Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

LANGUAGE SUPPORT FOR PARALLEL DISCRETE-EVENT SIMULATIONS

Rajive L. Bagrodia

Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024
rajive@cs.ucla.edu

ABSTRACT

A number of algorithms have been developed to support parallel execution of discrete-event simulation models. In general, these algorithms are complex and implementing them directly in a simulation model is a difficult and resource-intensive programming task. Parallel simulation languages and environments can be of considerable help in hiding the complexity of the underlying synchronization algorithm and providing a simpler virtual machine to the model designer. This tutorial is a survey of currently available software tools that facilitate the design of parallel discrete-event simulations.

1 INTRODUCTION

Parallel (or distributed) discrete-event simulation refers to the execution of a discrete-event simulation program on a parallel (or distributed) architecture. Interest in exploiting parallelism in the execution of discrete-event simulation models has increased in a number of application areas ranging from network simulations, personal communication systems, digital circuits, and parallel architectures. This has partly been fueled by the increasing availability of low cost parallel architectures like the multiprocessor, shared-memory, desktop workstations and low-end distributed- and shared-memory multicomputers. However, parallel simulations are still being utilized primarily in the research community, with only a limited penetration in the commercial modeling and sequential simulation community [Fujimoto, 1993]. A major impediment to the widespread use of parallel simulation is the complexity of implementing efficient parallel simulations, coupled with the paucity of tools that allow analysts who are unfamiliar with the intricacies of parallel simulation algorithms to explore its potential.

A number of algorithms have been defined for the parallel execution of simulation models [Fujimoto, 1990]. The next section gives a quick overview of the two primary classes: conservative and optimistic. A simulation tool may either be tailored to a specific (class of) synchronization protocol, or be capable of executing models with diverse protocols. The former category includes tools like TWOS [Jefferson, et al., 1987] that are designed to only use the Time Warp synchronization protocol, and SPEEDES [Steinman, 1991] that is designed to work with a set of optimistic protocols; the advantage of committing to a specific (class of) protocol is that typically the data structures in the underlying simulator can be tailored to exploit the specific features of the protocol. The latter category includes multi-algorithm environments like Maisie [Bagrodia and Liao, 1994], SPECTRUM [Reynolds, 1989] and YADDES [Preiss, 1989] that support both conservative and optimistic protocols. As the viability of specific protocols to support efficient parallel execution of models in different application domains is still not well understood, it is perhaps desirable to develop tools that give analysts some flexibility in the choice of synchronization protocols.

Parallel simulation tools may also be classified based on the generality of their programming interface. Available tools range from operating systems dedicated to support of parallel simulations to domain-specific tools. An operating system typically provides a kernel that exports the needed functionality via a set of function calls. Section 3 describes TWOS [Jefferson, et al., 1987] as a typical example. At the next level of abstraction are parallel simulation languages (PSLs) and libraries that provide programmers with a set of well-defined constructs to design (parallel) models. These tools are described in section 4. At the end of the spectrum are domain-specific tools that are designed to serve a specific application area like logic simulation of VLSI circuits. Such software obviously has limited applicability in areas out-

side the specific application domain that it targets. Domain-specific tools are discussed in section 5. We conclude the paper with a brief discussion of topics for future research in section 6.

2 Parallel Simulation Protocols

In a parallel discrete-event simulation, the system being simulated is modeled by a number of logical processes (LP) that interact by exchanging timestamped event messages. Each LP must eventually process incoming messages in their global timestamp order. Enforcing this requirement, referred to as the causality constraint, is the central problem in efficient execution of parallel simulations. Two primary approaches have been suggested to solve the synchronization problem: conservative and optimistic.

