Proceedings of the 2004 Winter Simulation Conference
R .G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

EFFICIENT PROCESS INTERACTION SIMULATION IN JAVA: IMPLEMENTING CO-ROUTINES WITHIN A SINGLE JAVA THREAD

Richard M. Weatherly
Ernest H. Page

The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102, U.S.A.

ABSTRACT

The genesis of a research effort to develop a Java-based process-oriented simulation framework is described. A key enabler to the framework is an efficient co-routine mechanism implemented within the context of a single Java thread. A design for such a co-routine mechanism is described and some initial results of an implementation within the IBM Jikes Reference Virtual Machine are given.

1 INTRODUCTION

The history of languages and environments for discrete event simulation is a rich one (Nance 1993). Debates on the suitability of general-purpose languages for the simulation task gave rise to the birth of special-purpose simulation languages in the 1960s and 1970s. These languages proliferated and flourished during the 1970s and 1980s. During this period, the notion of holistic programming *environments* emerged in the software engineering community with the Ada Programming Support Environment (Oberndorf 1988), and a shift from language-focus to environment-focus also occurred in the simulation community (Balci 1986; Balci et al. 1990) and has continued for the past two decades. That's at least one way to look at our history as a community. As Nance points out, though, histories are rarely quite as tidy as historians would like them to be.

In reality, the application of general-purpose programming languages to the simulation task has persisted since the earliest days of simulation. Simulation packages based on Pascal (Barnett 1986; Malloy and Sofa 1986; Seila 1986; Uyeno 1980), Ada (Samuels and Speigel 1987), Modula-2 (L'Ecuyer and Giroux 1987), C and C++ (Bagrodia 1991; Fishwick 1992; Lomow and Baezner 1990; Schwetman 1986 and 1994), Smalltalk (Drolet at al. 1991; Knapp 1986) and Prolog (Adelsberger 1984; Elmaghraby 1988; Le 1993), for example, have all appeared. Many of these packages were developed within the academic community as pedagogical aids. But several of the commercial simulation tools you see in the vendor spaces at the Winter Simulation Conference (WSC) are based on general-purpose languages. Within the military simulation domain, most of the larger systems contain a mixture of languages, as one might reasonably expect. Our informal survey of systems currently under development suggests that the use of general-purpose languages is dominating the use of simulation languages.

With the appearance of the Java programming language circa 1995, it wasn't long before Java-based simulation packages also began to appear. A large number of Java-based simulation packages have been developed, including: SimKit (Buss 2000, 2001), JSIM (Miller, Ge and Tao 1998), J-Sim (Tyan 2002), DESMO-J (2004), Silk (Kilgore 2000), SimJava (Howell and McNab 1998; Page, Moose and Griffin 1997), PsimJ (Garrido 2001; Garrido and Im 2004) and SSJ (L'Ecuyer, Meliani and Vacher 2002) to name only a few.

In this article we discuss a MITRE Sponsored Research project to develop a Java-based simulation framework. The primary goal for the project is not to create *yet-another-Java-based-simulation-language*. The primary goal of the project is to develop technology that supports the construction of large-scale simulation *systems* by small teams. This primary goal is enabled by a number of secondary objectives:

1. Apply and evaluate open-source technologies to develop the simulation framework.
2. Create a simulation-specific application programmer's interface that supports the construction of efficient simulation while retaining access to all existing Java language features.
3. Develop an efficient co-routine mechanism for Java.
4. Shepherd the integration the framework into the Java language standard through the Java Community Process, and
5. Apply the framework to realistic problems within the military and civil domains.

The focus of this article is on the third objective—developing an efficient co-routine mechanism for Java. The remainder of the article is organized as follows. Section 2 provides background and motivation for working within Java. Section 3 presents an initial study of Java thread performance. Section 4 describes our approach and design for a Java co-routine mechanism. Our initial implementation within the IBM Jikes Reference Virtual Machine, and some preliminary performance results are discussed in Section 5. Conclusions and directions for future work appear in Section 6.

2 MOTIVATION

A primary stimulus for this research was the experience of the first author as Chief Engineer for the Joint Simulation System (JSIMS) (see USJFCOM 2004) during the period 2000-2003. This experience revealed a number of challenges in the construction of large-scale simulation *systems*:

- First, traditional systems engineering and government acquisition processes are ineffective in this domain—staging requirements definition, then design, construction, and system integration. The inherent flexibility of simulation invites requirement inflation, especially when the cost of a requirement is not considered in the overall design.
- Second, large developer teams are expensive and unresponsive—teams of a hundred developers adjust slowly to changes in requirements or design.
- Third, the current infrastructure for building distributed simulations is too complex for the average developer.

