The Parsimony Project: A Distributed Simulation Testbed in Java

Bruno R. Preiss and Ka Wing Carey Wan
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L 3G1, Canada

Keywords: distributed, discrete-event simulation; optimistic synchronization; conservative synchronization; Java.

Abstract

The Parsimony Project is a vehicle for conducting research in distributed, network-centric computing. The primary objective is the development of a Java-based testbed for distributed discrete-event simulation. In this paper, we present requirements for the implementation of a distributed, discrete-event simulation system based on our earlier research in the area. We show how the Java language and virtual machine support directly these requirements. Finally, we describe briefly a suite comprised of eight different, yet compatible simulators. A user-defined simulation can be run using any of the simulators without modification or even recompilation.

1 Overview

This paper describes *The Parsimony Project*[1], the latest phase in an on-going program of research in parallel and distributed discrete-event simulation[2–5]. The Parsimony Project is the direct descendant of *Yaddes*, a simulation specification language and execution environments that were used to study the performance of distributed, discrete-event simulation[6, 7]. The Parsimony Project has emerged at this time because of the advent of the Java language and the related technologies which support distributed, network-centric computing. In Section 2 we review the requirements for distributed, discrete-event simulation that were identified in previous research and in Section 3 we show how the features of Java and its related technologies support directly the implementation of network-centric simulation. Section 4 presents the modeling methodology used in Parsimony and shows the relationships between the user-defined classes and system classes. In Section 5 we describe the the eight simulation engines currently available—a user-defined simulation can be run using any of these simulators without modification or even recompilation. Finally, in Section 6 we illustrate the Parsimony simulation paradigm using a simple queueing network example.

2 Requirements for Distributed Discrete-Event Simulation

On the basis of our earlier experience in the implementation of systems for parallel and distributed discrete-event simulation, we have identified the following requirements for the development language and environment:

2.1 Modeling Support

In parallel and distributed discrete-event simulation (PDES), the real-world system is viewed as a network of interacting, physical processes (self-contained, discrete-event systems)[8]. The physical processes exchange information periodically, at discrete points in time. Each such exchange of information is equivalent to an event.

The network of physical processes is simulated by a collection of logical processes that exchange messages. Each message carries the information associated with an event and the time at which the event occurs. The description of the state and the behavior of a logical process is called a *model*. The simulated system is a network of instantiated models.

Therefore, the development language must support the specification of models (state and behaviour). It must support the instantiation of models and the concurrent execution of model instances.

2.2 Dynamic Loading

A simulator must be extensible in the sense that user-defined models can be simulated without requiring the simulator itself to be recompiled.

This kind of extensibility can be support in one of several different ways: (1) user-defined models may be specified using a "simulation language" that is either compiled or interpreted by the simulator; (2) the user-defined model is specified in a conventional programming language which is then compiled and linked with a simulation library; or (3) compiled user-defined models can be linked dynamically to an existing simulator (e.g., using a mechanism like that of a dynamic-linked library). Ideally, models can be selected and linked into a running simulation on demand.

2.3 Support for Multiple Execution Threads

The modeling paradigm used in PDES leads naturally to an implementation consisting of a network of communicating logical processes. While it is possible to implement the required behaviour in a language that does not support concurrency explicitly, our experience has shown that it is far more natural and expedient to implement such a system using a language/environment that supports user-level concurrency (threads). Since concurrent programming is difficult to get right, we do not expect (or require) that the simulation user make use of threads. Rather, the modeling paradigm automatically yields a simulation that can be partitioned for parallel/distributed execution.

2.4 Transparent and Extensible Networking Support

In order to build a distributed simulation, it is necessary to be able to distribute model instances over multiple processors and to allow those instances to exchange information. However, both the models and the messages they send are defined by the simulation user. Therefore, the simulator must support the dynamic distribution of models, which comprise both state (data) and behaviour (code), as well as the exchange of arbitrary, user-defined messages. Thus, the development language/environment must support transparently extensible networking in that it automates the marshaling and unmarshaling of simulation entities (both model instances and messages).

3 How Java Supports Distributed Discrete-Event Simulation

In this section we discuss the requirements for the implementation of a distributed, discrete-event simulator and we show how those requirements are supported by the mechanisms provided in the Java language and the Java virtual machine.

3.1 Models as Classes, Events as Runnable Objects

That the object-oriented paradigm is a natural one for the specification of simulations, especially discrete-event simulations, is well understood. In particular, the class concept admits a natural mapping from the simulation application to programming language constructs. Specifically, classes, which encapsulate state and behaviour, are a natural way to represent simulation models and instances of such classes correspond to simulation entities.

In Parsimony we go one step further—events are represented as timestamped, runnable objects. (In Java, a runnable object is one that implements the interface java.lang.Runnable). Thus, to simulate an event, we simply run the corresponding object. (Events objects are run at most once.)

