Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

THE IDES FRAMEWORK: A CASE STUDY IN DEVELOPMENT OF A PARALLEL DISCRETE-EVENT SIMULATION SYSTEM

David M. Nicol

Department of Computer Science
Dartmouth College
Hanover, New Hampshire 03755, U.S.A.

Michael M. Johnson
Ann S. Yoshimura

Sandia National Laboratories
Livermore, California 94550, U.S.A.

ABSTRACT

This tutorial describes considerations in the design and development of the IDES parallel simulation system. IDES is a Java-based parallel/distributed simulation system designed to support the study of complex large-scale enterprise systems. Using the IDES system as an example, we discuss how anticipated model and system constraints molded our design decisions with respect to modeling, synchronization, and communication strategies.

1 INTRODUCTION

The use of parallel computers to execute discrete-event simulations has been a topic of research interest for nearly 20 years. Until recently, parallel computers could be found only in research labs, and application of parallel simulation technology was limited by the simple problem of lack of access. That has changed. Shared-memory multiprocessors have become a commodity product. Fast networks to link personal computers have become commodity products. It is now possible to order the pieces of a tremendously powerful distributed / parallel system *over the Internet* one day, receive and assemble it two days later.

But, while hardware to support large-scale simulations is readily accessible, software (typically) lags behind. In the enterprise computing world a number of tools, languages, and standards exist, e.g., Java and its development environments, CORBA and its implementations. However, systems to support large-scale distributed simulations are absent.

In this paper we discuss issues that arose in the development of a parallel/distributed simulation system which was intended from the start to support a certain type of application, on a variety of commercially available platforms. We anticipate that the lessons we learned in the course of designing and building this system have application to other systems as well.

Perhaps the main point we wish to convey is that *capability* is our main concern, not run-time performance. Of course, execution time is a consideration, but we view it as a constraint rather than an objective function. In the enterprise computing world issues of portability, maintainability, and conformance to standards are as important as fast run-time, so much so that it is acceptable to sacrifice execution speed to provide these other capabilities.

2 IDES DESIGN

IDES, an Infrastructure for Distributed Enterprise Simulation, is a parallel simulation framework for complex, large-scale enterprise simulations. IDES was developed to support Sandia National Laboratories in the study of issues of importance to national security. Many of these issues involve the analysis of complex systems. IDES is a policy driven simulation tool capable of performing decision directed analysis of complex system models. The goal of such analysis is to discover the emergent collective behavior of the system through the interaction of detailed individual sub-model simulations—the definition of enterprise simulation.

2.1 System Design Goals

To motivate the IDES system design, consider an example domain: simulation of a U.S. Health Maintenance Organization (HMO). The IDES design was governed by three goals. The first goal deals with the structure of the simulation framework to express the systems to be modeled: link low-level, complex sub-models with high-level, policy driven resource allocation techniques to perform cost / benefit trade-off analyses. In the HMO example, each patient is modeled with complex disease processes represented by differential equations—including risk for coronary artery disease. Medical treatment policies interact with disease models to affect the health outcome of patients.

The second goal mandates a type of question the simulation model must be able to answer. Using IDES, we want to study the use of screening techniques to detect an otherwise invisible system deterioration, itself a contributor to a catastrophic failure we would like to prevent. In the HMO example, we would say the early detection and treatment of coronary artery occlusion may extend life and saves later costs when heart failure might otherwise occur.

The third and final goal specifies the portability of the system: development of simulation models using IDES should be within the reach of systems analysts, and support deployment across heterogeneous computing architectures. IDES runs on single-processor systems, networks of workstations, and multiprocessor computers with shared or distributed memory. In addition, IDES incorporates a web-based interface for distributing simulation subcomponents across the enterprise network.

In support of these goals we have developed the IDES framework. IDES is an object-oriented simulation system capable of supporting complex, massive model, parallel discrete event simulations transparently across heterogeneous platforms.

2.2 System Constraints

In support of these design goals, a number of system constraints had to be overcome. First and foremost, IDES had to be capable of hosting massive models with relatively large state. The example HMO model includes ten million patients and one hundred regional hospitals and facilities. Enterprise simulations evolve differently than more traditional parallel simulation models such as queuing and PCS networks. For example, simulation entity behavior is not governed by a simple draw on a random number stream, but through the evaluation of complex, coupled state-evolution equations. Hence, the difficulty of extracting lookahead discourages the use of a purely conservative protocol.