One approach to resolving these problems uses small, responsive teams of developers to create significant simulations. Not only are small teams less expensive and more agile, but their speed and agility allows iterative development of simulations. Thus requirements can be refined as their cost becomes evident. But how can we enable small teams to build very large simulation systems quickly?

Large-scale simulation systems typically involve a great deal more than a core simulation. Such simulation systems require graphical user interfaces, database connections, XML and other data parsers, map displays and other visualization, and communication with other computers. Access to these ancillary functions is not provided to any real degree in extant discrete event simulation programming languages. Most of the large-scale military and civil simulation systems, therefore, are constructed using general-purpose programming languages. And with the rising popularity of Java, many of these systems are being built in Java.

Java, like other "mainstream" programming languages, has a variety of development tools, including integrated development environments (compilers, debuggers, build tools), and code profiling and optimization tools. The range of tools available to Java programmers exceeds that available to programmers in most simulation languages. Theoretically, at least, these tools make it easier to construct large systems of any sort—including simulations. Java literature and training are extensive. Java is the dominant teaching language for Computer Science in the U.S., which ensures a steady stream of graduates comfortable with the Java platform.

However, Java tends to scale poorly as a simulation language, especially in the context of the process interaction world view. The primary reason for this is a simple one: within standard Java Virtual Machines, Java threads are tied to underlying operating system threads. In Java-based process-oriented simulations, each object (or logical process) is typically assigned its own thread. On commodity Windows and Linux platforms, this limits the number of logical processes to a few thousand (see Section 3). Large-scale military and civil simulations, on the other hand, can require hundreds of thousands of logical processes.

One way to support very large numbers of logical process is to use multiple physical processors (i.e. parallel or distributed simulation). Another approach to provide an efficient co-routine mechanism for Java—one that does not rely on operating system threads.

3 A QUICK LOOK AT JAVA THREAD PERFORMANCE

To establish a baseline for the number of concurrent threads supportable in the Sun Microsystems Java Virtual Machine (JVM), we constructed a simple benchmark that creates logical processes until a `java.lang.Out Of Memory Error: unable to create new native thread` error is generated. The benchmark creates a "main" entity that advances time in equal steps. At each timestep, the main entity creates a new instance of a "null" entity. The null entities each execute an indefinite wait.

The benchmark was run on two platforms. The first is a Dell D600 (1.3 Ghz processor, 1 Gb RAM) running Windows XP and Java HotSpot(TM) Client VM 1.4.2_03-b02. The second is a Dell 4300 (two 2-GHz processors, 2 GB of RAM) running Linux kernel 2.4.20-24.9smp (Redhat 9) and Java HotSpot(TM) Client VM 1.4.2_02-b03 (a slightly older version of the JVM).

We examined the effects of varying the thread stack size (-Xss*n* argument to the JVM) and the memory allocation area (-Xmx*n* argument to the JVM). In each case, when varying one of these parameters, the other was left at its default value. The results, which appear in Table 1, indicate a limitation of, roughly, 8000 threaded entities for Windows and 4000 for Linux.

Table 1: Upper Bounds for Thread Creation in Sun JVM

Platform	JVM argument	#entities created
Dell, Windows	-Xss5k	7240
	-Xss10k	7240
	-Xss20k	7240
	-Xss50k	7240
	-Xss400k	1847
	-Xss10m	176
	-Xmx8m	7386
	-Xmx16m	7386
	-Xmx32m	7366
	-Xmx64m	7240
	-Xmx128m	6987
	-Xmx256m	6481
	-Xmx640m	4963
Dell, Linux	-Xss5k	Stack size too small
	-Xss96k	Stack overflow error
	-Xss100k	Stack overflow error
	-Xss200k	3459
	-Xss400k	3459
	-Xss10m	178
	-Xmx32m	3459
	-Xmx100m	3459
	-Xmx400m	2985

4 DESIGN OF A CO-ROUTINE MECHANISM FOR JAVA

Within our framework, recently dubbed *Tortuga*, interacting processes are created by extending class `Entity`, creating instances of those extended classes, and registering those instances with the simulation executive. Each entity has an `agenda()` method. The agenda is where simulation developers place code to model the proactive aspects of entity behavior. The Tortuga framework provides the usual set of services to allow an entity to sense the simulated environment and control the advance of simulation time.