Conservative algorithms [Misra, 1986] do not permit any causality error: each object in the simulation processes an incoming message only when the underlying synchronization algorithm can guarantee that it will not subsequently receive a message with a smaller timestamp. This constraint may introduce deadlocks, which are typically avoided by using *null messages*. A null message is a timestamped signal sent by an LP to indicate to other LPs a lower bound on the timestamp of its future messages. If an LP sends a null message with timestamp T at simulation time t, $T >= t$, we say that the LP has a lookahead of (T-t). In general, the larger the lookahead of an LP, the better its performance with conservative protocols. Efficient implementation of null messages is also facilitated if each LP maintains the set of its source and/or destination LPs. (Otherwise, *null* messages may be needlessly broadcast to all LPs). Conservative implementations are generally simple to develop and basically need support to implement *null message transmission* and *lookahead specification*.

In optimistic protocols [Jefferson 1985], an LP is allowed to process events in any order; however, the underlying synchronization protocol must detect and correct violations of the causality constraint. The simplest mechanism for this is to have each LP periodically save (or checkpoint) its state. Subsequently, if it is discovered that the LP processed messages in an incorrect order, it can be rolled back to an appropriate checkpointed state, following which the events are processed in their correct order. The rollback may also require that the LP unsend or cancel the messages that it had itself sent to other LPs in the system. An optimistic algorithm is also required to periodically compute a lower bound on the timestamp of the earliest global event, also called the Global Virtual Time or GVT. As the model is guaranteed to not contain any events with a timestamp smaller than GVT, all checkpoints timestamped earlier than GVT can be reclaimed. Thus the primary facilities needed to implement optimistic methods include *checkpointing*, *message cancellation*, *rollback and recomputation*, and *GVT computation*.

Recently, a new protocol has been suggested that allows each LP to individually select either the conservative or the optimistic execution mode [Jha and Bagrodia, 1994]. This protocol defines a local metric called EIT (for Earliest Input Time) for each LP. Conservative LPs can process all events that are timestamped earlier than its EIT, whereas optimistic LPs can use EIT to reclaim memory. A separate global control mechanism is defined to allow each LP to periodically update its EIT. The global control mechanisms could use algorithms similar to the ones used to compute GVT, or be based on null messages, or even use a combination of the two techniques.

3 Operating Systems

The Time Warp Operating System or TWOS developed at JPL [Jefferson, et al., 1987] is among the earliest operating systems for parallel simulation. A more recent example is the MIMDIX system [Madisetti, et al., 1992] that has been implemented on a BBN Butterfly and the GTW [Das, et al., 1994] implemented on a the Kendall Square Research KSR-1. TWOS and GTW both implement the Time Warp synchronization protocol, whereas MIMDIX implements an optimistic protocol based on the notion of probabilistic synchronization.

The TWOS system is a complete implementation of the Time Warp optimistic synchronization protocol. The programming interface supported by TWOS is a standard process-based model, where processes communicate using timestamped messages. Each process is required to be deterministic and the process is not allowed to use heap storage. Because interactions of an optimistic system with the outside world cannot be rolled back, input output is handled by special TWOS processes that delay outputs until the computation has converged to a time greater than the timestamp of the output.

TWOS messages are classified into *event* and *query* messages. A message is designated to be a query only if the code executed on its arrival at a process does not modify the local state of the process, and the only externally visible effects are the transmission of other query messages and/or a single reply message to the sender of the query. A message that is not a query message is an event message.

The body of a TWOS process consists of two pri-

mary sections that are distinguished syntactically: the event message section which contains code to be executed on the arrival of an event message and the query-message section for processing query messages.

The programming interfaces provided by TWOS were designed to enhance the transparency of the underlying synchronization protocol and provide only a limited interface to the simulator. The interface includes calls to send and receive messages and to read the current value of simulation time. In particular, no interface is provided to the programmer to modulate the behavior of the simulator by modifying various parameters like the frequency of checkpointing, GVT computation, or message cancellation strategy.

