Design Issues in General Purpose, Parallel Simulation Languages

Marc Abrams and Greg Lomow

TR 89-38

Technical Report TR 89-38
November 1989

Design Issues in General Purpose, Parallel Simulation Languages

Marc Abrams
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106

Greg Lomow
Jade Simulations International, Corp.
#80, 1833 Crowchild Trail NW
Calgary, Alberta
Canada T2M 4S7

ABSTRACT

This paper addresses the topic of designing general purpose, parallel simulation languages. We discuss how parallel simulation languages differ from general purpose programming languages. Our thesis is that the issues of distribution, performance, unusual implementation mechanism, and the desire for determinsm are the dominant considerations in designing a simulation language today. We then discuss the separate roles that special and general purpose simulation languages play. Next we use two languages, Sim++ and CPS, to illustrate these issues. Then we discuss eight design considerations: process versus event oriented-view, basic program structure, I/O, making implementation costs explicit to the programmer, providing dynamic facilities, memory management, the semantics of false messages in time warp, and program development methodology considerations. A number of conclusions are drawn from our experiences in language design.

This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211 while the first author was affiliated with Stanford University.

Table of Contents

Design Issues in General Purpose, Parallel Simulation Languages

1. Introduction

Most research in parallel, discrete-event simulation has focused on developing synchronization mechanisms, or *simulation protocols*, that minimize the running time of a simulation program. We consider a complementary direction to this research, addressing the problem of designing a simulation programming language that incorporates these parallel simulation protocols.

The precise definition of "simulation programming language" is explored in the following section. In this paper we discuss a number of design problems that arise, and discuss the solutions chosen in two languages that the authors have worked on, Sim++ [LOMO89] and CPS [ABRA89].

Other parallel simulation language efforts include Yaddes [PREI89], Rockwell's time-warp implementation [TINK89], and the JPL time-warp Operating System [JEFF87].

2. Simulation Versus General Purpose Programming Languages

Simulation languages facilitate writing computer programs that simulate a model of a system. The major difference between discrete-event simulation languages and most other programming languages is the introduction of processes, events, and simulation time. Processes are separate threads of execution in the simulation that are used to model the distinct physical processes in the system being simulated. Simulation time is an arbitrary, application-defined time scale that is independent of real time. Events represent points in simulation time when the model undergoes state changes. Processes schedule events for each other to mark when these state changes are to occur, to synchronize their actions, and to communicate. The actions of all processes and the scheduling of all events is tied to simulation time, and processes receive events scheduled for them in order of increasing simulation time.

Parallel simulation languages are simulation languages designed to execute in parallel on multiprocessors. Designing a parallel simulation language involves balancing

the goals of elegance and performance while accommodating the distributed nature of parallel simulations and the underlying synchronization mechanism. Elegance is important in the design of any programming language. It involves evaluating language features for orthogonality, simplicity, and ensuring that they present a coherent abstraction that is appropriate to the application domain. Since performance is important in parallel simulation, the design of parallel simulation languages must ensure that language features do not compromise overall system performance (typically measured in terms of execution time). Since parallel simulations are intended to run on multiprocessor architectures, the design of parallel simulation languages must ensure that language features scale easily on a multiprocessor and do not rely on shared memory or centralized synchronization algorithms. Finally, it is important that language features are compatible with and do not violate the assumptions made by the underlying synchronization mechanism. This last goal is at odds with most principles of language design. However, it is a consideration that has to be taken into account because research into parallel simulation has been driven by research into the synchronization mechanism. Throughout this paper, language features are examined and evaluated with these goals in mind.

Some aspects of simulation programs make them amenable to execution on multiprocessors. Simulations are naturally suited to running on multiprocessors since processes represent logically distinct threads of execution that can be run on separate processors. Furthermore, since events are the only mechanism for synchronizing processes they provide a simple, well-defined interface that is easily adapted to different multiprocessor architectures.

Other aspects of simulation languages impose stringent constraints on the design of parallel simulation languages.

