SPaDES/Java: Object-Oriented Parallel Discrete-Event Simulation

Yong Meng TEO and Yew Kwong NG
Department of Computer Science
National University of Singapore
3 Science Drive 2
Singapore 117543
email: teoym@comp.nus.edu.sg

Abstract

This paper describes the design, implementation and performance optimizations of SPaDES/Java, a process-oriented discrete-event simulation library in Java that supports sequential and parallel simulation. Parallel event synchronization is facilitated through a hybrid carrier-null, demand-driven flushing conservative null message mechanism. Inter-processor message communication is coordinated by a shared persistent memory implemented using Java Jini/JavaSpaces. We present the stepwise performance optimizations we have carried out, focusing mainly on reducing the cost of multithreading, null message synchronization overhead, and the cost of inter-processor communication. Two benchmark programs consisting of an open linear pipeline system and PHOLD representing a closed system are used. For PHOLD(16x16,16), our optimization reduces the null message ratio from 0.94 to 0.29 on eight processors. Based on our time and space instrumentation, we observed that the memory cost of null message synchronization accounts for less than 10% of the total memory required by the PHOLD simulation.

1. Introduction

Much research in parallel discrete-event simulation (PDES) has focused on designing suitable event synchronization algorithms to enhance the efficiency in simulating specific class of problems. Less attention has been devoted to address the complexity of modeling and developing parallel simulators. In recent years, several simulation environments and tools have been developed by the military and in research to support discrete-event simulation, frequently for specialized domains of interest. These efforts attempted to provide clear boundaries between the application-independent (the synchronization module) and application-specific (the problem under consideration) elements in discrete-event simulation.

Examples of extensible military application simulation support frameworks include Simkit [2], from the United States Naval Operations Research Department, and JADIS [17] from the United States Air Force. Simkit consists of three conceptual layers, and achieves modeling ease by providing a set of Java classes for constructing resident entities and resources in either the event-oriented or process-oriented paradigm, while allowing the user to customize at a higher level than conventional programming. JADIS, adopting an event-oriented paradigm, applies the Model-View-Controller (MVC) paradigm from Smalltalk to the development of Visual Interactive Simulations (VIS), emphasizing on user-initiated interactions to analyze aircraft repair time problems in the domain of airbase logistics.

JSIM [13,14], SimJava [18] and DESMO-J [10] are examples of sequential simulation libraries developed with commercial and educational objectives. JSIM is integrated with a web server and a database management system, and implements the query-driven simulation concept, principally comprising of three communicating processes and a number of data stores to maintain records of models, data and meta-data. Generated simulation results for the relevant parameters can be cached in the data stores and retrieved. SimJava's design principles were derived from the SIM++ library for computer architecture simulations in the HASE project [8]. It was implemented as a multi-threaded simulation environment, and an applet module was integrated to facilitate web-based distributed simulation, which is built upon a master-slave architecture [19]. DESMO-J, on the other hand, is the first library that provides support for a hybrid event- and process-oriented modeling views under the scope

Proceedings of the 35th Annual Simulation Symposium (SS'02)
1080-241X/02 $17.00 © 2002 IEEE

of one single model, by offering the modeler a choice of representation for each entity in the model. Like JSIM and SimJava, DESMO-J is embeddable into applets.

Table 1 summarizes the main features of several existing discrete-event simulation support libraries, including the ones mentioned above.

Table 1. Simulation Support Libraries

Library	Modeling view	Synchronization	Language
Simkit [2]	event or process-oriented	sequential	Java
JADIS [17]	event-oriented	sequential	Java
JSIM [14]	process interaction	distributed sequential	Java
SimJava [18]	event-oriented	multi-threaded seq	Java
DESMO-J [10]	event + process-oriented	multi-threaded seq	Java
Jwarp [1]	event-oriented	optimistic	Java
SPEEDES [12,22]	event-oriented	optimistic	C++
CPSim [15]	event-oriented	conservative	C

This paper introduces SPaDES/Java, a PDES library that supports the process-oriented modeling view and implemented in the Java language. SPaDES/Java provides transparency of the underlying event synchronization implementation and relieve the programmer from parallel programming, thereby enabling the simulationist to concentrate on modeling the problem at hand. Section 2 discusses the design and implementation of SPaDES/Java. Section 3 presents a chronology of performance optimizations to reduce the overhead and the cost of parallel simulation. Our concluding remarks are in Section 4.