The message passing routines provided by TWOS are relatively 'bare bones' as compared with the interface supported by most languages. Routines are provided to send and receive event and query messages at specific future (simulation) time. Thus *SendEventMessage(ReceiveTime, Receiver, Text)* will cause a message containing *text* to be delivered to process *Receiver* when its simulation time reaches *ReceiveTime*. In general, TWOS requires that the *ReceiveTime* of a message be strictly greater than the simulation time at which it is sent. Routine *ReadEventMessage(k,msg)* is used to remove the k^{th} message with timestamp equal to the current simulation time from the event message queue and return it in variable *msg*. Similar routines are provided for query messages. The simulation time of a process is set transparently by the simulator to the timestamp on the earliest message in its event queue; the process can read this value by calling the system routine $VirtualTime(VTime)$ which sets the value of parameter $VTime$ to the local simulation time of the LP.

The TWOS system has been used to execute a number of applications, including a war game simulation programmed using 130 processes that was found to yield a speedup of almost 10 on 32 nodes of a Caltech Mark III Hypercube. It was subsequently used in conjunction with the Modsim language to execute other models; the MODSIM/TWOS environment is described in the next section.

4 Languages and Libraries

The primary approaches used to design general purpose parallel simulation software include the following:

- Library based approaches represented by systems like YADDES [Preiss, 1989], OLPS [Abrams, 1988], and SPEEDES [Steinman, 1991]. These systems typically provide simulation and parallelism capabilities via calls to libraries implemented in standard sequential languages like C or C++. Their primary advantage is that the user does not have to learn a new language. A major drawback is that because no translator is used, the library routines must provide less functionality than is possible in a language, or they tend to become ungainly.

- Enhance sequential (simulation) languages with primitives for parallel simulation; examples include MAY [Bagrodia, et al., 1987], Sim++ [Baezner, et al., 1990] and Maisie [Bagrodia and Liao, 1990]. The ability to use a translator allows languages to provide a more succinct and 'natural' interface for the programmer as compared with simulation libraries. Both Maisie and Sim++ were influenced by the May simulation language, particularly in their choice of process scheduling, communication and synchronization constructs. Whereas Sim++ is used to execute models synchronized with the Time Warp protocol, Maisie was designed to provide efficient support for both conservative and optimistic protocols.

- Add simulation capability to parallel languages: examples include ModsimII [Bryan, 1989] that is derived from Modula-2 and SCE [Gill, et al., 1989], which is based on Ada.

- High-level notations based on declarative, equational, or logic programming environments: these include the environment based on UNITY and a notation based on Temporal logic. While such environments offer the potential for a high-level specification that can subsequently be executed with little or no modification using a variety of synchronization protocols, there is little experimental evidence as yet to support the viability of this approach for programming (in contrast to specifying) parallel simulations.

In the remainder of this section, we describe three systems, one from each of the first three categories: SPEEDES, Maisie, and ModsimII.

4.1 SPEEDES

SPEEDES (Synchronous Parallel Environment for Emulation and Discrete Event Simulation) is a C++ based simulation environment that implements a set of optimistic synchronization protocols including Time Warp and a windowing algorithm called Breathing Time Buckets. In this paper, we restrict our

attention to the programming interface provided by SPEEDES, a simulation environment built on C++.

A SPEEDES simulation consists of simulation objects and events. An event is created as a C++ object (the authors use the term 'a fully encapsulated object'), when a message is sent to a simulation object. A distinguishing feature of SPEEDES is that unlike most optimistic systems, it requires the user to explicitly checkpoint the state of a simulation object. Rather than checkpoint the entire state, SPEEDES uses incremental state saving where only those variables that are modified by an event are saved using a technique that the authors refer to as *Delta Exchange*.

An event is initialized by copying the data from the corresponding message that was used to create the event. It is then processed as follows: the event executes the corresponding method (presumably determined by the message) but does not modify the state of the object; rather the specific variables modified by the event are stored within the event itself. Future messages generated by the object are also stored in the event rather than forwarded to other objects. In the next step, the values computed in the previous phase are exchanged with the simulation object, such that the event now has the old values and the object contains the new values. (Note that an additional exchange will restore the state of the object.) In the final phase, the messages stored in the event are either canceled, if the event itself is canceled, or forwarded to other objects and the event itself is garbage collected.