The requirement that each process receive events scheduled for it in order of increasing simulation time is difficult to satisfy because of the distributed and decentralized nature of multiprocessor systems. This has led to implementation of complex synchronization mechanisms which, unfortunately, influence the design of the corresponding parallel simulation languages.

Another problem is determinism. A program is deterministic if, given the same input, multiple runs of the program always produce identical results. A program is non-deterministic if multiple runs of the same program do not necessarily produce the same results even if they are started with the same input. Almost all distributed parallel programs

are non-deterministic. This is because their execution is sensitive to differing processor clock speeds and message latency.

Non-determinism introduced by the run-time system is intolerable in simulations. To understand this, remember that the most difficult phases of any simulation project are validating the model and verifying that the simulation program correctly implements the model. These phases often involve establishing confidence intervals and doing other forms of statistical analysis on the output of the simulation. Non-determinism complicates any statistical analysis by introducing an external source of randomness that cannot be characterized and quantified. This is why all sequential simulation languages are deterministic. Maintaining determinism is the responsibility of the synchronization mechanism. To complicate matters further, determinism must be maintained no matter how many processors are used to execute the simulation, no matter how the entities are mapped to these processors, and no matter what the architecture of the underlying hardware.

Another critical reason for guaranteeing determinism is that it simplifies debugging and testing. When debugging a non-deterministic distributed program it can be difficult to re-create an error because separate runs can legitimately follow different paths of execution. Determinism ensures that a program follows the same path of execution each time it is run, allowing errors to be easily re-created.

3. General Versus Special Purpose Simulation Languages

We distinguish between *general purpose* and *special purpose* simulation languages. A general purpose language is designed to be capable of expressing any system which one wishes to simulate. In contrast, a special purpose language is tailored to a particular class of systems, for example logic circuits.

Special purpose languages take advantage of information about the class of systems being simulated to optimize performance. In addition, special purpose languages can be easier to use because they can present an abstraction that uses concepts specific to the class of systems being simulated.

In this paper we consider general purpose simulation languages. General purpose languages are appropriate for analysts who model a variety of systems in different domains. They are also appropriate when one wishes to vary the fundamental assumptions used in the model. This is because a special purpose simulation language may depend on a fixed set of assumptions in its design. For example, a logic simulator may be best

constructed using special purpose hardware. However, using special purpose hardware may make it impossible to vary an assumption, such as letting gate delays be random variables. Another class of simulations that call for general purpose languages are those that involve very complex processes or algorithms, in which the design of the algorithm is being studied. Examples include computer sytsem simulations and C^3I models.

4. Two Parallel Simulation Languages

In this section we present the key concepts of two independently developed simulation languages, Sim++ and CPS. Sim++ is designed for construction of production simulators, while CPS is designed as a research tool to experiment with parallel simulation protocols. We begin by describing concepts that are common to both. A number of design decisions summarized here are discussed at length in section 5.

A comment is in order on terminology. Misra [MISR86] describes a system to be simulated as consisting of a set of *physical processes* (PP's). The simulation of a system consists of a set of *logical processes* (LP's), one LP per PP, that pass messages. Sim++ and CPS both employ this view. CPS uses the term logical process, while Sim++ uses the term *entity* to mean an LP. We generally use the term "LP" in this paper, but sometimes use the term "entity" when discussing Sim++.

4.1 Concepts Common to Sim++ and CPS

Both languages provide the user with facilities for discrete-event simulation. Both are intended to be implemented by both sequential and parallel simulation protocols. A key criteria applied during the design of both languages was to examine how efficiently each language mechanism could be executed.

One important design issue in both languages was the choice of computation model. The model presented to the Sim++ and CPS programmer is one of loosely coupled processes that only communicate by passing messages and generally do not share read/write memory. (Sharing is only permitted in two forms. First, read-only variables may be shared. Second, Sim++ provides a language construct called *clusters* allowing limited sharing of data.) Note that the computation model provided to the user, processes that pass messages, is distinct from the language implementation, which may use shared memory to increase performance if the architecture provides it.

There are two reasons for choosing the loosely coupled computation model. First, it is consistent with the view of simulation as a set of LP's communicating through messages. This is the computation model used by many parallel simulation protocols, including the Bryant-Chandy-Misra (BCM) [BRYA77, CHAN79] and time-warp [JEFF85] protocols.