Since the data state of each component is large, we use multiple machines to acquire the memory needed. While a conservative approach to synchronization could use less memory than an optimistic approach, lack of lookahead limits the effectiveness of conservative synchronization. Consequently very large state coupled with lack of lookahead motivates use of Breathing Time Buckets (BTB) developed by Steinman (1992) to constrain optimism. Furthermore, sheer model size and portability concerns motivated investigation of impact of architecture on performance.

The state of simulation entities is computationally complex. In the HMO example, evaluation of complex disease models is computationally expensive. Parallelism is evident with a large population.

3 SYNCHRONIZATION

Synchronization is generally viewed as the key source of difficulty when executing discrete-event simulations. Conservative synchronization methods ensure that every bit of computation executed contributes directly to the final simulation state; optimistic methods support speculative computing where some computations may ultimately be undone. The task of building a parallel simulation framework is understandably easier with a conservative approach. However, there is ample evidence that reasonable performance can be achieved under conservative synchronization only if there is easily extracted lookahead in the simulation model. This simply means that without a great deal of effort it is possible to examine the state of a sub-model (the term we'll use to identify that portion of the simulation model that is cohesive in the sense that all simulation work associated with a sub-model will be done by the same CPU) and find a lower bound on the time when next that sub-model performs some action that affects the state of another sub-model. Dissemination of lookahead provides the slack needed between processors that permits them to make forward progress without concern for so-called straggler messages (messages with time-stamps less than the recipients local simulation clock).

Our initial intent was to use a synchronization protocol based on YAWNS by Nicol (1989, 1993). YAWNS is conservative, and when suitable lookahead is available, is provably scaleable. However, as we studied the class of model problems we began to see that lookahead would not be easy to extract without requiring the IDES user to provide more information about the model than we thought the user would typically care to provide. Consider again the HMO model. A patient's risk profile with regards to, say, heart disease, is dependent upon a number of risk factors including lifestyle choices, family history, and known health problems within ones family. A differential equation describes the probability distribution of the time of next heart attack, as a function of those risk factors. If any of those risk factors were to change, a heavy-weight computation would be required to recompute the probability distribution. The sort of lower-bound calculation needed to compute lookahead would have to identify the worst-case combination of risk factor values and assume they simultaneously changed to this worst case scenario, and then compute a worst-case time-to-heart-attack distribution. The only alternative is to require the modeler to provide this sort of worst case information (at the risk of the modeler being wrong!). We eschewed those constraints in favor of a limited form of optimism that constrains the sort of large-scale memory consumption that general Time Warp simulation is capable of requiring.

We next considered the Breathing Time Buckets (BTB) synchronization approach, as it is essentially an optimistic version of YAWNS. While being optimistic, it ensures that messages between sub-models are "correct" in the sense that they will not be canceled. In its simplest form, BTB works as follows. Simulation objects synchronize at points in simulation time (the determination of which is the point of the protocol). At a synchronization point, messages are exchanged between sub-models; as these messages are correct, they can be incorporated into their recipients' event lists. Next a sub-model executes events on its event list in time-stamp order, performing state-saving. As messages to other sub-models are generated, these are buffered but their so-called *receive-times* are noted, the times when the message affects the recipient (as opposed to the time when the sender sends it, which may be different). A sub-model tracks the minimum receive-time of any message it generated but not yet delivered. At the point when the time of next event is greater than or equal to the minimum such receive-time, the sub-model has reached its *local event horizon*. BTB defines the next synchronization point as the minimum local event horizon among all sub-models, this called the *global event horizon*. The global event horizon essentially establishes the least next time when an as-yet-unknown message can arrive at a sub-model and change its state. Therefore, all computation up to the global event horizon is known to be "good" in that even though computed speculatively, it did not depend upon a message from another sub-model. Of course, a sub-model may have been advanced beyond the global event horizon, and so (at least conceptually) is rolled back to the global event horizon.

A naïve way of determining the global event horizon is to have each sub-model execute all the way until reaching its local event horizon, and then engage in a global minimum-reduction operation to identify the least such. This would actually maximize the amount of memory used for state-saving in a BTB approach, in that each sub-model would be executed as far as could be possible, saving state the entire way. Clearly, to reduce state-saving costs one needs to disseminate local event horizons as they are discovered. Towards this end we developed an algorithm—the preemptive min-reduction—to attempt to identify and distribute the global event horizon quickly.

