THE UNIVERSITY OF CALGARY

Language Design for Parallel Simulation

by

Dirk Baezner

A thesis

submitted to the Faculty of Graduate Studies

in partial fulfillment of the requirements for the

degree of Master of Science

Department of Computer Science

Calgary, Alberta

February, 1991

ISBN 0-315-99537-8

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Language Design for Parallel Simulation" submitted by Dirk Baezner in partial fulfillment of the requirements for the degree of Master of Science. The research presented in this dissertation was supervised by Drs. Brian Unger and John Kendall.

Dr. John Kendall, Supervisor
Dean of Science

Dr. Brian Unger
Department of Computer Science

Dr. Graham Birtwistle
Department of Computer Science

Dr. Richard Fujimoto
Georgia Institute of Technology

Date February 11, 1991

Abstract

Much of the research into parallel simulation over the last decade focuses on the performance of various strategies for parallel simulation. Among the most successful of these are implementations of a paradigm called Virtual Time. Virtual Time simulations consist of independently executing processes that communicate and synchronize their actions solely by sending and receiving timestamped messages. The successful application of Virtual Time to a wide class of simulation problems has already been demonstrated. However, much of this success has been achieved without regard for the modeling and language design issues associated with this paradigm. Specifically, existing modeling practices and language constructs associated with sequential simulation may have to be modified or abandoned in favor of decompositional techniques dictated by Virtual Time.

The goal of this research is to assess the impact of Virtual Time on simulation language design. In addressing this goal, this thesis makes a number of contributions. First, the fundamental differences between Virtual Time and sequential simulation are identified and the impact of these differences on parallel simulation is examined. Second, a set of language design criteria for Virtual Time is developed based on the fundamental differences between Virtual Time and sequential simulation, and on the characteristics of existing languages for Virtual Time. Third, key limitations of existing languages for Virtual Time are identified. These limitations are shown to make even very basic simulation models difficult to implement using these languages. Fourth, a set of language constructs is developed that addresses the limitations of existing languages. These languages constructs are embodied within a new language for Virtual Time called *Sim*D that was designed and implemented as part of this research. *Sim*D was implemented on an existing implementation of Virtual Time, called Time Warp, developed and made available by Jade

Simulations International Corporation. Finally, the effectiveness of *Sim*D for developing well-structured, efficient, parallel simulations is demonstrated using several basic examples and two simulations implemented and executed on Time Warp.

Dedication

Für Oma

Acknowledgements

I would like to thank Brian Unger for his enormous contributions to this research. He supervised my initial research and subsequently continued to provide much needed guidance in further developing my ideas and this thesis. He was also the one who introduced me to the world of simulation and Time Warp, topics that will no doubt continue to dominate my interests for many years to come. I would also like to thank him for having the courage and wisdom to found Jade Simulations, thereby giving so many of us a chance to learn and grow simply by doing what we enjoy most – parallel simulation.

I would like to thank John Kendall for agreeing to supervise the remainder of my graduate programme and for his very helpful comments about this thesis. If you can distinguish my work from that of others upon which it is based, it is John who should be thanked. Most important of all, he provided a much appreciated sense of calm that contributed significantly to the successful completion of this thesis. The games of "telephone tag" were always good for a laugh as well.

I would like to extend a very special thank you to Greg Lomow for so many of his contributions throughout a very rewarding friendship. His enthusiasm for me and this research was inspiring. His insights and many helpful suggestions helped me with all aspects of this research. He also had the courage to review early drafts of my work and suggest numerous improvements. Of course, none of this diminishes the contributions of Lonnie, the "pupster". Although devoted to Greg, she provided me with many hours of joyful diversion as we chased each other around the office.

I would like to thank Richard Fujimoto for suggesting additional improvements to the thesis, and I would like to thank both him and Graham Birtwistle for participating in my thesis defense.

I would like to thank everyone at Jade Simulations for their assistance and friendship. Most notably, Gail Davies-Howard, Jeff Allan, and Mark Davoren. Gail somehow managed to arrange almost regular appointments between me and Brian Unger, whether he had time or not, and Jeff and Mark helped me clarify my ideas about parallel simulation and language design. I'm sure Mark would have done it even if he didn't owe me lunch. In addition, Jim Inkster was always more than willing to reminisce with me about the good old days of Simula.

I would like to thank my parents, my brother, and his family for their support, enthusiasm, and understanding during the many evenings and weekends it took to put all of this together.

Table of Contents

Approval Page ii

Abstract iii

Dedication v

Acknowledgements vi

Table of Contents viii

List of Tables xi

List of Figures xii

1 Introduction 1
- 1.1 Parallel Processors and Parallel Programming 1
- 1.2 Parallel Simulation and Virtual Time 2
- 1.3 Research Goal 4
- 1.4 Contributions of this Research 5
- 1.5 Thesis Overview 6

2 Departure from Sequential Simulation 8
- 2.1 Virtual Time 8
- 2.2 Impact of Message-Passing 10
- 2.3 Designing a Parallel Simulation 12
- 2.4 The Chandy-Misra Approach 14
- 2.5 Time Warp 16
- 2.6 Time Warp Transparency Issues 20
- 2.7 Summary 24

3 Languages for Virtual Time 26
- 3.1 The *Sim*++ Parallel Simulation Language 27
  - 3.1.1 Overview of C++ Concepts and Terminology 29
  - 3.1.2 *Sim*++ Data Types 30
  - 3.1.3 Decomposing a Simulation into Entities 32
  - 3.1.4 Simulation Primitives 34
  - 3.1.5 Conditional Selection of Events 36
  - 3.1.6 Type Checking 38
  - 3.1.7 Initialization and Execution of *Sim*++ Simulations 39
  - 3.1.8 Clusters for Shared Memory among Entities 41
  - 3.1.9 Preventing Side Effects of Causality Errors 43
  - 3.1.10 Language Support for Time Warp 44

3.2 Comparing *Sim++* to Other Languages 46
3.2.1 Extended Virtual Time 46
3.2.2 ModSim 50
3.2.3 Languages for Conservative and Optimistic Systems 52
3.3 Language Design Criteria for Virtual Time 55
3.4 Summary 58

4 Limitations of Existing Languages 60
4.1 The Barber Shop Queueing Model 60
4.2 Limiting the Size of the Barber Shop Wait Queue 63
4.3 A Variation on Barber Shop Termination 64
4.4 Adding Statistics Reporting to the Barber Shop Model 68
4.5 Extending *Sim++* for Modeling Resource Competition 74
4.6 Summary 78

5 *Sim*D: A Language Proposal for Virtual Time 80
5.1 Overview 80
5.2 *Sim*D Data Types 81
5.3 Predicates 88
5.4 Entities 91
5.5 Event Handlers 93
5.6 Initialization of *Sim*D Simulations 96
5.7 The Barber Shop Model Solved in *Sim*D 96
5.8 The Resource Competition Facility Solved in *Sim*D 100
5.9 Optimizing *Sim*D on Time Warp 103
5.10 Summary 104

6 Experience With *Sim*D 106
6.1 Overview of Experiments and Experimental Method 106
6.2 The Health Care System 110
6.2.1 Design Issues and Implementation 111
6.2.2 Simulation Parameters and Performance Results 113
6.3 The Mobile Communication Network 118
6.3.1 Design Issues and Implementation 118
6.3.2 Simulation Parameters and Performance Results 123
6.4 Summary 127

7 Conclusion 129
7.1 Critique of *Sim*D 129
7.2 Thesis Summary 130
7.3 Conclusions 133
7.4 Further Study and Development 136

Bibliography 139

A *Sim*D Implementation of the Health Care System 149

B *Sim*D Implementation of the Mobile Communication Network 159

List of Tables

6.1 Configuration of the Health Care System..... 115
6.2 Application Parameters for the Health Care System 115
6.3 Application Parameters for the Mobile Communication Network..... 124

List of Figures

3.1 C++ Declarations for *Sim*++ Data Types 31
3.2 C++ Declaration for Entity Class Automobile 34

4.1 C++ Code for Source and Barber Entities 61
4.2 C++ Declaration for Late Customer Predicate 67
4.3 C++ Code for Barber Entity with Transparent Reports 71
4.4 C++ Code for Event-Oriented Barber Entity 73

5.1 C++ Declarations for *Sim*D Data Types 83
5.2 C++ Insertion and Extraction Operators for Class Point 87
5.3 C++ Declaration for Class Entity 92
5.4 *Sim*D Predicate Chain 95
5.5 *Sim*D Barber Entity for Basic Barber Shop Model 97
5.6 *Sim*D Predicate Chain for Barber Entity During Wait 98
5.7 *Sim*D Predicate Chain for Barber Entity During Hold 98
5.8 *Sim*D Barber Entity for Extended Barber Shop Model 99
5.9 *Sim*D Predicate Chain for Extended Barber Shop Model 100
5.10 *Sim*D RES Facility 102
5.11 *Sim*D Predicate Chain for RES Facility 103

6.1 The Health Care System 111
6.2 C++ Code for Village Entity Class 112
6.3 C++ Code for Health Center Entity Class 113
6.4 Application Speedup for the Health Care System (no added computation) 117
6.5 Application Speedup for the Health Care System (10 milliseconds of added computation) 117
6.6 Application Speedup for the Health Care System (20 milliseconds of added computation) 118
6.7 The Mobile Communication Network 119
6.8 C++ Declarations for the Spatial Layer Interface 121
6.9 Application Speedup for the Mobile Communication Network (no added computation) 125
6.10 Application Speedup for the Mobile Communication Network (10 milliseconds of added computation) 125
6.11 Application Speedup for the Mobile Communication Network (20 milliseconds of added computation) 126

Chapter 1

Introduction

During the last decade, a great deal of attention has focused on the use of parallel processors to execute large, complex simulations whose computational requirements cannot be satisfied by most single processor computing systems. Such simulations often require many hours or days of execution time and tens or hundreds of megabytes of memory to execute a single experiment. Although many single processor computing systems can satisfy the memory requirements of such simulations using virtual memory, it is usually at the expense of increased execution time. In the absence of viable alternatives, researchers are forced to simplify the models they study to reduce their processing and memory requirements. Unfortunately, such simplifications also reduce our confidence that the simulation is a valid representation of the physical system that it models. Without an appropriate level of confidence, predictions about the physical system based on observations of the logical system are meaningless.

1.1 Parallel Processors and Parallel Programming

To address the limitations of sequential simulation, many researchers are turning to parallel processors for a solution. The term *parallel processor* is used throughout this thesis to refer to multiple instruction, multiple data stream (MIMD) architectures, ranging from tightly coupled multiprocessors that share memory to loosely coupled multicomputer networks that communicate solely via message-passing. Parallel processors offer the combined processing speed and memory of hundreds to thousands of processors.

However, developing software for parallel processors is significantly more difficult than for single processor computing systems. There are two reasons for this. First, typically, algorithms for parallel processors must be decomposed into multiple, concurrently executing and cooperating processes. The concurrency, synchronization, and communication among parallel processes introduces a level of complexity not present in sequential programming. Second, many parallel programs are inherently non-deterministic. This means that they do not necessarily produce the same results even when the same program is executed with the same input. Non-determinism results from variations in timing among parallel processes from one execution to the next. Non-determinism is intolerable in simulations because it introduces a degree of randomness that cannot be quantified. Non-determinism also complicates debugging because errors may not be repeatable, making it difficult to find the cause of an error and to verify when the error has been corrected. These problems must be addressed by appropriate paradigms and languages for parallel processors.

1.2 Parallel Simulation and Virtual Time

The execution of a simulation on a parallel processor is referred to as *parallel simulation.* The goals of parallel simulation are to reduce execution time and to allow larger and more complex systems to be simulated. The primary challenge in achieving these goals is to preserve the causal relationships present in the simulation model in the absence of traditional sequential techniques based on global knowledge and centralized control. The causality constraint can be expressed in relatively simple terms [Jef85b, page 407]:

> If an event A causes event B, then the execution of A and B must be scheduled in real time so that A is completed before B starts.

Violation of this constraint is referred to as a causality error [Fuj89].

One of the most successful paradigms for parallel simulation is Virtual Time [Jef85b]. Virtual Time simulations are composed of independently executing processes that communicate and synchronize their actions by sending and receiving timestamped messages. This paradigm is sufficiently general that it encompasses a number of approaches to parallel simulation typically classified as either *conservative* or *optimistic*. For comprehensive surveys of this and other techniques for parallel simulation see [Fuj89, Rig89]. Among existing techniques, Virtual Time has emerged as a dominant approach to parallel discrete-event simulation, and it is the sole focus of this thesis.

Conservative implementations of Virtual Time execute events only when they can guarantee that doing so does not violate the causality constraint. Because of inherent limitations in verifying that causality is preserved across multiple processors, conservative systems often delay executing events even when it is unnecessary to do so. This can result in deadlock. Much of the research into conservative systems has focused on techniques for deadlock prevention, and deadlock detection and recovery. The dominant conservative approach was developed by Chandy and Misra [Cha81, Mis86].

Optimistic implementations of Virtual Time execute events even when there is a possibility that doing so will violate the causality constraint. As a result, optimistic systems require the ability to detect and correct causality errors when they occur. Optimistic systems incur the additional overhead of error detection and correction for the freedom to execute even when there is a possibility that the resulting computation will be incorrect. The dominant optimistic approach, called Time Warp, was developed by Jefferson and Sowizral [Jef85a].

The Virtual Time paradigm has been used successfully in simulating large, complex systems with speedups of an order of magnitude and more reported for a number of

performance benchmarks executing on a variety of parallel processors [Bae89, Fuj87, Fuj88a, Leu89, Wie89]. However, much of this success has been achieved without regard for the modeling and language design issues associated with this paradigm. Specifically, existing modeling practices and language constructs associated with sequential simulation may have to be modified or abandoned in favor of decompositional techniques dictated by Virtual Time [Jef84]. Although several languages have already been developed for Virtual Time, this thesis will show that many of these languages contain inherently sequential constructs or are difficult to use for representing even very basic simulation models.

1.3 Research Goal

The goal of this research is to assess the impact of Virtual Time on simulation language design. This goal is divided into the following research questions addressed by this thesis:

1. How do the characteristics of Virtual Time differ from sequential simulation?
2. How do the characteristics of Virtual Time impact simulation language design?
3. Can languages for Virtual Time be used to develop well-structured, efficient, parallel simulations?

In addressing these questions, this thesis focuses on discrete-event simulation, the primary application of Virtual Time. In addition, this thesis focuses on procedural, rather than functional or logical languages. Although some attention has been given to the use of functional [Mar88] and logical [Cle90, Fut88, Li89] languages for parallel simulation, procedural languages continue to dominate the simulation literature. Finally, this thesis focuses on Time Warp as the preferred implementation of Virtual Time. This choice is justified by existing literature [Fuj87, Fuj88a, Fuj89, Jef87, Jef90] which demonstrates

that Time Warp has fewer limitations and comparable, often superior, performance to other implementations of Virtual Time. However, Time Warp implementations cannot currently be made completely transparent to users. This claim and its resulting implications are examined in Chapters 2 and 3.

1.4 Contributions of this Research

In assessing the impact of Virtual Time on simulation language design, this thesis makes the following contributions:

1. The fundamental differences between Virtual Time and sequential simulation are identified and the impact of these differences on parallel simulation is examined.
2. A set of language design criteria for Virtual Time is developed based on the fundamental differences between Virtual Time and sequential simulation, and on the characteristics of existing languages for Virtual Time. The degree to which the existing languages satisfy the proposed design criteria is examined.
3. Key limitations of existing languages for Virtual Time are identified. These limitations are shown to make even very basic simulation models difficult to implement using these languages.
4. A set of language constructs is developed that addresses the limitations of existing languages. These languages constructs are embodied within a new language for Virtual Time called *Sim*D that was designed and implemented as part of this research. *Sim*D was implemented on an existing implementation of Time Warp developed and made available by Jade Simulations International Corporation.
5. The effectiveness of *Sim*D for developing well-structured, efficient, parallel simulations is demonstrated using several basic examples and two simulations

implemented and executed on Time Warp. A performance study of these simulations is presented.

In addition to the contributions of this thesis, this research resulted in two refereed conference papers on the subjects of parallel simulation language design [Bae90] and parallel simulation performance using Time Warp [Bae91].

1.5 Thesis Overview

This thesis is intended to be self-contained with respect to concepts such as parallel simulation, Virtual Time, and Time Warp. It is assumed that the reader has some familiarity with discrete-event simulation and object-oriented programming.

Chapter 2 examines the characteristics of Virtual Time and Time Warp and how these differ from sequential simulation. The modeling and language design issues associated with Virtual Time are separated from those specific to Time Warp. Several proposals in the literature for enhancing the transparency of Time Warp are examined. The Chandy-Misra approach is also briefly summarized.

Chapter 3 surveys existing simulation languages for Virtual Time to determine how they address the issues identified in Chapter 2. One of these languages, *Sim++* [Jad90], is examined in some detail in that it serves as the foundation for much of the research presented in this thesis. A set of language design criteria for Virtual Time is proposed, based on the findings of Chapters 2 and 3.

Chapter 4 investigates by way of short examples the difficulties associated with using *Sim++* to model common types of process interactions. These difficulties are not unique to *Sim++*. They are common to several existing languages for Virtual Time. The

shortcomings illustrated in this chapter were the primary motivation for the *Sim*D language proposed in Chapter 5.

Chapter 5 proposes a new language for Virtual Time called *Sim*D, based in part on the language design criteria outlined in Chapter 3. *Sim*D incorporates simple, elegant solutions to the shortcomings identified in Chapter 4 without abandoning the basic philosophy and style of *Sim++*. The examples employed in Chapter 4 are reimplemented in *Sim*D as evidence of this claim.

Chapter 6 describes two parallel simulations that were implemented in *Sim*D and executed on Time Warp. The purpose in developing these simulations was to gain experience with *Sim*D and to test the implementation of *Sim*D on Time Warp. Performance results are presented for these simulations executing on Time Warp. The successful implementation and execution of these simulations is provided as evidence that *Sim*D can be used to develop well-structured, efficient, parallel simulations.

Chapter 7 critiques the *Sim*D language, summarizes the contributions of this research, presents conclusions drawn from this research, and describes areas requiring further study and development.

Chapter 2

Departure from Sequential Simulation

This chapter summarizes Virtual Time and Time Warp and outlines key areas of departure from sequential simulation. The summaries of Virtual Time and Time Warp are sufficient to understand the modeling and language design issues presented in the remainder of this thesis. More detailed descriptions are given by Jefferson [Jef85b]. Proposals in the literature for enhancing the transparency of Time Warp are also examined. These are important for minimizing implementation-specific modeling and language design issues. For completeness, the conservative Chandy-Misra approach is also briefly summarized. This approach is used in several existing languages for Virtual Time surveyed in Chapter 3.

2.1 Virtual Time

One view of physical systems commonly used in simulation is that of a collection of interrelated components that interact over time. This is the view captured by Virtual Time. A Virtual Time system consists of independently executing processes whose execution and interactions are tied to a logical clock that ticks *simulation time*. Each process has its own local view of this clock to which are tied the execution and interactions of that process.

In Virtual Time, all interactions are represented by timestamped messages. When a process sends a message, it must specify a *receiver* and *receive time* for the message. The receiver of a message is the process to which the message is being sent. The receive time of a message is the simulation time at which the receiver must receive the message. The receive time must always be greater than or equal to the simulation time of the sending

process. A process receives all messages sent to it in order of increasing receive time. As each message is received, the simulation time of the receiving process is advanced to the receive time of the message. This is the only way in which the simulation time of a process can advance. The processing of a message may involve updating the state variables of the receiving process, as well as sending zero or more messages. A process terminates when it has received all outstanding messages.

Ordering messages on increasing receive time is not in itself sufficient to guarantee determinism. The implementation of Virtual Time must additionally provide some mechanism for ordering messages with identical receive times. One possibility is to include an identifier in each message that is unique among all messages in the simulation and that is used to further order messages with identical receive times. The identifier must be independent of the number of processors on which the simulation is executed and the mapping of processes to processors. In this way, determinism is not affected by the run-time configuration of the simulation. The exact details of this and other possible mechanisms for determinism are beyond the scope of this thesis. It is assumed simply that some mechanism for guaranteeing determinism is provided by the implementation of Virtual Time.

Virtual Time differs fundamentally from sequential simulation in two ways. First, Virtual Time processes interact solely by sending and receiving timestamped messages. Each process has its own local state and does not directly access the states of other processes or communicate with other processes through shared memory. Second, Virtual Time simulations must be designed specifically for parallel execution if they are to achieve significant reductions in execution time. Although it is common in sequential simulations to decompose a problem into logical processes, this is typically a representational

convenience without regard (or need) for the ability of those processes to work in parallel. The impact of these differences is examined in detail in the sections that follow.

2.2 Impact of Message-Passing

The primary advantage of message-passing is that Virtual Time processes can execute concurrently on multiple processors and need not progress through simulation time at a uniform rate. As such, there is a great deal of freedom in how processes are executed, as long as causality is maintained. This differs from the *pseudo-concurrency* of sequential simulation in which only one process executes at a time and in which all processes advance through time simultaneously. Sequential simulations rely on these invariants to forgo the synchronization that would otherwise be required for multiple processes to access a common state.

By relying solely on message-passing for all process interactions, there is no need to detect or control other forms of interactions through shared memory. For example, on shared-memory parallel processors, it is possible to provide system calls that processes invoke before and after accessing a shared variable. The system calls could provide mutually exclusive access to the shared variable as well as the necessary synchronization with respect to simulation time. The disadvantage of this approach is that it relies on the discipline of the user to make these calls wherever necessary. In addition, shared variables would be much more difficult and costly to implement on distributed-memory parallel processors.

Another advantage of message-passing is its close correspondence with the event scheduling concepts of discrete-event simulation. For example, sending a message corresponds to scheduling an event, and receiving a message corresponds to the execution

of the event. These similarities facilitate the development of simulation constructs using Virtual Time [Lom88b].

Message-passing also provides a conceptually simpler form of process interaction when compared with traditional sequential techniques. For example, in the sequential simulation language Simula [Dah72], processes communicate using global memory or by directly accessing each other's state variables; to synchronize, these processes use a number of process scheduling constructs built into the language. This dichotomy between how processes communicate and how they synchronize can result in increased code complexity and errors [Bae90, Bir84]. In contrast, Virtual Time uses message-passing for both types of interactions. Thus, a message can represent communication between processes, synchronization between processes, or both.

In spite of the advantages of message-passing, the lack of shared memory can also increase the complexity of some simulation problems. For example, one of the most difficult types of problems to model using Virtual Time is one in which many or all processes regularly access a large, global state. A common technique for modeling problems of this type using Virtual Time is to subdivide the global state into sectors [Bec88, Cle90, Con90, Gol84, Lom88b]. Each sector is a separate process and is responsible for managing its part of the global state. Accessing the state is accomplished by sending messages to the appropriate sector process. A problem of this type is examined in Chapter 6.

Another problem with message-passing is that it is generally several orders of magnitude slower than interactions through shared memory. As such, messages cannot generally be employed as one-for-one substitutes for shared memory references. The communication overhead resulting from such an approach could easily dominate the execution of a parallel simulation. This implies that simulation models with many

interactions between processes may be unsuitable for Virtual Time or, at the very least, will require careful structuring in order to minimize communication overhead.

2.3 Designing a Parallel Simulation

Although the characteristics of Virtual Time make it suitable for execution on parallel processors, it is nevertheless possible to write Virtual Time simulations that perform poorly in a parallel execution. Such simulations might include one or more of the following:

1. *Processes that consume a significant fraction of the total processing time of a simulation.* The time required to execute these "hog" processes imposes a lower bound on the execution time of the simulation, regardless of the number of processors used to execute the simulation.
2. *Multiple processes that coordinate sequentially to complete a task.* In a simulation in which two or more processes coordinate sequentially to complete a task, there is little or no potential for parallelism among those processes if one or more such processes are idle waiting for other processes to execute.
3. *Processes with a high ratio of communication overhead to computation.* If the number of messages that processes send and receive is sufficiently high, the performance benefits derived from executing simulation processes in parallel may be outweighed by communication overhead.

All of the issues outlined above are performance-oriented and specific to Virtual Time. They are not factors in the design of sequential simulations.

The importance of performance in the *design* of a Virtual Time simulation can be seen as a major shortcoming of parallel simulation. As past experience with sequential

simulation has shown, performance considerations may be at odds with another key design objective: readability [Béz88, page 48]:

> It is a usual practice to transform a program in order to improve its performance. This could mean adding new information to the source program that is completely extraneous to the logical problem. This could also mean partially or completely destroying the structure of the program.

Because of the need to consider parallelism in parallel computation, this problem is even more acute than for sequential computation. Jenkins [Jen89, page 27] cites the experience of a team of researchers at Sandia National Laboratories who achieved between 502 and 637 times speedup running three general, parallel computations on a 1024 processor hypercube:

> When you go to several hundred or a thousand processors, in order to get [a high degree] of efficiency, you really have to rewrite your code in a special way. Right now there's no way to get around that. . . . If you pick the wrong algorithm, it definitely will not scale – you can pick an algorithm that doesn't have enough parallelism to use 1000 processors. Even if you pick the right algorithm, the way you implement that algorithm in the high-level language – the way you structure it when you convert it from your mathematical description into Fortran – is going to make a big difference in how that code performs on 1000 processors.

2.4 The Chandy-Misra Approach

The Chandy-Misra approach is a conservative implementation of Virtual Time. This approach consists of processes that send and receive timestamped messages along logical, directed channels connecting pairs of processes. The number of processes and the connectivity among processes are established at the beginning of the simulation and are dictated by the requirements of the application. For example, in an application in which any process may potentially communicate with any other process, a fully-connected communication topology is required in which each process has a directed channel to every other process. Processes that send messages to themselves to advance simulation time or to schedule a future activity have directed channels to themselves as well. Associated with each channel is a timestamp that corresponds to the receive time of the last message received along that channel. All messages sent along a channel must be in order of increasing receive time.

Using the established communication topology and the requirement that all messages sent along a channel be in order of increasing receive time, it is possible to guarantee that a process receives all messages sent to it in order of increasing receive time. Specifically, when a process attempts to receive its next message, the process is blocked until there is at least one waiting message on each of its incoming channels. The process then receives from among those channels the message with the lowest receive time. Unfortunately, blocking a process while awaiting additional messages does not guarantee that those messages will arrive. For example, it is possible that an incoming channel is empty because the associated sending process has no messages to send along that channel at the current simulation time. In fact, the sending process may not send any messages along that channel for the remainder of the simulation. In these cases, the receiving process would

remain blocked even if other messages were waiting on other incoming channels. If all processes are blocked waiting on one or more incoming channels for messages that never arrive, the system is deadlocked. The termination of a simulation is a special case of deadlock in which all processes are blocked and there are no remaining unreceived messages. Several variants of the Chandy-Misra approach exist to cope with deadlock, including techniques for deadlock prevention, and deadlock detection and recovery.

Deadlock prevention is accomplished by sending *null* messages along channels for which no application messages exist. In this way, there exist messages along all of a process' incoming channels. Null messages are not received by the application, however. Instead, they are used to advance the timestamp on the channels along which they are sent. In effect, a null message is a guarantee from the sending process that it will not send any future messages with a lower receive time than that of the null message along the same channel. In this way, the receiving process has sufficient information to determine whether or not a waiting application message can be received without the possibility of a future message with a lower receive time arriving along an alternate, empty channel. The amount by which a null message advances the timestamp of a communication channel is referred to as *lookahead* and is entirely application dependent. For example, the receive time of the next message a process sends may be entirely dependent upon the next message the process receives. Although techniques exist for automatically computing lookahead under certain, limited circumstances [Bag90], it is generally calculated explicitly by the application.

Deadlock detection and recovery is accomplished by continuously circulating a token among simulation processes. The token is a special message that gathers global knowledge about the system sufficient to detect deadlock. Once deadlock is detected, a recovery algorithm is invoked that locates the message in the system with the lowest receive time and arranges for it to be processed next. Instead of the circulating token, shared memory

parallel processor implementations can keep a global counter of the number of non-blocked processes [Fuj87, Ree88]. The system is deadlocked if this counter drops to zero and the system has not terminated.

2.5 Time Warp

Time Warp is an optimistic implementation of Virtual Time. Unlike the Chandy-Misra approach, there is no requirement for a statically defined communication topology, no concept of a channel between processes, and no requirement for messages to be sent in order of increasing receive time, even when those messages are sent to the same process. A Time Warp process can send to any process it can identify, including itself, and can send those messages in arbitrary order. In addition, Time Warp processes can be created and destroyed dynamically throughout the course of the simulation.

Time Warp, like all optimistic systems, relies on detection and correction of causality errors, not prevention. Whenever a process receives a message, no attempt is made to determine whether or not other messages with a lower receive time may yet be sent to the receiving process. As a result, it is possible for processes to receive messages out of order, resulting in causality errors that must subsequently be corrected. A causality error is detected when a message is received by a process whose simulation time has advanced beyond the receive time of that message. Such a message is referred to as a *straggler*. The erroneous computation resulting from a causality error may include an erroneous process state as well as erroneously sent messages based on that state. Unless the erroneous messages are eliminated before they are received and processed, secondary erroneous computations will result in other processes. These, in turn, may lead to tertiary erroneous computations, and so on.

Erroneous computations are eliminated using *process rollback* and *message cancellation* to restore a simulation back to a point that precedes the causality error. After rolling back, the affected processes resume executing, rereceiving the same messages as before, excluding erroneously sent messages that have been cancelled, and including the straggler. Although multiple processes may be involved in an erroneous computation, no global synchronization is required for rollback and message cancellation. Rollback and message cancellation are performed on a per process basis.

To support rollback and message cancellation, three data structures are associated with each process: an input queue, an output queue, and a state queue. The input queue consists of messages sent to a process and includes both the messages the process has received as well as those it has not. The messages received by a process remain in the input queue in case the process rolls back and must rereceive those messages. The output queue consists of copies of the messages sent by a process. The messages in the output queue are referred to as *anti-messages* and are used to cancel erroneously sent messages. The state queue consists of copies of the state of a process from checkpoints taken throughout the process' execution. During rollback, the process is restored to a state that precedes the causality error.