A second reason for choosing the loosely coupled model is the desire to be protable across a variety of machine architectures. Sim++ and CPS have been and are being implemented on a variety of computer architectures and operating systems. The parallel architectures on which we wished to implement the languages ranged from message-passing machines, such as the Intel Hypercube, to multi-processor machines with a single global memory with equal cost access and a hardware snoopy cache coherency mechanism, such as the Sequent Symmetry. Consequently the computation model provided to the programmer by both Sim++ and CPS is based on the lowest common denominator of these machine architectures -- loosely coupled processes that do not share memory.

Each thread of control in both languages corresponds to an LP. These threads are called *entities* in Sim++ and LP's in CPS.

Both languages advocate an object-oriented view of simulation programming. First, object-oriented programming is consistent with viewing a simulation as consisting of a set of LP's. Second, object-oriented programming is consistent with a computation model that provides no shared memory, when objects are mapped to processes.

Both languages requires the user to partition the simulation model into a set of LP's, each of which is implemented as an object. Furthermore, the efficiency of a parallel simulation will depend on how this partitioning is done. Both Sim++ and CPS treat each LP as a separate thread of control. Both languages provide the user with a view of cheap and plentiful threads -- the language implementations are designed to create and manage many more LP's than there are processors.

Both languages are transparently scalable. This term refers to the fact that the same Sim++ or CPS program can run on any number of processors without recompilation. In addition, there is no provision for specifying in the source code the number of processors or the identity of any processor. However, the mapping of LP's to processors may be optionally specified at execution time in both languages in a run-time configuration file.

Sim++ and CPS are, strictly speaking, not programming languages in their own right. Rather, they provide a library of objects implemented in C++ [STRO86], and impose on C++ a style of programming appropriate for loosely coupled processes. In both cases the language designers wanted to have the facilities of a general purpose language available, but did not want to undertake the task of designing an entirely new programming language. One inconvenience arising from this choice is that the simulation programmer must know the names of all objects in the library, because he cannot use these names for his objects. In Sim++ all of these objects are prefixed by *sim_*.

Sim++ and CPS are like other simulation languages, such as Simula, in that the computation model presented to the user includes a concept of time, which the user must explicitly manipulate. Unlike sequential simulation languages, both Sim++ and CPS present the user with a different clock in each LP, and each event, implemented as a message, is time-stamped.

As mentioned earlier, both Sim++ and CPS provide a library of objects to the user. In addition, the user defines a set of objects corresponding to LP's. The issue arises of what form the interface to all these objects should take: a message passing or a procedure call interface. Both languages choose a message passing interface, for simplicity of implementation. (In contrast, Tinker and Agre's time-warp implementation uses a procedure call interface [TINK89]).

A message passing interface has a disadvantage. Each message can have a user-defined, untyped data strcture associated with it. The language implementation cannot guarantee that the sender and receiver of a message interpret the user-defined body of each message correctly. Furthermore, the user must tell the implementation how long the message body is; if this value is inaccurate the message passing mechanism will malfunction. Sim++ will transmit a type name in the body of the message to reduce the problem, but the problem cannot be eliminated.

Both languages partition the lifetime of a simulation program into two phases: an initialization phase and an execution phase. This choice was made to simplify the implementation. Having a single phase would potentially increase performance, but at the same time would require semantics for communication with LP's that do not yet exist. This problem has been addressed in Tinker and Agre's system.

Because execution speed is a major consideration in Sim++ and CPS, both language implementations have extensive facilities for collecting statistics on the efficiency of the simulator itself.

4.2 Concepts Specific to Sim++

In *Sim++*, the simulation programmer constructs a simulation model by representing each class of physical processes in the system being simulated by a class of entities. Each class of entities is derived from class entity and each entity is represented by an instance of a particular class of entities.

The simulation programmer also defines a set of events that entities use to communicate and synchronize the actions. When an entity schedules an event it can associate information concerning the event by attaching an arbitrary data structure to it.