2. SPaDES/Java - Design and Implementation

The main objective of SPaDES/Java (Structured Parallel Discrete-Event Simulation in Java) is to provide a simulation support tool to alleviate the simulationists' burden of having to implement the event synchronization details, and parallelization of the simulation through parallel programming. A simulationist should concentrate on modeling the problem and analyzing the simulation results, which is the main objective of applying parallel simulation.

2.1 Modeling View

SPaDES/Java adopts the process-oriented modeling view, whereby entities in the real world are mapped to a set of processes each encapsulating its own state and behaviour in the conceptual model, and processes interact with one another through message passing. Furthermore, it is necessary for a process-oriented model to be mapped to an underlying operational model that is suitable for parallelization. The operational model of SPaDES/Java is based on the Virtual Time Paradigm [20]. Figure 1 illustrates the SPaDES/Java modeling view.

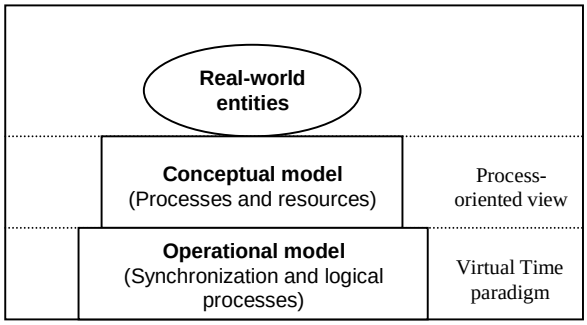

Figure 1. Process-oriented Modeling View in SPaDES/Java

2.2 Modeling Support

Processes in the conceptual model are categorized into *permanent* and *temporary* processes. A permanent process, also known as a resource, exists throughout the simulation duration. A temporary process (called process) is created and destroyed at any point during the simulation. This corresponds to modeling the arrival and the service completed for a process. In the operational model, resources are modeled as logical processes (LPs) and processes are modeled as time-stamped messages passed between LPs.

2.2.1 Processes and Resources

A SPaDES/Java process can be in any one of five states, namely *active*, *blocked*, *pending*, *non-existent* or *holding*, as shown in Figure 2.

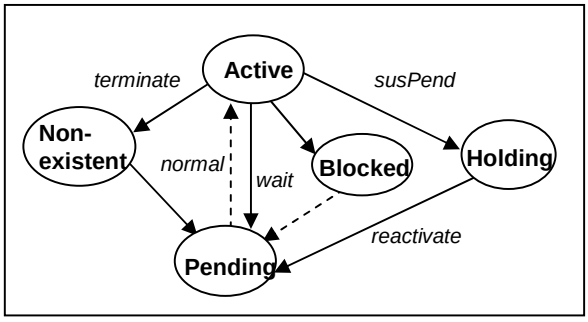

Figure 2: State Transition for a SPaDES/Java Process

When a process is first created, it is *non-existent* relative to the simulation until it is activated, causing it to enter the *pending* state. This process is timestamped and scheduled for execution in an event list. When the simulation clock advances to the timestamp of the process, the process enters the *active* state and begins executing its routine. During execution, it may request

service from a simulation entity called a resource. When a resource is unavailable, the process halts execution temporarily, causing it to be *blocked*. A process is also permitted to suspend from being executed, causing it to change into the *holding* state, and remain suspended till another process reactivates it. A process finally returns to the *non-existent* state when it has completed its thread of execution and leaves the system.