The two most common approaches to message cancellation are *aggressive cancellation* and *lazy cancellation*. Both approaches send anti-messages to cancel erroneously sent messages. An anti-message is always sent to the same process as the message it is intended to cancel and, like that message, is inserted into the input queue of the receiving process. The existence of a message and its anti-message in the same queue causes both to be deleted as if neither had ever existed. If the erroneous message has already been processed when the anti-message arrives, the process first rolls back to a state prior to the receipt of that message. This is referred to as a secondary rollback. A secondary rollback

may, in turn, result in tertiary rollbacks, and so on. Aggressive cancellation attempts to minimize the spread of erroneous computation by sending the anti-messages associated with an abandoned computation immediately upon rollback. However, sometimes the messages sent as a result of an erroneous computation may actually be correct. This might occur, for example, if the processing of a straggler does not change the state of the receiving process or only changes those portions of the state that do not affect all subsequently generated messages. To address this phenomenon, lazy cancellation withholds the anti-messages associated with an abandoned computation and compares them with the messages generated by the rolled back process in its subsequent forward computation. Only those messages not regenerated are cancelled. This eliminates the overhead associated with cancelling and resending identically regenerated messages. However, the delay required to confirm that an erroneously sent message must be cancelled may permit erroneous computation to spread further than with aggressive cancellation. Because the success of one cancellation strategy over another depends on the number of messages identically regenerated after rollback, the choice of which strategy to use is application-specific. A performance study by Lomow et al [Lom88a] suggests that lazy cancellation can outperform aggressive cancellation by as much as a factor of two in simulations involving feedback. Another study by Reiher et al [Rei90a] found that, although lazy cancellation typically performed slightly better than aggressive cancellation, the difference in performance was usually only 1-2%.

Fujimoto [Fuj88a] proposes *direct cancellation* as an optimization of Time Warp for shared-memory parallel processors. This approach does not use anti-messages to implement cancellation. Instead, each process maintains pointers to the messages it has sent. As such, it is possible to identify those messages to be cancelled simply by traversing the appropriate message pointers. Direct cancellation reduces both the memory and

execution overhead of message cancellation since there is no need to create, maintain, or send anti-messages. Since erroneous messages are cancelled more quickly, the spread of erroneous computation is also reduced.

Depending on the number and size of messages, anti-messages, and state checkpoints, the memory available on a parallel processor could easily be exhausted in simply maintaining the input, output, and state queues of Time Warp processes. Fortunately, the amount of historical information required by Time Warp to support rollback and message cancellation can be minimized. Time Warp regularly computes and distributes a value known as *global virtual time* (GVT). GVT is the minimum of all process simulation times and all unreceived message timestamps. This value defines the minimum simulation time to which any process can ever roll back. Messages and anti-messages with a receive time less than GVT are no longer required and can be deleted. Similarly, for each process, all state checkpoints but one with a simulation time less than GVT can also be deleted. The deletion of messages, anti-messages, and saved states earlier than GVT is referred to as *fossil collection*. Jefferson [Jef90] has shown that fossil collection, in combination with a flow control mechanism called *cancelback*, is sufficient for Time Warp simulations to execute in an amount of memory comparable to that of an equivalent sequential simulation. Executed under those constraints, a simulation is effectively serialized and unable to achieve speedup. Several times the minimum amount of memory is generally required for processes to execute optimistically in parallel. Efficient algorithms for computing and distributing GVT [Bel90] are therefore important in that the amount of memory freed through fossil collection is tied directly to the rate at which GVT advances.

GVT is also used as a commitment threshold for actions such as input, output, error handling, and termination. In one form or another, each of these actions involves interactions with the *external world* outside the scope of the simulation. Ordinarily, any

one of these actions on the part of a process is tentative in that the process may roll back as a result of an erroneous computation. For example, the act of reporting an error could itself be erroneous if it is the result of a causality error. After the affected process rolls back, the conditions that led to the error might no longer exist. As a result, interactions with the external world are not permitted until the interaction is guaranteed to be correct. In Time Warp, all actions at a simulation time greater than GVT are tentative and subject to rollback. However, all actions at or earlier than GVT are committed and free from rollback. Thus, when a process reports an error, the process is blocked until it rolls back or until GVT reaches the simulation time of the error call. In the latter case, the error is real and the execution of the simulation is aborted. Input, output, and termination are handled in a similar manner.

2.6 Time Warp Transparency Issues

Although Time Warp supports the same abstraction as Virtual Time, implementations of Time Warp cannot currently be made completely transparent to the user. This means that, in many cases, simulations must be designed specifically for Time Warp in order to achieve significant reductions in execution time or to execute at all. Specifically, simulations for Time Warp must deal with the issues of process state size, side effects resulting from causality errors, and interactions with the external world. This section examines these issues as well as proposed solutions. In addition, numerous researchers have proposed optimizations of Time Warp, sometimes at the expense of transparency. One such optimization, which appears several times in the literature, is examined.

The primary performance issue specific to Time Warp is process state size. Due to the overhead associated with saving and restoring states, Time Warp simulations must be

designed to minimize the size of process states. In one study, Fujimoto [Fuj88a] shows that the speedup achieved by Time Warp over an equivalent sequential execution declines from 9 to 2 as the size of process states is increased from 0 to 8000 bytes. Several approaches to reducing the overhead associated with large states have been developed. Abrams [Abr89] proposes dividing process states into *sub-states* and only saving those sub-states modified during the processing of a given message. Currently, this approach requires that the sub-states be identified explicitly by the application. In *Sim++*, it is possible to *write-lock* portions of a process' state that the application will never again modify. This approach is discussed in greater detail in Chapter 3. Like sub-states, write-locked memory is not transparent to the user. Fujimoto et al [Fuj88b] propose the use of special purpose hardware known as the *rollback chip* to eliminate much of the overhead associated with saving and restoring states. The rollback chip is designed to save and restore even large process states in a fraction of the time required using existing software techniques. In effect, the rollback chip only saves those portions of a state modified by the processing of a message. In addition, the rollback chip is transparent to an application and eliminates the need for facilities such as sub-states and write-locked memory that rely on an application to explicitly optimize state checkpoints. A prototype of this chip is currently being evaluated [Buz90]. Until the rollback chip is readily available for a wide variety of parallel processors, techniques such as sub-states and write-locked memory will continue to be required, as will the need to design simulations to minimize the size of process states.

Another important concern in Time Warp is ensuring that all side effects resulting from causality errors can be eliminated. Although Time Warp is capable of rolling back the state of a process and cancelling erroneously sent messages, there are other actions a process can take that cannot necessarily be rolled back. These include erroneous memory references outside the state of the process, division by zero, references through null pointers, infinite

loops, and stack overflow. Problems of this type are exacerbated by the fact that causality errors are not deterministic. In the worst case, the effects of a causality error may go undetected and lead to erroneous simulation results. Clearly, the potential problems arising from causality errors are ominous. However, my experience with *Sim++* suggests that these problems are relatively easy to avoid. Some discussion of how this is done is included in Chapter 3. Nevertheless, these problems do exist and current techniques for avoiding them rely, in general, on the discipline of the user. Resolution of these problems will depend primarily on the level of support provided by compilers and parallel processors.

A third difficulty concerns the restrictions imposed by Time Warp on a simulation's interactions with the external world. Specifically, Time Warp simulations are limited to those interactions for which the Time Warp implementation provides commitment. This typically includes basic facilities for input, output, error handling, and termination, but may exclude other, advanced facilities specific to a given parallel processor. One approach to this problem is to make the commitment mechanism available to the user. This is the approach used by Tipc [Ung90], a multi-lingual Time Warp implementation that augments existing sequential programming languages with primitives for sending and receiving timestamped messages. Tipc processes call **wait_for_gvt** prior to executing any operation that cannot be rolled back. **wait_for_gvt** blocks the calling process until GVT is equal to the process' simulation time. As long as the process executes at GVT, its actions are not subject to rollback. At the same time, however, GVT cannot advance until the simulation time of the process advances. If many processes execute repeatedly at GVT, the rate at which GVT advances is reduced, thereby reducing the amount of memory freed through fossil collection and available for optimistic execution. In addition to its performance implications, **wait_for_gvt** is also error prone in that it must be called prior

to each interaction with non-Tipc facilities. An alternative is to provide one type of process that always executes optimistically and is subject to rollback and another type of process that always executes at GVT. This approach is less error prone in that it clearly separates processes that can interact directly with the external world from those that cannot. This approach is used by *Sim++* and is discussed in greater detail in Chapter 3. Either approach can be used to transparently implement facilities requiring commitment.

Finally, a great deal of attention has focused on the development of optimized queries for Time Warp. A query is a read-only interrogation of a process' state by another process. A query consists of two parts: a query message, representing a request for information, and a reply message, representing the response. Because the process that initiates a query must wait for the response, a query is a form of sequential coordination between processes. As noted in Section 2.3, this type of programming practice can have a negative impact on performance. Nevertheless, when they exist, queries lend themselves to the following optimization. If Time Warp encounters a straggler query message, it locates the appropriate historical state to which the receiving process would ordinarily roll back, processes the query message using that state, and then restores the process to its future state to continue executing. This optimization eliminates the recomputation typically associated with a rollback by relying on the invariant that the processing of a query message does not alter the state of the receiving process. Typical approaches to this optimization [Bag90, Gat88, Jef87] require that the application explicitly identify when a message represents a query or a reply.

West [Wes88a] proposes an optimization to Time Warp called *lazy reevaluation* in which non-side-effecting messages are detected automatically by the Time Warp implementation. Specifically, after a process has processed a straggler message, the resulting state is compared to the process' ensuing state from its prior forward

computation. If the two states are identical, then the straggler message had no effect on the process' state and the process is restored to its future state to continue executing. If the two states differ, then all ensuing states are discarded and the process is forced to reprocess all messages following the straggler. This optimization is shown to improve performance by as much as 38% in applications tested. However, the success of lazy reevaluation is extremely application-dependent. Specifically, the number of non-side-effecting messages must be sufficiently large to outweigh the additional overhead associated with comparing process states. Unfortunately, if the non-side-effecting messages are queries, as is often the case, then a large number of such messages may degrade performance due to the sequential nature of queries. The advantage of lazy reevaluation over explicit query messages is that it is transparent to the user and, as such, does not promote an inefficient parallel programming practice. Indeed, queries have since been removed from the Time Warp Operating System described in [Jef87] because of this potential for abuse.

2.7 Summary

This chapter identified two fundamental differences between Virtual Time and sequential simulation. First, Virtual Time processes communicate and synchronize their actions solely by sending and receiving timestamped messages. Second, Virtual Time simulations must be designed specifically for parallel execution if they are to achieve significant reductions in execution time. An assessment of the impact of these differences showed that existing modeling practices and language design must be adapted to Virtual Time.

Throughout this chapter, a clear distinction was maintained between the Virtual Time paradigm and its implementation. This distinction reflects the view of this thesis that it is inappropriate to adapt modeling practices and language design to accomodate specific

implementations of Virtual Time. For example, it is inappropriate to rely on the discipline of the user to decompose a simulation to minimize process state size or to prevent side effects of causality errors. There are two reasons for this. First, such adaptations require that the user understand fundamental characteristics of the implementation of Virtual Time, even though those characteristics are not inherent to the Virtual Time paradigm or to parallel processing. Second, additional requirements imposed by a specific implementation of Virtual Time increase the complexity of the design and implementation of a parallel simulation and reduce its portability to other implementations of Virtual Time.

An examination of the literature suggests that state size and other Time Warp transparency issues can be resolved but will require support from compilers, parallel processors, and special-purpose hardware such as the rollback chip. However, the current state of the art in parallel simulation is such that these issues do factor into the design of a simulation. The language survey in the following chapter examines transparent and explicit techniques provided by existing parallel simulation languages for coping with these issues.

Chapter 3

Languages for Virtual Time

The fundamental differences between Virtual Time and sequential simulation present new challenges for simulation language design. Many existing sequential simulation languages provide modeling abstractions that encourage a close correspondence between a model and its implementation [Bir84, Bir86, Mul82]. However, since these languages are not designed for parallel processors, the abstractions they provide are not suitable for Virtual Time. Most notably, they rely on shared memory for process interactions and assume a pseudo-concurrency in which only one process executes at a time.

One language designed to address these problems is *Sim++* [Jad90], a simulation library embedded in the object-oriented programming language, C++ [Str86]. *Sim++* is a commercial product developed by Jade Simulations and is designed specifically for execution on Time Warp. Nevertheless, the modeling abstraction it provides is based on Virtual Time, rendering the underlying Time Warp executive relatively transparent. This chapter focuses primarily on *Sim++* in that it serves as the foundation for much of the research presented in this thesis, including the development of the *Sim*D language proposed in Chapter 5. *Sim++* was designed by a team of developers, including myself, and was implemented by me prior to beginning this research.

The selection of *Sim++* as a starting point for this research is appropriate for several reasons. First, *Sim++* provides modeling constructs for Virtual Time similar to those proposed by a variety of researchers. As such, it is representative of much of the ongoing research into language design for Virtual Time. Indeed, a number of its language features are entirely unique among existing and proposed languages. Second, since *Sim++* is

designed for Time Warp, it is not subject to the additional limitations imposed by conservative implementations of Virtual Time. Third, since *Sim++* is a C++ library, it is possible to experiment with variations of its modeling constructs without the need to write or modify a compiler. Access to *Sim++* and Jade Simulation's parallel processor were provided for this research through the courtesy of Jade Simulations.

An overview of *Sim++* is presented in Section 3.1 and is intended to provide a reading knowledge of *Sim++* sufficient to understand the examples and language design issues discussed in this thesis. Many of these issues have also been addressed by other researchers. Section 3.2 compares their solutions with those of *Sim++*. Section 3.3 proposes a set of language design criteria for Virtual Time based on the characteristics of the languages surveyed.

Sim++, *Sim*D, and all of the major examples and benchmarks presented in this thesis are implemented in C++. For the sake of brevity and clarity, a number of conventions have been adopted for the C++ code presented throughout this thesis. Typically, the declarations for local variables in functions are omitted. The context in which these variables are used is sufficient in most cases to determine their type. An ellipsis (i.e., ...) is used to indicate where C++ code, unnecessary to the discussion, has been omitted. The tokens **AND**, **OR**, and **NOT** are used to represent the C++ logical operators &&, ||, and !. *Italic font is used to represent pseudo code used in place of C++.* Finally, the symbol // denotes the beginning of a C++ comment that continues until the end of the current line.

3.1 The *Sim++* Parallel Simulation Language

Sim++ is a process-oriented, discrete-event simulation language embedded in the object-oriented programming language, C++. *Sim++* simulations can be executed sequentially on

a sequential simulator, or in parallel using Time Warp. The sequential simulator serves as the primary development environment for *Sim++* simulations as well as a baseline for speedup comparisons. *Sim++* programs can be moved between the sequential and Time Warp environments without source code modifications. *Sim++* programs are also transparently scaleable. This means that they can be executed on varying numbers of processors, also without source code modifications. The number of processors is specified at run-time in a user-supplied configuration file. *Sim++* programs are deterministic regardless of the run-time configuration used.

Sim++ simulations are defined in terms of *entities* and *events*. Entities are independently executing objects that communicate and synchronize their actions by scheduling and receiving events. Each entity has its own separate address space and, generally, cannot access the member variables and functions of other entities or communicate with other entities using shared memory. Entities and events correspond to Virtual Time processes and messages.

The actions of all entities and the scheduling of all events is tied to a logical clock that ticks time in an arbitrary, application-defined time scale called *simulation time*. Each entity has its own local view of this clock called the *entity's simulation time*. Each event is tied to this clock by means of a *scheduled event time* that specifies when a given entity should receive the event. This is the simulation time at which the event *occurs*. Typically, an entity receives events in order of increasing scheduled event time, the simulation time of the entity advancing in step with these event times. Entities can also defer and cancel events, and simulate the passage of time.

Typically, entities communicate solely by scheduling and receiving events. As an alternative, entities that communicate frequently can be grouped into *clusters*. All entities within a cluster are always executed on the same processor and are synchronized in such a

way that they are free to share memory and directly access each other's member variables and functions.

The execution of a *Sim++* simulation is divided into two phases: the *initialization phase* and the *execution phase*. The initialization phase serves to create entities, group entities into clusters, and initialize global variables. Entities may subsequently read but not modify the values of these global variables during the execution phase. The execution phase represents the main actions of a simulation and encompasses both the initialization and execution of entities.

Like most simulation languages, *Sim++* also provides facilities for random number generation, data collection and reporting, linked list manipulation, formatted input and output, tracing, and error reporting. These facilities are similar to those of other simulation languages [Bir86, Mul82] and will not be discussed further except as they appear in various examples throughout the thesis.

3.1.1 Overview of C++ Concepts and Terminology

C++ is an object-oriented extension of the C [Ker78] programming language. C++ provides support for common object-oriented language features such as data abstraction, encapsulation, inheritance, and polymorphism. The key language construct by which these abilities are possible is the *class*. A class is a user-defined data type consisting of named data elements, called *member variables*, and a set of operations, called *member functions*, that manipulate those data elements. An instance of a class is referred to as an *object*. Each object has its own private copy of the member variables of its class and is subject to the operations defined for that class. Typically, two of these operations include the creation and destruction of the objects themselves. A *constructor* is a member function that

specifies how instances of a class are created and initialized. A *destructor* is a member function that specifies how instances of a class are destroyed.

3.1.2 *Sim++* Data Types

This section describes the major data types provided by *Sim++*. The C++ declarations for these data types are shown in figure 3.1.

Class **sim_time** consists of real numbers greater than or equal to zero and is used to represent simulation times. Values of this type are used primarily in scheduling events and simulating the passage of time. For example, when an entity schedules an event, it must specify a simulation time delay. The sum of this delay and the scheduling entity's simulation time define the scheduled event time of the event.

Class **sim_type** consists of integers greater than or equal to zero and is used to represent event types. When an entity schedules an event, it must specify a value of type **sim_type**. This value becomes an attribute of the event and can be used to distinguish between different kinds of events.

Class **sim_entity_id** is used to represent entity identifiers. When an entity is created, a value of type **sim_entity_id** is returned to the application. This value uniquely identifies the newly created entity and is required to identify that entity when scheduling events for it. Class **sim_entity_id** defines two member functions, **name** and **class_name**, that can be used to determine an entity's name and class given its entity identifier.

Class **sim_event** is used to represent events. When an entity schedules an event, an object of type **sim_event** is created to represent that event. An entity *receives* an event when it receives the corresponding event object. Every event has a set of attributes,

```
class sim_time { ... };
class sim_type { ... };

class sim_entity_id {
   ...
public:
   char *name();
   char *class_name();
};

class sim_event {
   ...
public:
   sim_entity_id scheduled_by();
   sim_entity_id scheduled_for();
   sim_time event_time();
   sim_type type();
   void *body();
   int length();
};

class sim_event_id {
   ...
public:
   sim_entity_id scheduled_by();
   sim_entity_id scheduled_for();
   sim_time event_time();
   sim_type type();
};

const sim_time          SIM_NO_TIME;
const sim_type          SIM_NO_TYPE;
const sim_entity_id     SIM_NO_ENTITY_ID;
const sim_event         SIM_NO_EVENT;
const sim_event_id      SIM_NO_EVENT_ID;
```

Figure 3.1: C++ Declarations for *Sim++* Data Types

specified by the scheduling entity, that define the event. These attributes include the identity of the entity that scheduled the event, the identity of the entity for which the event is scheduled, the scheduled event time of the event, the type of the event, and the *body* of the event. The body of an event is used to pass arbitrary, application-specific data from the scheduling entity to the receiving entity. Class **sim_event** defines the member functions **scheduled_by**, **scheduled_for**, **event_time**, **type**, **body**, and **length** to access the attributes of an event. **scheduled_by** returns an entity identifier denoting the entity that

scheduled the event. **scheduled_for** returns an entity identifier denoting the entity for which the event is scheduled. **event_time** returns the simulation time for which the event is scheduled. **type** returns the event type of the event. **body** returns a pointer to the application-specific data in the body of the event. **length** returns the length, in bytes, of the event body. If the body of an event contains no data, **body** returns a null pointer and **length** returns zero.

Class **sim_event_id** is used to represent event identifiers. When an entity schedules an event, a value of of type **sim_event_id** is returned to the scheduling entity. This value uniquely identifies the scheduled event and can be used to subsequently cancel that event. Class **sim_event_id** defines the member functions **scheduled_by**, **scheduled_for**, **event_time**, and **type** to access the attributes of an event identifier. The values of these attributes are the same as those of the event that the event identifier represents.

For each data type, a special value is used to represent invalid or uninitialized instances of that type. For example, the value **SIM_NO_TIME** represents a non-existent simulation time and is the default value of all uninitialized variables of type **sim_time**. Similarly, the values **SIM_NO_TYPE**, **SIM_NO_ENTITY_ID**, **SIM_NO_EVENT**, and **SIM_NO_EVENT_ID** are defined for the remaining data types.

3.1.3 Decomposing a Simulation into Entities

Most physical systems can be viewed as a set of independently acting components that interact over time. In *Sim++*, these active components are represented by entities. Each entity is an instance of an entity class and has its own set of member variables. The entity class defines these member variables as well as the member functions that manipulate them. Typically, an entity class is defined for each type of active component in the simulation.

Unlike ordinary C++ objects, entities are active objects, like the components they represent. In other words, they execute continuously, scheduling and receiving events and updating their individual states. The state of an entity includes its member variables, its run-time stack, and any data structures dynamically allocated by that entity.

Entity classes are defined using the C++ class construct. They differ from ordinary classes in three ways, however. First, all entity classes must be derived from the pre-defined base class **sim_entity**. Unless derived from class **sim_entity**, an entity class inherits none of the attributes or abilities of entities (e.g., the ability to schedule and receive events). The interface to these attributes and abilities is provided through global functions. Second, all entity classes must define a constructor and body to represent the actions of entities of that class. The constructor of an entity serves to initialize the entity's member variables. The sole argument to an entity constructor is an *initialization* event that contains whatever run-time arguments are required to initialize the entity. The attributes of the initialization event are specified when the entity is created. The body of an entity defines the actions of that entity with respect to the physical component that the entity is intended to represent. Third, all entity class declarations must include a call to the macro **SIM_ENTITY**. This macro hides a number of additional declarations required by the implementation for the given entity class. As an example, the declaration of entity class **automobile** is shown in figure 3.2.

All entities share a common set of *entity attributes* that can be accessed by the functions **sim_current**, **sim_clock**, **sim_name**, and **sim_class_name**. **sim_current** returns an entity identifier that uniquely identifies the calling entity. Entities use this identifier when scheduling events for themselves. **sim_clock** returns the current simulation time of the calling entity. **sim_name** returns a string that is the name of the calling entity. The

```
class automobile : public sim_entity {
public:
   // member variables
   double Speed;
   double Fuel;
   ...

   // entity constructor
   automobile(sim_event &init_ev);

   // entity body
   void body();
};

SIM_ENTITY(automobile);


automobile::automobile(sim_event &init_ev)
{
   // initialize member variables
   Speed = 0.0;
   Fuel = 0.0;
   ...
}

void automobile::body() {
   // main actions of an automobile
   ...
}
```

Figure 3.2: C++ Declaration for Entity Class Automobile

name of an entity is specified when the entity is created. **sim_class_name** returns a string that is the name of the calling entity's class.

3.1.4 Simulation Primitives

This section describes the *Sim++* simulation primitives for scheduling, cancelling, and receiving events, and for simulating the passage of time.

The primitive **sim_schedule** is used to schedule events. An entity can schedule events for any entity it can identify, including itself. Arguments to **sim_schedule** include an entity identifier, a simulation time delay, an event type, and an optional data pointer and

length. These arguments serve to define the attributes of the event. The entity identifier uniquely identifies the entity for which the event is being scheduled. The sum of the simulation time delay and the scheduling entity's simulation time defines the scheduled event time of the event. The event type defines the type of the event. The data pointer and length define the body of the event; if omitted, the body of the event is null. **sim_schedule** returns an event identifier uniquely identifying the scheduled event. As an example, the call

```
ev_id = sim_schedule(sim_current(), 8.0, ARRIVAL, &i, sizeof(i));
```

schedules an event for the calling entity with a delay of 8.0 time units. **ARRIVAL** is assumed to be a constant of type **sim_type**. The body of the scheduled event contains a copy of **i**, where **i** is an arbitrary data structure. The return value of **sim_schedule** is assigned to the event identifier **ev_id**.

The primitive **sim_cancel** is used to cancel an event denoted by a given event identifier. Cancelling an event guarantees that it will not be received by the entity for which it was scheduled. As an example, the call

```
sim_cancel(ev_id);
```

cancels the event denoted by the event identifier **ev_id**. An event can only be cancelled at a simulation time earlier than its scheduled event time (i.e., before the event occurs).

The primitive **sim_wait** is used to receive events in order of increasing event time. As an example, the call

```
sim_wait(ev);
```

assigns the calling entity's next event to the event object **ev**. The simulation time of the calling entity is advanced to the event time of the received event. If an entity calls **sim_wait** after it has received its last event, the entity terminates.

The primitive **sim_hold** is used to simulate the passage of time. The simulated delay can be interrupted by any event scheduled for the calling entity with an event time that coincides with the delay. As an example, the call

```
remaining = sim_hold(10.0, ev);
```

simulates a delay of 10.0 time units. If the simulated delay is interrupted, the earliest interrupting event is assigned to the event object **ev**, the simulation time of the calling entity is advanced to the event time of the interrupting event, and **sim_hold** returns the amount of simulation time remaining of the original delay. If the simulated delay expires without interruption, the value **SIM_NO_EVENT** is assigned to **ev**, the simulation time of the calling entity is advanced by 10.0 time units, and **sim_hold** returns 0.0.

3.1.5 Conditional Selection of Events

Using the primitives **sim_wait** and **sim_hold**, an entity receives the events scheduled for it in order of increasing event time. Alternatively, an entity may wish to select its next event based on whether or not the event satisfies a particular set of conditions. The conditional selection of events is referred to as *event selection.* It allows an entity to receive events out of order or to simulate delays that are uninterruptable or that can only be interrupted by certain kinds of events. Events not selected at their scheduled event times are *deferred* for later reception. Deferring an event causes it to be received and enqueued on behalf of the receiving entity. Thus, even though an event is deferred, it is received in order of increasing receive time, as required by Virtual Time. *Sim++* provides four primitives for event selection: **sim_wait_for** for receiving events, **sim_hold_for** for simulating the passage of time, **sim_waiting** for counting deferred events, and **sim_select** for receiving deferred events.

The conditions by which an entity receives or defers events are specified using values known as *predicates*. Predicates are C++ objects that test the attributes of events. Generally, an application does not directly match predicates to events. Instead, predicates are passed as parameters to the event selection primitives which use them to determine whether an entity wishes to receive or defer a given event. Several commonly used predicates are defined by *Sim++*. Among these are the predicates **SIM_ANY** and **SIM_NONE** that match any and no events, respectively, and the predicate type **sim_type_p**. Instances of **sim_type_p** can be instantiated to match events of a specified type. As an example, the call

```
sim_type_p(ARRIVAL)
```

instantiates a predicate that matches only those events with the event type **ARRIVAL**. In addition to these pre-defined predicates and predicate types, new predicate types can be defined by individual applications to test any combination of event attributes.

The primitive **sim_wait_for** is similar to **sim_wait** except that the calling entity supplies an additional predicate argument that specifies the set of conditions an event must satisfy to be received by the entity. As an example, the call

```
sim_wait_for(sim_type_p(ARRIVAL), ev);
```

assigns the calling entity's next event of type **ARRIVAL** to the event object **ev**. The simulation time of the calling entity is advanced to the event time of the received event. All events of the calling entity that have not been received and that precede the selected event are deferred.

The primitive **sim_hold_for** is similar to **sim_hold** except that the calling entity supplies an additional predicate argument that specifies the set of conditions an event must satisfy to interrupt the simulated delay. As an example, the call

```
sim_hold_for(10.0, SIM_NONE, ev);
```

simulates an uninterruptable delay of 10.0 time units. All events of the calling entity that coincide with this delay are deferred. Alternatively, if a call to **sim_hold_for** is interrupted by an event satisfying a given predicate, only those events of the calling entity that have not been received and that precede the interrupting event are deferred.

The primitive **sim_waiting** is used to count deferred events satisfying a given predicate. As an example, the call

```
count = sim_waiting(SIM_ANY);
```

assigns to **count** the total number of deferred events of the calling entity.

The primitive **sim_select** is used to receive a deferred event satisfying a given predicate. As an example, the call

```
found = sim_select(sim_type_p(ARRIVAL), ev);
```

selects the calling entity's earliest deferred event of type **ARRIVAL**. If a deferred event of this type exists, it is assigned to the event object **ev** and **sim_select** returns the boolean result **true**. Otherwise, the value **SIM_NO_EVENT** is assigned to **ev** and **sim_select** returns **false**.

3.1.6 Type Checking

As previously noted, the body of a *Sim++* event can contain arbitrary, application-specific data. Unfortunately, to achieve this level of flexibility, the type information associated with data copied into and out of the body of events is lost. Without type checking, it is possible to copy data out of the body of an event into a variable whose type differs from that of the data. Errors of this type may go undetected and lead to erroneous simulation results. To address this problem, *Sim++* supports a simple form of run-time type checking that allows entities to include a type string in scheduled events. The type string is the type name of the data in the body of the event and is optionally specified as an argument to **sim_schedule**.