Incremental state saving is also supported by another mechanism referred to as the *rollback queue* which may be used by a simulation object to directly save the changes in its state; however this technique is also claimed to be significantly more expensive for incremental state saving than the delta exchange mentioned previously. Support for lazy cancellation is also provided, but once again it relies on the user to provide the appropriate functions that can be used to compare the incrementally saved states. Although incremental state saving is likely to yield solid performance benefits for many applications, reliance on the programmer to provide the code for the delta exchange in every event adds significant complexity to program design and maintenance.

The performance of the SPEEDES environment has been evaluated using a proximity detection problem, where it was found to yield a speedup of almost 20 on 32 nodes of a Caltech Mark III Hypercube using the SPEEDES implementation of the Time Warp synchronization protocol, where the speedup was measured against an optimized one node implementation of SPEEDES. Good speedup measurements have also been reported for a variety of queueing networks using their windowing protocol.

4.2 Maisie

Maisie [Bagrodia and Liao, 1994] is a C-based discrete-event simulation language. With few modifications, a Maisie program may be executed using a variety of simulation protocols that include a sequential algorithm, parallel conservative algorithms based on null messages [Misra, 1986] and conditional events [Chandy and Sherman, 1989a], a new conservative protocol that combines null messages with conditional events [Jha and Bagrodia, 1993], and a parallel optimistic algorithm [Chandy and Sherman, 1989b; Bagrodia, et al., 1991]. Maisie is also the first language to support semantic optimizations — the use of application semantics to reduce the overhead of both conservative and optimistic parallel simulation.

A Maisie program is a collection of entity definitions and C functions. An entity definition (or an entity type) describes a class of objects. An entity instance, henceforth referred to simply as an entity, represents a specific object in the physical system and may be created and destroyed dynamically. An entity is created by the execution of a **new** statement and is automatically assigned a unique identifier on creation. For instance, the following statement creates a new instance of a manager entity and stores its identifier in variable $r1$.

$r1$ = **new** *manager*{10};

Entities communicate with each other using buffered message-passing. Maisie defines a type called **message**, which is used to define the types of messages that may be received by an entity. Definition of a message-type is similar to a struct; the following declares a message-type called *req* with one parameter (or field) called *count*.

message *req* {int *count*; };

Every entity is associated with a unique message-buffer. A message is deposited in the message buffer of an entity by executing an **invoke** statement. The following statement will deposit a message of type *req* with timestamp *clock*()+t, where *clock* is the current value of the simulation clock, in the message buffer of entity $m1$.

invoke $m1$ **with** *req*(2) [**after** t]

If the after clause is omitted, the message is timestamped with the current simulation time. An entity accepts messages from its message-buffer by executing a **wait** statement. The wait statement has two components: an optional wait-time (t_c) and a required resume-block. If t_c is omitted, it is set to an arbitrarily large value. The resume-block is a set of

resume statements, each of which has the following form:

mtype(m_i) [**st** b_i] $statement_i$;

where m_i is a message-type, b_i an optional boolean expression referred to as a *guard*, and $statement_i$ is any C or Maisie statement. The guard is a side-effect free boolean expression that may reference local variables or message parameters. If omitted, the guard is assumed to be the constant *true*. The message-type and guard are together referred to as a *resume condition*. A resume condition with message-type m_i and guard b_i is said to be *enabled* if the message buffer contains a message of type m_i, which if delivered to the entity would cause b_i to evaluate to *true*; the corresponding message is called an *enabling message*.