Every entity is denoted by a unique entity identifier. Entity identifiers are used to identify the entity for which an event is scheduled.

The actions of an entity are described using the following basic set of simulation primitives:

clock returns the entity's simulation time

self returns the entity's entity identifier

schedule schedules an event for a specified entity at a particular future simulation time

cancel eliminates a previously scheduled event

wait blocks the entity until the simulation time of the next event

hold models a simulated delay interruptible by the occurrence of an event

Sim++ provide libraries for random number generation, data collection, linked-list manipulation, and distributed, parallel input and output.

4.3 Concepts Specific to CPS

CPS stands for "Common Programming Structure." The term *common* refers to the fact that common source code can be implemented, without change, by one of several simulation protocols. The current implementation of CPS, a system called the object library for parallel simulation (OLPS) [ABRA88], provides BCM with deadlock avoidance, time-warp, and sequential protocols. To achieve this, a CPS program requires more information than a program requires to be executed by time-warp or the BCM protocol alone. For the BCM protocol, this information includes a function for each LP to respond to null messages. For the time-warp protocol, this information is optional and, if provided, optimizes performance of the memory management syste (see section 5.6).

In CPS, a simulation model consists of a set of LP's connected by a communication graph. An LP consists of three components: a *sequencer*, a *responder*, and a *router*. Messages flow through an LP first through the sequencer, then through the responder, and finally through the router. These three components have the following functions:

Sequencer: Receives messages sent by upstream LP's. Provides an interface with other LP's that send messages to a given LP. The sequencer contains a buffer pool. The buffer pool allows concurrent writing and reading by multiple LP's. The sequencer passes a sequence of messages to the responder.

Responder: Receives individual messages from the sequencer, simulates the physical process that a given LP models, and passes a resultant list of messages to the router. The responder modifies a set of state variables that are local to an LP.

Router: Sends each message in each list received from the responder to downstream LP's. Provides an interface with other LP's that receive messages from a given LP. The router may contain a routing table.

One concept that CPS has that is not present in Sim++ is that of a global directory of all LP's, which is constructed during initialization. This directory may be queried during execution, for example, to obtain a list of all LP's that are instances of a particular type. This feature is useful in simulation models in which a portion of the model comes from a library. The LP's in the library can query the directory to find out the topology of the rest of the simulation model. Thus directories help build decomposable simulation models.

5. Key Design Considerations

5.1 Process versus Event Oriented Views

A fundamental choice in designing a simulation programming language, be it implemented by a sequential or parallel protocol, is whether the programmer is presented with a process or an event-oriented view. We conjecture that when a parallel protocol is used, the choice between process or event view may be a choice between ease-of-use or performance, respectively.

Sim++ uses the process view, based on the belief that from the simulation modeler's perspective, the process view is easier to use than the event view.

CPS, which uses the event-oriented view, was designed with the philosophy that ultimately parallel simulation would be desirable in situations where the number of LP's is much larger than the number of processors. This has two implications, on memory and on performance.

Memory implications: In a process-oriented simulation, each LP must be active during the entire simulation program (assuming that LP's cannot be created and destroyed during simulation). Each LP is implemented by one thread of control, for example by a light-weight process. In contrast, in an event-orient simulation, each LP must only be active for the period of time that it takes to execute a single event. Therefore an LP can be implemented by a function.

This distinction is important, because each LP needs a stack area while it is active. The fact that in a process-oriented simulation all LP's are active through the entire simulation implies that the portion of memory that needs to be dedicated to LP stacks is the sum of the maximum estimated stack size used by all LP's. In contrast, in an event-oriented simulation, the stack size required is only the sum of stacks required by all LP's active at any moment. Since the number of LP's is typically much larger than the number of processors, the stack size requirement for the process view is much larger than the stack size of the event view.

In practice, a programmer can ignore the stack size issue in an event-oriented simulator, but may need to estimate stack requirements in a process-oriented simulator. This value is chosen to be small enough to permit many LP's to be created and to leave enough free memory for the time-warp protocol. This value is chosen to be large enough

to make unlikely the possibility of a simulation aborting because an LP exceeded its stack size.