When accessing the body of the event, the receiving entity can optionally specify the type name of the variable into which it intends to copy the given data. *Sim++* compares the two type names to ensure that the type of the data in the body of the event is the same as the type of the variable into which the data will be copied. As an example, the call

```
sim_time time = sim_clock();
sim_schedule(..., &time, sizeof(time), "sim_time");
```

schedules an event with an event body containing a copy of the simulation time at which the event was scheduled. The type string **sim_time** corresponds to the type name of **time** and is included as an attribute of the event. In turn, the call

```
sim_wait(ev);
time = *(sim_time *) ev.body("sim_time");
```

is used to receive the scheduled event and safely copy the data out of the body of the event into the variable **time**. The type string **sim_time** is compared for equality to the type string attribute of the event object. The only way this type checking mechanism can fail is if both of the type strings specified in the two individual calls are identical but incorrect. As shown in the second of the examples above, this is highly unlikely because the type name of the variable into which the data will be copied appears alongside every occurrence of the corresponding type string specified when accessing the body of the event. (The type name is used in a *cast* operation that converts the untyped pointer returned by **body** to the type of the variable into which the data will be copied.)

3.1.7 Initialization and Execution of *Sim++* Simulations

The initialization phase is a special phase in the execution of a *Sim++* program that precedes the actual simulation, represented by the execution phase. The creation of entities and clusters and the initialization of global variables are restricted to the initialization phase. *Sim++* does not support the dynamic creation and destruction of entities nor does it permit

global variables to be modified during the execution phase. However, entities are permitted to read the values of global variables during the execution phase.

Since entities are not permitted to modify global variables, they cannot be used for communication among entities. Instead, global variables are typically used to store run-time parameters (e.g., the duration of a simulation) and entity identifiers denoting the entities in the simulation. They can also be used as data bases for static information (e.g., airport locations in an air traffic control simulation). The use of global variables reduces the amount of data that must be distributed to entities at creation and minimizes the size of entity states in that global data need not be duplicated within each entity's local state.

An application specifies the actions of the initialization phase by defining the function **sim_initialize**. To ensure that global variables are accessible to all entities on all processors of a parallel execution, identical invocations of **sim_initialize** are executed concurrently on all processors. In this way, the same values are assigned to the same global variables on every processor. However, to prevent **sim_initialize** from creating the same entities on every processor, *Sim++* only creates an entity on the one processor to which that entity has been assigned according to a user-supplied, run-time configuration file. It is impossible for any given invocation of **sim_initialize** executing on a given processor to tell which entities were or were not created on that processor. In this way, a simulation is indepedent of the number of processors on which it is executed as well as the way in which entities are mapped to processors.

Simulations create entities by calling the function **sim_create** and passing it the name of an entity class and an entity name. The entity class specifies the type of entity to be created. The entity name must be unique and it serves as the name of the newly created entity. Additional arguments to **sim_create** include an event type, and an optional data pointer and length. The event type, data pointer, and length are used to construct an

initialization event that is passed to the constructor of the newly created entity as its sole argument. **sim_create** returns an entity identifier denoting the newly created entity. As an example, the call

```
auto1_id = sim_create("automobile", "auto1", INIT);
```

creates an instance of entity class **automobile** called **auto1**. **INIT** is assumed to be a constant of type **sim_type**. The return value of **sim_create** is assigned to the entity identifier **auto1_id**. If **auto1_id** is a global variable, then all entities can use this identifier during the execution phase to schedule events for **auto1**.

The execution phase follows the initialization phase and encompasses both the initialization and execution of entities. The initialization of an entity is represented by its constructor. The execution of an entity is represented by its body. At the beginning of the execution phase, all entity constructors begin executing at simulation time 0.0. When an entity's constructor returns, its body is automatically executed. When the entity's body returns, the entity terminates. Alternatively, an entity also terminates if it waits for an event that never occurs. The simulation terminates when all entities have terminated.

3.1.8 Clusters for Shared Memory among Entities

A cluster is a group of entities that can share memory and directly access each other's member variables and functions. Clusters are intended to model physical systems that contain multiple, independently acting components whose interactions are too frequent to be efficiently represented by scheduling and receiving events. The use of clusters may reduce the number of events required for modeling such systems but at the expense of whatever parallelism the clustered entities could achieve by executing concurrently on multiple processors.

Clusters of entities can only be created during the initialization phase. Once a cluster has been created, all entities within that cluster remain so for the duration of the simulation. The functions **sim_begin_cluster** and **sim_end_cluster** are used to create clusters of entities. A call to **sim_begin_cluster** *opens* a new cluster while a call to **sim_end_cluster** *closes* the current cluster. All entities created while a cluster is open become part of that cluster and can directly access and share memory with other entities within that same cluster. Entities created when there is no open cluster are completely independent of other entities and cannot communicate or synchronize with other entities except by scheduling and receiving events. The following code fragment creates a cluster of two entities, **auto1** and **driver1**:

```
sim_begin_cluster();
auto1_id = sim_create("automobile", "auto1", INIT);
driv1_id = sim_create("driver", "driver1", INIT);
sim_end_cluster();
```

An entity accesses the states of other entities in its cluster in the same way that it accesses a dynamically allocated object within its own state. As such, the entity must first obtain a pointer to the entity whose state it intends to access. The function **sim_entity_ptr** is used to obtain a pointer to an entity given the entity's class name and entity identifier. As an example, the call

```
p = (automobile *) sim_entity_ptr("automobile", auto1_id);
```

assigns to **p** a pointer to the entity denoted by **auto1_id**. The given class name is used for type checking similar to that described for events. Only entities within the same cluster can obtain pointers to one another. **sim_entity_ptr** returns a null pointer if the calling entity is not part of a cluster containing the entity denoted by the given entity identifier. Given a pointer to entity **auto1**, the call

```
p->Speed
```

might be used by entity **driver1** to determine the current speed of the automobile.

Although clustering allows entities to communicate with relatively little overhead, the entities within a cluster must still schedule and receive events in order to synchronize with one another and to interact with other entities outside of the cluster.

3.1.9 Preventing Side Effects of Causality Errors

Sim++ refers to the side effects of causality errors as *transient errors*. Many such errors are trapped by *Sim++*. For example, if a simulation primitive is invoked with erroneous arguments (e.g., a negative delay to **sim_schedule**), *Sim++* invokes the error routine **sim_error** which blocks the calling entity until the entity rolls back or until the error is committed by the advance of GVT. In the former case, the error is transient and will be corrected. In the latter case, the error is real and the execution of the simulation is aborted. Real errors are typically detected and eliminated during the development of a simulation executing on the sequential simulator.

Not all transient errors are trapped by *Sim++*. Some must be trapped by the application itself. Most of these can be prevented by using the type checking mechanism described in Section 3.1.6 when moving data into and out of the body of events. The type checking mechanism invokes **sim_error** whenever a type conflict occurs. A causality error can lead to a transient type conflict when an entity expects to receive one type of event and actually receives another. Based on its expectations, the entity may try to copy data out of the body of the event that differs in type from the data actually in the body of the event. The type checking mechanism ensures that such transient type conflicts are handled transparently. For other transient errors, the application must detect them and call **sim_error** explicitly. For example, if an integer in the body of an event is to be used as divisor in an arithmetic operation, the application should first check that the integer is non-zero. The application can similarly check the validity of array indices and other sources of transient errors.

Although many of the techniques for preventing transient errors are additional burdens imposed on the user by Time Warp, many of these techniques are also appropriate for developing correct sequential simulations. For example, all of the errors noted here and in Chapter 2 as potential side effects of causality errors can occur in sequential simulations as well. What does differ is that error checks can be removed from sequential simulations for efficiency once the simulation is thought to be correct. Since transient errors can occur even in correct simulations executing on Time Warp, transient error checks are always required unless the resulting errors can be trapped by the language or Time Warp.

3.1.10 Language Support for Time Warp

All of the language features of *Sim++* presented so far are based on Virtual Time or extensions of Virtual Time. As such, they are not specific to Time Warp. Specifically, entities and events correspond to Virtual Time processes and messages, and the simulation and event selection primitives are modeling abstractions that can be implemented entirely in terms of an entity's ability to schedule and receive events [Lom88b]. The initialization phase and clusters are extensions of Virtual Time that reduce overall memory usage and event communication overhead. As such, they are appropriate for any implementation of Virtual Time.

This section presents two *Sim++* language features designed specifically for Time Warp: *write-locked memory* and *interface entities*. Write-locked memory is intended to reduce Time Warp state checkpoint overhead. Interface entities allow Time Warp simulations to interact directly with architecture-specific facilities not supported by *Sim++*. These facilities are important for two reasons. First, the existence of these facilities demonstrates shortcomings in Time Warp for which no viable, transparent solutions

currently exist. Second, the *Sim*D language interface proposed in Chapter 5 relies on write-locked memory to transparently optimize the *Sim*D implementation.

Write-locked memory allows an entity to *lock* and *unlock* portions of its state. By locking a portion of its state, an entity guarantees to the Time Warp implementation that it will no longer modify that portion of its state. Like global variables, the entity may subsequently read but not modify the values of data structures in that portion of its state. As a result, it becomes unnecessary to include that portion of the state in subsequent entity state checkpoints. By unlocking a previously locked portion of its state, an entity informs the Time Warp implementation that it intends to subsequently modify some or all of that portion of its state. Once unlocked, that portion of the state is once again subject to state checkpoints. The function **sim_write_lock** is used to lock dynamically allocated data structures. The ability to unlock a portion of an entity's state is not yet supported. No run-time error checking is performed to ensure that an entity does not attempt to modify write-locked memory. This facility relies instead on the discipline of the user.

Interface entities are a special type of entity that can interact directly with architecture-specific facilities not supported by *Sim++*. This entity type is intended to supplement the formatted input and output facilities provided by *Sim++*. Interface entities are derived from class **sim_interface_entity**. Unlike entities derived from class **sim_entity**, interface entities do not execute optimistically. Instead, interface entities only execute when GVT is equal to their entity simulation time. As a result, the events these entities schedule and receive, and the actions that they take are always known to be correct and not subject to rollback. Unfortunately, interface entities tend to slow the advance of GVT, potentially slowing the execution of an entire simulation. As such, few if any interface entities are used in most simulations.

3.2 Comparing *Sim++* to Other Languages

This section compares *Sim++* to other parallel simulation languages and systems, focusing primarily on alternative solutions to issues addressed by *Sim++* or to unique capabilities not supported by *Sim++*. In addition to the languages surveyed in this chapter, other noteworthy systems include the Time Warp Operating System (TWOS) [Jef87] and Tipc [Ung90]. Simulations written for TWOS have achieved speedups of an order of magnitude or more across a variety of application domains [Ebl89, Hon89, Pre89b, Wie89]. However, TWOS provides only a minimal interface for the development of event-oriented simulations [Jef87]. Tipc is a multi-lingual Time Warp implementation that augments existing sequential programming languages with primitives for sending and receiving timestamped messages. Each Tipc process is an independently executing sequential program within a Tipc system.

3.2.1 Extended Virtual Time

The simulation and event selection primitives provided by *Sim++* are based on a parallel simulation language called Extended Virtual Time defined by Lomow [Lom88b]. Extended Virtual Time combines the elements of Virtual Time and process-oriented simulation to provide a language for process-oriented parallel simulation. Extended Virtual Time defines primitives for scheduling, cancelling, receiving, deferring, and ignoring events, and for simulating the passage of time. Two strategies for implementing Extended Virtual Time are defined: an *application* approach and an *integrated* approach. The application approach implements Extended Virtual Time as an application layer above Virtual Time, implementing all of the language primitives solely in terms of a process' ability to send and receive messages. This approach incurs excessive overhead because it is restricted by the

semantics of Virtual Time. The integrated approach implements Extended Virtual Time by integrating the language primitives with an underlying Time Warp synchronization mechanism. This approach requires modifications to the Time Warp implementation but is shown to reduce execution time, memory usage, and communication overhead in a variety of benchmark simulations. Currently, *Sim++* is implemented using the application approach. Given the close correspondence between *Sim++* primitives and Extended Virtual Time primitives, the performance optimizations in the integrated version of Extended Virtual Time can also be implemented in *Sim++*. This is significant in that it allows simulations to be described using primitives based on Virtual Time while allowing those primitives to be transparently optimized for Time Warp. The remainder of this section summarizes these optimizations.

Deferred events in *Sim++* and Extended Virtual Time result when an entity receives events out of order based on whether or not the events satisfy a given predicate. All events not received at their scheduled event times are deferred on behalf of the entity for which they are scheduled. In this way, an entity need not explicitly receive and buffer events that it does not wish to process as scheduled. This same effect can be achieved through event selection, allowing the language to implicitly buffer deferred events on an entity's behalf. The application version of Extended Virtual Time maintains a list of deferred events as part of each entity's state. The language defers an event by adding a copy of the event to the entity's deferred event list. My own experience with *Sim++* suggests that tens of deferred events per entity are not uncommon in some queueing models. Since each event can be hundreds or more bytes in length, the size of an entity's state can be thousands or tens of thousands of bytes in size depending on the number of events the entity has deferred. The integrated version of Extended Virtual Time integrates the deferred event list with the input queue of the underlying Time Warp process. This approach requires that the semantics of

the input queue be modified to represent not only received and unreceived messages, but also deferred messages. The effects of rollback and fossil collection on the input queue must be modified to accomodate these new semantics. The advantage of the integrated approach is that it eliminates the deferred event list from an entity's state. This optimization takes advantage of the fact that, as long as an event is deferred, it is not subject to modification and therefore need not be included in entity state checkpoints.

Both *Sim++* and Extended Virtual Time provide primitives for simulating the passage of time. One way to represent a simulated delay is for an entity to schedule an event for itself in the future and then wait to receive that event. This is the approach used by the application version of Extended Virtual Time. The event is scheduled and received implicitly by the language primitive and is transparent to the application. Associated with each such event is a message in the underlying Time Warp process' input queue, an anti-message in the output queue, and a copy of the entity's state in the state queue when the event is received. The events used to implement simulated delays differ from other kinds of events in that they are always scheduled by an entity for itself and they contain no application-specific information except that they mark the end time of the delay. The integrated version of Extended Virtual Time uses this knowledge, combined with a scan of the current state of the input queue, to eliminate the need for both the message and the anti-message. A copy of the entity's state is still required, however. In effect, the message and anti-message are implied by the state in the state queue. Specifically, if a state exists for which there is no corresponding message in the input queue, then the missing message represents the end of a simulated delay. The receive time of the missing message can be determined from the timestamp on the state. As a result, the simulated delay can be implemented without requiring that the entity schedule an event for itself. This optimization eliminates the execution overhead associated with allocating and inserting a message and

anti-message into the input and output queues, and deallocating the message and anti-message during fossil collection. In addition, this approach requires less memory since it allocates fewer messages.

Both *Sim++* and Extended Virtual Time allow entities to cancel previously scheduled events. An event can only be cancelled at a simulation time earlier than its scheduled event time (i.e., before the event occurs). The application version of Extended Virtual Time cancels an event by transparently scheduling a *cancel* event to preempt the original event. The cancel event is scheduled for the same entity as the original event but at an earlier event time. In this way, an entity always receives a cancel event before the event that it preempts. The cancel event includes an event identifier denoting the event being cancelled. The receiving entity stores the event identifier in a cancelled event list in the entity's state. Each application event received by an entity is compared against the event identifiers in that entity's cancelled event list. If a matching event identifier is found, the newly received event is ignored and the event identifier is deleted from the cancelled event list. These actions are completely transparent to the application. The integrated version of Extended Virtual Time uses anti-messages to implement event cancellation. Specifically, when an entity cancels an event, the anti-message for that event is sent to annihilate the corresponding message in the receiving entity's Time Warp input queue. Changes to rollback, fossil collection, and the output queue of Time Warp processes are required to correctly implement event cancellation using anti-messages. For example, it must be possible to uncancel previously cancelled events in case of a rollback. Lomow proposes implementations of this optimization for both lazy and aggressive cancellation. The advantage of the integrated approach is that it eliminates the cancelled event list from an entity's state and, since the integrated approach uses an event's own anti-message for cancellation, no additional memory is required to maintain an anti-message for a cancel

event. However, in the integrated approach, an event can only be cancelled by the entity that scheduled the event since only it has the corresponding anti-message. In the application approach, the entity that schedules the cancel event need not be the same as the entity that scheduled the original event. Lomow et al [Lom91] describe a variation of the integrated approach with the same flexibility as the application approach.

3.2.2 ModSim

ModSim [Bry89, Wes88b] is a process-oriented, discrete-event simulation language designed to support the development and execution of large simulations. ModSim was developed for the U.S. Army and is based on a prototype design known as the Language for Concurrent Simulation (LCS) [Wes85]. One of the initial design objectives was that simulations written in LCS be capable of executing on Time Warp. Currently, ModSim executes in a number of sequential environments, but has not yet been completely implemented on Time Warp.

ModSim is an object-oriented language with a syntax similar to Modula-2. Classes in ModSim are referred to as *object types* while member variables and member functions are referred to as *fields* and *methods*, respectively. Like *Sim++*, ModSim defines a special type of object, called **ProcessObj**, to represent processes. ModSim processes are unique in that each process can model multiple, independently acting components. Each component is represented by a time-elapsing method referred to as an *activity*. From the user's perspective, each activity is an independently executing method. The coordination and execution of multiple activities within a single process is managed transparently by the process itself. As such, a ModSim process is similar to a cluster of entities in *Sim++*. Unlike the entities in a cluster, however, activities can be created and destroyed

dynamically throughout the course of a simulation. The same effect can be achieved in *Sim++*, but requires dynamic creation and destruction of entities.

A major representational difference between ModSim processes and *Sim++* clusters is the way in which the two constructs model the states of multiple, independently acting components. A ModSim process is a single object with one or more independently executing methods. A *Sim++* cluster is a group of one or more independently executing entity objects. As such, a ModSim process is more appropriate for modeling multiple, independently acting components that share a single state. A cluster is more appropriate for modeling multiple, independently acting components where each component has a well-defined state. This difference is primarily one of representational convenience and conceptual clarity since both approaches are merely data abstractions for the modeled state.

ModSim processes interact using **ASK** and **TELL** statements to invoke each other's **ASK** and **TELL** methods. **ASK** methods are synchronous, meaning that the calling process waits for the called method to complete before continuing to execute. In addition, **ASK** methods are non-time-elapsing. As such, the simulation time of the calling process is unaffected by a call to an **ASK** method. As an example, the call

```
ASK Auto1 TO IncreaseSpeedBy(10.0)
```

invokes the **ASK** method **IncreaseSpeedBy** of the process denoted by **Auto1**. Alternatively, an asynchronous **TELL** statement can be used to schedule a method to execute without requiring that the calling process wait for the method to complete. As an example, the call

```
TELL Ship1 TO SailTo("Alaska") IN 5.0
```

schedules the **TELL** method **SailTo** to begin executing in 5.0 time units. The scheduled method will execute as an independent activity in the process denoted by **Ship1**. Upon executing the **TELL** statement, the calling process continues to execute. **TELL** methods

can execute a **WAIT** statement that allows them to simulate the passage of time, wait for the completion of other tell methods, or synchronize with other methods using special objects known as *triggers*.

ASK and **TELL** statements are similar to event scheduling in *Sim++*, except that they define process interactions in terms of object methods and parameters. This approach has significant appeal since it is completely type-safe and adds few additional features to the object-oriented interface to support process interactions. For example, **ASK** methods are also used to define interactions among non-process objects. Thus, the call

```
ASK Auto1 TO IncreaseSpeedBy(10.0)
```

is identical regardless of whether **Auto1** denotes an object in the calling process' state or another process. This approach does have a drawback in the context of parallel simulation since the cost of object and process interactions may differ by several orders of magnitude. By using the identical syntax for both types of operations, these differences are not explicit to the user.

Although designed for Time Warp, ModSim contains a number of language constructs not suited to parallel execution. Chief among these are interactions through shared and global memory. The developers of ModSim concede that language constructs such as global variables and **ASK** methods for process interactions should be avoided when executing on Time Warp [Bry89]. **ASK** methods are deemed inefficient because they block the calling process for the time required to execute the **ASK** method in a remote process.

3.2.3 Languages for Conservative and Optimistic Systems

A number of researchers have proposed parallel simulation languages intended for execution on both conservative and optimistic implementations of Virtual Time. Because of

the fundamental differences between conservative and optimistic systems, all of these languages include facilities specific to individual implementations. In some cases these facilities simply enhance performance while in others they are mandatory for correctness.

Maisie [Bag90] extends the C programming language with constructs for message-based simulation. The central construct of Maisie is the **wait** statement. **wait** provides functionality similar to the event selection primitives in *Sim++*. It allows Maisie entities to receive and defer messages, and simulate the passage of time. **wait** is also used to transparently extract information from an application to support the underlying implementation of Virtual Time. For example, in certain limited circumstances, **wait** can be used to automatically calculate lookahead for conservative implementations. **wait** can also be used for optimizations of Time Warp similar to those defined for Extended Virtual Time to reduce the overhead associated with buffering deferred messages. In spite of these transparent optimizations, additional information must still be specified explicitly by the application. For the Chandy-Misra approach, entities must execute a system call for each entity identifier they distribute to other entities. The implementation uses this information to maintain a connectivity graph of the entities from which each entity can receive messages. When a Maisie program executes on Time Warp, entities can send special *probe* messages that are used for optimized, read-only queries of other entities' states. This optimization was discussed at length in Chapter 2.

Yaddes [Pre89a] is a simulation specification language based on the C programming language. The purpose of Yaddes is to evaluate the performance of different optimistic and conservative implementations of Virtual Time. For efficiency, Yaddes uses an event orientation rather than a process orientation. Yaddes programs are defined in terms of logical processes, input channels, and output channels. As such, they conform to the Chandy-Misra model for parallel simulation. This approach sacrifices some of the

flexibility of Virtual Time and Time Warp for a programming model that can be supported by both conservative and optimistic systems. For example, Yaddes programs must provide explicit information to prevent deadlock in conservative systems. However, the system calls that provide this information have no effect in sequential and Time Warp environments. As such, a Yaddes program written for a conservative implementation will execute sequentially and on Time Warp without source code modifications. The reverse is not true, however.

Common Programming Structure (CPS) [Abr89] is a C++ library for message-based simulation. Like Yaddes, the purpose of CPS is to evaluate the performance of different optimistic and conservative implementations of Virtual Time. CPS also uses an event orientation. Nevertheless, CPS has many features similar to *Sim++*. Most notably, initialization and execution phases, read-only global variables, read-only state (similar to write-locked memory), and a configuration file for mapping processes to processors. As with *Sim++*, the configuration file eliminates the need to specify the number of processors or the mapping of processes to processors directly in the application, making CPS programs transparently scaleable. One language feature of CPS not present in *Sim++* is a global directory of processes. The directory can be queried for lists of processes of a given type. The directory eliminates the need for an application to create and maintain its own arrays of process identifiers. The directory can also be used by separately compiled and initialized program modules to determine the number and types of other processes in a simulation.

3.3 Language Design Criteria for Virtual Time

This section outlines key language design criteria for Virtual Time. These criteria are based on the fundamental differences between Virtual Time and sequential simulation, and on the characteristics of the languages surveyed in this chapter. These criteria are in addition to basic design goals such as simplicity, expressiveness, and extensibility. Several of these criteria were first proposed by Abrams and Lomow [Abr90].

Parallel Efficiency. Unfortunately, no decomposition techniques exist that can enforce performance-oriented design considerations. However, languages for Virtual Time can encourage appropriate programming practices by providing language constructs suited to parallel execution. Of the languages surveyed, Extended Virtual Time most satisfies this language design criterion. Many of its primitives can be integrated with Time Warp, resulting in reduced execution time, memory usage, and communication overhead. At the same time, these primitives provide capabilities common to process-oriented simulation. The simulation and event selection primitives provided by *Sim++* are based on Extended Virtual Time and can be similarly optimized. A subset of these optimizations has also been proposed for Maisie.

Explicit Costs. In order for users to develop efficient parallel simulations, they must be aware of the costs associated with various alternative implementations of a simulation model. Specifically, if an operation in a parallel simulation differs significantly in cost from the equivalent operation in a sequential simulation, then those operations should appear obviously different to the user. This would suggest, for example, that languages for Virtual Time should not support interactions through shared memory where such interactions are implemented using message-passing. One alternative to this is the approach used by *Sim++* which allows shared memory interactions, but only for entities in the same

cluster. Of the languages surveyed, ModSim least satisfies this language design criterion. For example, the syntax of ModSim process interactions is identical to that of object interactions. On a parallel processor, the cost of these operations typically differs by several orders of magnitude. This language design criterion is based on the current state of the art in parallel processors. Improved hardware support to reduce message-passing overheads could reduce the need for this criterion.

Determinism. Given the same input, a simulation should produce the same results regardless of its run-time configuration. Determinism is crucial for repeatable simulation results and repeatable errors during debugging. All of the languages surveyed are deterministic as long as determinism is provided by the underlying implementation of Virtual Time.

Type-Safety. Even when simulations execute deterministically, the existence of multiple, concurrently executing processes as components of a single simulation make parallel simulations more difficult to debug than sequential simulations. As such, the ability of type-safety to promote the development of defect-free simulations is even more important for parallel simulation than for sequential simulation. Of the languages surveyed, ModSim is most type-safe with compile-time type checking for both object and process interactions. C++ provides compile-time type checking for object interactions, but not for the event and message-based interactions of *Sim++*, Extended Virtual Time, and CPS. *Sim++* provides optional run-time type checking for events.

Transparency of the Implementation. This criterion is based on the conclusion in Chapter 2 that it is inappropriate to adapt modeling practices and language design to accomodate specific implementations of Virtual Time. Users should instead concentrate on the requirements dictated by Virtual Time, while the underlying implementation should be transparent to the application. None of the languages surveyed fully satisfy this criterion,

either because they rely to some degree on the discipline of the user, or because they provide explicit language support for the implementation of Virtual Time. This is due to the current state of the art in parallel processors and parallel simulation. Depending on the implementation, transparency may only be possible with compiler and hardware support.

Transparent Scaleability. Simulations should be executable on varying numbers of processors and with different mappings of processes to processors without source code modifications. Typically, this requires some form of run-time configuration file. The advantage of this approach is that it is not necessary to modify and recompile a simulation simply to change its run-time configuration. And, by making the run-time configuration transparent to the application, the application cannot inadvertently violate determinism by executing different application code for different run-time configurations. Like determinism, transparent scaleability is generally provided by the implementation of Virtual Time.

Portability of Applications. Application programs should be capable of executing sequentially or in parallel on multiple operating systems and architectures without source code modifications. Together, transparent scaleability and portability of applications simplify the development of parallel simulations because it is possible to develop software on a workstation in a well supported environment while ensuring that the simulation can be moved, without source code modifications, to a parallel processor for production runs. *Sim++* interface entities are an example of a language feature that violates this design criterion. Interface entities are designed specifically for interactions with architecture-specific facilities not supported by *Sim++*. Since *Sim++* already provides facilities for formatted input, output, tracing, and error reporting, the majority of simulations do not require their own interface entities. As a result, interface entities can be seen as a reasonable exception to this design criterion.

Enforced Restrictions. Virtual Time imposes special restrictions on parallel simulations not present in sequential simulations. For example, interactions through shared memory are restricted to varying degrees in most of the languages surveyed. However, this and other restrictions are not enforced by these languages, relying instead on the discipline of the user. Violations of these restrictions that go undetected may lead to erroneous simulation results. The biggest obstacle is that many of these restrictions cannot be enforced without compiler support. In some cases, the restrictions cannot be enforced until run-time, thereby increasing execution overhead. An example of the latter is read-only global variables in *Sim++* and CPS, which can only be modified during the initialization phase of a simulation.

3.4 Summary

This chapter surveyed six parallel simulation languages, from which were developed eight language design criteria for Virtual Time: parallel efficiency, explicit costs, determinism, type-safety, transparency of the implementation, transparent scaleability, portability of applications, and enforced restrictions. None of the languages surveyed fully satisfies all of these criteria. In some cases, this is justified in that the purpose of languages such as CPS and Yaddes is to evaluate the performance of different optimistic and conservative implementations of Virtual Time. As such, these languages provide an interface adequate for researchers who wish to focus on the performance of Virtual Time implementations, rather than on language design for parallel simulation. In contrast, *Sim++* and ModSim are both intended for the development of real simulations by industry and government with little or no experience in parallel simulation. In the case of ModSim, portions of the language will need to be redesigned to eliminate inherently sequential constructs. In

addition, both languages will require compiler support to enhance the transparency of the underlying Time Warp implementation and to enforce restrictions. Currently, *Sim++* is implemented as a library in C++. A compiler and parallel implementation of ModSim are under development.

Chapter 4

Limitations of Existing Languages

This chapter investigates various difficulties associated with using *Sim++* to model common types of entity interactions. To illustrate these difficulties, a basic queueing model and several simple extensions to that model are presented, as well as a facility for resource competition among entities. The difficulties encountered are due primarily to an inability to extend or restrict the behaviour of the existing simulation and event selection constructs according to the specific requirements of the application. These shortcomings are not unique to *Sim++*. They can also be found in Extended Virtual Time and Maisie, which have constructs similar to the simulation and event selection primitives of *Sim++*. The shortcomings illustrated in this chapter were the primary motivation for the *Sim*D language proposed in Chapter 5. No further evaluation of ModSim was attempted since ModSim is still being revised for use with Virtual Time and Time Warp.

4.1 The Barber Shop Queueing Model

The barber shop is a simple queueing system in which customers arrive at randomly distributed times and wait for service. The barber shop has one barber capable of providing up to three services for each customer: a shave, a hair wash, and a haircut. All customers are served in the order they arrive.

This model can be implemented in *Sim++* using two entities: a *source* entity to generate customers arriving at the barber shop and a *barber* entity to serve customers. Customers

```
1     void source::body()
2     {
3        while (sim_clock() < Duration) {
4           sim_hold_for(Interarrival_time.sample(), SIM_NONE, ev);
5           sim_schedule(Barber_id, 0.0, CUSTOMER);
6        }
7     }
8
9     void barber::body()
10    {
11       while (true) {
12          if (sim_waiting(sim_type_p(CUSTOMER)))
13             sim_select(sim_type_p(CUSTOMER), ev);
14          else
15             sim_wait_for(sim_type_p(CUSTOMER), ev);
16
17          if (customer wants shave)
18             sim_hold_for(Shave_time.sample(), SIM_NONE, ev);
19          if (customer wants hair washed)
20             sim_hold_for(Wash_time.sample(), SIM_NONE, ev);
21          if (customer wants hair cut)
22             sim_hold_for(Cut_time.sample(), SIM_NONE, ev);
23       }
24    }
```

Figure 4.1: C++ Code for Source and Barber Entities

are represented by events scheduled by the source entity for the barber entity. C++ code for the body of each entity is shown in figure 4.1 (line numbers in the following description correspond to those in figure 4.1).