With the wait-time omitted, the wait statement is essentially a selective receive command that allows an entity to accept a particular message only when it is ready to process the message. For instance, the following wait statement consists of two resume statements. The resume condition in the first statement ensures that a *req* message is accepted only if the requested number of units are currently available (the requests are serviced in first-fit manner). The second resume statement accepts a *free* message:

```
wait until
{ mtype(req) st (units >= msg.req.count)
    /* signal requester that request is granted */
 or mtype(free) /* return units to the pool */
}
```

Maisie also provides a number of pre-defined functions that may be used by an entity to *inspect* its message buffer. For instance, the function **qsize**(m_t) returns the number of messages of type m_t in the buffer. A special form of this function called **qempty**(m_t) is defined, which returns *true* if the buffer does not contain any messages of type m_t, and returns *false* otherwise. In general the resume condition in a wait statement may include multiple message-types, each with its own boolean expression.

If two or more resume conditions in a wait statement are enabled, the timestamps on the corresponding enabling messages are compared and the message with the earliest timestamp is removed and delivered to the entity. If no resume condition is enabled, a timeout message is scheduled for the entity t_e time units in the future. The timeout message is canceled if the entity receives an enabling message *prior* to expiration of t_e; otherwise, the timeout message is sent to the entity on expiration of interval t_e. Thus the wait statement can be used to schedule conditional events. A **hold** statement is provided to unconditionally delay an entity for a specified simulation time.

The Maisie simulation environment has been implemented on a network of workstations and on distributed memory multicomputers like the IBM SP1. Maisie implementations have yielded good speedups for queuing network benchmarks with both conservative and optimistic algorithms, with speedups approaching close to linear for large grain computations [Jha and Bagrodia, 1993; Bagrodia, et al., 1991]. As reported in a companion paper in this volume, the IBM SP1 implementations have been used for gate level circuit simulations of circuits from the ISCAS85 benchmarks. For the largest circuits in the benchmark (which contained less than 3000 gates), the simulations yielded a speedup of almost 4 on 8 nodes of an IBM SP1 with both conservative and optimistic algorithms. Experiments with larger circuits are in progress.

Recently, an object-oriented extension of Maisie has been designed. This extension, called MOOSE for *M*aisie-based *O*bject-*O*riented *S*imulation *E*nvironment [Waldorf and Bagrodia, 1994] uses inheritance to derive parallel implementations of an object that may exploit specific knowledge about the application, architecture, or simulation algorithm to improve its efficiency.

4.3 ModsimII

ModsimII[Bryan, 1989] is an object-oriented simulation language based on Modula-2. Like Modula-2, it was designed to support programming-in-the-large and uses block structure, strong typing and code encapsulation. The primary components of a ModsimII program are *objects*. An object contains state variables (called *fields*) and functions (called *methods*). The fields in an object may only be modified by its local methods (in object-oriented terminology, all fields are considered to be *private*). Each method has two parts: *definition* part which is the public interface to the method and defines the name and parameter list, and the *implementation* part which contains the function body and is accessible only to derived objects. ModsimII supports multiple inheritance.

ModsimII provides construct for dynamic object creation, termination, communication and synchronization. The statement *NEWOBJ*(*obj_name*) is used to create a new instance of object *obj_name*. However ModsimII does not provide explicit facilities to group objects together to either share a common OS thread as in Sim++ or a common processor as in Maisie. The methods of the new object may be executed using either the *ask* or *tell* statements. An *ask* statement has remote procedure call semantics similar to the standard method invocation of object-

oriented languages; this statement does not involve passage of simulation time. In contrast, a *tell* statement is used to schedule a method invocation at the current or future simulation time. Like the Maisie invoke statement, tell statements are non-blocking. For instance, the statement

TELL obj TO func(actuals) IN time

schedules the execution of method *func* at the object *obj* at simulation time $CLOCK + time$, where $CLOCK$ is the current simulation time. The *IN time* component of a *TELL* statement may be omitted, in which case, the method is scheduled for execution at the current simulation time. Note that unlike Maisie, a ModsimII object cannot use a guard to temporarily disable the execution of a given method or to select internally among multiple methods that may be scheduled for execution at a given simulation time. All scheduling decisions are made by the ModsimII scheduler and are transparent to the programmer. A *WAIT* statement is also provided to allow an object to suspend itself in simulation time or until a specified method in another object has been executed.