The simple entity depicted in Figure 1 illustrates the major points in the control of interacting processes. When, at time *t*, an instance of class `MyEntity` is registered with the simulation executive, the executive invokes the `agenda()` method. Computation at lines 3 and 4 takes place at time *t*. In line 4, the entity requests a delay of 5.0 simulation time units using framework method `waitForTime()`. Method `waitForTime()` transfers control to the simulation executive. After simulation time advances 5.0 time units, the executive transfers control back to line 5 where code at that point experiences a simulation time value *t* + 5.0. When the `agenda()` code is complete, control returns to the simulation executive. The executive then excludes the entity from future scheduling consideration and makes it available for garbage collection.

```
1   class MyEntity extends class Entity {
2    public void agenda() {
3      // time is now t
4      waitForTime(5.0);
5      // time is now t + 5.0
8    }
9   }
```

Figure 1: Simple Entity

The interaction between the simulation executive and the entities that comprise a simulation can be generalized to four operations. These operations are defined below:

- **Elaborate(p)** - Invocation of the `agenda()` method of entity *p*. This is done by the executive after the entity is registered.
- **Yield to Executive** – Used by an entity to transfer control to the simulation executive when the entity needs to wait for the passage of time or some other change in the environment.
- **Yield to Process(p)** – Used by the simulation executive to transfer control from the simulation executive to entity *p* when some specified conditions are met.
- **Terminate(p)** – Used by the simulation executive or by an entity to remove entity *p* from further consideration by the simulation executive. This operation takes place implicitly after the `agenda()` method is complete.

All Java-based process oriented frameworks have operations similar to those above. In all cases some sort of thread manipulation is needed to achieve the co-routine semantics required by `Yield to Executive` and `Yield to Process`. Typically, a unique Java thread is assigned to the simulation executive and to each of the entities. The thread manipulation needed to implement the four operations are:

- **Elaborate(p)** – Create a new thread for entity *p*. Let the new entity thread begin execution of the entity `agenda()` method. Suspend the executive thread while this is taking place.
- **Yield to Executive** – Suspend the thread of the entity that invoked the `Yield to Executive` operation. Awaken the simulation executive thread.
- **Yield to Process(p)** – Suspend the simulation executive thread and awaken the thread for entity *p*.
- **Terminate(p)** – Awaken the simulation executive thread if needed and destroy the thread for entity *p*.

Given an understanding of how threads are used to coordinate the interaction between the simulation executive and a group of entities, we can make the following observations. Threads are a more capable, and unfortunately more complex and expensive, control mechanism than is really needed to support the four operations defined above. Specifically, there is no need for *true parallelism* as provided by threads. Additionally, as used by the four operations, all threads are awakened explicitly by executive or entity code. One of the powerful, and again expensive, features of threads is their ability to link awakening to asynchronous events such as hardware interrupts. But this capability is not needed to implement the four operations.

Note also that there are *n + 1* threads needed to implement a simulation with *n* entities. Effectively, only one thread is actually executing at any given time even if more than one physical processor is available. This means that there will be *n + 1* execution stacks in the JVM but only one of them will be changing at a time.

These observations led us to investigate a single thread approach. Given that it is not possible in "pure" Java to implement the four operations above without using multiple threads, the research goal became the isolation, definition, and implementation of the smallest set of JVM extensions needed to support co-routines. The experimental vehicle for the investigation is the IBM Jikes Reference Virtual Machine (RVM) (2004). The IBM RVM was chosen because it has good performance, is written in Java, was designed for experimentation, is available as open source, and is self hosting. This last point is significant as Jikes may be the only JVM written in Java that runs on itself and does not require a second JVM.

The essence of the project is to convert a single thread with its single method invocation stack into a cactus stack (see Sardesi, McLaughlin and Dasgupta 1998) without adversely effecting the operation of the RVM. The left side of Figure 2 shows the normal situation where each thread has a conventional, linear frame stack. As one method invokes another, a new frame is pushed onto the top of the stack and the frame pointer (FP) is advanced upward to keep track of the currently executing method. Each frame contains references to the method code for that invocation, the saved instruction pointed for the method that invoked it (the one beneath it), a pointer to the stack frame beneath it, and other saved state such as hardware registers.