Performance implications: The desire for many threads relative to the number of processors implies that an efficient thread-package is required. For parallel simulation to be as efficient as sequential simulation, the context switching time for threads should be on the same order of magnitude as the procedure call/return time. (This assumes that each event in a sequential simulation is executed by a separate procedure.)

The event-oriented paradigm allows each user-defined responder in CPS to be written as a function. The scheduler for a processor then executes a procedure call to invoke a responder, rather than executing a context switch. This is possible because a responder will never block, as all blocking occurs in the sequencer or router.

The importance of basing threads on procedure calls rather than context switches will grow when parallel processors are based on RISC architectures, because larger number of registers make the ratio of context switch time to procedure call time higher than on a conventional architecture processor.

5.2 Basic Program Structure

Given that the language designer has chosen between the process-oriented and event-oriented views, a question that soon arises is what form the basic program structure should take. For example, both of the languages discussed in this paper employ object-oriented programming. In this case two questions arise: how should the objects themselves be specified and organized in a source program (i.e., *macroscopic organization*), and how should the internal functioning of an object be specified (i.e., *microscopic organization*)? These two questions are examined below.

Macroscopic organization: Both Sim++ and CPS programs have no "main" function. Instead the programmer defines a set of LP classes, and then defines a function that creates instances of the LP classes. This forms the entire source program. The Sim++ or CPS run-time environment is then responsible for distributing the LP's to processors, initiating execution, and scheduling LP's. This form of program is quite a bit different from the form used in non-simulation programs and traditional, sequential simulation programs.

A Sim++ class representing an entity contains a member function called *body()*. This function performs the simulation associated with the LP, and typically contains

statements to wait for events, hold for certain periods of time, and schedule events for other entities. In a sense, each LP can be thought of as a separate program that interacts with other LP's using the simulation primitives provided by the simulation library.

A CPS class representing an LP consists of a constructor and two member functions:

RespondToUser(message M)* : This function is called by the run-time executive whenever a user-defined message *M* arrives for the LP. The function is defined by the user, and must perform the simulation necessary to respond to the message, and then return a list of zero or more messages that will be routed to other LP's in the system.

RespondToNull(message M)*:. This function is called by the run-time executive whenever the BCM protocol is used and a null message *M* arrives for the LP. This function is defined by the user to optionally generate a list of null messages with updated time-stamps to be routed to other LP's in the system.

Microscopic organization: Within a single LP (i.e., in function *body()* of Sim++ or functions *RespondToUser(...)* and *RespondToNull(...)* in CPS), both languages provide: (1) ways to control the ordering of messages received by an LP, and (2) ways to control how messages are scheduled or sent to other LP's.

Sim++ provides a notion of *predicates*. Predicates specify a set of conditions that an event must satisfy to be received by the LP There are two predefined predicates (testing for equality of events and equality of LP's that schedule events), and the user can define additional predicates. The simulation primitives to wait for events and hold for a delay may be called with a predicate to specify which events will be accepted and which events may interrupt the delay, respectively.

CPS provides the concepts of *sequencers* and *routers*, which order messages received by an LP and provide different rules to route messages to other LP's. CPS attempts to simplify the programming of common queueing disciplines and routing rules by providing a small library of useful sequencers and routers. Because CPS is organized as a library of C++ objects, the programmer can define his own sequencers and routers and add this to the library.

5.3 Distributed, Parallel Input/Output

Integrated language facilities for input and output are often overlooked in the design of parallel programming languages or specially tailored to a specific machine architecture. This is due to the fact that there is not a widely accepted model for I/O that is both efficient and adaptable across the spectrum of multiprocessor architectures. Examples of this spectrum include the fact that some machines provide multiple I/O channels while others only provide a single I/O channel, and some machines allow all processors to perform I/O while others only allow a subset of processors to perform I/O. Any portable I/O facility has to provide the user with a consistent interface across all of these machines while permiting the implementation to take advantage of whatever facilities are available. At the same time, the I/O mechanism must be capable of the high performance required by parallel programs. After all, it would be a pity to optimize the performance of a parallel simulation program only to find that the I/O mechanism is inadequate and is the bottleneck.