ModsimII uses the TWOS operating system to synchronize the execution of parallel ModsimII programs. Although the simulator has been used to obtain speedups with large-grained computations with an object count in the thousands, it was eventually unable to execute a division level combat model. As reported in Rich and Michelsen [1991], the primary problem was the inability to explicitly map multiple ModsimII objects into a single TWOS process. The default was to treat each (possibly fine-grained) ModsimII object as a TWOS process which caused large checkpointing and context-switching overheads to be incurred needlessly. The absence of Sny hooks from ModsimII into the underlying simulator was also found to be a major problem, as it made it impossible to control or alter the scheduling or thread management strategies used by TWOS.

5 Domain-specific Simulators

Parallel simulators that are targeted to specific application domains are also becoming popular. Two areas that show particular promise and recent research activity are digital circuit simulations and parallel architecture simulations.

A number of studies have investigated the viability of asynchronous parallel circuit simulations [Su and Seitz, 1989; Briner, et al., 1991; Soule and Gupta, 1992]. The most successful applications have used optimistic protocols with reported speedups measured on 32 nodes of a BBN Butterfly ranging from 7 for a small adder to almost 25 for a 31000 gate circuit [Briner, et al., 1991]. MIRSIM [Chen and Bagrodia, 1994] is a recent effort in the design of parallel circuit simulators for use by circuit designers. MIRSIM is a parallel Maisie implementation of IRSIM, an event-driven logic level simulation that incorporates a simple *linear model* of MOS transistors to compute transition delay of logic state. MIRSIM has been implemented with both conservative and optimistic synchronization algorithms.

Two recent efforts in the area of parallel architectural simulators are the Wisconsin Wind Tunnel (WWT) [Reinhardt, et al., 1993] and the LAPSE [Dickens, et al., 1994] simulators. WWT was designed to specifically evaluate cache-coherent shared-memory systems on the Thinking Machine CM5. It uses direct execution for all load and stores that hit in the local cache; cache misses are trapped by using the ECC bits in the CM5 causing an interrupt handler to be invoked which initiates the simulation of the corresponding cache miss event in logical time. The execution on multiple processors is synchronized by using a semi-synchronous conservative algorithm with a fixed window size.

LAPSE is a parallel simulator that uses a message-passing multicomputer to simulate the performance of parallel programs running on larger configurations of message-passing multicomputers. LAPSE also uses direct execution to 'simulate' sequential code (code between communication points) and traps calls to the message communication library. However, unlike WWT, the traps are generated at compile time by redirecting calls to the message-passing library in the application code as calls to appropriate LAPSE routines that simulate the corresponding event. It uses a conservative appointment-based protocol to synchronize the execution of the simulator on multiple processors. The simulator was found to yield a self-relative speedup (i.e., speedup is measured with respect to an implementation of the parallel simulator on 1 node as opposed to measuring speedups against a sequential implementation) of up to 47 for the simulation of a variety of numerical applications.

6 Research Issues

The area of parallel simulation languages and environments is clearly in its infancy. Much remains to be done in the design of general purpose parallel tools, in their application to specific domains, and in the identification of application characteristics that could be exploited by specific synchronization protocols. We conclude this paper with a brief look at a few areas that offer opportunities for research and development.

