Urbana, IL, Sept. 1994.

[98] G. Agha, S. Frølund, R. Panwar, and D. Sturman, "A linquistic framework for the dynamic composition of dependability protocols," in *Dependable Computing for Critical Applications 3*, Heidelberg: Springer-Verlag, Sept. 1993, pp. 345–363.

[99] S. Krishnan and L. V. Kalé, "Efficient, machine-independent checkpoint and restart for paralllel programs." Available as `ftp://ftp.cs.uiuc.edu/pub/CHARM/papers/Checkpoint_SC93.ps.Z`.

[100] A. B. Sinha, L. V. Kalé, and B. Ramkumar, "A dynamic and adaptive quiescence detection algorithm," Parallel Programming Laboratory, University of Illinois, Urbana, IL, Tech. Rep. 93-11, Sept. 1993.

[101] B. Ramkumar, "Machine independent "AND" and "OR" parallel execution of logic programs," Ph.D. dissertation, University of Illinois, Urbana, IL, Oct. 1990.

[102] F. Mattern, "Global quiescence detection based on credit distribution and recovery," *Information Processing Letters*, vol. 30, pp. 195–200, Feb. 1989.

[103] H. Abelson and G. J. Sussman, Eds., *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press, 1985.

[104] F. Brglez, D. Bryan, and K. Kominski, "Combinational profiles of sequential benchmark circuits," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, June 1989.

[105] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: W. H. Freeman and Company, 1979.

[106] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, (Addison-Wesley Series in Artificial Intelligence). Reading, MA: Addison-Wesley Publishing Company, 1984.

[107] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Transactions on Computers*, vol. 30, pp. 215–222, Mar. 1981.

[108] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM Journal of Research and Development*, vol. 10, pp. 278–291, July 1966.

[109] H. Fujiwara and S. Toida, "The complexity of fault detection problems for combinational logic circuits," *IEEE Transactions on Computers*, vol. 31, pp. 555–560, June 1982.

[110] S. Chandra and J. H. Patel, "Test generation in a parallel processing environment," in *Digest of Papers, International Conference on Computer Design*, pp. 11–14, Oct. 1988.

[111] S. Patil and P. Banerjee, "Fault partitioning issues in an integrated parallel test generation fault simulation environment," in *Proceedings of the IEEE International Test Conference*, Washington, D.C., pp. 718–727, Aug. 1989.

[112] S. Patil and P. Banerjee, "Performance trade-offs in a parallel test generation fault simulation environment," *IEEE Transactions on Computer Aided Design*, vol. 10, pp. 1542–1558, Dec. 1991.

[113] S. Patil, P. Banerjee, and J. Patel, "Parallel test generation for sequential circuits on general-purpose multiprocessors," in *Proceedings of the Design Automation Conference*, June 1991.

[114] P. Agrawal, V. D. Agrawal, and J. Villoldo, "Sequential circuit test generation on a distributed system," in *Proceedings of the Design Automation Conference*, June 1993.

[115] G. J. Li and B. W. Wah, "MANIP-2: A multicomputer architecture for evaluating logic programs," in *Proceedings of the International Conference on Parallel Processing*, pp. 123–130, Aug. 1985.

[116] B. W. Wah, G. J. Li, and C. F. Yu, "Multiprocessing of combinatorial search problems," *IEEE Computer*, vol. 18, pp. 93–108, June 1985.

[117] V. N. Rao and V. Kumar, "Parallel depth first search, part I: Implementation," *International Journal of Parallel Processing*, vol. 16, no. 6, 1987.

[118] V. N. Rao and V. Kumar, "Parallel depth first search, part II: Analysis," *International Journal of Parallel Processing*, vol. 16, no. 6, 1987.

[119] A. Motohara, K. Nishimura, H. Fujiwara, and I. Shirakawa, "A parallel scheme for test-pattern generation," in *Digest of Papers, International Conference on Computer-Aided Design*, pp. 156–159, Nov. 1986.

[120] S. Patil and P. Banerjee, "A parallel branch and bound approach to test generation," in *Proceedings of the Design Automation Conference*, Las Vegas, NV, pp. 339–345, June 1989.

[121] S. Patil and P. Banerjee, "A parallel branch and bound algorithm for test generation," *IEEE Transactions on Computer Aided Design*, vol. 9, pp. 313–322, Mar. 1990.

[122] S. Arvindam, V. Kumar, V. N. Rao, and V. Singh, "Automatic test pattern generation on parallel processors," Department of Computer Science, University of Minnesota, Minneapolis, MN, Tech. Rep. TR-90-29, May 1990.

[123] B. Ramkumar and P. Banerjee, "Portable parallel test generation for sequential circuits," in *Digest of Papers, International Conference on Computer-Aided Design*, pp. 220–223, Nov. 1992.

[124] J. S. Conery, "The AND/OR process model for parallel interpretation of logic programs," Ph.D. dissertation, University of California, Irvine, CA, June 1983.

[125] S. Patil, "Parallel algorithms for test generation and fault simulation," Ph.D. dissertation, University of Illinois, Urbana, IL, Sept. 1990.