In a discrete-event simulation, the execution of an event typically results in modification of the system state as well as the scheduling of future events. Since the state of the simulation resides in the instances of models, there needs to be a coupling between event objects and model instances.

Thus the coupling between event objects and model instances is naturally facilitated by Java's *inner classes*. The following code fragment illustrates the basic idea:

```
class Model
{
   State state = new State();
   class Event implements java.lang.Runnable
   {
      public void run()
      {  modify(state);
         schedule(new Event());
      }
   }
}
```

In Java, an inner class (e.g., Event) is a class defined within another class (e.g., Model). Every instance of a (non-static) inner class is implicitly bound to an instance of the outer class. Furthermore, the methods defined in the inner class (e.g., run) have direct access to the fields (e.g., state) of the outer class instance. This coupling has turned out to be an extremely useful Java feature and it is used extensively in the implementation of Parsimony.

3.2 Logical Processes as Threads

Java's support for concurrent programming directly facilitates the implementation of parallel/distributed simulations. The concurrency mechanisms in Java (e.g., monitor-like synchronization primitives) have precisely defined semantics which are supported directly by the Java compiler and the Java virtual machine[9, 10]. This is in direct contrast to languages such as C and C++ which provide no support for concurrency (in the language) and in which compiler optimizations that are perfectly legal for sequential code may violate the semantic requirements of concurrent code.

In Parsimony, the system to be simulated is viewed as a network of physical processes. The physical processes are simulated by instances of user-defined model classes. Each model instance is managed by an instance of a *logical process* system entity. Each logical process runs as a separate thread. The logical processes self-synchronize and ensure that events are executed in a proper sequence (i.e., without causality violations).

3.3 The Java Virtual Machine as Simulation Engine

A general-purpose simulation engine needs to be able to load and execute user-defined models. Since a user-defined model comprises both state and behaviour, this requirement is satisfied by the ability to dynamically load, link, and instantiate user-defined classes.

In this regard, the facilities of the Java language and virtual machine are ideal. A Java virtual machine can load classes on demand—the actual classes loaded need not be known to the simulator. Furthermore, a Java virtual machine can be made to load classes from a remote site. Using this feature, it is quite simple to implement a distributed simulation engine as a network of interconnected Java virtual machines.

The security features of Java (e.g., strict compile- and runtime type checking, the bytecode verifier and security managers) significantly enhance the robustness of the resulting simulator. In particular, it has been our experience that these features greatly assist in the debugging of simulation models.

3.4 Object Serialization and Remote Method Invocation

In a distributed simulator, logical processes and the model instances they manage are distributed over multiple processors. That is, they execute in separate Java virtual machines. In order to effect the simulation, the logical processes need to exchange user-defined information (event messages).

The distribution of model instances as well as the exchange of event messages, requires the ability to move objects from one Java virtual machine to another. Fortunately, Java supports directly and transparently *object serialization*[11]. I.e., any instance of a class that implements the interface java.io.Serializable is easily transfered from one Java virtual machine to another one over a TCP connection. The marshaling of the object in the sender and the unmarshaling at the receiver is automatic. In Parsimony, the distribution of model instances and the transmission of event messages is facilitated by Java's object serialization.

An even more powerful capability of Java that is used to great effect in Parsimony is the notion of *remote method invocation*[12]. Java implements remote procedure calls in an extremely transparent fashion. Since all objects in Java are manipulated through a reference, it does not matter to the user of the object whether the object to which a reference refers is a local object or a remote one. In the case of a remote method invocation (RMI), the arguments are serialized in the virtual machine of the caller and transmitted to the virtual machine of the callee. Similarly, the return value is serialized at the callee and returned to the caller.

The distributed Parsimony simulators turned out to be extremely easy to implement using RMI. We approached the problem in two steps. First, we implemented a threaded simulator in which each logical process executes as a separate thread. Then, we arranged to have the various threads distributed over multiple processors. By carefully writing the threads to eliminate the reliance on shared global data, the distribution of the threads was accomplished without the need to modify the code.

4 Modeling and Simulation in Parsimony

The first step in writing a simulation is to construct a model of the real-world system to be studied. The real-world system is modeled as a collection of communicating, discrete-event processes called *physical processes* (PPs). The state of a discrete-event PP changes discontinuously at discrete points in time. In addition, the PPs periodically exchange information at discrete points in time.

The simulation of the real-world system is obtained by writing a simulation program in which the behaviour of each PP is mim-

icked by a *logical process* (LP). Each LP comprises application-specific state and behaviour and, in addition, each LP maintains its own internal (future) event list. The exchange of information by the PPs is mimicked in the simulation by the exchange of messages by the LPs. Since the computer simulation does not execute in real time, each LP has its own notion of simulation time and each message is tagged with the time of the corresponding real-world event.