Resources are permanent entities providing services to processes. Each resource comprises of a queue, which maintains the processes requesting service from the resource. When a resource is busy, an arriving process is subsequently queued for service. Each resource is a collection of basic functional units, known as service units. The presence of multiple service units in a resource allows more than one process to be serviced in parallel. The default queuing discipline in SPaDES/Java simulations is *First-Come-First-Serve* (FCFS), where processes are ordered according to their timestamp values. Other queuing disciplines such as *Last-In-First-Out* (LIFO) are supported. In addition, a user can define a general *priority* queue [21] by overriding the generic *comparesWith* method of the process object class.

2.2.2 Parallel Simulator Template

A simulator may be programmed using a general template that extends from the SPaDES/Java library. The template consists of four main parts: definitions of the processes used in the simulation, process routines that define the simulation logic, code for initializing and starting the simulator, and message abstraction and reconstruction routine for the case of parallel simulation. Figure 3 shows the template with an example of the *M/M/1* (a single server queuing system) simulator program.

2.3 Library Implementation

2.3.1 Class Hierarchy

SPaDES/Java consists of four main simulation classes and a number of simulation support classes, as shown in Figure 4. *SimObject* is the base class for the simulation entities, namely the processes and resources, which are modeled as the *SProcess* and *Resource* classes respectively. All SPaDES/Java classes are inherited from Java's built-in *Thread* API [7] library.

```
1    // Executive instance
2    // import SPaDES/Java library and other packages to Java's library resources
3    import spades_Java.*;
4
5    // simulation initialization
6    public class simulation-kernel-name extends Executive {
7            << create resources >>
8            << create processes >>
9            << initialize future event list >>
10           << configure layout of simulator >>
11
12
13
14           public static void main(String args[]) {
15                  << create an instance, say, E, of this class >>
16                  E.initialize(args.length,args);
17                  E.startSimulation([duration]);
18           }
19   }
20
21   // Process type instance
22   // process class definition and routine definitions
23   class process_type extends SProcess {
24           << Executive instance type >>
25           public process_type(Executive E)
26           {       << construction of process_type instance >>}
27
28           public void execute()
29           {
30                  . . . .
31                  . . . .
32                  . . . .
33                  << process routine statements >>
34                  . . . .
35                  . . . .
36                  . . . .
37                  . . . .
38                  . . . .
39                  . . . .
40           }
41   }
```

```
1    // Executive instance
2    // import SPaDES/Java library and other packages to Java's library resources
3    import spades_Java.*;
4
5    // simulation initialization
6    public class MM1 extends Executive {
7         Resource server;
8         public void init() {
9              server = new Resource("Server", 1, 1);
10             mapProcess(server);
11             activate(job, 0);
12        }
13
14        public static void main(String[] args)  {
15             MM1 mm1 = new MM1();
16             mm1.initialize(args.length, args);
17             mm1.startSimulation(10000);
18        }
19   }
20
21   // Process type instance
22   // process class definition and routine definitions
23   class Job extends SProcess {
24        MM1 mm1;
25        public Job(MM1 m)
26        {    mm1 = m;}
27
28        public void execute()
29        {
30             switch(phase) {
31                case 1: {
32                   Job job = new Job(mm1);
33                   activate(job, 1+exponential(10));
34                   work(1+exponential(1), mm1.server, 1);
35                   phase = 2;
36                }
37                case 2:
38                {  terminate();}
39             }
40        }
41   }
```

Figure 3. SPaDES/Java Programming Template

The *RandNoGenerator* class supports the generation of random variate values using the linear congruential (LCG) algorithm [11]. The *NullMsg* (models a null message in the conservative protocol) and the *SProcess* classes both implement the *Entry* class in the Java-Jini/JavaSpaces library [9], which provides the message coordination control for multi-processor simulation.