The source entity repeatedly executes the following sequence of activities. The source entity calls **sim_hold_for** (line 4) to model the interarrival time between successive customers. The delay representing this interarrival time is generated by the distribution object **Interarrival_time.** When called, the member function **sample** of this object returns a value drawn from the random number distribution that the object represents. The predicate **SIM_NONE** specifies that the given delay is uninterruptable. Since no events are ever scheduled for the source entity, the call to **sim_hold_for** would not be interrupted anyway and the choice of predicate is arbitrary. Upon expiry of the simulated delay, the call to **sim_hold_for** returns and the source entity schedules an event for the

barber entity representing the arrival of the next customer (line 5). It is assumed that **Barber_id** is an entity identifier denoting the barber entity and **CUSTOMER** is a constant of type **sim_type** representing the arrival of a customer. The event is scheduled with a delay of 0.0.

As noted, the barber provides up to three services for each customer. To represent the time required for a barber to provide one of these services to a customer, the barber entity calls **sim_hold_for** (lines 18, 20, and 22) signifying that it wishes a specified amount of simulation time to elapse. The specified delays are generated by the distribution objects **Shave_time**, **Wash_time**, and **Cut_time**. In each call to **sim_hold_for** in the barber entity, the predicate **SIM_NONE** specifies that the given delay is uninterruptable. **sim_hold_for** defers any events scheduled for the barber entity with an event time that coincides with one of these simulated delays. Since only events representing arriving customers are ever scheduled for the barber entity, deferred events implicitly represent the queueing of customers arriving for service.

Each time the barber entity finishes serving a customer, it immediately begins serving the next customer, if any. The barber entity terminates when there are no more customers to serve. In an actual barber shop, whenever the barber finishes serving a customer, the next customer is either the customer at the beginning of the queue of waiting customers or, if no customers are currently waiting, the barber waits for the next customer to arrive. To represent this in *Sim++*, the barber entity first tests if there are any deferred customer events by calling **sim_waiting** (line 12). If one or more such events exist, the first of these is selected by calling **sim_select** (line 13). If there are no deferred customer events, the barber entity calls **sim_wait_for** (line 15) to await the next customer event scheduled by the source entity.

The barber entity specifies the predicate **sim_type_p(CUSTOMER)** as an argument to each of the primitives **sim_waiting, sim_select,** and **sim_wait_for.** This predicate matches only those events whose type is that of **CUSTOMER.** Since customer events are the only kind of event scheduled for the barber entity, the predicate **SIM_ANY** could also have been specified. However, the former is more specific, thereby enhancing the readability of the resulting code.

4.2 Limiting the Size of the Barber Shop Wait Queue

This section investigates an extension to the barber shop model that places an upper limit on the number of customers that can enqueue for service at any one time. In this extended model, an arriving customer can only enqueue for service if there is room in the queue; otherwise, the customer is turned away.

The most natural approach to the implementation of this extended model would mirror the preceding model description, as shown in the following code fragment:

```
for each new customer event
   if (sim_waiting(sim_type_p(CUSTOMER)) < Limit)
      defer customer event
   else
      ignore customer event
```

Unfortunately, the queueing of customer events via event deferral cannot be encoded explicitly by the user. Instead, event deferral is implicit and no application-specific restrictions or extensions to the underlying deferred event list are possible. In addition to limiting queue sizes, it may be desirable to collect queueing statistics about deferred events, or to queue events in different lists according to application-specific criteria. In the extended barber shop model, the only solution is for the barber entity to circumvent the language queueing facility and explicitly receive all arriving customer events at the application layer and then buffer those events in the application or ignore them when the

specified limit has been reached. By circumventing the language queueing facility, the performance optimizations possible for Extended Virtual Time are lost.

4.3 A Variation on Barber Shop Termination

This section investigates another extension to the basic model in which the barber shop is locked at the end of each day to prevent more customers from entering the shop. Customers already waiting in the shop when it closes are served to completion, after which the barber can leave.

At least two implementations alternatives of this extended model can be identified:

1. The source entity can be modified to cease generating customers once the barber shop is locked. In this way, no additional customer events beyond those already deferred by the barber entity will be scheduled.
2. The barber entity can process only those customer events whose event time precedes the point at which the barber shop was locked. Additional customer events generated by the source entity will be automatically deferred by the barber entity but will be ignored since they represent customers that arrived after the barber shop was locked.

The first implementation alternative is inappropriate for two reasons. First, the proposed modification to the source entity does not reflect the model as outlined. Specifically, locking the barber shop does not imply that no more customers will arrive at the barber shop, only that they will no longer be served. Indeed, the barber entity may wish to count the number of customers arriving after closing to determine how many customers are being turned away. Second, the proposed modification does not reflect the

modularity of the model's decomposition. Specifically, since it is the behaviour of the barber shop that has been altered for this model, then it is the corresponding barber entity that should reflect those changes.

Since the second implementation alternative exhibits neither of the aforementioned shortcomings, it would seem to be the preferable alternative. This alternative is identical to that of the basic model in figure 4.1 except that the code for serving a customer is only executed if the customer arrived before closing time:

```
if (ev.event_time() < Closing_time) {
   if (customer wants shave) ...
   if (customer wants hair washed) ...
   if (customer wants hair cut) ...
}
```

Although this implementation is an accurate, modular representation of the model, it requires that the barber entity explicitly receive all customer events, even if they are to be ignored. As such, the structure of the barber entity body no longer represents only the serving of customers, but rather the combined tasks of serving customers and ignoring customers after closing time. Also, the barber entity's deferred events represent both waiting customers as well as late-arriving customers that are to be turned away. Although the barber shop model is sufficiently simple that the combination of the above tasks can be realized without significant loss of clarity, many realistic models perform many such distinct tasks that, if similarly integrated within the structure of a single function, would be significantly more difficult to implement and debug.

It can be argued that if a component of a physical system performs multiple, distinct tasks, then that component should be represented by multiple entities. For example, the barber could be represented by a cluster of two entities, one for each task. The additional entity represents the entrance to the barber shop, receiving all arriving customer events generated by the source entity and discarding those with an event time greater than or equal

to closing time. The remaining events are rescheduled for the barber entity within the same cluster. This approach has the added implementation overhead of an additional entity type and the added run-time overhead of scheduling up to two events for each arriving customer.

Rather than explicitly receiving and discarding events to be ignored, Extended Virtual Time defines an *ignoring* facility that allows an entity to specify the set of events that it does not want to receive and that it wants to treat as if they never arrived. The **ignore** primitive provided by Extended Virtual Time is declared as follows:

```
void ignore(predicate &p);
```

ignore is a non-blocking primitive whose predicate argument **p** specifies the set of events to be ignored. After an entity calls **ignore(p)**, all subsequent calls to other simulation primitives will automatically intercept and discard events satisfying **p** when they occur. From the viewpoint of the application, however, it is as if the events never arrived. A subsequent call to **ignore** can override **p**, allowing the calling entity to ignore different events at different times throughout the course of the simulation. Using this facility, there would be no need to modify the structure of the barber entity to cope with late-arriving customers. Instead, the barber entity could call **ignore** as part of its entity initialization, as follows:

```
ignore(Late_customers);
```

where **Late_customers** is a predicate that matches customer events with an event time greater than or equal to the barber shop's closing time. The body of the barber entity, as shown in figure 4.1, would be unaffected.

Unlike other predicates used in examples thus far, **Late_customers** is application-specific and cannot be represented using predefined predicate types provided by *Sim++*.

```
1     class late_customer_p : public predicate {
2        sim_time Closing_time;
3     public:
4        late_customer_p(sim_time closing_time) {
5           Closing_time = closing_time;
6        }
7
8        boolean match(sim_event &ev) {
9           if (ev.type() == CUSTOMER &&
10                ev.event_time() >= Closing_time)
11             return true;
12          else
13             return false;
14       }
15    };
```

Figure 4.2: C++ Declaration for Late Customer Predicate

Instead, application-specific predicates are derived from a special predicate base class, **sim_predicate**. These application-specific predicates are created and used in the same manner as the predefined predicate types provided by *Sim++*. Figure 4.2 shows the declaration of a predicate class, **late_customer_p**, for matching events representing customers who arrive after closing at the barber shop (line numbers in the following description correspond to those in figure 4.2). Every predicate class has a member function **match** (lines 8-14) that takes an event as its only parameter. This function tests the attributes of the event and returns **true** if the event's attributes satisfy the conditions of the predicate; otherwise, the function returns **false**. **match** is never actually invoked by the application. Instead, instances of a predicate class are passed as parameters to *Sim++* simulation primitives and they automatically invoke this function to test the attributes of events. Since instances of a predicate class are objects, they can be parameterized when they are created. Class **late_customer_p** has one such parameter, **closing_time** (line 4). When an instance of **late_customer_p** is created, **closing_time** is assigned to the member variable **Closing_time** (line 5). In this way, the object *remembers* the closing time of the barber shop. As an example, the declaration

```
late_customer_p Late_customers(100.0);
```

creates an instance of class **late_customer_p** called **Late_customers** that matches any customer event with an event time greater than or equal to simulation time 100.0. A call to the member function **match** of this predicate compares the event time of the given event against the remembered closing time. By representing closing time as a parameter to the predicate, the closing time of the barber shop need not be specified explicitly within the definition of **match**. Instead, it can be specified as an input parameter to the simulation, and varied from one run to the next, thereby varying the number of customers that will be served or ignored.

Ignoring is similar in strengths and weaknesses to event deferral. Like event deferral, ignoring provides an enhanced abstraction for discrete event modeling by automatically manipulating events according to application-specific predicates. Also like event deferral, no application-specific restrictions or extensions to the ignoring facility are possible. In other words, there is no way to count or otherwise manipulate ignored events. In the barber shop model, for example, it may be desirable to forward ignored customers events to another barber entity to represent a model in which late-arriving customers seek out another barber shop with longer operating hours. In addition, although ignoring can be used to solve the extended barber shop model, the declaration of the application-specific predicate is significantly longer than the code changes required for the barber entity to explicitly receive and discard unwanted events.

4.4 Adding Statistics Reporting to the Barber Shop Model

This section investigates an extension to the barber shop implementation for generating statistics reports at regular intervals. An additional *report* entity is used that schedules a report event for the barber entity for each statistics interval. The barber entity responds to

each such event by printing a statistics report appropriate to the application. The following code fragment shows the body of the report entity, as described:

```
while (sim_clock() < Duration) {
   sim_hold_for(Report_interval, SIM_NONE, ev);
   sim_schedule(Barber_id, 0.0, REPORT);
}
```

The report entity is similar to the source entity except that the report interval, denoted by **Report_interval**, is constant and the events scheduled by the report entity are of type **REPORT** rather than **CUSTOMER**.

Since the barber entity must now receive and process two types of events, its implementation must once again be modified. At least four implementation alternatives for the barber entity can be identified:

1. All calls to **sim_wait_for** and **sim_hold_for** in the barber entity can be modified to return report events whenever they occur. In other words, when the barber entity calls **sim_wait_for** to await the next customer, the event returned by **sim_wait_for** is either a customer event or a report event. Similarly, when the barber entity calls **sim_hold_for**, the simulated delay will either expire without interruption, or be interrupted by a report event.
2. The barber entity can redefine **sim_wait_for** and **sim_hold_for** to transparently intercept and process report events.
3. The barber entity can defer the report event until it has served the current customer to completion.
4. The barber entity can be reimplemented in an event-oriented style.

The first implementation alternative is similar to the extended barber shop model in Section 4.3 in which the structure of the barber entity body represented the combined tasks of serving customers and ignoring customers after closing time. Here, by explicitly

receiving and processing report events, the structure is similarly overloaded to both serve customers and print statistics reports. It has already been noted how the combination of such unrelated tasks within the structure of a single function can complicate the resulting implementation.

In addition, this implementation alternative leads to a significant increase in and duplication of code. For example, the call

```
sim_wait_for(sim_type_p(CUSTOMER), ev);
```

must be changed to

```
do {
   sim_wait_for(sim_type_p(CUSTOMER, REPORT), ev);
   if (ev.type() == REPORT) print_report();
} while (ev.type() != CUSTOMER);
```

where **sim_type_p(CUSTOMER, REPORT)** is a predicate that matches an event of type **CUSTOMER** or of type **REPORT**, and **print_report** is assumed to be a member function of the barber entity. Similarly, each non-interruptable delay of the form

```
sim_hold_for(Shave_time.sample(), SIM_NONE, ev);
```

must be changed to an equivalent interruptable delay

```
delay = Shave_time.sample();
while (delay > 0.0) {
   delay = sim_hold_for(delay, sim_type_p(REPORT), ev);
   if (delay > 0.0) print_report();
}
```

If the simulated delay is interrupted by a report event, the member function **print_report** is called, after which the simulated delay is resumed.

The use of interruptable delays to receive and process report events can be seen as an abuse of the interrupt facility. Specifically, interruptable delays are intended to model interruptable activities in the system being simulated. For example, the barber may need to interrupt his service to a customer to answer a telephone call. Report events do not

```
void barber::await_customer(sim_event &ev)
{
   do {
      sim_wait_for(sim_type_p(CUSTOMER, REPORT), ev);
      if (ev.type() == REPORT) print_report();
   } while (ev.type() != CUSTOMER);
}

void barber::serve_customer(sim_time delay)
{
   while (delay > 0.0) {
      delay = sim_hold_for(delay, sim_type_p(REPORT), ev);
      if (delay > 0.0) print_report();
   }
}

void barber::body()
{
   while (true) {
      if (sim_waiting(sim_type_p(CUSTOMER)))
         sim_select(sim_type_p(CUSTOMER), ev);
      else
         await_customer(ev);

      if (customer wants shave)
         serve_customer(Shave_time.sample());
      if (customer wants hair washed)
         serve_customer(Wash_time.sample());
      if (customer wants hair cut)
         serve_customer(Cut_time.sample());
   }
}
```

Figure 4.3: C++ Code for Barber Entity with Transparent Reports

represent interrupts in an actual barber shop. Instead, they are used only to generate information about the simulated system.

The second implementation alternative is similar to the first except that it eliminates much of the duplication of code by redefining the primitives **sim_wait_for** and **sim_hold_for** to transparently intercept and process report events. C++ code defining the resulting barber entity member functions and body is shown in figure 4.3. By replacing **sim_wait_for** with **await_customer** and **sim_hold_for** with **serve_customer**, the resulting barber entity body is even more readable than the basic model of figure 4.1. However, the implementation of **serve_customer** still relies on

interrupts to receive report events. In addition, both **await_customer** and **serve_customer** combine the conceptually distinct tasks of awaiting and serving customers with the printing of statistics reports. Finally, although the number of primitives that are redefined for this implementation is limited to two, it may be impractical to adopt this approach for other, more extensive models that make greater use of available primitives. In the worst case, it may be necessary to redefine all primitives simply to intercept and process one type of event.

In the third implementation alternative, the barber entity defers report events that coincide with the serving of a customer. After serving a customer to completion and before beginning to serve the next customer, the barber entity could execute the following code fragment to process a deferred report event:

```
if (sim_waiting(sim_type_p(REPORT))) {
   sim_select(sim_type_p(REPORT), ev);
   print_report();
}
```

Although this implementation alternative is considerably simpler than either of the first or second alternatives, it can only be used if statistics reports need not be printed at exactly the event time of the report events scheduled for the barber entity. Once again, the implementation of the barber entity body is overloaded with the processing of both customer and report events. Deferred events are similarly overloaded to represent both waiting customers and reminders to print statistics reports.

The fourth implementation alternative restructures the barber entity in an event-oriented style. All events in the simulation model are represented by events explicitly scheduled and received by the application. For example, the start and end times of each service provided by the barber are represented by events rather than by a series of simulated delays. C++ code for the barber entity body for this implementation alternative is shown in figure 4.4.

```
void barber::body()
{
   while (true) {
      sim_wait(ev);

      switch (ev.type()) {
         case CUSTOMER:
            handle_arriving_customer(ev);
            break;
         case END_OF_SHAVE:
            handle_finished_shave(ev);
            break;
         case END_OF_HAIR_WASH:
            handle_finished_hair_wash(ev);
            break;
         case END_OF_HAIRCUT:
            handle_finished_haircut(ev);
            break;
         case REPORT:
            print_report();
            break;
         default:
            sim_error("unknown event type");
      }
   }
}
```

Figure 4.4: C++ Code for Event-Oriented Barber Entity

All events scheduled for the barber entity are received by the single call to **sim_wait** in the body of the barber entity. For each event received by the barber entity, an *event handler* function is called to carry out the actions associated with that event. When the event handler function returns, the barber entity proceeds to receive the next event. The processing of an event may include updating the state of the barber entity and scheduling zero or more additional events. Event handlers do not advance simulation time, however. In other words, they do not execute simulated delays nor do they receive events. Instead, they represent only the instantaneous state transitions associated with the given event.

This implementation alternative differs significantly from the process-oriented style employed in previous examples in that the body of the barber entity no longer mirrors the sequence of activities as described for the barber shop model. As such, the conceptual gap

between the model and the implementation is increased. In addition, the barber entity is less modular for interrelated tasks. For example, depending on which services a customer requires, the barber entity might begin cutting a customer's hair immediately upon the customer's arrival, after shaving the customer, or after washing the customer's hair. Since each of these tasks is managed by a different event handler, each of these event handlers must be prepared to schedule an **END_OF_HAIRCUT** event according to the requirements of the individual customer. The scheduling of this event marks the beginning of the haircut and the subsequent receipt of this event marks the end of the haircut. Other tasks are similarly divided among multiple event handlers.

In spite of the shortcomings of the event-oriented style, it is significantly easier to add an independent task such as regularly scheduled statistics reports to an event-oriented barber entity. Statistics reporting is referred to as an independent task in that it does not depend on the prior execution of other events within the barber entity, nor does it cause other events to be executed. Simply by extending the **switch** statement in the body of the barber entity and adding additional event handlers, any number of independent tasks may be added to the barber entity.

4.5 Extending *Sim++* for Modeling Resource Competition

This section investigates the *extensibility* of the *Sim++* interface by adding an application-independent facility for mutually exclusive competition for resources. The inspiration for this facility comes from the Demos **RES** facility [Bir86]. Here, a simplified parallel implementation of the **RES** facility is described. This section illustrates the difficulties associated with implementing this type of facility in *Sim++* and, in particular, the difficulties associated with making this facility application-independent. In other words, it

is assumed that this facility is intended as a set of library routines, rather than tailoring it to individual applications.

The representation of systems as a collection of entities competing for scarce resources is a natural way to represent many problems. Birtwistle [Bir79] describes a simulation of a harbour in which ships, moving in and out of port, are towed to and from harbour jetties by tugboats. A ship moving into port to unload cargo must acquire access to a jetty at which to dock and two tugboats to tow it to the jetty. Once unloaded, the ship requires one tugboat to tow it out of port. In this simulation, tugboats and jetties can both be modeled as resources for which ships must compete.

In the parallel implementation of the **RES** facility, each resource is represented by an integer denoting the number of units of that resource available. Each such integer resource is encapsulated within a separate *resource manager* entity that provides mutually exclusive access to that resource through primitives **acquire** and **release**, declared as follows:

```
void acquire(sim_entity_id manager, int quantity);
void release(sim_entity_id manager, int quantity);
```

An entity calls **acquire** to request **quantity** units of a resource from the resource manager denoted by **manager**. If the requested units are not available immediately, **acquire** blocks the calling entity until a time when the resource manager can satisfy the request. All events received by the calling entity while blocked in **acquire** are deferred. An entity calls **release** to return previously acquired resources back to the specified resource manager. In the simulation of the harbour, if a ship entity requests two tugboats and all tugboats have already been acquired, the ship will be blocked until sufficient tugboats are subsequently released by other entities.

Since resource managers are entities, interactions between resource managers and entities calling **acquire** and **release** must be represented by events. **acquire** can be

implemented by scheduling an *acquire* event for the specified resource manager and awaiting a *granted* event in reply:

```
sim_schedule(manager, 0.0, ACQUIRE, &quantity, sizeof(quantity));
sim_wait_for(sim_type_p(GRANTED), ev);
```

The body of the event scheduled by **acquire** includes a copy of **quantity**, the number of units of the requested resource. All events received by the calling entity while awaiting a reply from the resource manager are automatically deferred by **sim_wait_for**. **release** can be implemented by scheduling a *release* event for the specified resource manager, informing it of the number of units of the resource being returned:

```
sim_schedule(manager, 0.0, RELEASE, &quantity, sizeof(quantity));
```

Instead of indefinitely blocking the calling entity when awaiting a reply from a resource manager, **acquire** could alternatively be designed to accept an application-specific predicate that specifies what events can interrupt a resource request:

```
boolean acquire(sim_entity_id manager, int quantity,
                sim_predicate &p, sim_event &ev);
```

If an event satisfying **p** is received by the calling entity before the resource request is satisfied, the resource request is cancelled and the interrupting event is returned to the calling entity in **ev**. If the resource request is satisfied without interruption, **acquire** returns the boolean result **true**; otherwise, acquire returns **false**. Any events received by the calling entity while blocked that do not satisfy the predicate **p** are deferred. The following code fragment implements these actions:

```
sim_schedule(manager, 0.0, ACQUIRE, &quantity, sizeof(quantity));
sim_wait_for(p OR sim_type_p(GRANTED), ev);
if (ev.type() != GRANTED) release(quantity);
return ev.type() == GRANTED;
```

acquire cancels a resource request by calling **release** before the resource request has been granted. For its part, if the resource manager receives a release event for a request that has not yet been granted, the request is assumed to have been cancelled. **acquire** cannot use

sim_cancel to cancel the resource request because the **acquire** event was scheduled with zero delay. In order to wait for either an event satisfying the application-specific predicate **p** or a reply from the resource manager, the call to **sim_wait_for** specifies the combined predicate ***p OR sim_type_p(GRANTED)***. Unfortunately, the predicate interface provided by *Sim++* does not allow predicates to be combined in this way. Nor is it possible to define a single predicate that specifies all of the conditions of the combined predicate, since **p** is known only to the application, while **sim_type_p(GRANTED)** is known only to the implementation of **acquire**. **acquire** could explicitly receive all events and test each against both individual predicates, but, as already noted in Section 4.2, there is no way to explicitly defer an event once it has been received.

A second difficulty concerns conflicting, simultaneous actions of the calling entity and the resource manager. Since entities execute concurrently, it is conceivable that a resource manager will schedule a granted event for a resource request at the *same* simulation time as the **acquire** primitive cancels the request as a result of receiving an event satisfying the given application-specific predicate. From its point of view, the resource manager receives the release event scheduled by **acquire** and assumes that the previously acquired resources are being returned (albeit in zero simulation time). However, having returned from its call to **acquire**, the calling entity is no longer prepared to receive the granted event from the resource manager. This unwanted event is referred to as an *orphan* event. The orphan event is used solely in the implementation of **acquire** and should not be seen by the application. Unfortunately, the next action the calling entity takes could be a call to **sim_wait** or a similar primitive through which it would receive the orphan event. Once again, **sim_cancel** cannot be used to cancel the orphan event because it too was scheduled with zero delay. To prevent the application from receiving the orphan event, it is necessary to redefine all existing primitives that an application is likely to invoke and that could

receive the orphan event, including **sim_wait**, **sim_hold**, **sim_wait_for**, and **sim_hold_for**. This differs from the barber shop example in which it was possible to transparently intercept report events by redefining only those primitives actually used by the application. Since the resource competition facility is intended to be application-independent, there is no way to know which primitives will not be invoked. As a result, all of them must be capable of intercepting the orphan event, resulting in significant duplication of code, both to redefine primitives that already exist, and to intercept and discard the orphan event in each primitive. This approach also lacks modularlity since many existing primitives must be redefined to support the new facility.

4.6 Summary

This chapter developed examples of a queueing model and a resource competition facility using the simulation and event selection primitives provided by *Sim++*. The ability to receive, defer, or otherwise manipulate events according to application-specific requirements is an appropriate abstraction for discrete event modeling because it eliminates much of the explicit manipulation of events that would otherwise be required. However, as illustrated in this chapter, these primitives also have significant limitiations. The most prevalent of these is the inability to restrict or extend the semantics of the event selection primitives. One reason for this is that the implicit actions and data structures associated with event selection are inaccessible to the application. Another reason is an interdependence among primitives that discourages modularity. The result is that these primitives must frequently be used *as is* or circumvented altogether. A related problem is the need to abuse modeling concepts to address non-modeling issues. Finally, predicates appear to be a limiting factor as well in that there is no way to combine predicates in the

same way as logical expressions, and application-defined predicates can significantly increase code size.

Many of the difficulties encountered with *Sim++* are inherent in any set of constructs that implicitly manipulate events without provision for application-specific restrictions or extensions. As such, similar problems exist in Extended Virtual Time and Maisie. Without appropriate solutions to these problems, even very basic models can be quite difficult to implement, requiring repeated and significant restructuring to cope with simple model extensions. An appropriate solution to these problems must encourage modular, application-specific and application-defined event selection. One such solution is presented in the next chapter.

Chapter 5

*Sim*D: A Language Proposal for Virtual Time

This chapter proposes a new language for Virtual Time called *Sim*D. *Sim*D is designed to support both process-oriented and event-oriented simulation design. The primary goal in developing *Sim*D was to address weaknesses in the event selection constructs of *Sim++*, Extended Virtual Time, and Maisie, without sacrificing the potential performance optimizations developed for Extended Virtual Time. In developing *Sim*D, the interfaces to many of the data types and language constructs found in *Sim++* have been improved.

Sections 5.1 through 5.6 describe the key elements of *Sim*D. Sections 5.7 and 5.8 reimplement the barber shop model and the resource competition facility in *Sim*D to demonstrate its effectiveness in addressing the modeling difficulties described in Chapter 4. *Sim*D was also used in the implementation of two parallel simulations described in Chapter 6. Section 5.9 discusses the potential for optimizing *Sim*D by incorporating the performance optimizations developed for Extended Virtual Time.

5.1 Overview

*Sim*D is designed to support both process-oriented and event-oriented simulation design. The process-oriented capabilities of *Sim*D are similar to those of *Sim++*, with support for simulated delays, scheduling, cancelling, and conditional selection of events. The event-oriented capabilities of *Sim*D provide a facility, called *event handlers*, whereby entities can intercept and process events before they can be received by the application or after they have been refused by the application. The combination of process-oriented and event-

oriented capabilities provides the necessary support required to address the modeling difficulties described in Chapter 4. For example, capabilities such as deferred and ignored events are implemented by the application using event handlers. As such, the actions and data structures associated with these capabilities are those of the application and can be easily modified to incorporate application-specific extensions or restrictions. This addresses the key difficulty associated with the implicit actions and data structures of *Sim++*, Extended Virtual Time, and Maisie. Further evidence of this claim is presented in sections 5.7 and 5.8.

*Sim*D's process-oriented and event-oriented capabilities can be implemented entirely in terms of a process' ability to send and receive timestamped messages. As such, *Sim*D is independent of any particular implementation of Virtual Time. Like *Sim++*, *Sim*D includes support for separate initialization and execution phases, clusters of entities, and read-only global variables. However, *Sim*D is not a complete simulation language. It lacks facilities for data collection and reporting, random number generation, error handling, and file and console input and output. The current implementation of *Sim*D *coexists* with *Sim++* and is therefore able to use *Sim++* facilities that are not yet available in *Sim*D. Where appropriate, the names of *Sim*D facilities are identical to the equivalent facilities in *Sim++*, except that the **sim_** prefix that appears in *Sim++* names is omitted (this was necessary to prevent namespace collisions since both languages coexist at run-time).

5.2 *Sim*D Data Types

*Sim*D defines data types similar to those of *Sim++* for representing simulation time, event types, entity identifiers, events, and event identifiers. This section summarizes significant

differences between the *Sim*D and *Sim*++ data types. The C++ declarations for the *Sim*D data types are shown in figure 5.1.

*Sim*D provides the macro **EVENT_TYPE** for generating unique event types. As an example, the declaration of event types **ARRIVAL** and **DEPARTURE** would be

```
EVENT_TYPE(ARRIVAL);
EVENT_TYPE(DEPARTURE);
```

The **EVENT_TYPE** macro expands these declarations to

```
sim_type ARRIVAL        = unique_event_type("ARRIVAL");
sim_type DEPARTURE      = unique_event_type("DEPARTURE");
```

where **unique_event_type** is a function that returns a unique integer value each time it is called. The equivalent declarations in *Sim*++ would be

```
sim_type ARRIVAL        = 1;
sim_type DEPARTURE      = 2;
```

The *Sim*++ approach is error prone in that it is possible to assign the same value to two or more event types, particularly if those variables are located in different modules of a simulation.

*Sim*D maintains a list of the event type strings passed to **unique_event_type** for use in trace output generated by the simulation primitives. As an example, trace output of the form

```
airplane1 at 10.7: scheduled ARRIVAL event for airport1
                   with delay 8.2
```

is optionally generated for each scheduled event. The equivalent trace output generated by *Sim*++ would be

```
airplane1 at 10.7: scheduled event of type 1 for airport1
                   with delay 8.2
```

The *Sim*++ trace output can be considerably more difficult to interpret, particularly when there are a large number of event types.

```
class sim_time { ... };
class sim_type { ... };

class entity_id { ... };

class class_id {
   ...
   char *class_name();
   int size();
   boolean includes(entity_id eid);
   entity_id operator[](int i);
};

class event {
   ...
   entity_id sched_by();
   entity_id sched_for();
   sim_time sched_at();
   sim_time sched_to();
   sim_type type();

   event &operator<<(char c);
   event &operator<<(int i);
   event &operator<<(double d);
   event &operator<<(char *s);

   event &operator>>(char &c);
   event &operator>>(int &i);
   event &operator>>(double &d);
   event &operator>>(char *s);
};

class event_id {
   ...
   entity_id sched_by();
   entity_id sched_for();
   sim_time sched_at();
   sim_time sched_to();
   sim_type type();
};

const sim_time    NO_SIM_TIME;
const sim_type    NO_SIM_TYPE;
const entity_id   NO_ENTITY_ID;
const class_id    NO_CLASS_ID;
const event       NO_EVENT;
const event_id    NO_EVENT_ID;
```

Figure 5.1: C++ Declarations for *Sim*D Data Types

One data structure unique to *Sim*D is the class identifier. Class identifiers are represented by class **class_id** and are used to denote classes of entities. Applications use class identifiers to determine the number and identity of entities of a given entity class or the number and identity of all entities in a simulation. This eliminates the need for an application to create and maintain arrays of entity identifiers. In addition, separately compiled and initialized program modules can each determine what entities exist in a simulation, without requiring the application to explicitly pass this information between modules.