In a normal reduction a processor offers a value to the reduction operator and then blocks until all processors have offered values and the reduction is performed. A processor interacts with a preemptive min-reduction somewhat differently. Each processor has a "working minimum" in the case of BTB the least observed receive time on generated messages. As the computation progresses, the working minimum changes in a monotonically non-decreasing fashion.

The reduction framework in a processor maintains a "partially reduced'' value, initially infinity, to reflect the minimum value reported to that processor in the course of the preemptive-reduction. Periodically (say, after each event) a processor compares its time of next event with the partially reduced value. If the former value is smaller, the processor's progress has been preempted by knowledge of the existence of a local event horizon, somewhere, that is smaller than the processor's own. It then engages in the reduction logic, offering the partially reduced value as its own.

It blocks until the reduction is completed and the global event horizon is identified. Alternatively, if a processor reaches its local event horizon without being preempted, it simply engages in the min-reduction. All that is needed to implement this algorithm is user code access to the partially reduced value that is typically in tree-based reduction algorithms. We have based our implementation on the non-committal barrier synchronization by Nicol (1995).

4 IDES Modeling

There are a large number of factors that potentially affect performance of the IDES system. We thought it prudent, prior to building IDES, to anticipate some of the performance considerations, by first building an analytic model of IDES and study *its* behavior. We have already reported on that work by Nicol, Johnson, Yoshimura, and Goldsby (1997), we only sketch the approach and results here.

The model recognizes that the key elements governing a sub-model's behavior with respect to synchronization are (1) its time of next event, and (2) its minimum known receive time on generated messages. A sub-model's state is described by a pair of real numbers, recording these two elements. Stochastic assumptions are made about changes in those two elements as events are processed. A sub-model reaches its local event horizon when its time-of-next event component dominates its receive-time component. One such model is advanced for every sub-model in the system; additional assumptions about communication delay and construction of reduction trees model the inclusion of a preemptive min-reduction calculation. The end result of the model is a probability distribution of the time required to execute one BTB window. Solution of the model is computational rather than closed form.

In order to include further detail (and temporarily avoid the effort of building a numerically stable solver), we developed a simulation of this model. Performance studies using the simulation revealed the sensitivity of performance to the delay through a network interface that is shared by all processors in an SMP. This result has immediate bearing on the issue of hardware

acquisition—ironically, the systems most prone to having the network interface be a performance bottleneck are the high end larger scale (and more costly) SMP servers. Actual studies are needed to assess whether the advantage of local communication between sub-models in the same SMP is enjoyed. Another point of interest was that perfect load balance is difficult if not impossible to achieve when the workload is stochastically driven. The inherent variance in the workload behavior induces a certain level of imbalance. An important conclusion to draw from this study is that complex load-balancing schemes are unlikely to be significantly more effective than simple schemes—a conclusion that has obvious bearing on IDES system design. A final lesson we learned from the simulation study was that a performance optimization we considered with regards to handling communication was usually quite effective, and should hence be considered for inclusion in the IDES system.

We believe that the effort we applied to developing analytic and simulation models of IDES helped us to understand much more deeply how such a system must operate, and the sort of performance sensitivities we could expect from the system once built. Armed with this confidence, we proceeded to implementation.

5 IDES IMPLEMENTATION

The IDES design has been implemented separately in both C++ and Java. This paper deals exclusively with the Java implementation.

5.1 Class Structure

The two main simulation classes are *Entity* and *Message*. All simulation objects are represented by the Entity class which encodes individual state and behavior. Entities communicate with one another by sending Messages which contain routing information as well as message content.

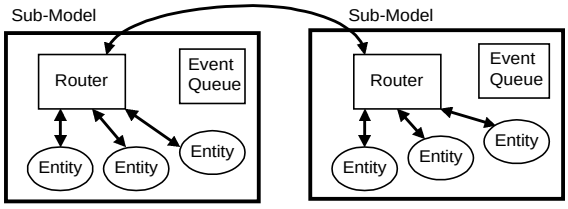

Figure 1: IDES Model Decomposition

Two additional base classes complete the IDES framework: *EventQueue* and *Router*. In IDES, a simulation is decomposed into a number of sub-models, each consisting of a subset of all simulation Entities (Figure 1). Each sub-model contains an EventQueue and a Router.

```
protected double wakeup (double time) {

   // CHECKPOINT THE STATE OF THE OBJECT, AND
   //   UPDATE ENTITY TO THE CURRENT TIME.
   checkpoint(time);
   update(time);

   // PERFORM INTERNAL ENTITY EVENTS.
   performInternalEvent();

   // RESPOND TO EXTERNAL MESSAGES.
   while(!messages_.isEmpty())
     performMessage(messages_.dequeue());

   // DETERMINE TIME OF NEXT WAKEUP.
   return forecast();
}
```