[126] S. T. Patel and J. H. Patel, "Effectiveness of heuristics measures for automatic test pattern generation," in *Proceedings of the Design Automation Conference*, pp. 547–552, 1986.

[127] S. J. Chandra and J. H. Patel, "Experimental evaluation of testability measures for test generation," *IEEE Transactions on Computer Aided Design*, vol. 8, pp. 93–97, Jan. 1989.

[128] R. H. Bell, Jr., R. H. Klenke, J. H. Aylor, and R. D. Williams, "Results of a topologically partitioned parallel automatic test pattern generation system on a distributed-memory multiprocessor," in *ASIC '92*, Sept. 1992.

[129] T. M. Niermann, "Techniques for sequential circuit automatic test generation," Ph.D. dissertation, University of Illinois, Urbana, IL, Mar. 1991.

[130] E. G. Ulrich and T. Baker, "Concurrent simulation of nearly identical digital networks," *IEEE Computer*, vol. 7, pp. 39–44, Apr. 1974.

[131] D. B. Armstrong, "A deductive method for simulating faults in logic circuits," *IEEE Transactions on Computers*, vol. 21, pp. 462–471, May 1972.

[132] W.-T. Cheng and M.-L. Yu, "Differential fault simulation — a fast method using minimal memory," in *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 424–428, June 1989.

[133] S. Seshu, "On an improved diagnosis program," *IEEE Transactions on Electronic Computers*, vol. 14, pp. 76–79, 1965.

[134] T. M. Niermann, W.-T. Cheng, and J. H. Patel, "PROOFS: A fast, memory efficient sequential circuit fault simulator," *IEEE Transactions on Computer Aided Design*, pp. 198–207, 1992.

[135] P. Banerjee, *Parallel Algorithms for VLSI Computer-Aided Design*. Englewood Cliffs, NJ: PTR Prentice Hall, 1994.

[136] P. A. Duba, R. K. Roy, J. A. Abraham, and W. A. Rogers, "Fault simulation in a distributed environment," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 686–691, June 1988.

[137] W. A. Rogers and J. A. Abraham, "CHIEFS: A concurrent hierarchical and extensible fault simulator," in *Proceedings of the IEEE International Test Conference*, pp. 710–716, 1985.

[138] T. Markas, M. Royals, and N. Kanopoulos, "On distributed fault simulation," *IEEE Computer*, vol. 7, pp. 40–52, Jan. 1990.

[139] L. Soule and T. Blank, "Parallel logic simulation on general purpose machines," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 166–171, June 1988.

[140] R. B. Mueller-Thuns, D. G. Saab, R. F. Damiano, and J. A. Abraham, "Portable parallel logic and fault simulation," in *Digest of Papers, International Conference on Computer-Aided Design*, pp. 506–509, Nov. 1989.

[141] J. F. Nelson, "Deductive fault simulation on hypercube multiprocessors," in *Proceedings of the 9th AT&T Conference on Electronic Testing*, Oct. 1987.

[142] S. Ghosh, "NODIFS: A novel, distributed circuit partitioning based algorithm for fault simulation of combinational and sequential digital designs on loosely coupled parallel processors," LEMS, Division of Engineering, Brown University, Providence, RI, Tech. Rep., 1991.

[143] S. Patil, P. Banerjee, and J. Patel, "Parallel test generation for sequential circuits on general purpose multiprocessors," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, San Fransisco, CA, June 1991.

[144] S. P. Smith, W. Underwood, and M. R. Mercer, "An analysis of several approaches to circuit partitioning for parallel logic simulation," in *Proceedings of the International Conference on Computer Design*, pp. 664–667, 1987.

[145] C.-P. Kung and C.-S. Lin, "Parallel sequence fault simulation for synchronous sequential circuits," *Proceedings of the European Design Automation Conference*, pp. 434–438, Mar. 1992.

[146] E. P. Markatos and T. J. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, pp. 200–211, Apr. 1994.

[147] O. Plata and F. F. Rivera, "Classes versus prototypes in object-oriented languages," in *Conference Proceedings of the 1994 International Conference on Supercomputing*, pp. 186–194, July 1994.

VITA

Steven Michael Parkes attended the University of California, Davis, and received the B.S. and M.S. degrees in Electrical Engineering in 1982 and 1989, respectively. In 1983, Parkes joined the Grass Valley Group, Nevada City, California, where he was responsible for hardware and software design of production digital video equipment. In 1987, Parkes enrolled in the University of Illinois at Urbana-Champaign; he received the Ph.D. in 1994.

Parkes has received fellowships from the IEEE, Motorola, the Regents of the University of California, Digital Equipment Corporation, and the National Science Foundation. His Ph.D. research was supported by the Semiconductor Research Corporation. He has consulted for Xerox and the Grass Valley Group. He is active in parallel and distributed computing, object-oriented and modern programming languages, and CAD.

Parkes is founder and president of Sierra Vista Research, which was founded in 1993 to develop object-oriented environments for concurrent computing. He currently resides in Los Gatos, California.