An entity is a member of an entity class if it is an instance of that class or if it is a member of an entity class derived from that class. As an example, the call

```
cid = class_id("airplane");
```

creates a class identifier denoting all entities of or derived from entity class **airplane**. Similarly, since all entities in *Sim*D are derived from class **entity**, the call

```
cid = class_id("entity");
```

creates a class identifier denoting all entities in the simulation.

The information required by *Sim*D to determine which entities belong to which entity classes is provided by the macro **ENTITY_CLASS**. **ENTITY_CLASS** replaces the macro **SIM_ENTITY** in *Sim++*. The arguments to **ENTITY_CLASS** include the name of an entity class, and the name of the entity class from which it is derived. For most entity classes, the underlying entity class is **entity**. However, it is also possible to derive an entity class from a previously derived entity class. As an example, the following code fragment shows the declaration of entity class **vehicle** and entity class **truck**, and the corresponding calls to **ENTITY_CLASS**:

```
class vehicle : public entity { ... };
class truck : public vehicle { ... };
ENTITY_CLASS(vehicle, entity);
ENTITY_CLASS(truck, vehicle);
```

Class **truck** is derived from class **vehicle**, and class **vehicle** is derived from class **entity**.

The operations permitted on class identifiers are defined by the member functions **class_name**, **size**, **includes**, and the index operator ([ ]). **class_name** returns the name of the entity class denoted by a class identifier. **size** returns the number of entities in the entity class denoted by a class identifier. **includes** tests whether a given entity identifier is a member of the entity class denoted by a class identifier. The index operator provides a convenient array notation for accessing the entity identifiers of entities in an entity class. As an example, the call

```
cid = class_id("airplane");
eid = cid[0];
```

assigns to **eid** the entity identifier of the first entity in entity class **airplane**. (*Sim*D follows the C++ convention that array indexing begins with 0.)

Class **event** is used to represent events in *Sim*D. Class **event** has similar attributes to class **sim_event** in *Sim*++, but the names of member functions to access those attributes differ somewhat. The member function **sched_at** of class **event** has no equivalent in class **sim_event**. **sched_at** returns the simulation time at which the event was scheduled.

*Sim*D events are created explicitly by the application before being scheduled. An entity creates an event by invoking the constructor of class **event** and passing it the appropriate arguments used in initializing the attributes of the event. As an example, the call

```
ev = event(auto1_id, 8.0, ARRIVAL);
```

creates an event with attributes

```
ev.sched_by()      = self()
ev.sched_for()     = auto1_id
ev.sched_at()      = time()
ev.sched_to()      = time() + 8.0
ev.type()          = ARRIVAL
```

where **self** denotes the entity identifier of the calling entity and **time** denotes the current simulation time.

*Sim*D events include an implicit event body used to pass application-specific data from the scheduling entity to the receiving entity. The event body is organized as a stack and is accessed through insertion (<<) and extraction (>>) operators. The scheduling entity inserts data into an event using insertion operators and the receiving entity extracts data from the event using extraction operators. The insertion and extraction operators are member functions of class **event**. As an example, the call

```
ev << i << j;
```

copies the values in data structures **i** and **j** into the body of the event. *Sim*D defines insertion and extraction operators for all of the basic C++ types (integers, real numbers, characters, and strings). From these operators, new insertion and extraction operators can be defined for user-defined types. As an example, figure 5.2 shows the declaration of class **point** and its associated insertion and extraction operators. The approach illustrated in this figure was used to define insertion and extraction operators for many of the data types defined by *Sim*D.

Once the attributes and data of an event have been established, an entity can schedule the event with the call

```
schedule(ev);
```

*Sim*D events offer several advantages over *Sim++* events. First, the insertion operators for the basic C++ types automatically include type information with each data item inserted into the body of an event. This information is used by the extraction operators for the basic C++ types to ensure that a data item being extracted from the body of an event is of the same type as the data item actually in the body of the event. Although this type checking

```
class point {
   double X;
   double Y;
   double Z;
   ...
};

event &operator<<(event &ev, point &p)
{
   ev << p.X << p.Y << p.Z;
   return ev;
}

event &operator>>(event &ev, point &p)
{
   ev >> p.Z >> p.Y >> p.X;
   return ev;
}
```

Figure 5.2: C++ Insertion and Extraction Operators for Class Point

mechanism is transparent to the application, it is not foolproof. For example, it is possible to insert user-defined data of one type into an event and extract user-defined data of another type out of the event if the member variables of both data types are of the same basic C++ types. This problem can be addressed by associating a unique integer with each user-defined data type and including that integer in the body of the event. This would allow the extraction operator for a data type to ensure that it is extracting the correct type of data. Second, the insertion operators for *Sim*D events allow multiple data items to be inserted into the body of an event. In *Sim++*, the body of an event is a copy of the single data structure specified when the event was scheduled. To include multiple data items in a *Sim++* event, it would be necessary to copy the individual data items into a single, composite structure from which the event body could then be created. Typically, a new structure would have to be defined specifically for this purpose. Third, since all data items are inserted into the body of an event in terms of the basic C++ components that make up the data item, it would be straightforward to provide a capability to automatically print out the contents of an event as an extension of the tracing facility, or to insert the data into an

event in a machine-independent format that would allow events to be passed between entities executing in a heterogeneous multicomputer network.

Class **event_id** is used to represent event identifiers in *Sim*D. Like *Sim*D events, event identifiers are created explicitly by the application. An entity creates an event identifier by calling the constructor for class **event_id** and passing it an event as its argument. As an example, the call

```
ev_id = event_id(ev);
```

creates an event identifier denoting the event **ev**. The attributes of the event identifier are identical to the attributes of the event it identifies. The need to create event identifiers explicitly is less convenient than in *Sim*++ where they are created and returned by **sim_schedule**. However, profiling of *Sim*++ showed that 30% of the language overhead associated with scheduling an event came from creating and returning event identifiers. Since event identifiers are only required for event cancellation, it is unreasonable that every simulation and every scheduled event should incur this overhead. The *Sim*D approach associates the cost of event cancellation with those applications that use it.

5.3 Predicates

*Sim*D predicates are created using special objects called *predicate generators*. The predicate generators include

```
sched_by_p
sched_for_p
sched_at_p
sched_to_p
type_p
```

Each predicate generator corresponds to an event attribute. The predicate generator for a given event attribute is used to create predicates that test that attribute of an event. For

example, **sched_at_p** is used to create predicates that test the event attribute **sched_at**, which specifies the simulation time at which an event was scheduled.

Predicates are created by applying a relational operator to a predicate generator. As an example, the call

```
p = (sched_at_p <= 10.0);
```

creates a predicate that matches any event scheduled on or before simulation time 10.0. Similarly, the call

```
p = (sched_by_p == auto1_id);
```

creates a predicate that matches any event scheduled by the entity denoted by entity identifier **auto1_id**. In each case, the value on the right hand side of the relational operation must be of the same type as the event attribute corresponding to the predicate generator. In the examples shown, 10.0 is a value of type **sim_time** and **auto1_id** is a value of type **entity_id**. The types **sim_time** and **entity_id** correspond to the types associated with the event attributes **sched_at** and **sched_by**, respectively.

Using the logical operators **AND** and **OR**, predicates can be combined to test an event attribute for more than one value or to test multiple event attributes. As an example, the call

```
p = (sched_at_p <= 10.0) AND (sched_by_p == auto1_id);
```

creates a predicate that matches any event scheduled on or before simulation time 10.0 by the entity denoted by **auto1_id**. Similarly, the call

```
p = (type_p == ARRIVAL) OR (type_p == DEPARTURE);
```

creates a predicate that matches any event of type **ARRIVAL** or **DEPARTURE**.

Using the logical operator **NOT**, predicates can be also be negated. As an example, the call

```
p = NOT(type_p == ARRIVAL);
```

creates a predicate that matches any event not of type **ARRIVAL**.

*Sim*D also defines two special predicates, **any_p** and **none_p**, that match any and no events, respectively.

*Sim*D predicates offer several advantages over *Sim++* predicates. First, *Sim*D predicates are more easily defined than *Sim++* predicates. Since *Sim*D predicates are completely specified when and where they are created and used, the application need never define predicate classes, as is sometimes the case in *Sim++*. Second, *Sim*D predicates are more readable than *Sim++* predicates. This is because the conditions an event must satisfy to match a *Sim*D predicate are always explicit where and when the predicate is created and used, whereas in *Sim++*, the conditions are encapsulated within a predicate class, typically in a different source file from that in which the predicate is used. Third, *Sim*D predicates can be combined and negated in the same was as logical expressions. The inability to logically combine predicates was noted as a significant shortcoming of *Sim++* in Chapter 4. Fourth, *Sim*D predicates are safer than *Sim++* predicates. Since *Sim++* predicates are sometimes defined by the application, the event selection primitives that use them must rely on the discipline of the user to define predicates solely for testing event attributes. Other operations such as modifying the contents of an event or invoking a simulation primitive are illegal actions for a predicate, but these restrictions cannot be enforced. In contrast, *Sim*D predicates are specified solely in terms of data types and operations defined by the language. This means, however, that it is not possible for a *Sim*D predicate to examine the data in the body of an event, since that data can be of arbitrary type. My experience with *Sim++* and *Sim*D suggests that the ability of a predicate to examine the data in the body of an event is not required. Instead, section 5.8 shows how event handlers are used to intercept and examine the contents of an event prior to the receipt of that event by the application.

5.4 Entities

In *Sim*D, all entities are derived from class **entity**. Class **entity** defines attributes and capabilities required for simulation. Unlike *Sim++*, *Sim*D does not distinguish between simulation and event selection capabilities. The entity member functions that provide these capabilities in *Sim*D are collectively referred to as simulation primitives. Figure 5.3 shows the declaration of class **entity** and related operations.

The member function **self** returns an entity identifier denoting the calling entity. The member function **name** returns the name of the calling entity. The member function **class_name** returns the name of the calling entity's class. The function **time** returns the current simulation time. The function **current** returns a pointer to the object representing the calling entity.

The simulation primitive **schedule** is used to schedule an event. **schedule** differs from **sim_schedule** in that *Sim*D events must be created explicitly by the application before being scheduled.

The simulation primitive **cancel** is used to cancel a previously scheduled event.

The simulation primitive **wait** combines the capabilities of the *Sim++* primitives **sim_wait** and **sim_wait_for**. As an example, the calls

```
wait(ev);
wait(ev, type_p == ARRIVAL);
```

are equivalent to

```
sim_wait(ev);
sim_wait_for(sim_type_p(ARRIVAL), ev);
```

The second argument to **wait** is optional and defaults to the predicate **any_p**.

The simulation primitive **hold** combines the capabilities of the *Sim++* primitives **sim_hold** and **sim_hold_for**. As an example, the calls

```
class entity {
   ...

   // entity attributes
   entity_id self();
   const char *name();
   const char *class_name();

   // simulation primitives
   void schedule(event &ev);
   void cancel(event_id &ev_id);
   void wait(event &ev, predicate &p);
   sim_time hold(sim_time delay, event &ev, predicate &p);
   void terminate();

   // event handling capabilities
   void prehandler(predicate &p, entity_function *f);
   void posthandler(predicate &p, entity_function *f);
   void forward(event &ev);
};

entity *current();
sim_time time();
```

Figure 5.3: C++ Declaration for Class Entity

```
hold(10.0);
hold(10.0, ev);
hold(10.0, ev, type_p == ARRIVAL);
```

are equivalent to

```
sim_hold_for(10.0, SIM_NONE, ev);
sim_hold(10.0, ev);
sim_hold_for(10.0, sim_type_p(ARRIVAL), ev);
```

The second and third arguments to **hold** are optional. When the third argument is omitted, it defaults to the predicate **any_p**, meaning that the simulated delay can be interrupted by any event. When the second and third arguments are both omitted, the third argument defaults to **none_p**, meaning that the simulated delay is uninterruptable.

The simulation primitive **terminate** is used to terminate the actions of an entity. The simulation terminates when all entities have terminated.

*Sim*D does not provide primitives for manipulating deferred events. Deferred events, ignored events, and similar simulation capabilities can be represented using event handlers, as described in the following sections.

5.5 Event Handlers

Three of the languages surveyed in Chapter 3, *Sim*++, Extended Virtual Time, and Maisie, provide a capability to defer events. Extended Virtual Time additionally provides a capability to ignore events. Events are deferred or ignored on behalf of an entity by the language, without requiring an application to explicitly receive events to be deferred or ignored. Chapter 4 noted that there are other actions an entity may wish to perform to process events, again without requiring that the application explicitly receive each event. These included the ability to limit the number of deferred events, to generate statistics reports, and to ignore non-application, orphan events. Rather than augmenting a language with these specific capabilities, a general mechanism is proposed whereby an application can specify arbitrary actions for processing events, without explicitly receiving those events. This capability is provided in *Sim*D using event-oriented simulation techniques. Many of the modeling difficulties described in Chapter 4 can be solved quite easily using *Sim*D's combined process-oriented and event-oriented capabilities. In addition, capabilities such as ignored events and deferred events have straightforward, modular implementations using event-oriented simulation techniques.

*Sim*D supports the creation of *event handlers* for processing events without explicitly receiving events through calls to the simulation primitives **wait** or **hold**. An event handler is an entity member function that is automatically invoked to process events that satisfy a given predicate. Event handlers represent instantaneous state transitions associated with a

given event. This means that event handlers do not execute simulated delays or receive other events. (The notion of *receiving events* is used exclusively in *Sim*D to refer to calls to **wait** or **hold** and excludes the implicit invocation of event handlers. The predicate specified in calls to **wait** or **hold** is referred to as the *active predicate.*)

*Sim*D defines two types of event handlers: *prehandlers* and *posthandlers.* Prehandlers are used to intercept events before they can be received by the application. Posthandlers are used to intercept events after they have been refused by the application. Any event not received by the application or intercepted by an event handler will result in a run-time error. *Sim*D defines the entity member functions **prehandler** and **posthandler** to create event handlers. The arguments to **prehandler** and **posthandler** include a predicate that specifies the conditions an event must satisfy for the event handler to be called, and a pointer to an entity member function that serves as the event handler. Typically, event handlers for an entity are created during the entity's initialization, as defined by the entity's constructor. An entity can change the predicates associated with existing event handlers by calling **prehandler** or **posthandler** and specifying a different predicate for a previously established event handler.

The predicates associated with event handlers need not be unique. It is permissible for multiple event handler predicates to match the same event. It is also permissible for the active predicate to also match the event. When more than one predicate matches the same event, the first matching predicate determines how the event will be processed. If the matching predicate is associated with an event handler, the corresponding entity member function is invoked. If the matching predicate is the active predicate, the event will be received by the application. Collectively, the active and event handler predicates are referred to as a *predicate chain.* The relative positions of predicates in the predicate chain

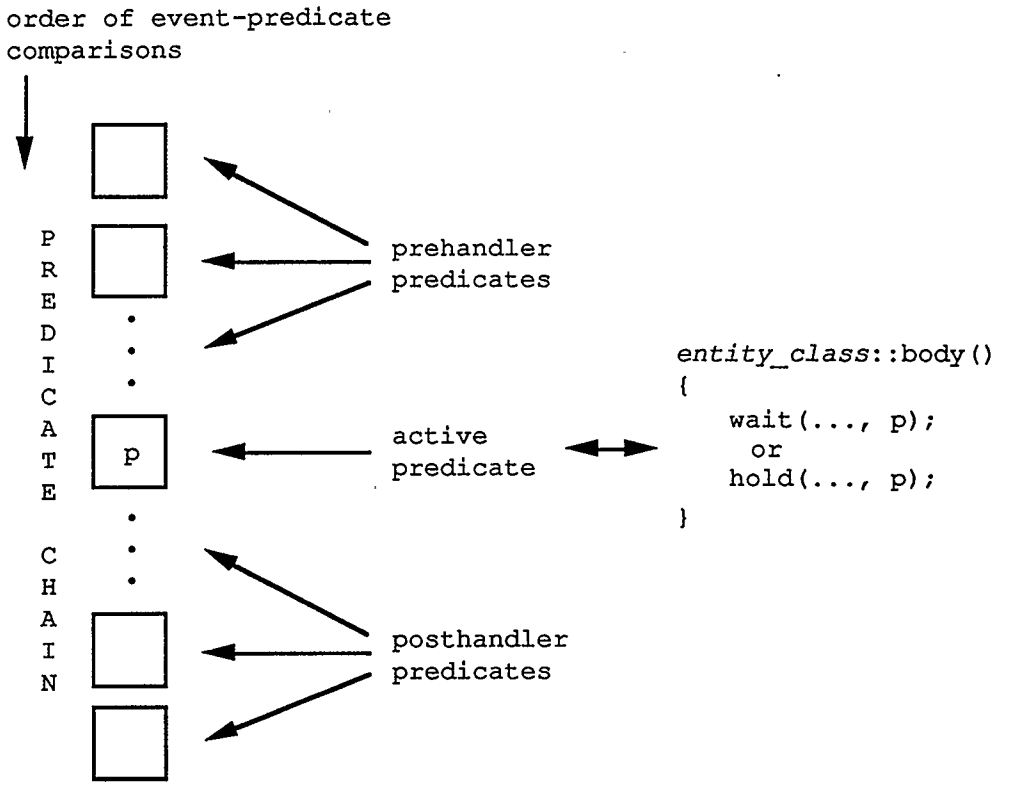

Figure 5.4: *Sim*D Predicate Chain

are shown in figure 5.4. To find a matching predicate, an event is compared first to the predicates associated with prehandlers, in the order that the prehandlers were created. Next, the event is compared to the active predicate. Finally, the event is compared to the predicates associated with posthandlers, in the order that the posthandlers were created.

By default, once an event has been intercepted by an event handler, the event does not continue along the predicate chain. However, my experience with event handlers has shown that there are many cases when it is desirable for an event handler to intercept an event, examine or modify the attributes or body of the event, and then allow the event to continue along the predicate chain to be intercepted by another event handler or to be received by the application. *Sim*D defines the entity member function **forward** to allow an

event handler to forward an event to the next predicate along the predicate chain. The event does not actually continue along the predicate chain until the executing event handler completes.

5.6 Initialization of *Sim*D Simulations

The *Sim*D initialization phase is represented by an arbitrary number of initialization functions. In this way, it is possible to define a separate initialization function for each independent program module in a simulation. For each initialization function, there must be a corresponding call to the macro **INITIAL_FUNCTION**. The argument to **INITIAL_FUNCTION** is the name of the initialization function. **INITIAL_FUNCTION** hides a number of declarations required by the implementation to locate and execute all of the initialization functions at run time. The command line arguments used to execute a *Sim*D simulation specify what initialization functions to execute, the order in which those functions should be executed, and what arguments to pass to each initialization function. The initialization phase ends when all initialization functions have finished executing.

5.7 The Barber Shop Model Solved in *Sim*D

This section shows how *Sim*D can be used to address the difficulties associated with the *Sim*++ implementation of the barber shop model described in Chapter 4.

The *Sim*D implementation of the barber entity for the basic barber shop model is shown in figure 5.5. In *Sim*D, there is no built-in deferred event list. The queue of waiting customers in the barber entity is represented instead by the event list denoted by **Waiting**.

```
barber::barber(event &ev)
{
   ...

   Waiting = event_list("waiting customers");
   posthandler(type_p == CUSTOMER, barber::handle_customer);
}

void barber::body()
{
   while (true) {
      if (Waiting.cardinal(type_p == CUSTOMER) > 0)
         ev = Waiting.retrieve(type_p == CUSTOMER);
      else
         wait(ev, type_p == CUSTOMER);

      if (customer wants shave)
         hold(Shave_time.sample());
      if (customer wants hair washed)
         hold(Wash_time.sample());
      if (customer wants hair cut)
         hold(Cut_time.sample());
   }
}

void barber::handle_customer(event &ev)
{
   Waiting.store(ev);
}
```

Figure 5.5: *Sim*D Barber Entity for Basic Barber Shop Model

Waiting denotes an instance of class **event_list** defined by *Sim*D. Class **event_list** supports operations for enqueueing events, dequeueing and counting events that satisfy a given predicate, and collecting and reporting queueing statistics. Events are enqueued in the event list by a posthandler that intercepts all events of type **CUSTOMER** that are refused by the application. When there are no waiting customers, the barber entity explicitly waits for the next arriving customer by calling **wait**. Figure 5.6 shows the predicate chain associated with the barber entity during the call to **wait**. While serving a customer, the barber entity executes uninterruptable delays with **hold**, thereby refusing to receive other events that coincide with the simulated delays. Figure 5.7 shows the

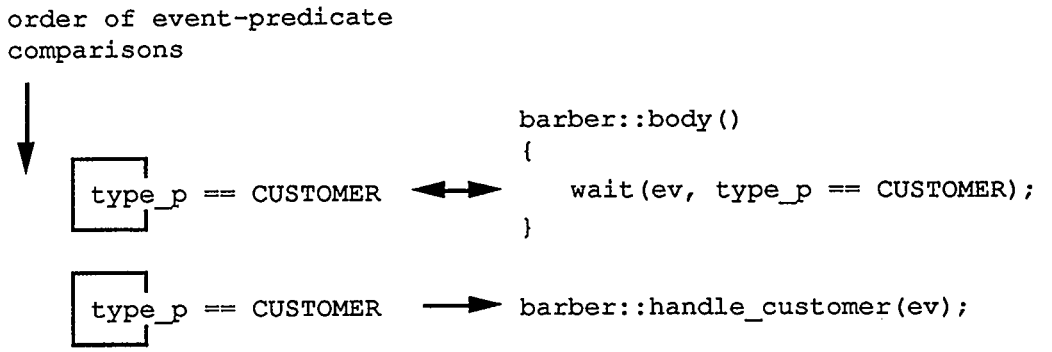

Figure 5.6: *Sim*D Predicate Chain for Barber Entity During Wait

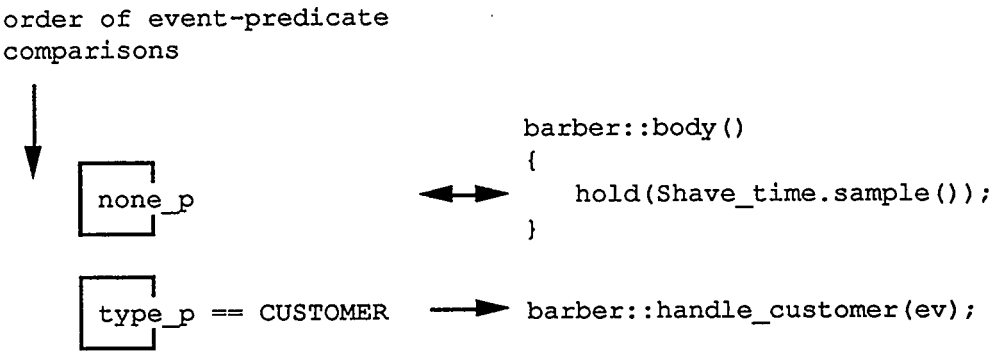

Figure 5.7: *Sim*D Predicate Chain for Barber Entity During Hold

predicate chain associated with the barber entity during one of these calls to **hold**. During each call to **hold**, the active predicate is **none_p**, and all events of type **CUSTOMER** are intercepted by the posthandler **barber::handle_customer** and enqueued in the event list **Waiting**.

To restrict the size of the queue of waiting customers, the implementation of the posthandler can be changed to ignore arriving customers when there is no room in the queue. To ignore additional customers after closing time, a prehandler can be used to intercept late customers. To generate statistics reports, a prehandler can be used to intercept and process report events (it is again assumed that the report events are scheduled by a

```
barber::barber(event &ev)
{
   ...

   prehandler(type_p == CUSTOMER AND sched_at_p >= Closing_time,
              barber::handle_late_customer);

   prehandler(type_p == REPORT, barber::handle_report);

   Waiting = event_list("waiting customers");
   posthandler(type_p == CUSTOMER, barber::handle_customer);
}

void barber::handle_late_customer(event &ev)
{
   Late_customers += 1;
}

void barber::handle_report(event &ev)
{
   ...
}

void barber::body()
{
   while (true) {
      if (Waiting.cardinal(type_p == CUSTOMER) > 0)
         ev = Waiting.retrieve(type_p == CUSTOMER);
      else
         wait(ev, type_p == CUSTOMER);

      if (customer wants shave)
         hold(Shave_time.sample());
      if (customer wants hair washed)
         hold(Wash_time.sample());
      if (customer wants hair cut)
         hold(Cut_time.sample());
   }
}

void barber::handle_customer(event &ev)
{
   if (Waiting.cardinal(type_p == CUSTOMER) < Limit)
      Waiting.store(ev);
   else
      Lost_customers += 1;
}
```

Figure 5.8: *Sim*D Barber Entity for Extended Barber Shop Model

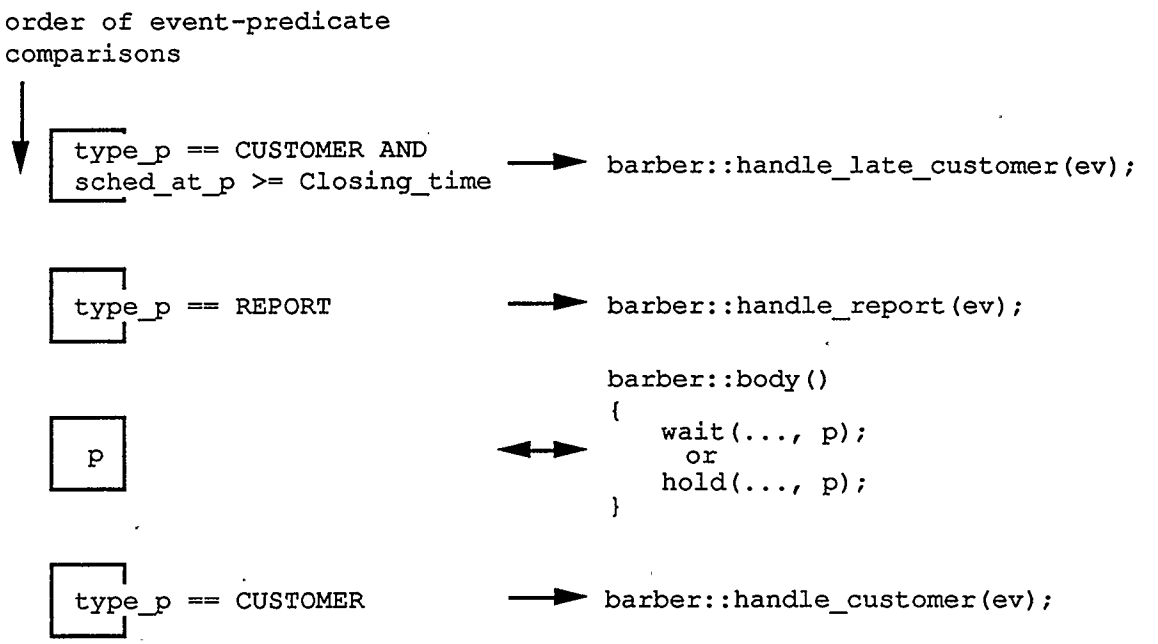

Figure 5.9: *Sim*D Predicate Chain for Extended Barber Shop Model

separate *report* entity). An implementation of the barber entity that incorporates all of these changes is shown in figure 5.8. The corresponding predicate chain is shown in figure 5.9. Despite all three changes, the main actions of the barber entity, as defined by the entity body, remain unchanged, and there is no need to redefine any of the existing simulation primitives.

5.8 The Resource Competition Facility Solved in *Sim*D

This section shows how *Sim*D can be used to address the difficulties associated with the *Sim++* implementation of the Demos **RES** facility described in Chapter 4.

The two problems associated with the *Sim++* implementation of the **RES** facility were the inability to combine predicates in the same way as logical expressions, and the inability to intercept and discard orphan events without redefining all existing primitives. The need

to combine predicates arises because there are two predicates involved in the use of the **RES** facility: a predicate that matches the event scheduled by a *resource manager* in response to a request for resources, and an application predicate that specifies what application events can interrupt a request for resources. The **acquire** primitive described in Chapter 4 must wait for an event that matches *either* of these two predicates. Orphan events result when an entity cancels a resource request at the same simulation time as the request is granted by the resource manager. Since these actions can occur simultaneously in two different entities, the entity that cancelled the resource request must be prepared to eliminate the event scheduled for it by the resource manager so that the event cannot be received by the application.

In *Sim*D, the ability to combine predicates using logical operators is used to address the first of the problems encountered with *Sim++*. And, a prehandler is used to intercept and discard orphan events resulting from cancelled resource requests. Implementations of **acquire** and **release**, and the associated prehandler, **demos::handle_granted**, are shown in figure 5.10 (line numbers in the following description correspond to those in figure 5.10). It is assumed that entities which compete for resources using **acquire** and **release** will be derived from entity class **demos**. The predicate chain for a typical call to **acquire** is shown in figure 5.11.