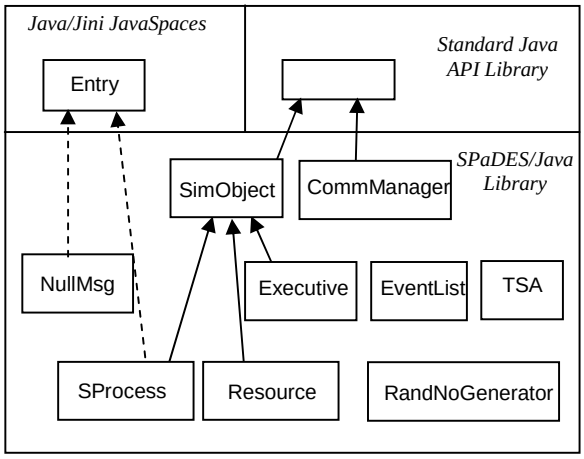

Figure 4: SPaDES/Java Class Hierarchy

2.3.2 Event Synchronization Protocol

SPaDES/Java supports two modes of execution, sequential simulation and parallel simulation based on the conservative *null message* protocol [5].

The sequential execution protocol comprises of a global Future Event List (FEL) that orders processes based on their timestamp values, inherently maintaining event causality. Each simulation pass involves executing the routine of the process at the head of the FEL, inserting the process into the relevant resource's queue, and executing blocked processes at that resource where relevant. Null message synchronization support for parallel simulation is discussed in Section 3.3.

2.3.3 Process Primitives

Six simulation primitives as listed in Table 2 allow the simulationist to model the state-transition of a SPaDES/Java process.

The simulationist can manipulate these primitives to simulate the behaviour of each process during the simulation, by overriding the *execute( )* method of the *SProcess* class. The detailed syntax of SPaDES/Java is in the Appendix.

Table 2. SPaDES/Java Process Primitives

Primitive	Parameter(s)	Functionality
activate	process, time	Schedules another process for execution at a future timestamp value.
reactivate	process, time	Schedules another process that was previously in the *holding* state into the event list, for execution at a future timestamp value.
work	resource, time, units	Requests for service from a resource for a specified time duration.
wait	time	Models the passage of simulation time.
susPend	-	Suspends itself from its routine of execution till it is reactivated by another process.
terminate	-	Models departure from the simulation system.

2.3.4 Time and Space Instrumentation

Teo et. al developed a methodology based on partial order set theory to study the memory required by a simulation [24]. As proposed in this methodology, the memory required for a simulator is divided into three main components:

- M^{prob} - The memory that is required to model the states of the real world system. This is measured by summing up the maximum queue lengths for each LP, $\max(Q_i)$, i.e. $M^{prob} = \sum_{i=1}^{LP} \max(Q_i)$.
- M^{ord} - The memory overhead of the selected event ordering. This is measured by summing up the maximum event list lengths for each LP, $\max(EL_i)$, i.e. $M^{ord} = \sum_{i=1}^{LP} \max(EL_i)$.
- M^{sync} - The memory cost in implementing the synchronization protocol that supports the selected event ordering. This is measured by summing up the maximum lengths of the null message buffer for each LP, $\max(NM_i)$, i.e. $M^{sync} = \sum_{i=1}^{LP} \max(NM_i)$.

To support the above instrumentation for different event orderings, a time and space analyzer (TSA) has been integrated into our library. The *TSA* class comprises of a monitor over the simulator, keeping track of the status of processes and the maximum lengths of the LPs' queues, event lists and null message buffers throughout the entire simulation. In addition, the memory requirements for known event orderings such as partial, time-interval, timestamp and total event orderings can be derived in a simulation run.

3. Performance Optimizations

The maintenance of interacting processes in a simulation influences its runtime efficiency. In parallel simulation, event synchronization and inter-processor communication overheads amplify the inefficiency. This section discusses a number of optimizations to reduce these overheads.

3.1 Event Lists

SPaDES/Java originally used vectors, which are array-based expandable data structures provided in the Java API library, to implement the event lists. Vectors are convenient tools for implementing priority queues such as the event lists. Insertion and deletion operations on a vector-implemented event list, however, cost *O(n)* time complexity [3] each, due to the need to translate processes to adjacent positions in the event list. Using such an inefficient queuing structure for the scheduling of events result in performance bottleneck, particularly for the global FEL in sequential simulation.