Figure 2: *Entity* Event Processing Routine

Execution of simulation events for Entities on the sub-model is controlled by the sub-model's EventQueue. The role of the EventQueue is simply to hand the thread of execution control to the appropriate Entity at the appropriate simulation time, by invoking the Entity's *wakeup* routine (Figure 2). In this routine, the Entity executes the events that should occur at that time, including response to and sending of Messages if required. It then gives execution control back to the EventQueue, having forecast (Figure 3) the time of next wakeup. Hence each entry in the EventQueue consists of an Entity reference and the simulation time at which the Entity should be "woken up."

```
protected double forecast() {

   // CALCULATE EARLIEST INTERNAL EVENT.
   wakeupTime_ = forecastInternal();

   // CALC. EARLIEST MESSAGE RECEIVE TIME.
   if (!(messages_.isEmpty())) {
      double messageTime =
         messages_.headKey();
      if (messageTime < wakeupTime_)
         wakeupTime_ = messageTime;
   }

   // RETURN EARLIEST TIME.  THE ENTITY WILL
   //   BE WAKEN UP AT THIS TIME.
   return wakeupTime_;
}
```

Figure 3: *Entity* Forecast

The Router is responsible for routing and filtering all Messages that are sent to and from the Entities on the Router's sub-model. The Router is also responsible for establishing synchronization windows with other Routers in the simulation, according to the algorithm discussed above.

5.2 Decomposition Mechanism

Entities are arranged in a hierarchy in which parent Entities are responsible for instantiating child Entities. We refer to the highest level parents as the *top-level Entities*.

For a particular simulation run, each top-level Entity must be assigned to a specific sub-model. We implement this mapping as a matrix of size (number of top-level Entities) x (maximum number of sub-models allowed). For any top-level Entity, given the number of sub-models in the simulation, the corresponding matrix entry identifies the assigned sub-model.

Invocation of the IDES executable code instantiates a single sub-model to which two arguments must be passed: (1) the total number of sub-models in the simulation and (2) the unique identifier for this particular sub-model. Each sub-model will then instantiate only the top-level Entities that have been assigned to it, based on the matrix described above.

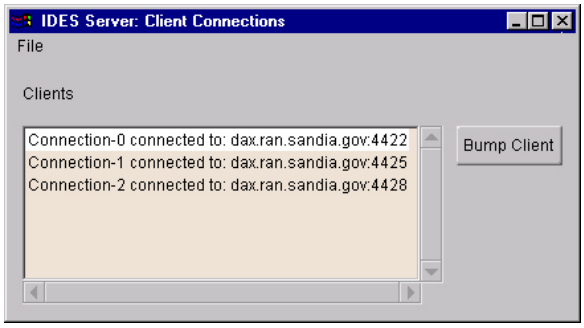

Figure 4: IDES Code Distribution Server

It should be noted that the Entity to sub-model assignment is an initial (simulation start-up) assignment only. We do not restrict Entities from migrating from one sub-model to another during a simulation run.

5.3 Code Distribution

The IDES distribution mechanism is also implemented in Java. At start up, the IDES Server is running on every machine that may be used as a host for the simulation run. The Server's user interface (Figure 4) allows the owner of the machine to control the use of the machine by remote IDES Clients. The IDES Client (Figure 5) is run by the simulation owner (the "user"). For a simulation run, the user indicates (1) the directory in which the simulation executable code resides, and (2) the machines on which the simulation is to be run. As each machine is selected, the IDES Client contacts it to ensure that the IDES Server is running there, ready to accept transmission of the simulation code.