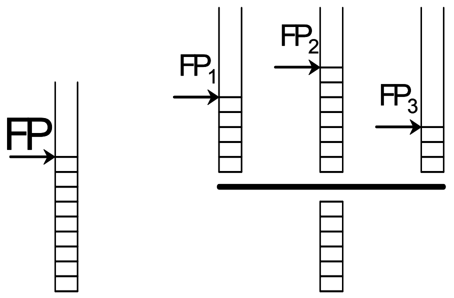

Figure 2: Conventional and Cactus Stacks

In a cactus stack, as seen on the right of Figure 2, there are multiple *arms* at a single level and a common base. The cactus stack data structure allows arbitrary levels of branching but only one level is needed for this design. Each arm has its own frame pointer, FP_i, and is used to manage the execution of a single entity. The common base contains the frames created by the runtime environment to start the thread and those of the simulation executive.

The four operations needed to coordinate the interaction between the simulation executive and the entities, defined above, also govern the major cactus stack actions. When the simulation executive uses Elaborate(p) to create a new entity, a new arm is added to the stack. Symmetrically, operation Terminate(p) removes the arm on which entity *p* was executing. Operation `Yield to Executive`, when invoked within arm *i,* causes execution at FP_i to stop and execution at the top frame of the common stack base to resume. `Yield to Process(p)` is always called from the simulation executive, the top frame of common stack base, and causes execution to resume at frame FP_i where arm *i* is the stack for entity *p*.

5 IMPLEMENTATION AND RESULTS FOR THE IBM JIKES REFERENCE VIRTUAL MACHINE

Most of the work within the Jikes RVM focused on creating the cactus stack semantics within the context of the single conventional stack organization expected by the RVM design. Our approach swaps the cactus stack arms on and off the conventional thread stack in a way that does not disturb normal RVM function.

5.1 Elaborate(p)

Creating a new entity does not require any special stack manipulation. The simulation executive uses `Elaborate(p)` to start the execution of an entity for the first time. This eventually results in the invocation of the `agenda()` method on entity *p*. Figure 3 shows the state of the stack after the simulation executive invokes `Elaborate(p)` which in turn has invoked the `agenda()` of entity *p*. The frame pointer, FP, points to the `agenda()` frame showing that the `agenda()` is currently executing.

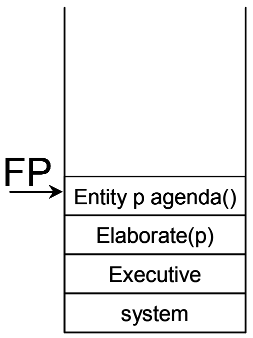

Figure 3: Initial Invocation of an Entity Agenda

5.2 Yield to Executive

The left side of Figure 4 shows the continued execution of entity *p*. The `agenda()` code invoked framework method `waitForTime(5.0)` indicating that entity *p* should be suspended until 5.0 units of simulation time have passed. Framework method `waitForTime()` does the necessary bookkeeping to schedule the reactivation of entity *p* and then uses the coordination operation `Yield to Executive` to pass control to the simulation executive.

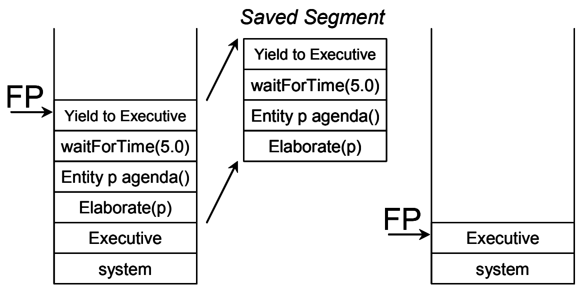

Figure 4: Saving Entity Execution State

`Yield to Executive` searches the stack from the point of its invocation downward until it finds the stack frame for the simulation executive. It then copies the stack frames from the top of the stack downward to the frame just above the simulation executive to a holding pool elsewhere in memory. Finally, `Yield to Executive` repositions the frame pointer, FP, to the simulation executive stack frame. As shown on the right side of Figure 4, this movement of the FP makes it appear to the simulation executive that its invocation of `Elaborate(p)` has returned.