As in *Sim++*, **acquire** schedules an *acquire* event for the resource manager denoted by **manager** to request **quantity** units of a resource (lines 28-32). The resource manager will respond with a *granted* event once **quantity** units of the resource are available. In order to distinguish the resource manager's response from orphan granted events, the event identifier for the acquire event, denoted by **Acquire_id** (line 30), is included in the body of the acquire event (line 31). The resource manager returns the event identifier in the

```
1     class demos : public entity {
2        event_id Acquire_id;
3        ...
4        void handle_granted(...);
5        boolean acquire(...);
6        void release(...);
7     };
8     ENTITY_CLASS(demos, entity);
9
10    demos::demos(...)
11    {
12       ...
13       Acquire_id = NO_EVENT_ID;
14       prehandler(type_p == GRANTED, handle_granted);
15    }
16
17    void demos::handle_granted(event &ev)
18    {
19       ev >> acquire_id;
20       if (Acquire_id == acquire_id) {
21          forward(ev);
22       }
23    }
24
25    boolean demos::acquire(entity_id manager, int quantity,
26                           event &ev, predicate &p)
27    {
28       ev = event(manager, 0.0, ACQUIRE);
29       ev << quantity;
30       Acquire_id = event_id(ev);
31       ev << Acquire_id;
32       schedule(ev);
33
34       wait(ev, p OR type_p == GRANTED);
35       if (ev.type() != GRANTED) release(manager, quantity);
36       Acquire_id = NO_EVENT_ID;
37
38       return ev.type() == GRANTED;
39    }
40
41    void demos::release(entity_id manager, int quantity)
42    {
43       ev = event(manager, 0.0, RELEASE);
44       ev << quantity;
45       schedule(ev);
46    }
```

Figure 5.10: *Sim*D RES Facility

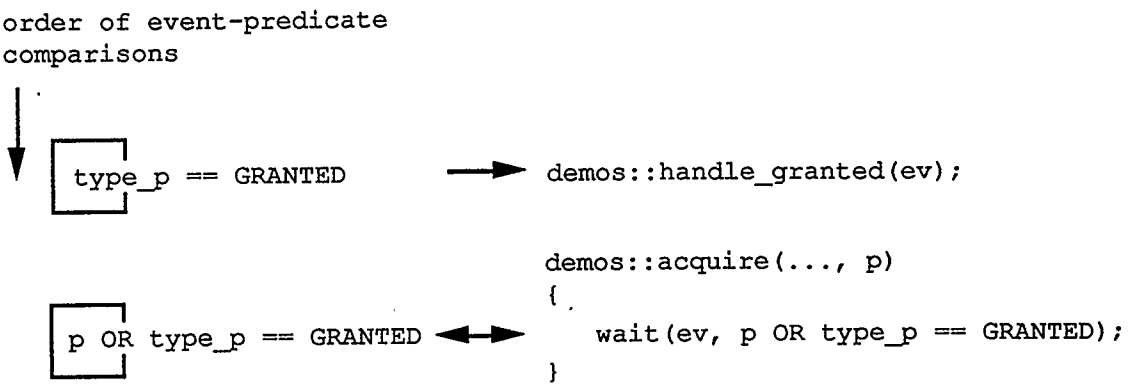

Figure 5.11: *Sim*D Predicate Chain for RES Facility

granted event to allow the prehandler (lines 17-23) to uniquely distinguish the expected granted event from orphan granted events. The prehandler intercepts all events of type **GRANTED** before they can be received by the application. Only the granted event awaited by the current call to acquire is forwarded along the predicate chain to be received by the call to **wait** (line 34). Other, orphan granted events are intercepted by the prehandler and are ignored.

5.9 Optimizing *Sim*D on Time Warp

It was noted in Chapter 3 that because of the similarities between the simulation constructs of *Sim++* and Extended Virtual Time, it is possible to modify the implementation of *Sim++* to incorporate the optimizations developed for Extended Virtual Time. Although many of the data types and language constructs of *Sim*D have revised interfaces, the implementations of **cancel** and **hold** in *Sim*D are as described for the application version of Extended Virtual Time. As a result, both of these primitives can be optimized as described in Chapter 3.

One apparant drawback of *Sim*D is that it is not possible to achieve the same state size reductions as achieved in Extended Virtual Time by integrating an entity's deferred event list with its underlying Time Warp input queue. However, write-locked memory may provide a suitable alternative. Since *Sim*D events can only be accessed through their attribute functions, insertion operators, and extraction operators, it is possible for an event to be write-locked until one of the attributes of the event is changed or until data is moved into or out of the body of the event. Each member function of class **event** that results in the contents of the event being altered would be responsible for unlocking the memory associated with the event. In this way, events that are queued in an application-specific event list would remain write-locked and would be excluded from state save operations. Since write-locked memory is not yet fully supported by the Time Warp implementation used in the development of *Sim*D, this optimization has not been implemented.

Since neither *Sim*++ nor *Sim*D incorporate any of the optimizations of Extended Virtual Time, neither implementation was found to have any significant advantage over the other in terms of overall performance, although *Sim*D does have slightly lower event scheduling overhead in cases where event identifiers are not required.

5.10 Summary

This chapter proposed a new language for Virtual Time called *Sim*D. Among the contributions of this language are improved interfaces to many of the data types and language constructs provided by *Sim*++, and a combined process-oriented and event-oriented modeling paradigm that was shown to address weaknesses in the event selection constructs of *Sim*++, Extended Virtual Time, and Maisie. It was argued that the unique characteristics of *Sim*D make resulting simulations simpler, more readable, more type safe,

and more modular. Although differences between *Sim*D and Extended Virtual Time do not permit all of the performance optimizations developed for Extended Virtual Time to be incorporated into *Sim*D, alternative optimizations for achieving comparable results were suggested.

One of the most important and unexpected lessons to arise from the development of *Sim*D is the role of documentation and implementation as factors affecting the design of the language. Although it was obvious from the very beginning that the use of *Sim*D in the development of applications would provide useful feedback towards improving the resulting design, documentation and implementation played equally important roles towards this end as well. Many prototype designs of *Sim*D features which were easy to use had to be further refined or even abandoned because they could not be easily described or because their implementation was too complex or inefficient. Indeed, the presentation of *Sim*D in this chapter is intended as evidence that the resulting design can be described in relatively simple and concise terms, and that key features can be described as a progression of concepts, beginning with basic data types and predicates, and ending with event handlers.

Chapter 6

Experience With *Sim*D

This chapter describes two discrete-event simulations that were implemented in *Sim*D and executed on Time Warp on a transputer-based, distributed-memory Meiko Computing Surface. The purpose in developing these simulations was to gain experience with *Sim*D and to test the implementation of *Sim*D on Time Warp. Both simulations have previously been used as benchmarks to evaluate the performance of Time Warp and Extended Virtual Time [Bae89, Bae91, Lom88a, Lom88b]. For each simulation, the model and its implementation in *Sim*D are described, and performance results are presented for the simulation executing on Time Warp. The purpose of this chapter is to demonstrate that *Sim*D can be used with Time Warp to develop well-structured, efficient, parallel simulations. For a detailed performance study and analysis of both simulations, see Lomow's thesis [Lom88b]. Both models were developed by Lomow and portions of the model descriptions that follow are, with minor simplifications for brevity, as they appeared in his thesis.

6.1 Overview of Experiments and Experimental Method

All experiments were executed on a sequential simulator on a single Computing Surface node with 16 megabytes of memory, and in parallel on Time Warp on 8, 16, 24, and 32 Computing Surface nodes with 4 megabytes of memory per node. Approximately 1 megabyte of memory per node is required by the Computing Surface operating system and for the executable object code. The sequential simulator is used as the basis for speedup

comparisons. The sequential simulator excludes all of the overhead associated with Time Warp for process rollback, message cancellation, calculating GVT, and interprocessor communication. The sequential simulator uses a linear list to represent the input queue of each process, and a global splay tree [Sle85] to order the next event time of all processes. The next event time of a process corresponds to the receive time of the first message in the process' input queue. Processes execute in order of increasing next event time.

Each of the speedup graphs presented in the following sections is presented in terms of *application speedup*. Application speedup is defined as the sequential application time divided by the parallel application time for a given number of nodes. Application time is the amount of time required to execute the initialization and execution phases of the application. Application time excludes startup and shutdown overhead such as the time required to download the executable object code to each node of the Computing Surface. This overhead amounts to approximately 4 seconds per node and easily dominates the total execution time for many of the parallel runs used in this study. For example, the time required for startup and shutdown on 32 nodes is approximately 2 minutes. The application speedup graphs are therefore intended to represent simulations in which this overhead is only a small fraction of the total execution time.

The sequential and parallel application times are each the average of 9 runs. A different initial seed for random number generation was used for every 3 runs. The maximum deviation from the mean for each set of 9 runs was less than 5% for all experiments. The sequential application times varied in length from 2 to 14 minutes. The parallel application times varied in length from 1 to 4 minutes. The speedup achieved for these runs was verified to be representative of the speedup achieved with longer runs of both simulations.

The use of sequential application time as the basis for speedup comparisons is generally consistent with the presentation of speedup in the literature [Fuj87, Fuj90, Lom88b]. It

can be argued, however, that the sequential application time is too high since it represents the time required to sequentially execute an application designed for parallel execution. A more appropriate basis for speedup comparisons might be the time required to execute a purely sequential implementation of each application, using global and shared memory for process interactions. For the sake of simplicity and consistency with the literature, the speedup results presented in this chapter are based on the sequential application time. It remains to be seen how well existing implementations of Virtual Time compare with sequential simulation techniques using global and shared memory.

In addition to the application speedup achieved by Time Warp, each graph also plots an estimate of the potential speedup of the application on 8, 16, 24, and 32 processors. Although the speedup for any problem executing in parallel is limited by the number of processors on which the problem is executed, real simulations are further limited by causal relationships between events and intra-processor and inter-processor communication overhead. The potential speedup is calculated by a performance analysis tool that takes these additional limitations into consideration. Using data collected by Time Warp during the execution of a simulation, the tool simulates a parallel execution of the same simulation, assuming a "perfect" implementation of Virtual Time with the following characteristics:

1. Each entity executes its events in the exact order that it would in a sequential execution.
2. Reasonable delays (appropriate to the type of parallel processor) are incurred for sending or receiving a message.
3. Two or more entities mapped to the same processor never execute simultaneously.

The resulting potential speedup excludes the additional overhead associated with existing conservative and optimistic approaches to parallel simulation. Although Berry [Ber86] has

shown that Time Warp is not theoretically limited by the potential speedup calculated by this tool, my experience and that of researchers at JPL [Rei90] suggests that the conditions under which Time Warp can achieve or exceed the potential speedup do not arise in practice. This is due primarily to the much higher overheads of Time Warp, including the time required to regularly save the states of processes, the time required to create and maintain anti-messages, the time required to perform rollbacks, and the time required to calculate GVT and perform fossil collection.

For the potential speedup calculated for this study, intra-processor communication overhead was fixed at 1.0 milliseconds and inter-processor communication overhead was fixed at 2.0 milliseconds. The value for intra-processor communication overhead is equivalent to the time required to schedule and receive an event in *Sim*D on the optimized sequential simulator. This includes the time required to create and schedule the event, context switch to the receiving entity, and receive the event. The value for inter-processor communication overhead is based on experiments performed on the Computing Surface using the message-passing kernel of the Time Warp implementation used in this study. In addition to the overhead associated with intra-processor communication, inter-processor communication further includes the time required to transmit the message between processors via the message-passing kernel. The value for inter-processor communication overhead excludes any delays that might typically be experienced by the message-passing kernel when the receiving processor is busy performing input or output, or executing the application. These additional delays have been ignored since they cannot be estimated with any reasonable degree of accuracy. As a result, the value for inter-processor communication overhead is a lower bound on the actual overhead. A more accurate estimate would tend to lower the potential speedup figures calculated for this study.

Both simulations include a variety of parameters that vary the behaviour of the application, resulting in diverse execution characteristics for these simulations executing in parallel. For the results reported here, many of the parameters chosen are "typical" configurations that represent neither a worst case nor a best case in terms of the potential speedup of either simulation [Lom88b]. The results achieved appear consistent with those of other studies of these simulations, although no direct comparison was attempted since the scope of this study is limited to demonstrating the effectiveness of *Sim*D for developing these simulations. Indeed, one of the key differences between this and earlier studies is that this study contained up to 10 times as many entities. The larger number of entities is intended to demonstrate that *Sim*D and Time Warp are suitable for large problems containing many hundreds of entities.

6.2 The Health Care System

The health care system is an hierarchical queueing system consisting of villages and health centers. There is one health center for each village and each village/health center pair is referred to as a health care node. When villagers become ill, they travel to their local health center for assessment and, when possible, treatment. If they cannot be treated locally, patients are referred up the hierarchy to the next, more sophisticated health center where the assessment/treatment/referral process is repeated. It is assumed that patients can always be treated at the root of the health care system. Once treated, patients return to their home village. Figure 6.1 illustrates a simplified view of this health care system.

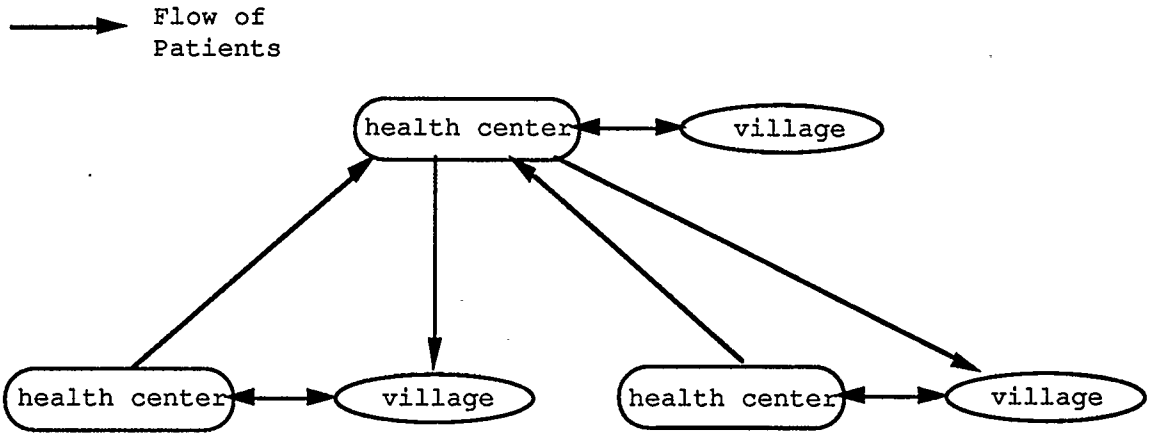

Figure 6.1: The Health Care System

6.2.1 Design Issues and Implementation

For the performance results that follow, villages and health centers are represented by separate entities, and patients are represented by events. Although it has been shown [Bae91] that implementing each health care node as a single entity can result in as much as 40% additional speedup, the approach used here results in more readable code. Appendix A presents a complete *Sim*D implementation of the health care system with separate village and health center entities.

Each village entity performs two distinct tasks: generate patients for the local health center, and receive returning villagers. The body of the village entity is used solely to generate patients. The interarrival time between successive patients is modeled using a call to **hold**. A separate event handler is used to receive returning villagers. C++ code for the body of the village entity and this event handler is shown in figure 6.2. By intercepting returning villagers with an event handler, the call to **hold** will never be interrupted. To interrupt the call to **hold** with returning villagers would be an abuse of interruptable delays, since returning villagers do not represent an interrupt in the model.

```
void village::handle_treated(event &treated_ev)
{
   Outstanding = Outstanding - 1;    // outstanding villagers
   if (Outstanding == 0) generate_report();
}

void village::body()
{
   while (time() < DURATION) {
      Outstanding = Outstanding + 1;

      hold(Interarrival_time.sample());
      Generated = Generated + 1;

      patient_ev = event(Local_health_center, 0.0, PATIENT);
      patient_ev << self();    // return patient to this village

      schedule(patient_ev);
   }
}
```

Figure 6.2: C++ Code for Village Entity Class

Health center entities are very similar to the barber entity described in Chapter 4. A health center entity treats one patient at a time, in the order that patients arrive at the health center. The body of the health center entity is used solely to model the assessment and treatment of patients. The simulation time during which a patient undergoes assessment and treatment is modeled using calls to **hold**. A separate event handler is used to receive and enqueue patients that arrive while another patient is being treated. C++ code for the body of the health center entity and this event handler is shown in figure 6.3.

A summary entity is used to gather and summarize statistics about the application. This entity is a sequential bottleneck since it must receive and process an event from every other entity in the simulation after those entities have terminated. For the results that follow, the sequential and parallel application times exclude the time required to execute the summary entity at the end of the simulation. The time required to execute the summary entity on

```
void health_center::body()
{
   while (TRUE) {
      // get the next patient
      if (Patients.empty())
         wait(patient_ev, type_p == PATIENT);
      else
         patient_ev = Patients.retrieve(type_p == PATIENT);

      hold(ASSESS_TIME / Personnel);
      Assessed = Assessed + 1;

      if (Treatable.sample()) {
         hold(TREAT_TIME / Personnel);
         Treated = Treated + 1;
         patient_ev >> home_village;
         treated_ev = event(home_village, 0.0, TREATED);
         schedule(treated_ev);
      }
      else {
         Referred = Referred + 1;
         patient_ev >> home_village;
         patient_ev = event(Next_health_center, 0.0, PATIENT);
         patient_ev << home_village;
         schedule(patient_ev);
      }
   }
}

void health_center::handle_patient(event &patient_ev)
{
   Patients.store(patient_ev);
}
```

Figure 6.3: C++ Code for Health Center Entity Class

Time Warp is approximately 1 minute. That time is a significant factor in many of the parallel runs used in this study, but is less significant in simulations that execute for longer periods of time.

6.2.2 Simulation Parameters and Performance Results

The configuration of the health care system used for all experiments consisted of 341 health care nodes organized as a full, 4-way branching tree of height 5. This configuration results

in 682 entities with an approximate state size of 6K per entity, including language overhead and the state saved portion of an entity's run-time stack. Approximately 1K of this state is due to language overhead. The average computation per event for this simulation is 2.6 milliseconds. Approximately 1 millisecond of this time is due to language overhead. For all experiments, the village and health center entities associated with a given health care node were always mapped to the same processor. Health care nodes were mapped to processors using a *static mapping* that attempts to distribute the total computation as evenly as possible over the available processors. Static mapping requires that the simulation be executed at least once in order to produce the necessary performance data to generate the mapping. Static mapping assumes that subsequent runs of the same simulation will be sufficiently similar to the first that the original mapping will result in better performance than if entities are mapped to processors at random or round robin. This is generally true for the configurations of both simulations presented in this chapter, since only the random number seeds are varied between runs. However, in general, if a simulation is executed multiple times with different sets of input parameters, it is possible that the simulation will exhibit radically different execution characteristics for each run. Reiher and Jefferson [Rei90b] present preliminary results for a dynamic load management scheme intended to address this problem.

The number of health care personnel at each health center is based on the health center's level in the tree. Health centers nearer to the root of the tree represent larger hospitals and have correspondingly more personnel. These increased resources are modeled by a corresponding decrease in service time. Table 6.1 gives the number of health care nodes at each level of the system and the number of health care personnel at each health center.

Level	Health Care Nodes	Health Care Personnel per Health Center
0	1	16
1	4	8
2	16	4
3	64	2
4	256	1

Table 6.1: Configuration of the Health Care System

Parameter	Value
Duration	100.0 time units
Arrival Rate	0.3 patients per unit time
Treatment Probability	0.9 (1.0 at root)
Assessment Time	0.3 time units
Treatment Time	1.0 time units

Table 6.2: Application Parameters for the Health Care System

The values of other application parameters are shown in table 6.2. The interarrival time between patients generated by a village is drawn from a negative exponential distribution with a mean arrival rate of 0.3 patients per unit time. The probability that a patient can be treated at a given health center is 0.9 (except at the root of the health care system where the probability is 1.0). The assessment and treatment times are fixed at 0.3 and 1.0 time units, respectively, divided by the number of health care personnel at the health center.

Since each village is mapped to the same processor as its local health center, inter-processor communication only occurs when a patient is referred to a health center on a different processor or when a patient returns from a remote health center. Since, on average, 90% of patients are treated locally, there is relatively little inter-processor communication, suggesting that this configuration will result in a high degree of parallelism among processors. This has indeed been the case with earlier studies of this model [Bae89, Lom88a, Lom88b]. A reduced treatment probability has been found to degrade performance, as inter-processor communication and synchronization increases.

Figure 6.4 shows the application speedup achieved by Time Warp for this simulation. Figure 6.5 shows the application speedup achieved by Time Warp for the same simulation with 10 milliseconds of additional computation per event. Figure 6.6 shows the application speedup achieved by Time Warp for the same simulation with 20 milliseconds of additional computation per event. The additional computation is artificial, but is still indicative of realistic simulation problems. For example, a *Sim++* simulation of an existing telecommunication network averaged 40 milliseconds of computation per event.

With no additional computation per event, the potential speedup ranges from 5.8 to 14.8 as the number of processors is varied from 8 to 32. Time Warp achieves 28% of the potential speedup on 8 processors, with a drop to 17% on 32 processors. The decline is due primarily to an increase in the number of rollbacks and anti-messages, as the execution of the simulation becomes increasingly asynchronous with the addition of processors. This effect can be seen to varying degrees in all of the graphs for both simulations, although it is most pronounced when the amount of computation per event is low. With less computation per event, entities are more likely to execute forward in simulation time more quickly and are therefore more likely to execute events out of order. With 20 milliseconds of additional computation per event, Time Warp achieves 61% of the potential speedup on 8 processors, with a drop to 48% on 32 processors. In this case, the application speedup achieved by Time Warp ranges from 4.7 to 12.6 as the number of processors is varied from 8 to 32. [Bae91] reports speedups as high as 18 on 32 processors when each health care node is implemented as a single entity.

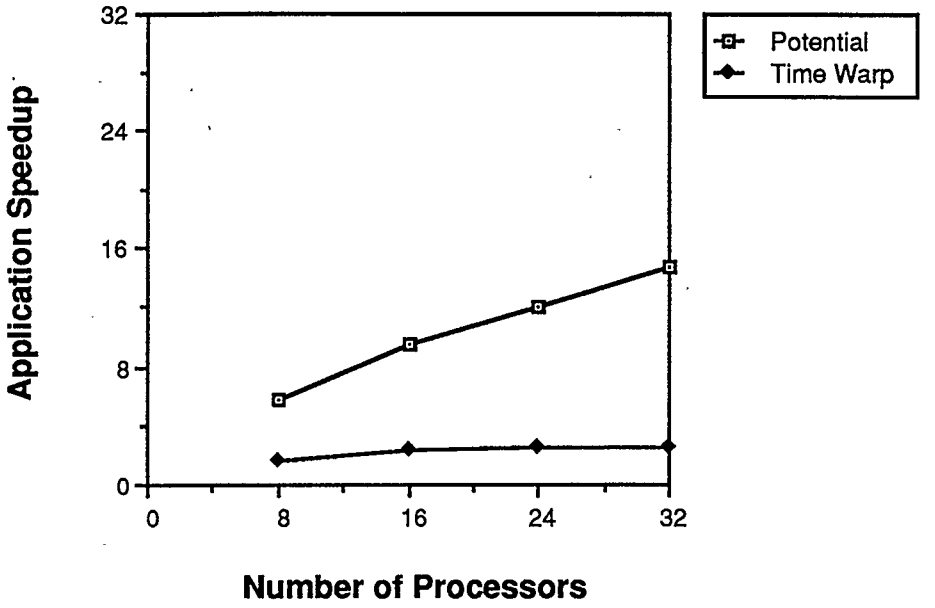

Figure 6.4: Application Speedup for the Health Care System
(no added computation)

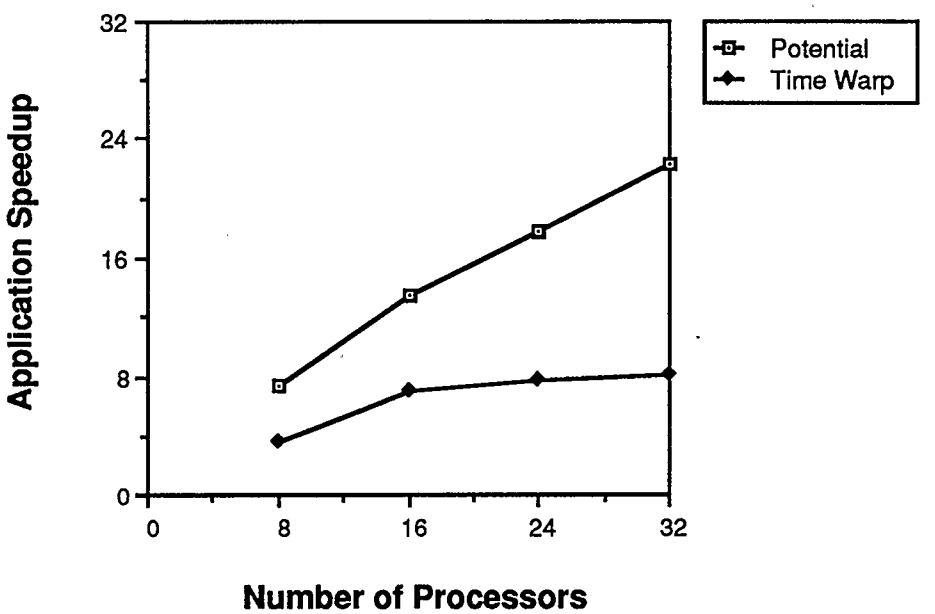

Figure 6.5: Application Speedup for the Health Care System
(10 milliseconds of added computation)

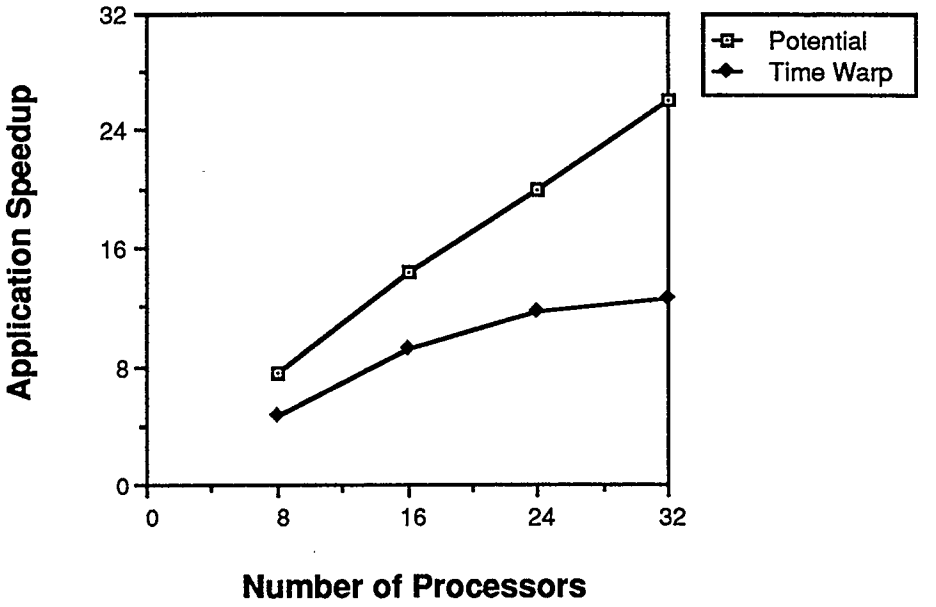

Figure 6.6: Application Speedup for the Health Care System
(20 milliseconds of added computation)

6.3 The Mobile Communication Network

The mobile communication network consists of multiple, indepedently acting physical components, called platforms, that broadcast and receive bulletins while moving on a fixed-size, two-dimensional surface. For simplicity, it is assumed that the surface is a torus with no fixed edges. Bulletins denote information packets transmitted among platforms and they model a simplified form of radio communication. When a platform broadcasts a bulletin, all platforms within range of the broadcast receive the information packet. Figure 6.7 illustrates the basic components of the mobile communication network.

6.3.1 Design Issues and Implementation

Simulating the mobile communication network is considerably different than simulating a

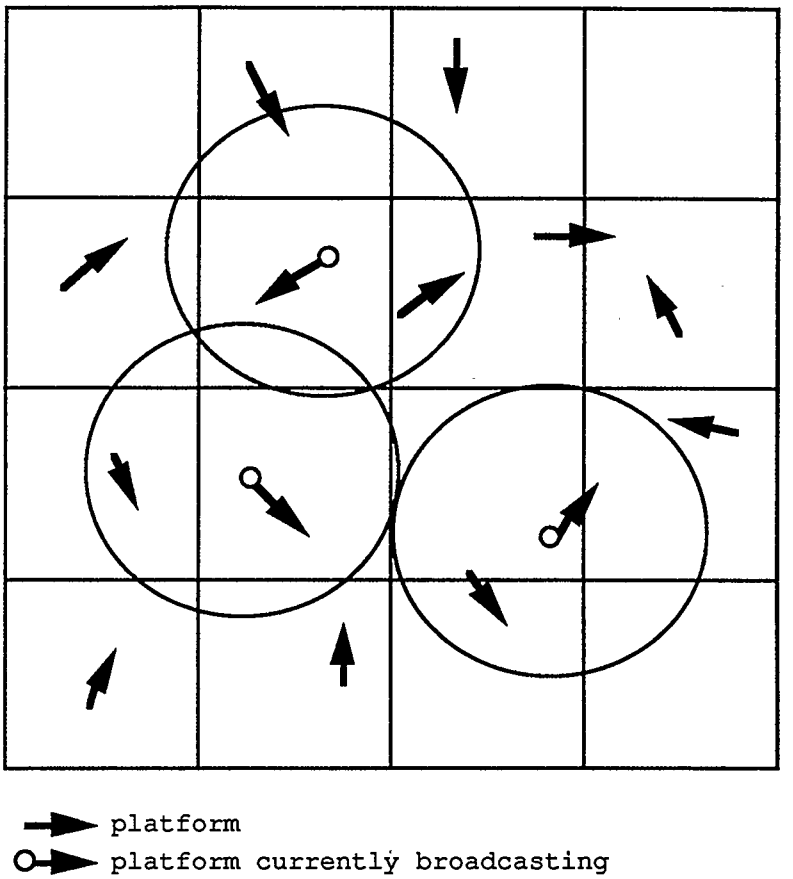

Figure 6.7: The Mobile Communication Network

queueing network. In the health care system, when a patient moves from a village to a health center, or from one health center to another, it is clear where the patient is moving from and where the patient is moving to. In contrast, when a platform broadcasts a bulletin, it does not explicitly identify which platforms are to receive the bulletin. Instead, the bulletin is delivered to exactly those platforms within range of the broadcast at the instant the broadcast takes place.