To ensure the correctness of the simulation, each LP must satisfy a *local causality constraint*—events and messages must be processed by an LP in non-decreasing timestamp order.

In Parsimony, we isolate the modeling domain from the domain of the simulation engine (the *simulator*). The user writes the application-specific *simulation* code. The Parsimony system provides the generic *simulator* code. In this way, the user needs only to be concerned with the details of the application and not with the nitty-gritty details of writing a causally-correct simulator.

4.1 The Entity Model and System Model Classes

Having obtained a model of the real-world system that consists of a network of PPs, the user prepares the simulation program by writing *entity model* classes and a *system model* class. Figure 1 shows the relationship between the user-defined entity and system model classes and the classes provided by Parsimony.

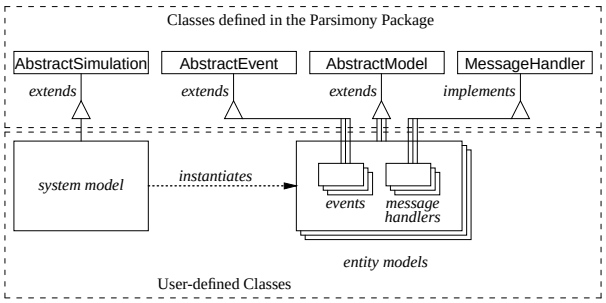

Figure 1: Relationship between User-defined and Parsimony System Classes

Each *entity model* class is derived from the Parsimony.AbstractModel abstract class. An entity model encapsulates state and defines *events* and *message handlers*. As described in Section 3.1, each event is an instance of a runnable inner class defined within a model class. As shown in Figure 1, events are derived from the Parsimony.AbstractEvent class.

The model class inherits from the AbstractModel base class the methods schedule and send. The former is used to schedule events in the future, the latter is used to send messages to other LPs. Every model that receives messages defines one or more *message handlers*. As shown in Figure 1, a message handler is an inner class defined within a model class. When a message is received at an LP, it is bound to a message handler. The combination of message handler and a message is equivalent to an event—it is a runnable object that, when run, modifies the state of the model as required.

To complete the specification of the simulation, the user defines a *system model* class. As shown in Figure 1, the system model class is derived from the Parsimony.AbstractSimulation abstract class.

A simulation is a *runnable* object. Its run method is required to create the network of LPs that make up the simulation. It inherits from the base class the methods createChannel and createProcess. The former is used to establish communication channels between LPs. The latter is used to instantiate the LPs themselves.

4.2 Achieving the Separation of Concerns

An important element in the success of our approach has been the separation of the user-defined, application-specific domain from the domain of the simulator. Figure 2 illustrates how this has been achieved.

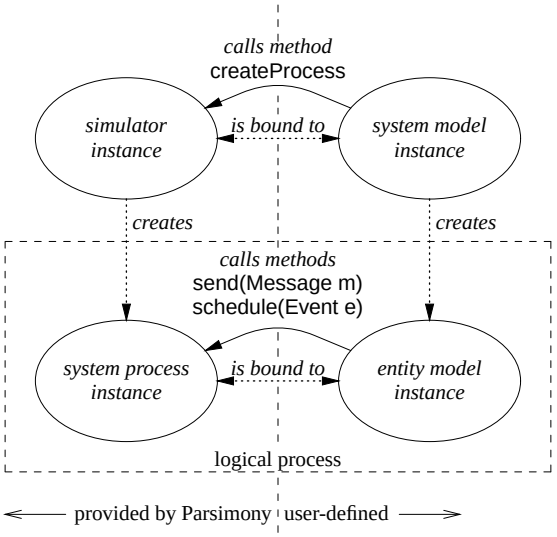

Figure 2: Achieving the Separation of Concerns

As discussed in the preceding section, a complete simulation is obtained by binding an instance of a user-defined system model class with an instance of a Parsimony simulator class. To begin execution, the simulator *runs* the system model. The system model instantiates the network of LPs and establishes the communication channels between the LPs.

Because each LP comprises both user-defined and system-defined behaviour, each LP is, in fact, a pair of objects—an instance of a user-defined entity model and an instance of a simulator process class. The entity model instance encapsulates application-specific state and behaviour. The simulator process instance encapsulates the LP's local event list and it implements the send and schedule methods.

The bindings between the system model instance and simulator instance and between entity model instance and simulator process instance are established dynamically at run-time. Consequently, the user-defined simulation code is completely independent of the simulator code, allowing the latter to be implemented in many different ways. In the following section, we describe eight different simulators currently implemented. The beauty of Parsimony is that, because of the separation of concerns, the user-defined simulation can be run using any one of the eight simulators *without recompilation*!