5.3 Yield to Process(p)

This operation resumes entity *p* execution after is has been suspended by the `Yield to Executive` operation. It locates the stack segment for entity p in the holding pool and copies it back to the frame stack where it will then resume execution. In the Jikes RVM, stack frames contain absolute memory pointers to other frames in the stack. This means that when a segment is moved out of the holding pool and back onto the frame stack it must be replaced in exactly the same memory locations from which it was found. This leads to a collision between the stack frame supporting operation `Yield to Process(p)` and the `Elaborate(p)` stack frame in the stack segment being restored. Said another way, the `Yield to Process(p)` operation will overwrite its own stack frame while coping the entity *p* stack segment back on to the frame stack.

A solution to the problem is to divide the restoration of the stack segment into two parts. In the first part, `Yield to Process(p)` determines where the top of the frame stack will be when the entity *p* segment is restored. At this position it creates the stack frame for the invocation of method `Restore Segment`. This is shown on the left side of Figure 5. In the second part, `Restore Segment` copies the stack segment to the memory that has been reserved beneath itself. It safely overwrites the `Yield to Process(p)` stack frame in the process because the active part of the frame stack is at `Restore Segment` and above.

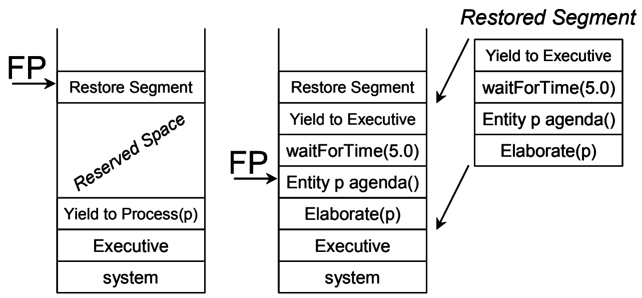

Figure 5: Restoring Entity Execution State

Once the entity *p* stack segment is copied into place, the FP is moved down three stack frames. This is shown in the middle of Figure 5. Control leaves `Restore Segment` and enters `waitForTime(5.0)` which itself returns. From the perspective of the entity *p* `agenda()`, the method `wait-ForTime(5.0)` has just returned and the next statement in the `agenda()` can be executed.

5.4 Terminate(p)

This operation is used when the simulation executive determines that an entity will never need to be restored. Usually this occurs when the entity `agenda()` is complete. `Terminate(p)` does not require extraordinary frame stack manipulation. It simply removes the entity *p* stack segment from the holding pool.

5.5 Initial Results

Initial results confirm the potential power of the cactus stack approach. Controlled benchmarks are able to create and manage up to 42,000 entities using as little as 100MB of memory. This compares favorably with the modest numbers of entities (less that 8,000) that can be created using conventional thread-based implementations. Before a robust implementation of the cactus stack can be considered for practical use, more work must be done on the interface to the garbage collection system.

Support for garbage collection is a powerful feature of Java. The essence of garbage collection is to separate portions of memory that are reachable by program reference from memory that has been allocated but is no longer reachable. Memory that is no longer reachable is garbage and can be collected and made available for future reallocation. Garbage collectors may examine the frame stack in the process of locating all the reachable memory in a program. The cactus stack manipulation adds an additional step to conventional garbage collection. After step to conventional garbage collection. After the garbage collector examines the frame stack it must also examine all the stack segments in the holding pool. This is needed because a stack segment may contain the only reference to an object that, if not taken into account, could be mistaken for garbage. The segment may also have references to objects that are moved by the garbage collector and need to be updated with the new object location. This also includes references to code objects, such as saved instruction pointers, that must also be updated if a code object moves.

6 CONCLUSIONS AND FUTURE WORK

Work to extend one of the Jikes RVM garbage collectors has been fruitful. The extended garbage collector successfully includes saved stack segments in its analysis and properly updates references to relocated objects. Work remains to improve the efficiency of the stack segment holding pool and the overall implementation robustness.

Once the IBM Jikes implementation of the single thread simulation executive is made robust enough for production use, attention will turn to the Sun JVM. It is hoped that similar success with the Sun JVM will support the argument that co-routines would be a useful extension to the Java language and a useful tool to the growing Java-based simulation market.

ACKNOWLEDGMENTS

The work described here is MITRE Sponsored Research funded through the MITRE Technology Program. The support of Al Grasso, Steve Huffman and Dave Lehman is gratefully acknowledged. The concepts and implementation described in this paper reflect significant contributions by Fred Kuhl, Rob Mikula, Keven Ring, Dave Seidel and Bill Weiland.