Shared variable languages The inclusion of shared variables in PSLs is of significant potential benefit. First, many applications are not 'naturally' decomposed into a set of message communicating LPs that do not share any memory. Second, from an efficiency viewpoint, the absence of shared data might imply duplication; for instance initial system configurations that are most conveniently stored as global write-once data, often need to be duplicated in the absence of shared memory support in the PSL. The primary problem in supporting shared memory in a PSL is that efficient implementations of shared memory on distributed memory machines (called virtual shared memory or distributed shared memory) use memory consistency models that are considerably weaker than the sequential consistency typically assumed by a PSL. Efficient mappings from the sequential consistency to weaker consistency models is an active research area in the parallel computing community, but few transparent solutions are known. The coherence problem is further compounded by the need to maintain timestamp ordering for all references to shared memory, particularly when the set of LPs that can potentially access specific variables is not known. Recent work by Mehl and Hammes [1993] provides some insight into the techniques to combine distributed shared memory with parallel simulation. Much still remains to be done towards efficient integration of shared memory programming primitives in PSLs.

Semantic Optimizations While keeping the simulator transparent to the programmer has the advantage of presenting a simpler programming interface to the simulationist, it makes it harder for the programmer to exploit application-specific information to reduce execution overheads. The Maisie environment provides language level constructs to allow programmers to interact with the underlying implementation. For instance, for conservative algorithms, it is possible to specify the dynamic lookahead of each LP and to maintain connectivity information. For optimistic simulators, constructs may be provided to reduce the rollback distance for straggler messages [Bagrodia and Liao, 1994]. Research in semantic optimizations is still in its early stages and much remains to be done in terms of providing well-defined interfaces to allow programmers to interact with the underlying simulator.

Application partitioning and load management The performance of any parallel program, including parallel simulations, is particularly sensitive to the decomposition strategy used to distribute the work on the multiple processors. However, available parallel simulators provide minimal support for even static load balancing and most do not support dynamic load balancing or process migration. Investigation of existing algorithms, development of new techniques designed for specific parallel simulators and applications, and their incorporation into parallel simulators, all present numerous opportunities for further research.

Program Development Environments The model development environments that are available with most PSLs are primitive when compared against the development environments and user interfaces that are available for sequential simulators. Considerable work remains to be done towards providing suitable graphical user interfaces (GUIs) for model definition and optimization for execution with specific parallel simulators. Domain-specific parallel simulators can derive particular benefit from such tools.

ACKNOWLEDGEMENTS

This research was supported by NSF PYI Award ASC-9157610 and by the US Dept of Justice/FBI, ARPA/CSTO under contract J-FBI-93-112.

REFERENCES

Abrams, M. 1988. The object library for parallel simulation (OLPS). In *Proceedings of the 1988 Winter Simulation Conference*, 210-219, December.

Baezner, Dirk, Greg Lomow, and Brian W. Unger. 1990. Sim++: The transition to distributed simulation. In *Proc 1990 SCS Multiconference on Distributed Simulation*, 211-218, San Diego, California, January.

Bagrodia, R. L., K. M. Chandy, and W-T. Liao. 1991. A unifying framework for distributed simulations. *ACM Transactions on Modeling and Computer Simulation*, October.

Bagrodia, R. L., K. M. Chandy, and J. Misra. 1987. A message-based approach to discrete-event simulation. *IEEE Transaction on Software Engineering*, Vol. 13, No. 6, 205-210, June.

Bagrodia, R. and W-T. Liao. 1990. Maisie: A language and optimizing environment for distributed simulation In *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, San Diego, California, January.

Bagrodia, R. and W-T. Liao. 1994. Maisie: A language for design of efficient discret-event simulations. *IEEE Transactions on Software Engineering*, April.

Briner, J., J. Ellis, and G. Kedem. 1991. Breaking the barrier of parallel simulation of digital systems. In *Proc ACM/IEEE Design Automation Conf.*

Bryan, Otis. 1989. MODSIM II - an object oriented simulation language for sequential and parallel processors. In *Proc of the 1989 Winter Simulation Conference*, 172–177, Washington, D.C., Dec.

Chandy, K. M. and R. Sherman. 1989a. The conditional event approach to distributed simulation. In *Proceedings of the SCS Simulation Multiconference on Distributed Simulation*, 93-99, March.