We re-implement the event lists using *binary minheaps* [3]. Inserting into and deleting a process from a heap-based event list costs only *O(lg n)*, because in a heap comprising of *n* processes, a maximum of only $\lfloor lg\ n \rfloor$ processes need to be migrated whenever insertion or removal of another process occurs.

3.2 Multi-Threading

SPaDES/Java processes and resources can be implemented as independent threads of execution. However, experiments have shown that excessive threads execution can lead to sub-optimal performance, due to the overhead of thread context switching [21]. An extreme option is to implement the simulation entities as ordinary objects, without any potential for concurrent execution. This compels sequential execution of all events, which defeats the objective of parallel simulation.

SPaDES/Java eliminates multi-threading for processes, but keeps the LPs as mutually exclusive threads. This compromise approach is based on the knowledge that the number of LPs in a simulation is statically determined because they exist permanently throughout the entire run, while the number of processes varies dynamically at runtime. Implementing the processes as objects prevents thread context switch overhead to be a dominant problem especially in open system [16] simulation where system throughput is less than inter-arrival rates, i.e. the population of processes in the system expands exponentially as the simulation progresses.

3.3 Null Message Synchronization

Other than ensuring the progress of the parallel simulation, null messages are merely overhead and do not contribute directly to the simulation. We define the *null message ratio* (NMR) of a simulation as the total number of null messages divided by the total event traffic. The original synchronization protocol adopted in the SPaDES/Java parallel execution kernel was the Chandy/Misra/Byrant (CMB) [5] null message algorithm, whereby LPs transmit null messages along their output channels at the end of every simulation pass. This is extremely costly in terms of NMR.

We first incorporated Wood-Turner's carrier-null scheme [25] to reduce null message overhead in problems with cyclic topologies. Tay et. al proposed a *flushing algorithm* [23], which attempts to resolve the problem of an LP sending null messages with repeated timestamp values along an output channel. We augment this algorithm with the demand-driven null messaging mechanism [4] to produce a hybrid null message algorithm applied on each output channel, *C*, of an LP, as shown in Figure 5.

```
DEMAND-DRIVEN-FLUSHING(C)
    N = remove most recently queued null message in C;
    for all messages m in C
        do
            if m is a null message
                then C = c - {m};
        end for
    if null-msg-requested
        then  send N to the LP at the other end of C;
    else
        C = C U {N};
```

Figure 5. Demand-driven Flushing

Now, rather than sending null messages on every simulation pass, an LP will only transmit null messages if requested by another LP. Furthermore, the output channel includes a flushing mechanism so that only the most updated null messages are sent.

3.4 Inter-Processor Communications

The first implementation of SPaDES/Java relies on Java's Remote Method Invocation (RMI) [19] to serialize and transport processes and null messages between processors. Being a subclass of the *SimObject* and *Thread* classes, the *SProcess* object encapsulates an enormous amount of data references, including global and user-defined data. Consequently, the serialization of processes becomes a significant performance bottleneck. The logical size of a process can be shrunk before it is propagated across the RMI channel to reduce communication cost. In SPaDES/Java, the

global information held in a process is discarded, and user-defined data only is transmitted to the destination processor. Once it arrives, the original process is reconstructed. Two methods, *extract* and *reconstruct*, are provided in the inter-processor communications module to perform the above-mentioned deflation and reconstruction of processes respectively. They are to be overridden by the simulationist in the simulator program.

Next, we attempt to centralize the coordination of message transmissions between remote processors by replacing the RMI protocol in the communications module with a single JavaSpaces [9]. JavaSpaces is a special service of Java Jini to support persistent data storage, and facilitated by a shared memory tuple space. The abstract operations on a space facilitate the transmission of null messages between LPs mapped to different processors. Transmission of processes and null messages now go through this shared memory.

The major steps involved, for LP_i, in the initialization phase and the transmission of messages between remote LPs are as follows:

Initialization:

- Write a *SProcess* entry template into the JavaSpaces to invoke the *notify* operation, indicating that LP_i is ready to accept event messages from remote LPs. This template must specify, in its `sender` attribute, the IDs of the LPs linked to the input channels to LP_i.
- Write a *NullMsg* entry template into the JavaSpaces, with its attributes `request=true` and `sender=i`. This is to invoke the *notify* operation on the space, indicating that LP_i is ready to accept null message requests from remote LPs.
- Start the simulation routine for LP_i.