REFERENCES

Adelsberger, H.H. 1984. Prolog as a Simulation Language, In: *Proceedings of the 1984 Winter Simulation Conference*, ed. S. Sheppard, U. Pooch, and C. Pegden, 500-504, Dallas, TX, 28-30 December.

Bagrodia, R.L. 1991. Iterative Design of Efficient Simulations Using Maisie, In: *Proceedings of the 1991 Winter Simulation Conferenc*e, ed. B. Nelson, W. Kelton, and G. Clark, 243-247, Phoenix, AZ, 8-11 December.

Balci, O. 1986. Requirements for Model Development Environments, *Computers and Operations Research*, **13**(1):55-67.

Balci, O., R.E. Nance, E.J. Derrick, E.H. Page, and J.L. Bishop. 1990. Model Generation Issues in a Simulation Support Environment, In: *Proceedings of the 1990 Winter Simulation Conference*, ed. O. Balci, R. Sadowski, and R. Nance, 257-263, New Orleans, LA, 9-12 December.

Barnett, C.C. 1986. Simulation in Pascal with Micro PASSIM, In: *Proceedings of the 1986 Winter Simulation Conference*, ed. J. Wilson, J. Henriksen, and S. Roberts, 151-155, Washington, DC, 8-10 December.

Buss, A. 2000. Component-Based Simulation Modeling, In: *Proceedings of the 2000 Winter Simulation Conference*, ed. R. Barton, J. Joines, P. Fishwick, and K. Kang, 964-971, Orlando, FL, 10-13 December.

Buss, A. 2001. Discrete Event Programming with Simkit, *Simulation News Europe*, 32/33:15-25, November.

DESMO-J Available online via `<www.desmoj.de>` (accessed July 14 2004).

Drolet, J.R., C.L. Moodie and B. Montreuil. 1991. Object-Oriented Simulation with Smalltalk-80: A Case Study, In: *Proceedings of the 1991 Winter Simulation Conference*, ed. B. Nelson, et al., 312-322, Phoenix, AZ, 8-11 December.

Elmaghraby, A.S. 1988. A Prolog Simulation for a Delphi-based Problem Solver, *Simulation Digest*, **19**(4):36-43.

Fishwick, P.A. 1992. SimPack: Getting Started with Simulation Programming in C and C++, in: *Proceedings of the 1992 Winter Simulation Conference*, J. Swain, D. Goldsman, R Crain, and J. Wilson, 154-162, Arlington, VA, 13-16 December.

Garrido, J.M. 2001. *Object-Oriented Discrete Event Simulation with Java: A Practical Introduction*, Kluwer New York, NY: Academic / Plenum Publishers.

Garrido, J.M. and K. Im. 2004. Teaching Object-Oriented Simulation with PsimJ Simulation Package, In: *Proceedings of the 42nd Annual ACM Southeast Regional Conference*, 422-427, Huntsville, AL, 2-3 April.

Howell, F. and R. McNab. 1998. Simjava: A Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling, In: *Proceedings of the First International Conference on Web-Based Modeling and Simulation*, San Diego, CA, January.

IBM Jikes Reference Virtual Machine. 2004. Available online via `<www.ibm.com/developerworks/oss/jikesrvm/>` (accessed July 14 2004)

Kilgore, R.A. 2000. Silk, Java and Object-Oriented Simulation, In: *Proceedings of the 2000 Winter Simulation Conference*, ed. R. Barton, et al., 246-252, Orlando, FL, 10-13 December.

Knapp, V. 1986. The Smalltalk Simulation Environment, In: *Proceedings of the 1986 Winter Simulation Conference*, ed. J. Wilson et al., 125-128, Washington, DC, 8-10 December.

L'Ecuyer, P. and N. Giroux. 1987. A Process-Oriented Simulation Package Based on Modula-2, In: *Proceedings of the 1987 Winter Simulation Conference*, ed. A. Thesen, H. Grant, W. Kelton, 136-145, Atlanta, GA, 14-16 December.

L'Ecuyer, P., L. Meliana and J. Vaucher. 2002. SSJ: A Framework for Stochastic Simulation in Java, In: *Proceedings of the 2002 Winter Simulation Conference*, ed. C. Chen, P. Sanchez, D. Ferrin and D. Morrice, 234-242, San Diego, CA 8-11 December.