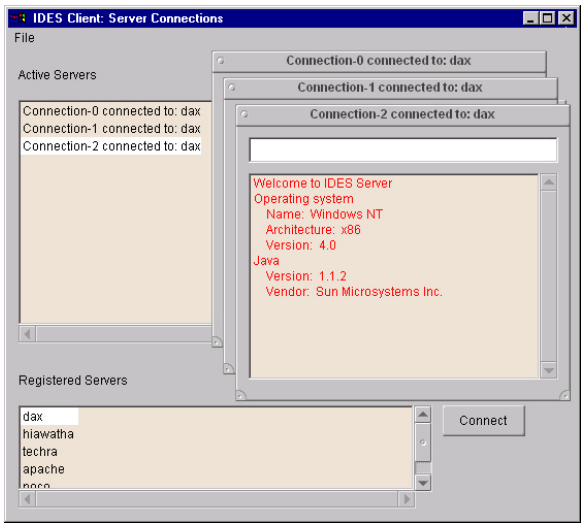

Figure 5: IDES Code Distribution Client

Upon user command, the IDES Client transmits to each participating Server the following data: (1) the simulation executable code, (2) the identification number for the sub-model to be instantiated, and (3) the total number of sub-models in the simulation. The Server then invokes the executable on its machine, creating the proper sub-model. The Client also sends to each Server the addresses and sub-model identification numbers for all other participating machines. This information is passed to the executing sub-model whose Router then uses it to establish a communication link to the Router in each of the other sub-models. The simulation is now ready to run.

5.4 State Saving Mechanism

Within BTB, individual simulation sub-models are allowed to optimistically surge forward, speculatively executing events on their events lists in time-stamp order. Since receipt of a message with receive time less than the current event execution time necessitates a state rollback, sub-models must perform state saving.

Driven by the need to support massive models and thus limit the amount of saved state, we first considered the naïve approach of state saving only once at the window boundary. The simulation would then be allowed to process forward speculatively until detection of the event horizon. With the event horizon determined, all simulation sub-models would be rolled back to the beginning of the window and run forward again to stop at the event horizon. While this scheme minimizes the amount of saved state, it necessitates execution of the simulation twice.

Next we considered going to an incremental mechanism whereby individual state variables are saved as they are changed. However, implementing this scheme in

Java appeared complicated and overly taxing on the user of the system. In addition, experiments showed that due to the coupling of state variables in the objects of interest to IDES, execution of a typical event touched most state variables anyway.

```
abstract public class Entity extends Persistent
implements Serializable
{ …}
```

Figure 6: *Entity* Class Declaration

In the face of these considerations, we implemented what is commonly known as "copy'' state-saving—see Franks, Gomes, Unger, and Cleary (1997) for a discussion of various state-saving policies. Immediately prior to receipt of a message or processing of an event, the system checkpoints the mutable state of the affected entity. The state saving mechanism relies on the Java implementation of object serialization. All IDES object classes are required to derive from Entity (Figure 6), which itself derives from Persistent.

The class Persistent contains the routines for checkpointing and rollback of individual Entity state. This is accomplished through an internal ordering of ByteArrayOutputStreams serialized through an ObjectOutputStream. In the IDES object class hierarchy, all classes from the Entity down are required to implement Serializable (Figure 7). The one drawback to this mechanism is the requirement that object images must be restored to a new address. In most cases, the user wants to *update* the state of an existing object with only those variables that could possibly change since the last checkpoint, and not replace all Entity state values completely. In order to accomplish this, our implementation relies on the Serializable mechanism to restore the state of transient (or non-persistent) variables into a new address space. Then a Persistent routine, *clone*, copies the contents of the newly restored object image into the original image.

```
public class
Car extends Entity implements Serializable {
   public Car (Router router,
               String name,
               int dealerId,
               double maintenanceInterval,
               double messageDelay) { }
…}
```

Figure 7: *Car* Class Declaration

5.5 Example Simulation Problem

Our example problem domain is an automobile franchise comprised of *Dealers*, *Owners*, and their *Cars*.

Dealers sell and service Cars. They also on occasion will issue recalls on certain defective Cars they have sold. Services on Cars include both routine maintenance work and recall repairs.

Owners purchase Cars from Dealers. They may request service from any Dealer, but recalls will always be received from the original (selling) Dealer.

Cars deteriorate with time (Figure 8). Routine maintenance slows the rate of deterioration, but cannot prevent it completely. Defects in Cars can be corrected by recall repair work. The useful life of a Car is affected by the presence of defects and the service work received over the life of the Car. When a Car dies, its Owner purchases a new Car from the same Dealer from which the first Car was purchased.