Transmission of Process:

- When sending a *SProcess* object, *P*, to a remote LP_j, deflate *P* and set `P.sender=i` and `P.receiver=j`. `Write` *P* into the JavaSpaces.

Transmission of Null Message:

- When requesting a null message from a remote LP_j, `write` a *NullMsg* entry into the space, with its `sender=j`.

The JavaSpaces handles messages as follows:

Process:

- If an LP `write`s in a *SProcess* object, *P*, search for a *SProcess* template *T* such that `T.sender` ∈ `P.sender` and `T.receiver` = `P.receiver`.
- Send *P* to the LP whose ID is indicated by `T.receiver`.

Null Message:

- If a null message request, *R*, has been written into space, search for a request template, *T*, with `T.sender=R.sender`.
- Inform the LP, which wrote in *T*, of the this request *R*.
- When a null message, *N*, has been written in by an LP_k, siphon *N* to the relevant LPs that had requested for null messages from LP_k.

Figure 6 illustrates the coordination of a process transmission by a JavaSpaces running on a separate processor.

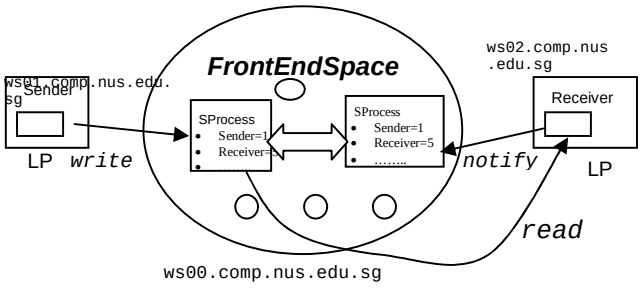

Figure 6. Interprocessor Communication using JavaSpaces

3.5 Performance and Space Analysis

We conducted a series of experiments using a cluster of PCs (Pentium II, 400 MHz, 256MB of memory) to study the effect of optimizations on performance. Two benchmark programs are used. A linear pipeline (n, ρ) representing an open system of n servers connected in series with traffic intensity of ρ. PHOLD (nxn,m) is a closed, strongly connected system with a network of nxn nodes, and with m jobs per node at the start of the simulation [6]. We observe that implementing the event lists as minheaps improves runtime by an average of 30%, and reduction in the level of multi-threading improves runtime (in seconds) by a further 7%. Table 3 and Table 4 summarize the performance improvements from these accumulative optimizations on selected parameters (problem size, workload, number of processors).

Table 3. Null Message Optimizations - One processor

Synchronization Protocol	Pipeline(16, 0.8)		PHOLD (16x16, 16)	
	NMR	Runtime	NMR	Runtime
CMB	0.94	3770	0.99	13990
+ carrier null	0.94	3775	0.69	5651
+ flushing	0.70	2917	0.57	4580
+ demand-driven	0.61	1989	0.44	1563

Table 4. PHOLD (16x16, 16) – Reducing Inter-processor Communication

#procs	Runtime		NMR	
	RMI	JavaSpaces	RMI	JavaSpaces
1	1563	1572	0.44	0.44
4	1288	899	0.44	0.34
8	1006	487	0.44	0.29

The step-wise enhancements due to the null message protocol and the centralization of inter-processor communication reduce the runtimes by more than half, NMR reduces from more than 0.9 to 0.29, and event rate is doubled using eight processors. Table 5 presents a profile of memory usage for both benchmarks.

Table 5. Profile of Memory Usage

Space Usage	Pipeline(16, ρ)				PHOLD (16x16, m)		
	ρ				M		
	0.2	0.4	0.6	0.8	1	8	16
M^{prob}	101	192	320	740	256	2048	4096
M^{ord}	51	52	54	56			
M^{sync}	335	341	348	352	665	651	638
M (total)	487	585	722	1148	921	2699	4734

We observe that for a closed system such as the PHOLD, the memory cost due to null message synchronization reduces when message density (m) increases. The parallel simulation memory overhead for PHOLD(16x16,16) is less than 10% of the total amount of memory required to run the simulation.