A straightforward implementation of this model would have each broadcasting platform schedule an event for every other platform in the simulation and have the receiving platform decide whether or not it was actually within range. Unfortunately, for any but a small

number of platforms, the communication overhead of scheduling one event for every entity for each broadcast easily dominates the execution of the simulation, and no speedup is possible.

A typical approach to this type of problem [Bec88, Cle90, Con90, Gol84, Lom88b] is to divide the two-dimensional surface into sectors and represent each sector by a sector entity. This is the approach used for the performance results that follow. Each sector entity knows the current trajectory of all platforms in its sector. Knowing the trajectory of a platform allows the sector entity to determine the current position of the platform at any instant in simulation time. As platforms change their trajectory, or move from one sector to another, the platforms are responsible for informing the appropriate sector entities. When a platform broadcasts a bulletin, it schedules an event for each sector entity that coincides with the area of the broadcast range. The sector entities, in turn, forward the event to all platforms determined to be in range of the broadcast.

The implementation of the mobile communication network is divided into two layers: a *spatial layer* and an *application layer*. The spatial layer implements the sector entities and provides primitives for modeling motion and broadcasting. The application layer uses these primitives to implement the mobile communication network model. All interactions with sector entities are performed by the spatial layer and are completely transparent to the application layer. The separation of these layers is complete – there is no application-specific knowledge in the spatial layer and there is no knowledge in the application layer about how the motion and broadcasting primitives are implemented. Indeed, several variations of the spatial layer were implemented and tested, without changes to the application layer. Appendix B presents a complete implementation of the application layer of the mobile communication network. The source code associated with the

```
class point {
   ...
   point(double x, double y);
};
event &operator<<(event &ev, point &pos);
event &operator>>(event &ev, point &pos);

class velocity {
   ...
   velocity(double direction, double speed);
};
event &operator<<(event &ev, velocity &vel);
event &operator>>(event &ev, velocity &vel);

class mobile_entity : public entity {
   ...
   void set_position(point &pos);
   void set_velocity(velocity &vel);
   point current_position();
   velocity current_velocity();
   void broadcast(double range, event &ev);
};
```

Figure 6.8: C++ Declarations for the Spatial Layer Interface

implementation of the spatial layer has been omitted. A substantial portion of that code deals with floating point errors and boundary conditions that obscure the key elements involved in managing motion and broadcasts.

The interface provided by the spatial layer is shown in figure 6.8. The primitives for modeling motion and broadcasting are member functions of entity class **mobile_entity**. The application layer further derives **mobile_entity** for the specific requirements of the application. All instances of an entity class derived from **mobile_entity** are referred to as *mobile* entities. The primitives **set_position** and **set_velocity** allow a mobile entity to initialize or change its position or velocity, respectively. The primitives **current_position** and **current_velocity** allow a mobile entity to determine its current position and velocity, respectively. The value returned by **current_velocity** is that specified in the most recent call to **set_velocity**. The value returned by **current_position** is a function of the position specified in the most recent call to

set_position, the amount of simulation time that has elapsed since the call to **set_position**, and the current velocity. The primitive **broadcast** schedules a given event for all mobile entities whose current position at the simulation time of the broadcast falls within the specified range.

In addition to a sectored implementation of the spatial layer, a *shared memory* implementation was also developed. In the shared memory implementation, there are no sector entities and all mobile entities are part of a single cluster. In this way, the spatial layer can always directly access the member variables and functions of all mobile entities in the simulation. Although this approach has no potential for speedup from parallel execution, it is representative of how this simulation might be implemented for efficient, sequential execution. For example, to implement **broadcast**, the spatial layer of the calling entity directly references the current position of each mobile entity in the simulation to determine which of those entities are within range of the broadcast and should receive the given event. As a result, the shared memory approach requires fewer events and is easier to implement than the sectored approach. Specifically, the shared memory implementation of the spatial layer required 268 lines of source code and the sectored implementation required 779 lines of source code.

Although the difference in code size suggests that an implementation of the mobile communication network for parallel execution is more difficult than for efficient sequential execution, this is not necessarily the case. Depending on the characteristics of the application, the sectored implementation may also be required for efficient sequential execution. For example, when the number of platforms is very large (e.g., 500), the shared memory implementation executing on the sequential simulator was found to execute as much as 35% slower than the sectored implementation executing on the sequential simulator, even though the sectored implementation scheduled five times as many events.

The reason for this is that the amount of computation required by the shared memory implementation to determine which of the 500 platforms is within range of each broadcast is greater than the amount of computation required to manage motion and broadcasting in the sectored implementation. A "best case" sequential implementation of this simulation would combine the use of sectors and shared memory, although no such implementation was attempted for this study. For the results that follow, the sectored implementation executing on the sequential simulator was the basis for all potential and application speedup figures.

Once again, a summary entity is used to gather and summarize statistics about the application. The sequential and parallel application times exclude the time required to execute the summary entity at the end of the simulation.

6.3.2 Simulation Parameters and Performance Results

The configuration of the mobile communication network used for all experiments consisted of 500 platforms and 16 sectors, for a total of 516 entities. The state size of each platform entity is approximately 6K, including language overhead and the state saved portion of an entity's run-time stack. Approximately 1K of this state is due to language overhead. The state size of each sector entity is approximately 10K. The average computation per event for this simulation is 3.66 milliseconds. Approximately 1 millisecond of this time is due to language overhead. For all experiments, platform and sector entities were mapped to processors using the static mapping scheme described in Section 6.2.2.

The values of key application parameters are shown in table 6.3. The size of the two-dimensional surface is fixed at 100 x 100 units. The size of each of the 16 sectors used for these experiments is fixed at 25 x 25 units. The broadcast range of each platform is fixed at

Parameter	Value
Duration	20.0 time units
Surface Size	100 x 100 units
Number of Sectors	16 (25 x 25 units each)
Number of Platforms	500
Broadcast Range	5.0 units
Speed	1.0 units per unit time
Event Rate	1.0 events per unit time
Broadcast Probability	0.5

Table 6.3: Application Parameters for the Mobile Communication Network

5.0 units. Platforms travel in a straight line with a fixed speed of 1.0 units of distance per unit time. The rate at which platforms broadcast a bulletin or change direction is referred to as the event rate and is based on a drawing from a negative exponential distribution with a mean of 1.0 events per unit time. This means that, on average, a platform broadcasts a bulletin or changes direction every unit of simulation time. There is equal probability that each such event will be a broadcast or a change in direction.

With a broadcast range of 5.0 units, each broadcast covers an area of 78.5 units^2 or, approximately, 0.8% of the total surface. As a result, each broadcast is typically confined to a single sector. Assuming that platforms are evenly distributed over the entire surface, each broadcast reaches 3 platforms, excluding the broadcasting platform. This means that, on average, a broadcasting platform schedules one event for the current sector, and that sector schedules three events for the platforms in range of the broadcast. All of the results that follow use this configuration of the application. However, with only minor changes to the application parameters, it would be possible to significantly impact the performance of the simulation. For example, by doubling the broadcast range, a single broadcast would typically reach 15 platforms, requiring 12 additional events per broadcast.

Figure 6.9 shows the application speedup achieved by Time Warp for this simulation.

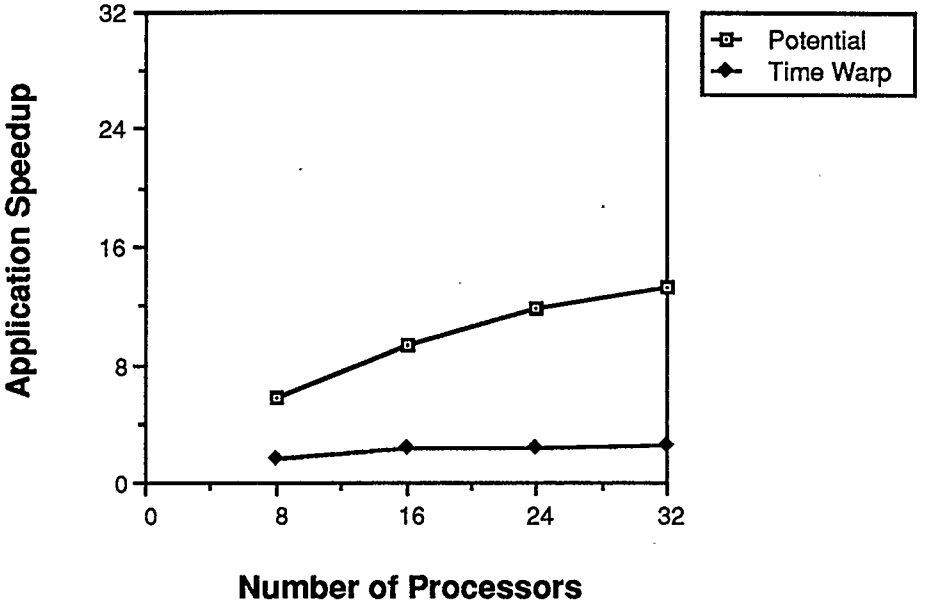

Figure 6.9: Application Speedup for the Mobile Communication Network (no added computation)

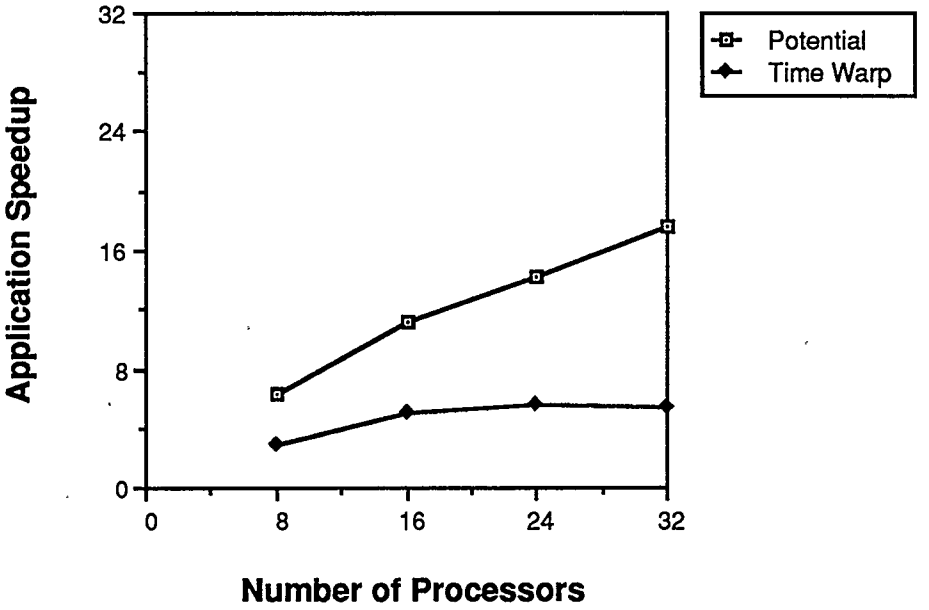

Figure 6.10: Application Speedup for the Mobile Communication Network (10 milliseconds of added computation)

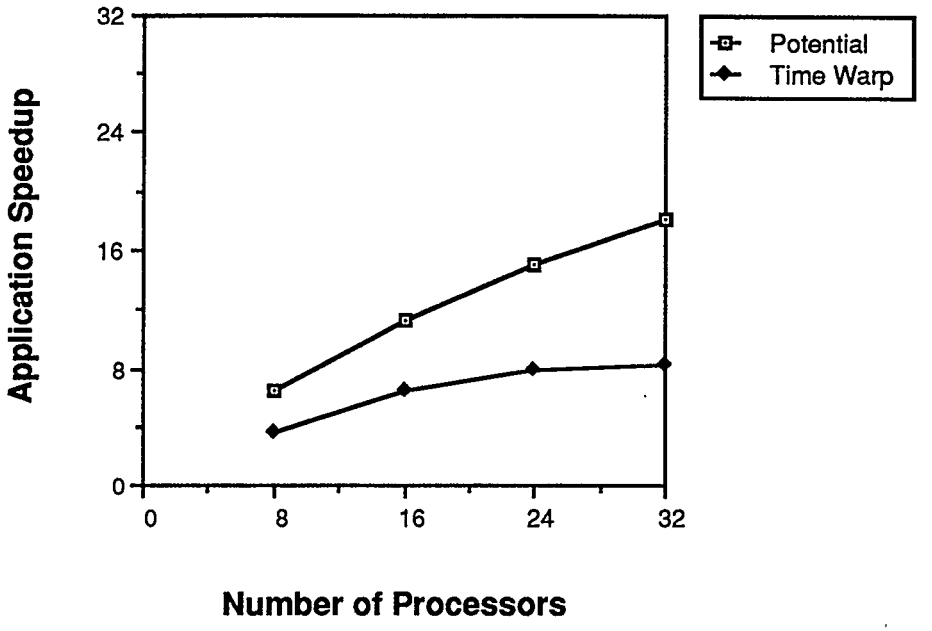

Figure 6.11: Application Speedup for the Mobile Communication Network (20 milliseconds of added computation)

Figure 6.10 shows the application speedup achieved by Time Warp for the same simulation with 10 milliseconds of additional computation per application event. Figure 6.11 shows the application speedup achieved by Time Warp for the same simulation with 20 milliseconds of additional computation per application event. Application events include only those events that are visible at the application layer. This excludes events used by platforms to interact with sector entities. It is assumed that the amount of computation required by sector entities to manage motion and broadcasting is already representative of realistic applications using sectors. For the configuration of the mobile communication network used for these experiments, approximately 40% of all events were used to interact with sector entities and were excluded from additional computation. As a result, with 20 milliseconds of additional computation per application event, the average computation over all events rose from 3.66 milliseconds to only 15.1 milliseconds.

With no additional computation per event, the potential speedup ranges from 5.7 to 13.3 as the number of processors is varied from 8 to 32. With 20 milliseconds of additional computation per application event, the potential speedup ranges from 6.4 to 18.2. For this simulation, Time Warp achieves application speedups ranging from a low of 1.6 to a high of 8.2. On 8 processors, the application speedup achieved by Time Warp is 28-56% of the potential speedup, depending on the amount of computation per event. On 32 processors, the application speedup achieved by Time Warp is 20-45% of the potential speedup.

6.4 Summary

This chapter described two discrete-event simulations that were implemented in *Sim*D and executed on Time Warp on a transputer-based, distributed-memory Meiko Computing Surface. The development of these simulations was one of several sources of feedback into the design of *Sim*D presented in Chapter 5. The mobile communication network is of particular interest for evaluating *Sim*D in that the implementation of the model required both a spatial layer and an application layer. Using *Sim*D, it was possible to completely separate the implementation of the two layers, making it possible to experiment with several alternative implementations of the spatial layer without changes to the application layer. This suggests that the constructs provided by *Sim*D are appropriate for implementing moderately complex parallel simulations consisting of multiple layers of software. The *Sim*D implementation of the resource competition facility shown in Chapter 5 is further evidence of this claim.

In addition to the health care system and the mobile communication network, I have also used *Sim*D to implement two other simulations. The first is a simulation of an

adaptive routing algorithm for a multi-hop, message-passing system. The second is a simulation of the Chandy-Misra approach to parallel simulation using deadlock detection and recovery. Whereas the purpose in simulating the health care system and the mobile communication network was to gain experience with *Sim*D and test its implementation on Time Warp, these other simulations were used to study the simulated system, not the simulation or the language.

The existing implementation of *Sim*D executes without errors, and simulations implemented in *Sim*D have been shown to execute efficiently on Time Warp. Speedups as high as 18 on 32 processors have been achieved with a *Sim*D implementation of the health care system. Nevertheless, the existing implementation of *Sim*D is a prototype and a more complete implementation is intended.

Chapter 7

Conclusion

This chapter presents a critique of *Sim*D, summarizes the contributions of this research, presents conclusions drawn from this research, and describes areas requiring further study and development.

7.1 Critique of *Sim*D

Much of Chapters 4 through 6 was devoted to demonstrating the superiority of *Sim*D over other languages for Virtual Time. It was shown how *Sim*D's data types and language constructs provide improved interfaces to equivalent constructs in *Sim++*. In addition, it was shown how the combined process-oriented and event-oriented capabilities of *Sim*D can be used to address limitations in *Sim++*, Extended Virtual Time, and Maisie. Chapter 4 showed how these limitations complicate the development of even very basic simulation models. Chapter 5 showed how the unique characteristics of *Sim*D can make simulations simpler, more readable, more type safe, and more modular. Chapter 6 demonstrated the effectiveness of *Sim*D for developing moderately complex simulations that execute efficiently on Time Warp.

In spite of *Sim*D's strengths, it does not address a number of difficulties. First, since *Sim*D is implemented in C++, it is plagued by many of the weaknesses of C++. For example, C++ does not support run-time type checking, automatic initialization of variables, or automatic garbage collection. In addition, C++ does not prevent references through invalid pointers or array references outside the bounds of an array. Based on my

experience, the lack of these capabilities makes C++ significantly more difficult to use than Simula or comparable languages where these capabilities are present. Second, *Sim*D does not satisfy all of the language design criteria proposed in Chapter 3. Most notably, *Sim*D does not enforce restrictions related to shared and global memory, nor are all transient errors caused by Time Warp transparent to the application. These limitations cannot be overcome without compiler support. Third, *Sim*D applications are required to invoke macros like **EVENT_TYPE** and **ENTITY_CLASS** to support the language implementation at run time. Eliminating these macros also requires compiler support. Finally, the existing implementation lacks facilities for data collection and reporting, random number generation, error handling, and file and console input and output. The current implementation of *Sim*D coexists with *Sim++* and is therefore able to use *Sim++* facilities that are not yet available in *Sim*D.

7.2 Thesis Summary

The goal of this research was to assess the impact of Virtual Time on simulation language design. This goal was divided into the following research questions addressed by this thesis and summarized below:

1. How do the characteristics of Virtual Time differ from sequential simulation?
2. How do the characteristics of Virtual Time impact simulation language design?
3. Can languages for Virtual Time be used to develop well-structured, efficient, parallel simulations?

Two fundamental differences between Virtual Time and sequential simulation were identified and the impact of these differences on parallel simulation was examined. First,

Virtual Time processes interact solely by sending and receiving timestamped messages. The primary advantage of message-passing is that processes can execute concurrently on multiple processors and need not progress through simulation time at a uniform rate. As such, there is a great deal of freedom in how processes are executed, as long as causality is maintained. The primary disadvantage of message-passing is that it is generally several orders of magnitude slower than interactions through shared memory. As such, messages cannot generally be employed as one-for-one substitutes for shared memory references. This makes it difficult to model problems in which many or all processes regularly access a large, global state. Second, Virtual Time simulations must be designed specifically for parallel execution if they are to achieve significant reductions in execution time. Specifically, it is necessary to limit the amount of computation performed by any one process, maximize the ability of processes to work in parallel, and minimize communication overhead. All of these requirements are specific to Virtual Time. They are not factors in the design of sequential simulations.

A set of language design criteria for Virtual Time was developed based on the differences between Virtual Time and sequential simulation, and the characteristics of existing languages for Virtual Time surveyed for this research. The proposed criteria are parallel efficiency, explicit costs, determinism, type-safety, transparency of the implementation, transparent scaleability, portability of applications, and enforced restrictions. The characteristics of languages for Virtual Time suggested by these criteria are as follows.

1. Languages for Virtual Time should encourage efficient, parallel programming practices with language constructs that have efficient, parallel implementations.
2. Operations in a parallel simulation that differ significantly in cost from equivalent operations in a sequential simulation should appear obviously different to the user.

3. Given the same input, a simulation should produce the same results regardless of the number of processors on which the simulation is executed or the mapping of processes to processors used.
4. Languages for Virtual Time should be fully type safe, including message-based interactions between processes.
5. The implementation of Virtual Time should be transparent to the application.
6. The number of processors used to execute a simulation, and the mapping of processes to processors should be transparent to the application.
7. Application programs should be capable of executing sequentially or in parallel on multiple operating systems and architectures without source code modifications.
8. Languages for Virtual Time should enforce restrictions imposed by the Virtual Time paradigm. Most notably, the language should prevent interactions between processes through shared or global memory.

Key limitations of existing languages for Virtual Time were identified. The most prevalent of these is the inability to restrict or extend the semantics of the event selection primitives provided by *Sim++*, Extended Virtual Time, and Maisie. These primitives allow a process to specify what events it is willing to receive at any given time. However, since the implicit actions and data structures associated with event selection are inaccessible to the application, the event selection primitives must frequently be used *as is* or circumvented altogether. In addition, attempts to define new primitives based on those provided by *Sim++* was found to be quite difficult. This is due to a strong interdependence among *Sim++* primitives that discourages modularity. Specifically, it was shown that it would be necessary to redefine many or all of the existing primitives in order to define the new primitives. These limitations were shown to make even very basic simulation models difficult to implement, requiring repeated and significant restructuring to cope with simple

model extensions. These limitations were the primary motivation for the development of *Sim*D.

A new language for Virtual Time called *Sim*D was defined. Among the contributions of this language are improved interfaces to many of the data types and language constructs provided by *Sim*++, and a combined process-oriented and event-oriented modeling paradigm that was shown to address the limitations of other languages for Virtual Time, as outlined above. Although differences between *Sim*D and Extended Virtual Time do not permit all of the performance optimizations developed for Extended Virtual Time [Lom88b] to be incorporated into *Sim*D, alternative optimizations were suggested for achieving comparable results. *Sim*D was implemented on an existing implementation of Time Warp developed and made available by Jade Simulations International Corporation.

The effectiveness of *Sim*D for developing well-structured, efficient, parallel simulations was demonstrated using several basic examples and two simulations implemented and executed on Time Warp. The primary purpose in developing these simulations was to gain experience with *Sim*D and to test the implementation of *Sim*D on Time Warp. The development of these simulations was one of several sources of feedback into the design of *Sim*D. A performance study of these simulations was presented.

In addition to the contributions of this thesis summarized above, this research resulted in two refereed conference papers on the subjects of parallel simulation language design [Bae90] and parallel simulation performance using Time Warp [Bae91].

7.3 Conclusions

The following conclusions are drawn from this research.

1. *Existing modeling practices and language design must be adapted to Virtual Time.* Unlike sequential simulations, Virtual Time simulations communicate and synchronize the actions of processes solely by sending and receiving timestamped messages, and Virtual Time simulations must be designed specifically for parallel execution if they are to achieve significant reductions in execution time. Although it is common in sequential simulations to decompose a problem into logical processes, this is typically a representational convenience without regard (or need) for the ability of those processes to work in parallel. Furthermore, sequential simulations rely on shared memory for process interactions. These abstractions are not suitable for Virtual Time. Instead, modeling practices and language design for Virtual Time must encourage techniques and language constructs that limit the amount of computation performed by any one process, maximize the ability of processes to work in parallel, and minimize communication overhead.
2. *Modeling practices and language design should not be adapted to accomodate specific implementations of Virtual Time.* There are two reasons for this. First, such adaptations require that the user understand fundamental characteristics of the implementation of Virtual Time, even though those characteristics are not inherent to the Virtual Time paradigm or to parallel processing. Second, additional requirements imposed by a specific implementation of Virtual Time increase the complexity of the design and implementation of a parallel simulation and reduce its portability to other implementations of Virtual Time.
3. *Currently, implementations of Time Warp cannot be made completely transparent to users.* This means that, in many cases, simulations must be designed specifically for Time Warp in order to achieve significant reductions in execution time or to execute at all. Specifically, simulations for Time Warp must deal with the issues of

process state size, side effects resulting from causality errors, and interactions with the external world. Although some of these issues can be addressed through compiler support, many require hardware support such as that provided by the rollback chip.

4. *The language design criteria proposed in this thesis – parallel efficiency, explicit costs, determinism, type-safety, transparency of the implementation, transparent scaleability, portability of applications, and enforced restrictions – promote the development of parallel simulations that are correct, efficient, and scaleable.* Many of these criteria are already proven in that they represent the best elements of existing languages for Virtual Time.
5. *Many existing languages for Virtual Time contain inherently sequential constructs or are difficult to use for representing even very basic simulation models.* ModSim contains a number of language constructs not suited to parallel execution. Chief among these are interactions through shared and global memory. *Sim++*, Extended Virtual Time, and Maisie support *event selection* whereby a process can specify what events it is willing to receive at any given time. Unfortunately, none of these languages provide any ability to restrict or extend the semantics of the event selection primitives. As a result, these primitives must frequently be used *as is* or circumvented altogether.
6. *SimD improves the structure of parallel simulations compared to other languages for Virtual Time. SimD*'s data types and language constructs provide improved interfaces to equivalent constructs in *Sim++*. In addition, the combined process-oriented and event-oriented capabilities of *SimD* can be used to address limitations in *Sim++*, Extended Virtual Time, and Maisie. These characteristics of *SimD* make resulting simulations simpler, more readable, more type safe, and more modular.

7. *Sim*D *can be used to develop parallel simulations that achieve significant reductions in execution time.* Speedups as high as 18 on 32 processors were achieved with the *Sim*D implementation of the health care system with 20 milliseconds of added computation per event [Bae91]. With no added computation per event, speedup never fell below 2.6 on 32 processors for either application presented in this thesis. These speedups were achieved without the benefit of optimizations such as integrating the simulation primitives with the underlying implementation of Time Warp [Lom88b] or special purpose hardware such as the rollback chip [Fuj88b]. Either of these optimizations should further improve the performance of these simulations on Time Warp.
8. *The language design process must include documentation, implementation, and experience with the language as feedback into the design.* In the development of *Sim*D, many prototype designs of *Sim*D features had to be further refined or even abandoned because they could not be easily described, because their implementation was too complex or inefficient, or because they were difficult to use.

7.4 Further Study and Development

The critique of *Sim*D identified a number of limitations with the existing language and its implementation in C++. The current implementation lacks a number of facilities common to simulation languages. A more complete implementation is intended and will include these additional facilities. The addition of compiler support is not planned. Although it would be possible to more fully satisfy the language design criteria proposed in this thesis using compiler support, significant changes to C++ would also be required to address many of its inherent weaknesses. Such changes are likely to be less successful than

designing a completely new language that incorporates the desired design criteria and language capabilities into the design of the language. This is the approach currently being used in the development of ModSim.

One area of parallel simulation language design not addressed by this research involves the dynamic creation and destruction of processes throughout the course of the simulation. Dynamic process creation and destruction generally also requires dynamic process migration since the implementation of Virtual Time cannot predict how much memory a given process will require at run time or the amount of computation that process will perform. Dynamic process migration continues to be an active area of research [Rei90b]. The impact of dynamic process migration on the language implementation is significant. Most notably, the state of a process must be completely independent of the area of memory in which it resides so that the process can migrate to a different area of memory on a different processor. Alternatively, the implementation of the language must be able to identify all absolute addresses in the state of a process and adjust those addresses when the process is migrated. Although solutions to these problems can be implemented in software, much of the overhead associated with software techniques can be eliminated using the rollback chip [Fuj88b].

Finally, a topic related to language design for Virtual Time is modeling techniques for Virtual Time. The fundamental differences between Virtual Time and sequential simulation require that simulations be designed specifically for Virtual Time if they are to achieve significant reductions in execution time. Specifically, it is necessary to limit the amount of computation performed by any one process, maximize the ability of processes to work in parallel, and minimize communication overhead. Very little research has been done on the impact of modeling techniques on the readability of simulations. Baezner et al [Bae89] examines the impact of various modeling techniques on the performance of simulations

executed on Time Warp, but does not address the impact of these changes on the readability of the simulation. Baezner et al [Bae90] examines the impact of *Sim++* clusters on the performance and readability of simulations. Finally, the results reported by Baezner et al [Bae91] for a *Sim*D implementation of the health care system executing on Time Warp are 40% better than those reported in this thesis. The additional speedup was achieved by implementing each health care node as one entity, rather than two. This approach results in less readable code, however. Generally, the impact of modeling techniques on the readability of parallel simulations has yet to be addressed for a broad range of applications.

In addition to the readability of parallel simulations, other modeling issues include the decomposition of a model for significant reductions in execution time across a wide range of input parameters, and increasing the level of detail in a model without significant restructuring of the implementation for continued, acceptable performance. Unfortunately, it is not always possible to know a priori what the range of input parameters to a simulation is likely to be, or what their impact will be on parallel performance. In addition, it is often the case that simulations are subject to change after they have been designed and implemented, where such changes could significantly alter parallel performance. The impact of these issues has yet to be addressed.

Bibliography

[Abr89] M. Abrams. A Common Programming Structure for Bryant-Chandy-Misra, Time Warp, and Sequential Simulators. *Proceedings of the 1989 Winter Simulation Conference*, pages 661-670, December 1989.

[Abr90] M. Abrams and G. Lomow. *Design Issues in General Purpose, Parallel Simulation Languages*. Technical Report No. TR-89-38, Department of Computer Science, Virginia Tech, January 1990.

[Bae89] D. Baezner, J. Cleary, G. Lomow, and B.W. Unger. Algorithmic Optimizations of Simulations on Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 21 No. 2, pages 73-78, The Society for Computer Simulation, March 1989.

[Bae90] D. Baezner, G. Lomow, and B.W. Unger. *Sim++*: The Transition to Distributed Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 22 No. 1, pages 211-218, The Society for Computer Simulation, January 1990.

[Bae91] D.Baezner, G. Lomow, and B.W. Unger. Jade's Parallel Simulation Environment: *Sim++* and TimeWarp. To appear at *6th Distributed Memory Computing Conference*, April 1991.

[Bag90] R.L. Bagrodia and W. Liao. Maisie: A Language and Optimizing Environment for Distributed Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 22 No. 1, pages 205-210, The Society for Computer Simulation, January 1990.