Sim++ addresses this set of problems by providing layered I/O facilities.

At the highest layer, Sim++ provides a set of routines machine independent routines that allow processes to open, close, read, and write files and terminals in a manner reminiscent of most sequential programming languages. Each implementation of Sim++ implements these routines in a fashion appropriate to the underlying machine.

At the next layer, Sim++ provides a set of file servers and console servers that are implemented using pre-defined processes. Processes interact with these servers in the same way that they interact with any process - using events. The facilities provided by this layer are more rudimentary than those provided by the highest layer. However, they allow the user to specially tailor the I/O mechanisms for the application and machine architecture. For example, this layer allows I/O servers to be assigned to specific processors that are attached to external devices and permits user level buffering of I/O.

The lowest layer of I/O facilities permits users to write their own I/O servers. This allows users to have complete control over the I/O mechanism. This is useful when a machine has I/O devices that cannot be properly supported using the high level I/O routines or the pre-defined I/O servers.

CPS provides a simpler but less flexible model for I/O. CPS extends the C++ concept of input and output streams by adding the concept of commitment. Modifications

to a stream may be rolled back when the time-warp protocol is used. An instance of a stream is private to a single LP.

5.4 Making Costs of Operations Explicit to the Programmer

One common principle in language design is the use of similar notations for similar operations. However, application of this principle may be inappropriate when similar operations have radically different costs. When this is the case, using similar notations may mislead the user into thinking that the operations have similar costs. This leads to the principle that the run-time costs of operations should be made explicit.

Accessing process attributes provides an example of this principle. Suppose a simulation consists of two processes referred to as P1 and P2, both defining an attribute called A. In Simula, P1 could access its copy of A as P1.A and P2's copy of A as P2.A. Using the same notation in both cases is appropriate because both operations have the same run-time cost. In a parallel simulation language the cost of P1 accessing P2's attribute may be two orders of magnitude more expensive than accessing its own attribute. This difference in cost is due to the fact that P1 and P2 may be running on different processors and accessing a remote attribute may involve message passing. In this case, using the same notation for both operations would be inappropriate because it might mislead the user into believing that the costs associated with the two operations are similar.

For this reason, non-local references in both Sim++ and CPS must be coded using message passing (unless both entities are in the same cluster in Sim++).

5.5 Dynamic Changes during Simulation Execution

Another key decision in simulation language design is whether to support any form of dynamic changes to the simulation model or the simulation itself during execution. Changes to the simulation model include allowing creation and destruction of LP's, and allowing changes to the topology of communication between LP's while the simulation is executing. Changes to the simulation itself include changing the mapping of LP's to processors while the simulation is executing. These two categories are discussed below.

Dynamic changes to the simulation model: First we discuss the issue of whether the communication topology can change during execution. This is allowed in both Sim++ and CPS (unless the BCM protocol is used with CPS).

CPS allows optional specification of static links. This is required for CPS when the BCM protocol is used. It is also useful to improve the efficiency of LP migration for the following reason. For efficiency, if two LP's are mapped to the same processor and one LP sends a message to the other, then the communication should occur through shared memory without locking. Specification of the topology allows an LP, when migrated, to examine whether all of its output arcs are on the same processor. If they are, future communication occurs in shared memory without locking. (The same effect could be achieved without knowing the links, but would require a test each time a message is sent to see if the recipient is on the same processor.) Sim++ circumvents the problem by requiring the programmer to put two LP's into the same static cluster if they can communicate without locking. However the mapping of entities to clusters may not change during execution.

The other possibility is dynamic process creation and destruction. Neither Sim++ nor CPS allow this. Both languages have made this decision first because the need is not critical if the language implementation supports lots of cheap threads, and second because dynamic creation/destruction requires some sort of dynamic load balancing. In addition, Sim++ does not provide dynamic creation and destruction because it is harder to preserve determinism. Meanwhile, CPS does not allow this because the BCM protocol is difficult to implement when processes can dynamically be created and destroyed.