4. Conclusions

We have presented a PDES library, SPaDES/Java, that is based on the process-oriented modeling view, and whose objective is to provide the simulationist with an abstract modeling environment for distributed simulation. The modeling primitives provided hide the underlying event synchronization mechanism, enabling the simulationist to focus on using simulation as a tool to study systems and the logical correctness of the problem, instead of causal correctness.

SPaDES/Java adopts the demand-driven flushing null message algorithm, on top of the carrier-null mechanism, to reduce the overhead of event synchronization. Inter-processor communication is facilitated by a JavaSpaces: a component service of the Java Jini. A shared persistent memory helps to further reduce null message overhead and improve simulation runtime. A time and space analyzer integrated into SPaDES/Java provides instrumentation to profile the space overhead in supporting parallel simulation.

Acknowledgements

This research is supported by the Ministry of Education (Singapore) and PSA Corporation under grants R-252-000-020-112 and R-252-000-020-490. SPaDES/Java can be downloaded for educational purpose at http://www.comp.nus.edu.sg/~pasta/spades-java/spadesJava.html.

References

[1] P. Bizarro, L. M. Silva and J. G. Silva, "*Jwarp: A Java Library for Parallel Discrete-Event Simulations*", Poster Paper at ACM Workshop on Java for High-Performance Network Computing, 1998.

[2] A.H. Buss and K.A. Stork, *"Discrete Event Simulation on the World Wide Web Using Java"*, Proceedings of the Winter Simulation Conference, pp. 780-785, 1996.

[3] T.H. Cormen, C.A. Leiserson and R.L. Rivest, *Introduction to Algorithms*, McGraw Hill, 1989.

[4] P. Fouliras, *"A Null-Event Demand-Driven Parallel Simulation Algorithm for SIMD Computers"*, Technical Report QMW-DCS-1994-669, Queen Mary and Westfield College, Department of Computer Science, 1994.

[5] R.M. Fujimoto, *"Parallel Discrete Event Simulation"*, Communications of the ACM, vol. 33, pp. 31-52, 1990.

[6] R.M. Fujimoto, *"Performance of Time Warp under Synthetic Workloads"*, Proceedings of the SCS Multiconference on Distributed Simulation, pp. 23-28, 1990.

[7] G. Hilderink, J. Broenink, W. Vervoort and A. Bakkers, *"Communicating Java Threads"*, Proceedings of the WoTUG-20 Parallel Programming and Java Conference, 1997.

[8] F. W. Howell, P. E. Heywood and R. N. Ibbett, *"Hase: A flexible toolset for computer architects"*, The Computer Journal, vol. 38, pp. 755-764, 1995.

[9] S. Hupfer, "The Nuts and Bolts of Compiling and Running JavaSpaces Programs", Java Developer Connection, Sun Microsystems, Inc., 2000.

[10] T. Lechler and B. Page, "DESMO-J: An Object Oriented Discrete Simulation Framework in Java", Proceedings of the European Simulation Symposium '99, 1999.

[11] D. H. Lehmer, *"Mathematical methods in large-scale computing units"*, Proceedings of the 2nd Symposium on Large Scale Digital Calculating Machinery, pp. 141-146, 1949.

[12] T. McGuinness, *"Executing Multiple Parallel Applications Using the SPEEDES Communications Library"*, Proceedings of the Department of Defense

High Performance Computing Modernization Program Users Group Conference, 2001.

[13] J.A. Miller, R. Nair, Z. Zhang and H. Zhao, "*JSIM: A Java-Based Simulation and Animation Environment*," Proceedings of the 30th Annual Simulation Symposium, pp. 31-42, 1997.

[14] R.S. Nair, J.A. Miller and Z. Zhang, *"A Java-based Query Driven Simulation Environment"*, Proceedings of the Winter Simulation Conference, pp. 786-793, 1996.