Le, T.V. 1993. FPOSS: A Fuzzy Prolog-based Object-Oriented Simulation System, In: *Proceedings of the 6th Australian Joint Conference on Artificial Intelligence*, Melborne, November.

Lomow, G. and D. Baezner. 1990. A Tutorial Introduction to Object-Oriented Simulation and Sim++, In: *Proceedings of the 1990 Winter Simulation Conference*, ed. O. Balci, et al., 149-153, New Orleans, LA, 9-12 December.

Malloy, B. and M.L. Soffa. 1986. Simcal: The Merger of Simula and Pascal, In: : *Proceedings of the 1986 Winter Simulation Conference*, ed. J. Wilson, et al., 397-403, Washington, DC, 8-10 December.

Miller, J.A., Y. Ge and J. Tao. 1998. Component-Based Simulation Environments: JSIM as a Case Study Using Java Beans, In: *Proceedings of the 1998 Winter Simulation Conference*, ed. D. Medeiros, E. Watson, J. Carson, and M. Manivannan, 373-381, Washington, DC, 13-16 December.

Nance, R.E. 1993. A History of Discrete Event Simulation Programming Languages, *ACM SIGPLAN Notices*, 28(3):149-175.

Oberndorf, P.A. 1988. The Common Ada Programming Support Environment (APSE) Interface Set (CAIS), *IEEE Transactions on Software Engineering*, 14(6):742-748.

Page, E.H., R.L. Moose, Jr. and S.P. Griffin. 1997. Web-Based Simulation in Simjava using Remote Method Invocation, In: *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Andradottir, K. Healy, D. Withers, and B. Nelson, 468-474, Atlanta, GA, 7-10 December 1997.

Samuels, M.L., and J.R. Spiegel. 1987. The Flexible Ada Simulation Tool (FAST) and its Extensions, In: *Proceedings of the 1987 Winter Simulation Conference*, ed. A. Thesen, et al., 175-184, Atlanta, GA, 14-16 December.

Sardesai, S., D. McLaughlin and P. Dasgupta. 1998. Distributed Cactus Stacks: Runtime Stack-Sharing Support for Distributed Parallel Program*s, In: Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV,13-16 July.

Schwetman, H. 1986. CSIM: A C-Based Process-Oriented Simulation Language, In: *Proceedings of the 1986 Winter Simulation Conference*, ed. J. Wilson, et al., 387-396, Washington, DC, 8-10 December.

Schwetman, H. 1994. CSIM17: A Simulation Model-Building Toolkit, In: *Proceedings of the 1994 Winter Simulation Conference*, ed. J. Tew, S. Manivannan, D. Sadowski, and A. Seila, 464-470, Orlando, FL, 11-14 December.

Seila, A. F. 1986. Discrete Event Simulation in Pascal with SIMTOOLS, In: *Proceedings of the 1986 Winter Simulation Conference*, ed. J. Wilson, et al., 141-150, Washington, DC, 8-10 December.

Tyan, H.-Y. 2002. Design, realization and Evaluation of a Component-Based Compositional Software Architecture for Network Simulation, Ph.D. Thesis, The Ohio State University, Columbus, OH.

USJFCOM. 2004. Joint Simulation System (JSIMS). Available online via `<www.jfcom.mil/about/fact_jsims.htm>` (accessed July 14 2004).

Uyeno, D. 1980. PASSIM, A Discrete Event Simulation Package for Pascal, *Simulation*, **35**(5):479.

AUTHOR BIOGRAPHIES

RICHARD M. WEATHERLY is a Consulting Engineer with The MITRE Corporation. He received a Ph.D. in Electrical Engineering from Clemson University in 1984. He led the technical development of the Aggregate Level Simulation Protocol, DoD High Level Architecture (HLA) for Modeling and Simulation, and the HLA Runtime Infrastructure Verification Facility. His email address is `<weather@mitre.org>`.

ERNEST H. PAGE is a Principal Scientist with The MITRE Corporation. Since receiving the Ph.D. in Computer Science from Virginia Tech in 1994, he has worked primarily in the defense simulation arena. He served as ACM SIGSIM Chair (1999-2001) and is currently an Associate Editor for the *Journal of Defense Modeling and Simulation,* and the ACM *Transactions on Modeling and Computer Simulation*, and Area Editor for Military Applications for SCS *Simulation*. He has served on the Board of Directors for Winter Simulation Conference since 2001. His email address is `<epage@mitre.org>`.