Chandy, K. M. and R. Sherman. 1989b. Space-time and simulation. In *Distributed Simulation Conference*, Miami.

Chen, Yu-an and R. Bagrodia. 1994. Parallel switch-level circuit simulation. Technical report, Computer Science Dept, UCLA.

Das, Samir, Richard Fujimoto, Kiran Panesar, Don Allision, and Ingrid Hybinette. 1994. GTW: A time warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference*, Washington D.C., December.

Dickens, P., P. Heidelberger, and D. Nicol. 1994. A distributed memory lapse: Parallel simulation of message-passing programs. In *Workshop on Parallel and Distributed Simulation*, 32-38, July.

Fujimoto, R. 1990. Parallel discrete event simulation. *CACM*, Vol. 33, No. 10, 30-53, October.

Fujimoto, R. 1993. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3):213–230.

Gill, D. H., F. X. Maginnis, S. R. Rainier, and T. P. Reagan. 1989. An interface for programming parallel simulations. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, 151-154, Tampa, Florida, March.

Jefferson, D., B. Beckman, and F. Wieland, et al. 1987. Distributed simulation and the time warp operating system. In *Symposium on Operating Systems Principles*, Austin, Texas, October.

Jefferson, D. 1985. Virtual Time, *ACM TOPLAS*, 7(3):404-425.

Jha, Vikas and Rajive Bagrodia. 1993. Parallel implementations of Maisie using conservative algorithms. In *Winter Simulation Conference*, Dec.

Jha, Vikas and Rajive Bagrodia. 1994. A unified framework for conservative and optimistic distributed simulation. In *1994 Workshop on Parallel and Distributed Simulation*, Edinburgh, July.

Madisetti, Vijay K., David A. Hardaker, and Richard M. Fujimoto. 1992. The mimdix operating system for parallel simulation. In *6th Workshop on Parallel and Distributed Simulation (PADS92), 1992 SCS Western Simulation MultiConference on Parallel and Distributed Simulation*, 65-74, Newport Beach, CA, January.

Mehl, Horst and Stefan Hammes. 1993. Shared variables in distributed simulation. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, 68-76, San Diego, California, May.

Misra, J. 1986. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39-65.

Preiss, B. R. 1989. The Yaddes distributed discrete event simulation specification language and execution environments. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, March.

Reinhardt, S. K., M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. 1993. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, May.

Reynolds, Paul F. 1989. Comparative analysis of parallel simulation protocols. In *Proceedings of the 1989 Winter Simulation Conference*, 671–679, Washington, D.C., December.

Rich, David O. and Randy E. Michelsen. 1991. An assessment of the modsim/twos parellel simulation environment. In *Proceedings of the 1991 Winter Simulation Conference*, 509-518.

Soule, Larry and Anoop Gupta. 1992. An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation. In *6th Workshop on Parallel and Distributed Simulation (PADS92)*, 129–138, Newport Beach, CA, January.

Steinman, Jeff. 1991. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel and Distributed Simulation, SCS Multiconference*, 95-103, Anaheim, CA, January.

Su, Wen-king and C.L. Seitz. 1989. Variants of the Chandy-Misra-Bryant distributed simulation algorithm. In *1989 Simulation Multiconference: Distributed Simulation*, Miami, Florida, March.

Waldorf, J. and R. Bagrodia. 1994. MOOSE: A concurrent object oriented language for simulation *International Journal of Computer Simulation*, To appear.

RAJIVE L. BAGRODIA received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Bombay in 1981 and the M.A. and Ph.D. degrees in Computer Science from the University of Texas at Austin in 1983 and 1987, respectively. He is currently an Associate Professor in the Computer Science Dept. at UCLA. His research interests include parallel languages, parallel simulation, distributed algorithms, and software design methodologies. He was selected as a 1991 Presidential Young Investigator by NSF.