[15] D.M. Nicol, "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations", Journal of the Association for Computing Machinery, pp. 304-333, 1993.

[16] D. M. Nicol, "Parallel discrete-event simulation of FCFS stochastic queueing networks", SIGPLAN Notice, vol. 23, pp. 124-137, 1988.

[17] S. Narayanan, N.L. Schneider, C. Patel, T.M. Caricco, J. DiPasquale and N. Reddy, *"An Object-based Architecture for Developing Interactive Simulations Using Java"*, Proceedings of the 1997 Simulation Conference, vol. 69, pp. 153-171, 1997.

[18] E.H. Page, R.L. Moose and S.P. Griffin, *"Web-based Simulation in SimJava Using Remote Method Invocation"*, Proceedings of the Winter Simulation Conference, pp. 468-474, 1997.

[19] E.H. Page, R.L. Moose and S.P. Grifflin, *"Implementation Notes for a Distributed SimJava"*, MITRE Technical Report, The MITRE Corporation, 1997.

[20] R. Righter and J.C. Walrand, *"Distributed Simulation of Discrete-event Systems"*, Proceedings of the IEEE, vol. 77, no. 1, pp. 99-113, 1989.

[21] A. Silberschatz and P. Galvin, *"Operating System Concepts, 5th Edition"*, Chapter 8, Addison Wesley, 1998.

[22] J. Steinman, "SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-event Simulation", Proceedings of the SCS MultiConference, pp. 95-103, 1991.

[23] S. C. Tay, Parallel Simulation Algorithm and Performance Analysis, PhD Thesis, Department of Computer Science, National University of Singapore, 1998.

[24] Y. M. Teo, B. S. S. Onggo and S. C. Tay, *"Effect of Event Orderings on Memory Requirement in Parallel Simulation"*, Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 41-48, IEEE Computer Society Press, USA, 2001.

[25] K. R. Wood and S. J. Turner, *"A Generalized Carrier-null Method for Conservative Parallel Simulation"*, Proceedings of the 8th Workshop on Parallel and Distributed Simulation, Edinburgh, UK, IEEE Computer Society Press, pp. 50-57, 1994.

Appendix

The syntax of SPaDES/Java in EBNF form is shown below, using the following metasymbols:

::= connects the left hand side (LHS) and right hand side (RHS) of a rule.
| separates alternative RHS items, as an ``or" operator.
[ ] encloses an optional RHS item.
{ } encloses a repeated RHS item that can appear 0 or more times.

1. Simulation processes

```
simulation_process ::= resource | process
```

2. Resource initialization

```
resource     ::= resource_identifier( name, unit )
resource_identifier ::= string
name  ::= string
unit  ::= integer
```

3. Process initialization

```
process      ::= process_identifier( size, [priority] )
process_identifier  ::= string
priority  ::= integer
```

4. Simulation primitives

```
primitive ::= self_primitive | others_primitive
self_primitive  ::= work( resource, time, unit )
                  | wait( time ) | susPend()
                  | terminate()
others_primitive::= activate( process, time )
                  | reactivate( process, time )
time  ::= double | stat_functions
```

5. Time and statistical functions

```
stat_functions ::= normal(mean, stddev)
                 | gamma(alpha, beta)
                 | exponential(mean) | chi_square(n)
                 | weibull(alpha, beta) | uniform()
                 | binomial(p, n)
                 | neg_binomial(p, n) | geometric(p)
                 | poisson(mean) | triang(mode)
                 | erlang(n) | laplace()
mean   ::= double
mode   ::= double
stddev ::= double
alpha  ::= double
beta   ::= double
p   ::= double
n   ::= integer
```

6. Process routines

```
process_body ::= switch(phase) '{'
                 { case phase ':' {others_primitive}
                                   self_primitive
                                   [phase = value] }
                               }'
value   ::= integer
```

7. Statistical referencing

```
resource_statistics ::= arrivals() | departures()
                      | utilization() | waiting_time()
                      | queueLength() | response()
                      | maxQueueLength()
```