Dynamic changes to the simulation itself: We identify three ways to balance processor utilization: (1) changing a static assignment of LP's to processors, (2) changing the assignment dynamically by allowing LP's to migrate, or (3) letting multiple processors share a ready queue of LP's waiting to run.

Option (1) is easy to implement, while options (2) and (3) are currently active research topics. An ideal solution would exist if good algorithms to do (2) and (3) were developed that could work transparently to the simulation program. However, the current design of both Sim++ and CPS are based on the limited mechanism known today.

Sim++ makes the first choice of static assignment. Sim++ provides a tool to report on processor utilization after the simulation completes; this may be used to change the assignment on future runs to balance utilization.

CPS makes the second choice. It provides a mechanism to migrate LP's during simulation, but leaves to the user the choice of a policy specifying *when* to migrate. Thus

the language provides a primitive operation to migrate an LP. This operation may be called by the user or, in the future, by a yet to be discovered load balancing algorithm.

Migration may be more effective for simulation than it is in general operating system contexts. This is because a simulation may schedule the same LP's through the entire course of a simulation. In contrast an operating system is continually scheduling new jobs whose historical behavior is unknown. Thus there may be more regularity in simulation compared to the set of jobs being scheduled by an operating system.

5.6 Influence of Language Construct on Memory Management

Every programming language presents a certain model of memory to the programmer. The programmer may have control over aspects of memory usage through the language constructs available to him. (For example, the C++ programmer can allocate and de-allocate memory using the operators *new* and *delete*.)

Memory management is a major consideration when the time-warp protocol is used to implement a parallel simulation language. The difficulty is that time-warp maintains multiple versions of an object in memory during execution that are used during roll-back. This suggests that a language designer may want to allow the programmer to optionally specify how he expects objects in his simulation to be used at run time; such information may reduce the memory requirements of a program.

In the case of CPS, the language provides a concept called *sub-states*. A programmer may optionally specify a partitioning of the objects comprising the state of an LP into a set of sub-states. The time-warp implementation will then manage separate versions for each sub-state. In addition, both Sim++ and CPS allow objects that comprise an LP's state but are read-only during simulation to be identified as such, so that the implementation only needs to keep a single version of the objects. (This is called *read-only state* in CPS and *write-locked memory* in Sim++.)

As an example, consider a simulation in which one or more LP's collects statistics in a histogram of 1000 bins. Without the sub-state concept, an implementation of time-warp would need to allocate a new 1000 bin histogram each time a new version of the LP state is created - even though only a few bins of the histogram were modified in the current version. In contrast, in CPS the programmer could partition the histogram into a set of bins. A new version would only be created for sets whose bins were modified.

Sub-states can potentially decrease the running time of a time-warp based simulation. However they require the programmer to know enough about the dynamics of simulation to make a good choice of sub-state partitions.

In CPS the programmer may optionally specify when a version has been changed, so that new versions need not be allocated every time an LP is run. The user may also specify how frequently new versions are created.

Both Sim++ and CPS employ the C++ memory allocation and de-allocation operators *new* and *delete*.

5.7 Semantics of False Messages in Time-Warp

When a parallel simulation is executing using time-warp as the synchronization mechanism, it is possible for a process to receive an event out of order and, as a result, enter an error state that could otherwise never occur. While the correct event will arrive eventually and the mis-ordering will be corrected, the error must be detected and handled in the interim. The existing message that caused the transient error is called a false message in the terminology of Lin and Lazowska [LIN89].

The problems associated with transient errors can be illustrated with an example. Suppose that the next true event that a process expects to receive contains an integer greater than zero that is to be used as the divisor in an arithmetic operation. Further suppose that the process receives a false event that contains the value zero. When the process extracts this value from the event and executes the division operation, a divide-by-zero error will occur.

Detecting and handling transient errors is the responsibility of the time-warp executive. However, the amount of effort necessary to detect and handle all possible errors has led to a situation where no current implementation is able to do this.

This means that the user is left with the responsibility of detecting and handling some transient errors. Sim++ provides the function *error()* for this purpose. Upon detecting a transient error, the process calls error() This blocks the process until some action occurs that resolves the error. Either the correct event will arrive and the process will rollback from the transient error or the run time executive will determine that no

outstanding event will resolve the error in which case the simulation is aborted. In the later case, the error is the result of an erroneous application.