[Bec88] B. Beckman, M. Di Loreto, K. Sturdevant, P. Hontalas, L. Van Warren, L. Blume, D. Jefferson, and S. Bellenot. Distributed Simulation and Time Warp (Part 1: Design of Colliding Pucks). *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 19 No. 3, pages 56-60, The Society for Computer Simulation, February 1988.

[Bel89] R. Belanger, B. Donovan, K. Morse, S. Rice, and D. Rockower. *ModSim: A Language for Object-Oriented Simulation (Reference Manual)*, CACI Products Company, October 1989.

[Bel90] S. Bellenot. Global Virtual Time Algorithms. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 22 No. 1, pages 122-127, The Society for Computer Simulation, January 1990.

[Ber86] O. Berry. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism.* PhD thesis, University of Southern California, February 1986.

[Béz88] J. Bézivin. Design and Implementation Issues in Object-Oriented Simulation. *Simuletter*, Vol. 19 No. 2, pages 47-53, June 1988.

[Bir79] G.M. Birtwistle. *Demos: A System for Discrete Event Modelling on Simula.* The Macmillan Press Ltd., 1979.

[Bir84] G. Birtwistle, G. Lomow, B. Unger, and P. Luker. Process Style Packages for Discrete Event Modeling: Using Simula's Class Simulation. *Transactions of the Society for Computer Simulation*, Vol. 1 No. 2, pages 175-195, December 1984.

[Bir86] G. Birtwistle, G. Lomow, B. Unger, and P. Luker. Process Style Packages for Discrete Event Modeling: Demos: A Process Based Simulation Package. *Transactions of the Society for Computer Simulation*, Vol. 3 No. 4, pages 279-316, October 1986.

[Bry89] O.F. Bryan, Jr. ModSim II – An Object Oriented Simulation Language for Sequential and Parallel Processors. *Proceedings of the 1989 Winter Simulation Conference*, pages 122-127, December 1989.

[Buz90] C.A. Buzzell, M.J. Robb, and R.M. Fujimoto. Modular VME Rollback Hardware for Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 22 No. 1, pages 153-156, The Society for Computer Simulation, January 1990.

[Cha81] K.M. Chandy and J. Misra. Asynchronous Distributed Simulation Via a Sequence of Parallel Computations. *Communications of the ACM*, Vol. 24 No. 11, pages 198-206, April 1981.

[Cle90] J.G. Cleary. Colliding Pucks Solved Using a Temporal Logic. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 22 No. 1, pages 219-224, The Society for Computer Simulation, January 1990.

[Con90] D. Conklin, J. Cleary, and B. Unger. The Sharks World (A Study in Distributed Simulation Design). *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 22 No. 1, pages 157-160, The Society for Computer Simulation, January 1990.

[Dah72] O.J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula 67 Common Base Language*, Norwegian Computing Center Pub. S-52, Norwegian Computing Center, Oslo, 1972.

[Ebl89] M. Ebling, M. Di Loreto, M. Presley, F. Wieland, and D. Jefferson. An Ant Foraging Model Implemented on the Time Warp Operating System. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 21 No. 2, pages 21-26, The Society for Computer Simulation, March 1989.

[Fuj87] R.M. Fujimoto. *Performance Measurements of Distributed Simulation Strategies*. Technical Report No. UUCS-87-026a, Computer Science Department, University of Utah, November 1987.

[Fuj88a] R.M. Fujimoto. *Time Warp on a Shared Memory Multiprocessor*. Technical Report No. UUCS-88-021a, Computer Science Department, University of Utah, November 1988.

[Fuj88b] R.M. Fujimoto, J. Tsai, and G.C. Gopalakrishnan. *Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp*. Technical Report No. UUCS-88-011, Computer Science Department, University of Utah, July 1988.

[Fuj89] R.M. Fujimoto. Parallel Discrete Event Simulation. *Proceedings of the 1989 Winter Simulation Conference*, pages 19-28, December 1989.

[Fuj90] R.M. Fujimoto. Performance of Time Warp Under Synthetic Workloads. *Proceedings of the SCS Multiconference on Distributed Simulation*,

Simulation Series Vol. 22 No. 1, pages 23-28, The Society for Computer Simulation, January 1990.

[Fut88] I. Futo. Distributed Simulation on Prolog Basis. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 19 No. 3, pages 160-165, The Society for Computer Simulation, February 1988.

[Gat88] B. Gates and J. Marti. An Empirical Study of Time Warp Request Mechanisms. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 19 No. 3, pages 73-80, The Society for Computer Simulation, February 1988.

[Gol84] A.P. Goldberg. *Object-Oriented Simulation of Pool Ball Motion.* Master's thesis, University of California, Los Angeles, 1984.

[Hon89] P. Hontalas, B. Beckman, M. Di Loreto, L. Blume, P. Reiher, K. Sturdevant, L. Van Warren, J. Wedel, F. Wieland, and D. Jefferson. Performance of the Colliding Pucks Simulation on the Time Warp Operating System. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 21 No. 2, pages 3-7, The Society for Computer Simulation, March 1989.

[Jad90] *The Sim++ Programmer Reference Manual*, Jade Simulations International Corporation, 1990.

[Jef84] D. Jefferson. Future Directions in Simulation at the Conference on Simulation in Strongly Typed Languages (Panel). *Proceedings of the Conference on Simulation in Strongly Typed Languages*, Simulation Series

Vol. 13 No. 2, pages 123-124, The Society for Computer Simulation, February 1984.

[Jef85a] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 15 No. 2, pages 63-69, The Society for Computer Simulation, January 1985.

[Jef85b] D.R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, Vol. 7 No. 3, pages 404-425, Association for Computing Machinery, July 1985.

[Jef87] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Di Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, L. Van Warren, J. Wedel, H. Younger, and S. Bellenot. Distributed Simulation on the Time Warp Operating System. *Operating Systems Review (Proceedings of the Eleventh ACM Symposium on Operating Systems Principles)*, Vol. 21 No. 5, pages 77-93, November 1987.

[Jef90] D. Jefferson. Virtual Time II: Storage Management in Distributed Simulation. To appear in *Principles of Distributed Computation.*

[Jen89] R.A. Jenkins. New Approaches in Parallel Computing. *Computers in Physics*, pages 24-32, January 1989.

[Ker78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall Inc., 1978.

[Leu89] E. Leung, J. Cleary, G. Lomow, D. Baezner, and B. Unger. The Effects of Feedback on the Performance of Conservative Algorithms. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 21 No. 2, pages 44-49, The Society for Computer Simulation, March 1989.

[Li89] X. Li. *CSP* – A Distributed Logic Programming Language for Discrete Event Simulation.* PhD thesis, University of Calgary, June 1989.

[Lom88a] G. Lomow, J. Cleary, B. Unger, and D. West. A Performance Study of Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 19 No. 3, pages 50-55, The Society for Computer Simulation, February 1988.

[Lom88b] G. Lomow. *The Process View of Distributed Simulation.* PhD thesis, University of Calgary, September 1988.

[Lom89] G. Lomow and D. Baezner. A Tutorial Introduction to Object-Oriented Simulation and *Sim++*. *Proceedings of the 1989 Winter Simulation Conference*, pages 140-146, December 1989.

[Lom91] G. Lomow, S.R. Das, R.M. Fujimoto. User Cancellation of Events in Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 23 No. 1, pages 55-62, The Society for Computer Simulation, January 1991.

[Mar88] J. Marti. RISE: The Rand Integrated Simulation Environment. *Proceedings of the SCS Multiconference on Distributed Simulation,*

Simulation Series Vol. 19 No. 3, pages 68-72, The Society for Computer Simulation, February 1988.

[Mis86] J. Misra. Distributed Discrete-Event Simulation. *Computing Surveys*, Vol. 18 No. 1, pages 39-65, March 1986.

[Mul82] A. Mullarney, B.E. Rector, and G.D. Johnson, *Simscript II.5 Programming Language*, CACI, 1982.

[Pre89a] B.R. Preiss. The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 21 No. 2, pages 139-144, The Society for Computer Simulation, March 1989.

[Pre89b] M. Presley, M. Ebling, F. Wieland, and D. Jefferson. Benchmarking the Time Warp Operating System with a Computer Network Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 21 No. 2, pages 8-13, The Society for Computer Simulation, March 1989.

[Ree88] D.A. Reed, A.D. Malony, and B.D. McCredie. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering*, Vol. 14 No. 4, pages 541-553, April 1988.

[Rei90a] P. Reiher, R. Fujimoto, S. Bellenot, and D. Jefferson. Cancellation Strategies in Optimistic Execution Systems. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 22 No. 1, pages 112-121, The Society for Computer Simulation, January 1990.

[Rei90b] P. Reiher and D. Jefferson. Virtual Time Based Dynamic Load Management in the Time Warp Operating System. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 22 No. 1, pages 103-111, The Society for Computer Simulation, January 1990.

[Rig89] R. Righter and J.C. Walrand. Distributed Simulation of Discrete Event Systems. *Proceedings of the IEEE*, Vol. 77 No. 1, pages 99-113, January 1989.

[Sle85] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the Association for Computing Machinery*, Vol. 32 No. 3, pages 652-686, July 1985.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1986.

[Ung90] B. Unger, J. Cleary, A. Dewar, and Z. Xiao., A Multi-Lingual Optimistic Distributed Simulator. *Transactions of the Society for Computer Simulation*, Vol. 7 No. 2, pages 121-151, June 1990.

[Wag91] D.B. Wagner. Algorithmic Optimizations of Conservative Parallel Simulations. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 23 No. 1, pages 25-32, The Society for Computer Simulation, January 1991.

[Wes85] J. West. *Object-Oriented Distributed Simulation.* Technical Report, CACI, 1985.

[Wes88a] D. West. *Optimising Time Warp: Lazy Rollback and Lazy Reevaluation.* Master's thesis, University of Calgary, January 1988.

[Wes88b] J. West and A. Mullarney. ModSim: A Language for Distributed Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 19 No. 3, pages 155-159, The Society for Computer Simulation, February 1988.

[Wie89] F. Wieland, L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson. Distributed Combat Simulation and Time Warp: The Model and its Performance. *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series Vol. 21 No. 2, pages 14-20, The Society for Computer Simulation, March 1989.

Appendix A

*Sim*D Implementation of the Health Care System

```
1   #include <SimD.h>
2
3   // globals for command-line arguments
4   extern sim_time DURATION;
5   extern double TREAT_PROB;
6   extern sim_time ASSESS_TIME;
7   extern sim_time TREAT_TIME;
8   extern double ARRV_RATE;
9   extern int SEED;
10  extern int NUMBER_OF_LEVELS;
11  extern int BRANCHING_FACTOR;
12
13  // event types
14  EVENT_TYPE(INITIAL);
15  EVENT_TYPE(PATIENT);
16  EVENT_TYPE(TREATED);
17  EVENT_TYPE(REPORT);
18
19  // summary entity class
20  // (statistics collection and reporting)
21  class summary : public entity {
22      sim_tally_obj Generated;
23      sim_tally_obj Assessed;
24      sim_tally_obj Treated;
25      sim_tally_obj Referred;
26      sim_tally_obj QLength;
27      sim_tally_obj QWait;
28  public:
29      summary(event &initial_ev);
30      void body();
31  };
32  ENTITY_CLASS(summary, entity);
33
34  // village entity class
35  class village : public entity {
36      int Outstanding;
37      int Generated;
38      entity_id Local_health_center;
39      sim_negexp_obj Interarrival_time;
40  public:
41      village(event &initial_ev);
42      void handle_treated(event &treated_ev);
43      void body();
44      void generate_report();
45  };
46  ENTITY_CLASS(village, entity);
47
48  // health center entity class
49  class health_center : public entity {
50      int Assessed;
51      int Treated;
52      int Referred;
53      int Personnel;
54      entity_id Next_health_center;
55      sim_draw_obj Treatable;
56      event_list *Patients;
57  public:
```

```
58          health_center(event &initial_ev);
59          void handle_report(event &report_ev);
60          void body();
61          void treat_patient(event &patient_ev);
62          void refer_patient(event &patient_ev);
63          void handle_patient(event &patient_ev);
64      };
65      ENTITY_CLASS(health_center, entity);
```

```
66      // initialize the summary entity
67      summary::summary(event &initial_ev)
68      {
69          // initialize statistics objects
70          Generated = sim_tally_obj("Generated");    // patients generated
71          Assessed = sim_tally_obj("Assessed");      // patients assessed
72          Treated = sim_tally_obj("Treated");        // patients treated
73          Referred = sim_tally_obj("Referred");      // patients referred
74          QLength = sim_tally_obj("QLength");        // length of patient queues
75          QWait = sim_tally_obj("QWait");            // wait time in patient queues
76      }
77
78      // main actions of summary entity
79      void summary::body()
80      {
81          int i;
82          int generated;
83          int assessed;
84          int treated;
85          int referred;
86          double qlength;
87          double qwait;
88          class_id vill_id, heal_id;
90          event report_ev;
91          sim_file_id file_id;
92
93          // wait for all villages to report
94          vill_id = class_id("village");
95          for (i = 0; i < vill_id.size(); i = i + 1) {
96              wait(report_ev, type_p == REPORT);
97              report_ev >> generated;
98              Generated.update(generated);
99              sim_trace(1, "received report from %s", report_ev.sched_by().name());
100         }
101
102         // instruct all health centers to report
103         heal_id = class_id("health_center");
104         for (i = 0; i < heal_id.size(); i = i + 1) {
105             report_ev = event(heal_id[i], 0.0, REPORT);
106             schedule(report_ev);
107         }
108
109         // wait for all health centers to report
110         for (i = 0; i < heal_id.size(); i = i + 1) {
111             wait(report_ev, type_p == REPORT);
112             report_ev >> qwait >> qlength >> referred >> treated >> assessed;
113             Assessed.update(assessed);
114             Treated.update(treated);
115             Referred.update(referred);
116             QLength.update(qlength);
117             QWait.update(qwait);
118             sim_trace(1, "received report from %s", report_ev.sched_by().name());
119         }
120
121         // write report file
122         file_id = sim_fopen("report", "w");
123         sim_fprintf(file_id, "%s", sim_tally_heading());
```

```
124         Generated.freport(file_id);
125         Assessed.freport(file_id);
126         Treated.freport(file_id);
127         Referred.freport(file_id);
128         QLength.freport(file_id);
129         QWait.freport(file_id);
130         sim_fclose(file_id);
131     }
```

```
132     // initialize village entity
133     village::village(event &initial_ev)
134     {
135        initial_ev >> Interarrival_time;
136        initial_ev >> Local_health_center;
137
138        Outstanding = 0;
139        Generated = 0;
140
141        prehandler(type_p == TREATED, village::handle_treated);
142     }
143
144     // prehandler to intercept returning villagers
145     void village::handle_treated(event &treated_ev)
146     {
147        sim_trace(1, "received treated villager from %s",
148                treated_ev.sched_by().name());
149
150        Outstanding = Outstanding - 1;
151        if (Outstanding == 0) generate_report();
152     }
153
154     // main actions of village entity
155     void village::body()
156     {
157        event patient_ev;
158
159        while (time() < DURATION) {
160           Outstanding = Outstanding + 1;
161
162           // hold until it is time to generate the next patient
163           hold(Interarrival_time.sample());
164           Generated = Generated + 1;
165
166           patient_ev = event(Local_health_center, 0.0, PATIENT);
167           // where to return the patient after treatment
168           patient_ev << self();
169
170           sim_trace(1, "sending sick villager to %s",
171                   Local_health_center.name());
172
173           schedule(patient_ev);
174        }
175     }
176
177     // send statistics report to summary entity
178     void village::generate_report()
179     {
180        event report_ev;
181
182        sim_trace(1, "sending report");
183
184        report_ev = event(entity_id("summary1"), 0.0, REPORT);
185        report_ev << Generated;
186        schedule(report_ev);
187     }
```

```
188     // initialize health center entity
189     health_center::health_center(event &initial_ev)
190     {
191        initial_ev >> Treatable;
192        initial_ev >> Next_health_center;
193        initial_ev >> Personnel;
194
195        Assessed = 0;
196        Treated = 0;
197        Referred = 0;
198
199        // queue for waiting patients
200        Patients = new event_list("patients");
201        Patients->reset();
202
203        prehandler(type_p == REPORT, health_center::handle_report);
204        posthandler(type_p == PATIENT, health_center::handle_patient);
205     }
206
207     // prehandler to intercept report event
208     void health_center::handle_report(event &report_ev)
209     {
210        sim_trace(1, "sending report");
211
212        report_ev = event(entity_id("summary1"), 0.0, REPORT);
213        report_ev << Assessed << Treated << Referred;
214        report_ev << Patients->qlength().avg() << Patients->qwait().avg();
215        schedule(report_ev);
216     }
217
218     // main actions of health center entity
219     void health_center::body()
220     {
221        event patient_ev;
222
223        while (TRUE) {
224           // get the next patient
225           if (Patients->empty())
226              wait(patient_ev, type_p == PATIENT);
227           else
228              patient_ev = Patients->retrieve(type_p == PATIENT);
229
230           if (Treatable.sample())      {
231              // treat and return patient home
232              treat_patient(patient_ev);
233           }
234           else {
235              // the patient cannot be treated at this level
236              refer_patient(patient_ev);
237           }
238        }
239     }
240
241     // treat and return the patient to the patient's home village
242     void health_center::treat_patient(event &patient_ev)
243     {
244        entity_id home_village;
```

```
245         event treated_ev;
246
247         // assess and treat the patient
248         hold((ASSESS_TIME + TREAT_TIME) / Personnel);
249         Assessed = Assessed + 1;
250         Treated = Treated + 1;
251
252         patient_ev >> home_village;
253         treated_ev = event(home_village, 0.0, TREATED);
254
255         sim_trace(1, "returning treated patient to %s", home_village.name());
256
257         schedule(treated_ev);
258      }
259
260      // send the patient up the hierarchy to a more sophisticated health center
261      void health_center::refer_patient(event &patient_ev)
262      {
263         entity_id home_village;
264
265         // assess but do not treat the patient
266         hold(ASSESS_TIME / Personnel);
267         Assessed = Assessed + 1;
268         Referred = Referred + 1;
269
270         patient_ev >> home_village;
271         patient_ev = event(Next_health_center, 0.0, PATIENT);
272         patient_ev << home_village;
273
274         sim_trace(1, "referring patient to %s", Next_health_center.name());
275
276         schedule(patient_ev);
277      }
278
279      // posthandler for enqueueing patients that arrive while another patient
280      // is being treated
281      void health_center::handle_patient(event &patient_ev)
282      {
283         // enqueue the patient
284         Patients->store(patient_ev);
285      }
```

```
286     // default settings for command-line arguments
287     sim_time DURATION           = 100.0;
288     double TREAT_PROB           = 0.9;
289     sim_time ASSESS_TIME        = 0.3;
290     sim_time TREAT_TIME         = 1.0;
291     double ARRV_RATE            = 0.3;
292     int SEED                    = 10079;
293     int NUMBER_OF_LEVELS        = 5;
294     int BRANCHING_FACTOR        = 4;
295
296     // the initial function; this function is run from the command-line
297     void initialize(int argc, char *argv[])
298     {
299        event initial_ev;
300        class_id heal_id;
301        int i, j, cur_node;
302        int next_health_center;
303        int last_position;
304        int branch_count;
305
306        // parse the command-line arguments
307        if (argc > 1) DURATION = atof(argv[1]);
308        if (argc > 2) TREAT_PROB = atof(argv[2]);
309        if (argc > 3) ASSESS_TIME = atof(argv[3]);
310        if (argc > 4) TREAT_TIME = atof(argv[4]);
311        if (argc > 5) ARRV_RATE = atof(argv[5]);
312        if (argc > 6) SEED = atoi(argv[6]);
313        if (argc > 7) NUMBER_OF_LEVELS = atoi(argv[7]);
314        if (argc > 8) BRANCHING_FACTOR = atoi(argv[8]);
315
316        ASSERT(DURATION > 0.0);
317        ASSERT(TREAT_PROB >= 0.0);
318        ASSERT(ASSESS_TIME >= 0.0);
319        ASSERT(TREAT_TIME >= 0.0);
320        ASSERT(ARRV_RATE > 0.0);
321        ASSERT(SEED > 0);
322        ASSERT(NUMBER_OF_LEVELS > 0);
323        ASSERT(BRANCHING_FACTOR > 1);
324
325        sim_trace(1, "run-time arguments");
326        sim_trace(1, "DURATION = %f", DURATION);
327        sim_trace(1, "TREAT_PROB = %f", TREAT_PROB);
328        sim_trace(1, "ASSESS_TIME = %f", ASSESS_TIME);
329        sim_trace(1, "TREAT_TIME = %f", TREAT_TIME);
330        sim_trace(1, "ARRIV_RATE = %f", ARRV_RATE);
331        sim_trace(1, "SEED = %d", SEED);
332        sim_trace(1, "LEVELS = %d", NUMBER_OF_LEVELS);
333        sim_trace(1, "BRANCHING = %d", BRANCHING_FACTOR);
334
335        // create the health care hierarchy
336        next_health_center = -1;
337        last_position = 0;
338        branch_count = BRANCHING_FACTOR - 1;
339        heal_id = class_id("health_center");
340
341        for (i = 1; i <= NUMBER_OF_LEVELS; i = i + 1) {
342           for (j = 1; j <= (int) pow(BRANCHING_FACTOR, i - 1); j = j + 1) {
```

```
343             initial_ev = event(NO_ENTITY_ID, 0.0, INITIAL);
344             cur_node = last_position + j - 1;
345             if (cur_node == 0) {
346                 initial_ev << (int) pow(2, NUMBER_OF_LEVELS - i);
347                 initial_ev << NO_ENTITY_ID;
348                 initial_ev << sim_draw_obj("treatable", 1.0,
349                                     sim_randint(1, 10000, SEED));
350             }
351             else {
352                 initial_ev << (int) pow(2, NUMBER_OF_LEVELS - i);
353                 initial_ev << heal_id[next_health_center];
354                 initial_ev << sim_draw_obj("treatable", TREAT_PROB,
355                                     sim_randint(1, 10000, SEED));
356             }
357
358             create("health_center", initial_ev);
359             sim_trace(1, "created health center %d", cur_node);
360
361             initial_ev = event(NO_ENTITY_ID, 0.0, INITIAL);
362             initial_ev << heal_id[cur_node];
363             initial_ev << sim_negexp_obj("interarrival time", ARRV_RATE,
364                                 sim_randint(1, 10000, SEED));
365
366             create("village", initial_ev);
367             sim_trace(1, "created village %d", cur_node);
368
369             branch_count += 1;
370             if (branch_count == BRANCHING_FACTOR) {
371                 branch_count = 0;
372                 next_health_center += 1;
373             }
374         }
375         last_position += (int) pow(BRANCHING_FACTOR, i - 1);
376     }
377
378     create("summary", NO_EVENT);
379
380     sim_trace(1, "initialized simulation");
381 }
382
383 INITIAL_FUNCTION(initialize);
```

Appendix B

*Sim*D Implementation of the Mobile Communication Network

```
1   #include <SimD.h>
2   #include "spatial.h"
3
4   // globals for command-line arguments
5   extern sim_time     DURATION;
6   extern int          PLATFORMS;
7   extern double       RANGE;
8   extern double       SPEED;
9   extern double       EVENT_RATE;
10  extern double       BCAST_PROB;
11  extern int          SEED;
12
13  // event types
14  EVENT_TYPE(INITIAL);
15  EVENT_TYPE(BULLETIN);
16  EVENT_TYPE(REPORT);
17
18  // summary entity class
19  // (statistics collection and reporting)
20  class summary : public entity {
21  public:
22      summary(event &initial_ev);
23      void body();
24  };
25  ENTITY_CLASS(summary, entity);
26
27  // platform entity class
28  class platform : public mobile_entity {
29      sim_negexp_obj Event_rate_gen;
30      sim_draw_obj Bcast_prob_gen;
31      sim_uniform_obj Direction_gen;
32
33      int Bcasts_sent;
34      int Bcasts_rcvd;
35  public:
36      platform(event &initial_ev);
37      void handle_bulletin(event &bulletin_ev);
38      void body();
39      void generate_report();
40  };
41  ENTITY_CLASS(platform, mobile_entity);
```

```
42      // initialize the summary entity
43      summary::summary(event &initial_ev)
44      {
45      }
46
47      // main actions of summary entity
48      void summary::body()
49      {
50          int i;
51          event report_ev;
52          int bcasts_sent;
53          int bcasts_rcvd;
54          int total_bcasts_sent;
55          int total_bcasts_rcvd;
56          sim_file_id file_id;
57
58          file_id = sim_fopen("report", "w");
59
60          // wait for all platforms to report
61          total_bcasts_sent = 0;
62          total_bcasts_rcvd = 0;
63          for (i = 0; i < PLATFORMS; i += 1) {
64              wait(report_ev, type_p == REPORT);
65              report_ev >> bcasts_rcvd >> bcasts_sent;
66
67              // print a report for each platform
68              sim_fprintf(file_id, "Report for %s", report_ev.sched_by().name());
69              sim_fprintf(file_id, "");
70              sim_fprintf(file_id, "    bcasts sent      = %d", bcasts_sent);
71              sim_fprintf(file_id, "    bcasts rcvd      = %d", bcasts_rcvd);
72              sim_fprintf(file_id, "");
73
74              total_bcasts_sent += bcasts_sent;
75              total_bcasts_rcvd += bcasts_rcvd;
76          }
77
78          // print totals
79          sim_fprintf(file_id, "Totals");
80          sim_fprintf(file_id, "");
81          sim_fprintf(
82              file_id, "    total bcasts sent      = %d",
83              total_bcasts_sent);
84          sim_fprintf(
85              file_id, "    total bcasts rcvd      = %d",
86              total_bcasts_rcvd);
87          sim_fprintf(file_id, "");
88
89          sim_fclose(file_id);
90      }
```

```
91      // initialize the platform entity
92      platform::platform(event &initial_ev)
93      {
94          point start_pos;
95
96          initial_ev >> Direction_gen;
97          initial_ev >> Bcast_prob_gen;
98          initial_ev >> Event_rate_gen;
99          initial_ev >> start_pos;
100
101         Bcasts_sent = 0;
102         Bcasts_rcvd = 0;
103
104         set_position(start_pos);
105         set_velocity(velocity(Direction_gen.sample(), SPEED));
106
107         prehandler(type_p == BULLETIN, platform::handle_bulletin);
108     }
109
110     // prehandler to intercept bulletins
111     void platform::handle_bulletin(event &bulletin_ev)
112     {
113         Bcasts_rcvd += 1;
114     }
115
116     // main actions of platform entity
117     void platform::body()
118     {
119         while (time() < DURATION) {
120             // hold until next broadcast or direction change
121             hold(Event_rate_gen.sample());
122
123             if (Bcast_prob_gen.sample() == TRUE) {
124                 // broadcast a bulletin
125                 broadcast(RANGE, event(NO_ENTITY_ID, 0.0, BULLETIN));
126                 Bcasts_sent += 1;
127             }
128             else {
129                 // change direction
130                 set_velocity(velocity(Direction_gen.sample(), SPEED));
131             }
132         }
133
134         generate_report();
135         terminate();
136     }
137
138     // send statistics report to summary entity
139     void platform::generate_report()
140     {
141         event report_ev;
142
143         report_ev = event(entity_id("summary1"), 0.0, REPORT);
144         report_ev << Bcasts_sent << Bcasts_rcvd;
145         schedule(report_ev);
146     }
```

```
147     // default settings for command-line arguments
148     sim_time    DURATION        = 20.0;
149     int         PLATFORMS       = 500;
150     double      RANGE           = 5.0;
151     double      SPEED           = 1.0;
152     double      EVENT_RATE      = 1.0;
153     double      BCAST_PROB      = 0.5;
154     int         SEED            = 10079;
155
156     // the initial function; this function is run from the command-line
157     void initialize(int argc, char *argv[])
158     {
159         int i;
160         event initial_ev;
161
162         // parse the command-line arguments
163         if (argc > 1) DURATION    = atof(argv[1]);
164         if (argc > 2) PLATFORMS   = atoi(argv[2]);
165         if (argc > 3) RANGE       = atof(argv[3]);
166         if (argc > 4) SPEED       = atof(argv[4]);
167         if (argc > 5) EVENT_RATE  = atof(argv[5]);
168         if (argc > 6) BCAST_PROB  = atof(argv[6]);
169         if (argc > 7) SEED        = atoi(argv[7]);
170
171         ASSERT(DURATION > 0.0);
172         ASSERT(PLATFORMS > 0);
173         ASSERT(RANGE > 0.0);
174         ASSERT(SPEED >= 0.0);
175         ASSERT(EVENT_RATE > 0.0);
176         ASSERT(BCAST_PROB >= 0.0);
177         ASSERT(SEED > 0);
178
179         sim_trace(1, "duration = %f",       DURATION);
180         sim_trace(1, "platforms = %d",      PLATFORMS);
181         sim_trace(1, "range = %f",          RANGE);
182         sim_trace(1, "speed = %f",          SPEED);
183         sim_trace(1, "event rate = %f",     EVENT_RATE);
184         sim_trace(1, "bcast prob = %f",     BCAST_PROB);
185         sim_trace(1, "seed = %d",           SEED);
186
187         // create all platforms
188         for (i = 0; i < PLATFORMS; i += 1) {
189             initial_ev = event(NO_ENTITY_ID, 0.0, INITIAL);
190
191             initial_ev << point(
192                 sim_uniform(0.0, MAX_X, SEED),
193                 sim_uniform(0.0, MAX_Y, SEED));
194             initial_ev << sim_negexp_obj(
195                 "event rate", EVENT_RATE, sim_randint(1, 10000, SEED));
196             initial_ev << sim_draw_obj(
197                 "bcast prob", BCAST_PROB, sim_randint(1, 10000, SEED));
198             initial_ev << sim_uniform_obj(
199                 "direction", 0.0, 360.0, sim_randint(1, 10000, SEED));
200
201             create("platform", initial_ev);
202         }
203
```

```
204             create("summary", NO_EVENT);
205         }
206
207         INITIAL_FUNCTION(initialize);
```