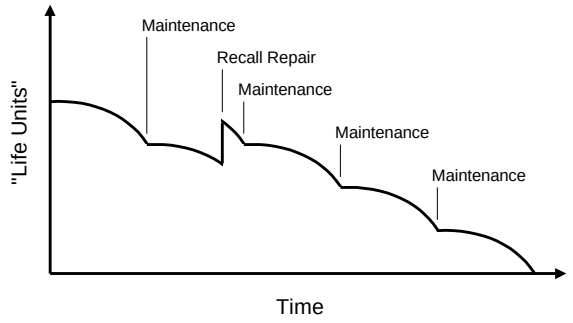

Figure 8: *Car* Deterioration Model

The following code sample (Figure 9) is from the Dealer class, in which a Dealer performs a recall event.

```
private void performInternalEvent () {
   if (time_ == recallTime_) {

      // Send message to car to be recalled.
      sendMessage(new Message(myId_,
                  recallCarId_,
                  currentTime_,
                  (currentTime_ + 0.5),
                  Message.RECALL));
   }
}
```

Figure 9: *Dealer* Sending a Message

The *sendMessage* routine is used to send a Message to the Car to be recalled. In creating the Message, the sending and receiving Entity identifications, the send and receive times, and the type of the Message must be specified.

```
private void performMessage (Message msg) {
   if (msg.type() == Message.RECALL) {

      // PERFORM RECALL.
      lifeUnits_ += .1;
      if (lifeUnits_ > 1.0)
         lifeUnits_ = 1.0;
   }
}
```

Figure 10: *Car* Recall Message Handler

Response to a received Message is done in *performMessage* (Figure 10). The example above is for a Car that has received a recall Message.

After having decoded the recall message, the Car performs the recall—here simply an adjustment of the Car's life units—and then returns immediately to the event processing loop. Next the Entity must determine the future wakeup time based on pending internal events and messages—a function performed by forecast. Since the recall affected the life units of the Car, and hence the internal state of the Entity, the forecast routine must determine when the next internal Entity event will occur.

```
protected double forecastInternal() {

   // EVALUATE DIFFERENTIAL EQUATIONS
   //   TO DETERMINE PREDICTED DEATH TIME.
   double nextTime = calcDeathTime();

   // SCHEDULE MAINTENACE IF PRIOR TO DEATH.
   if (nextTime > maintenanceTime_)
      nextTime = maintenanceTime_;
   return nextTime;
}
```

Figure 11: *Car* Forecast Internal Event

Forecast internal event (Figure 11) calculates the time of next internal event for an Entity. In the simple example given for a Car, the only possible two internal events are the demise of the Car, or a request for maintenance. Once the minimum has been determined, the forecast routine (Figure 3) then decides if the next internal event, or receipt of a pending message, will result in the next Entity wakeup.

6 SUMMARY

The IDES system being developed at Sandia National Laboratories is a parallel simulation framework for supporting the study of complex large-scale enterprise systems. This paper chronicles the development of IDES and how its goals of capability and portability affected our design decisions.

ACKNOWLEDGMENTS

This work was supported in part by the United States Department of Energy under Contract DE-AC04-94AL85000. Nicol's work was supported in part by NSF grant CCR-9625894 and DARPA contract N66001-96-C-8530.

REFERENCES

Franks, S., F. Gomes, B. Unger, and J. Cleary. 1997. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 72-79. IEEE Computer Society Press.

Nicol, D.M. 1993. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM* 40(2): 304-333.

Nicol, D.M. 1995. Non-committal barrier synchronization. *Parallel Computing* (21): 529-549.

Nicol, D.M., M.M. Johnson, A.S. Yoshimura, and M.E. Goldsby. 1997. Performance modeling of the IDES framework. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 38-45. IEEE Computer Society Press.

Nicol, D.M., C. Michael, P.M. Inouye. 1989. Efficient aggregation of multiple LPs in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*,. 680-685.

Steinman J. 1992. SPEEDES: A multiple synchronization environment for parallel discrete-event simulation. In *International Journal in Computer Simulation* (2): 251-286.

AUTHOR BIOGRAPHIES

DAVID M. NICOL received the Ph.D. in Computer Science from the University of Virginia in 1985 and is presently an Associate Professor of Computer Science at Dartmouth College. He has published extensively on topics in performance analysis, parallel computing, and parallel discrete-event simulation. He is on the editorial boards of the INFORMS Journal on Computing, and the ACM Trans. on Modeling and Computer Simulation.

ANN S. YOSHIMURA received the Ph.D. in Chemical Engineering from Princeton University in 1988 and is presently a Senior Member of the Technical Staff at Sandia National Laboratories. Her research interests include parallel discrete event simulation and combinatorial algorithms.

MICHAEL M. JOHNSON is a Senior Member of the Technical Staff at Sandia National Laboratories / California. He received the M.Sc. degree in Computer Engineering from the University of California, San Diego in 1991. His research interests include parallel simulation, computer graphics, and embedded systems design.