5.8 Program Development Methodology Considerations

Developing parallel programs is estimated to cost ten times as much as developing sequential programs. This is due to a variety of factors including: programmers are less familiar with parallel programming techniques; testing and debugging is more difficult due to the distributed and non-deterministic nature of parallel programs; and fewer tools are available for supporting this style of programming. It is interesting to note that simulation programmers are probably the best prepared among all programmers for moving from sequential programming to parallel programming. This is because they are already familiar with programs that involve concurrently executing processes.

Both CPS and Sim++ support software development by providing tracing facilities. Tracing information can be generated automatically by the system or by the user for user defined events.

Sim++ provides a software development methodology that is supported by three run-time executives: sequential, emulated distributed, and distributed. The sequential environment runs on a single processor and uses a centralized event-list. The emulated distributed executive runs on a single processor but emulates many of the conditions present in a distributed environment, including the possibility of false messages and rollback. The distributed executive runs on a multiprocessor using time-warp as the synchronization mechanism.

Initially a Sim++ program should be run using the sequential executive. This allows the simulation to be efficiently debugged and tested using standard sequential programming tools. After all errors detected using the sequential executive have been corrected, the program should be run using the emulated distributed executive. This executive simulates a multiprocessor scheduling mechanism and permits errors that would normally only manifest themselves on the distributed executive to be detected and corrected. Finally, the program can be executed using the distributed executive running on a multiprocessor.

6. Conclusions

Based on our experience in designing Sim++ and CPS, the key challenge we faced has been to balance a unique set of trade-offs that do not arise when designing other types of programming languages. In implementing a parallel simulation language, we cannot trade space and time off against each other; often there are strict constraints that must be met for both time and space simultaneously. Consequently, it is difficult to achieve robust performance in the face of small modifications to the program, across different numbers of processors, and across machine architectures.

Since implementation methods are still evolving, it is premature today to design a perfect simulation language. New implementation methods, such as ones for load balancing, will affect language design.

References

[ABRA88] M. Abrams. The Object Library for Parallel Simulation (OLPS). *Proc. Winter Simulation Conference*, San Diego, CA (Dec. 1988) 210-219.

[ABRA89] M. Abrams. A Common Interface for Bryant-Chandy-Misra, TimeWarp, and Sequential Simulators. *Proc. Winter Simulation Conference*, Washington, D.C. (Dec. 1989).

[BRYA77] R.E. Bryant. *Simulation of Packet Communication Architecture Computer Systems*. Tech. Rep. MIT, LCS, TR-188, M.I.T, Cambridge, MA (1977).

[CHAN79] K.M. Chandy, V. Holmes and J. Misra. Distributed Simulation of Networks. *Computer Networks 3*, (1979) 105-113.

[JEFF85] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the TimeWarp Mechanism. *Proc. Dist. Sim. 1985*. Society for Computer Simulation, San Diego (Jan. 1985), 63-69.

[JEFF87] D. Jefferson, *et al.* The Time Warp Operating System. *11th Symp. on Operating System Principles*. Austin, Texas (Nov. 1987).

[LOMO89] G. Lomow and D. Baezner. *Object Oriented Simulation and Sim++. Proc. Winter Simulation Conference*, Washington, D.C. (Dec. 1989).

[MISR86] J. Misra. Distributed Discrete-event Simulation. *ACM Computing Surveys 18, 1* (March 1986) 39-66.

[PREI89] B.R. Preiss. The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments. *Dist. Sim. 1989* Society for Computer Simulation, Tampa (March 1989) 139-144.

[SCS] *Proc. Distributed Simulation.* Society for Computer Simulation, San Diego, CA (1985, 1988, 1989).

[STRO86] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading, M.A. (1986).

[TINK89] P. A. Tinker and J. R. Agre, Object Creation, Messaging, and State Manipulation in an Object Oriented Time Warp System. *Dist. Sim. 1989* Society for Computer Simulation, Tampa (March 1989) 79-84.