5 Simulators

To date, we have implemented eight different simulators. The simulators are completely interchangeable. I.e., a given user-defined simulation will run with any one of the simulators, and in (almost) all cases the the results of the simulations are identical. Slight differences arise because the implementations do not impose a total order on events—events having exactly the same timestamp may execute in a different order in different simulators. Thus, different behaviour is only possible if the user-defined models are affected by the execution order of simultaneous events.

Because Java supports the dynamic loading of classes, our implementation dynamically loads the *simulator* as well as the *user-defined simulation*. I.e., the choice of simulator is only bound at run time. Table 1 lists the eight simulators currently implemented and briefly describes their characteristics.

5.1 The Distributed Simulators

The Parsimony package includes three different distributed simulators—DistributedMLSimulator, DistributedCMBSimulator, and DistributedTWSimulator. A distributed simulator is one in which a given user-defined simulation is executed using multiple Java virtual machines running on different host computers.

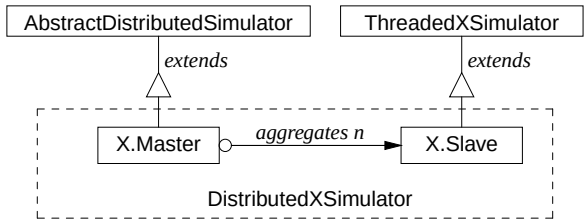

Figure 3: Making a Distributed Simulator from a Threaded One

In each case, we began by implementing a non-distributed simulators using multiple Java threads—ThreadedMLSimulator, ThreadedCMBSimulator, and ThreadedTWSimulator. In these simulators each logical process executes as a separate thread. What differs is how the threads synchronize in order to guarantee that the local causality constraint is not violated.

Figure 3 illustrates how we derived the implementations of the distributed simulators from the non-distributed, threaded ones. A distributed simulator consists of a single, master simulator and one or more slave simulators. The master simulator runs the system model entity and the logical processes are distributed among the slave simulators.

This distribution is easily achieved by making use of the Java remote method invocation mechanism. As Figure 3 shows, we implement a slave simulator by extending the threaded simulator. Because Java remote method invocation is transparent, the extension is trivial. (In each case only a few tens of lines of code were required!)

6 An Example—A Single-Server Queueing Network

In this section we present an example that illustrates how a queueing system is modelled in Parsimony. Figure 4 shows a simple queueing system. The system is comprised of three physical processes—the source process, the queue-and-server process, and the sink process. The behaviour of each of theses processes is modelled by a separate model class.

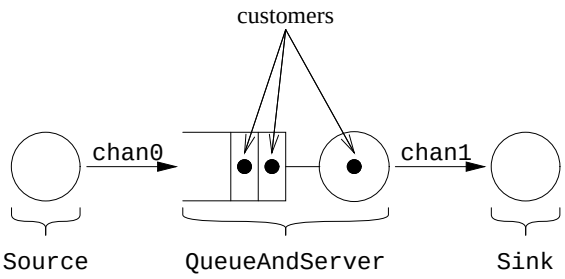

Figure 4: A Simple Queueing Network

6.1 Source Model

The code which describes the source model is given below. As shown in Figure 4, the source model has no inputs and a single output. The number of inputs and outputs are specified in the call to the super-class constructor.

The source process shown models a Poisson process. I.e., the customer inter-departure times are exponentially distributed. The Parsimony package includes several random variable classes, including the ExponentialRV used here.

The source model comprises a single event type—a customer departure. Customer departure events are represented as instances of the inner class, Departure. Invoking the run method of a departure event sends a message which represents a customer to the queue-and-server model, and then creates and schedules a new departure event.

```
class Source extends AbstractModel
{
   RandomVariable interDepartureTime;

   public Source (long mean)
   {  super(0, 1);
      interDepartureTime =
         new ExponentialRV(mean);
   }

   public void initialize (long time)
      { schedule(new Departure(time)); }

   class Departure extends AbstractEvent
   {
      Departure(long time) { super(time); }

      public void run ()
      {  send(new VoidMessage(getTime()));
         schedule(new Departure(
            Math.round(getTime() +
            interDepartureTime.nextDouble())));
      }
   }
}
```

6.2 Sink Model

As shown in Figure 4, the sink model has a single input and no outputs. The behaviour of the sink model is achieved by defining a message handler to process input messages. A the run method

Table 1: Characteristics of the Various Simulators

SequentialSimulator	A single-threaded, sequential simulator implemented using a common event list.
MultiListSimulator	A single-threaded, sequential simulator in which each logical process maintains its own event list.
ThreadedMLSimulator	A multi-threaded simulator in which each logical process runs as a separate thread. The threads are scheduled by a central scheduler.
DistributedMLSimulator	A distributed, multi-threaded simulator in which each logical processes runs as a separate thread in a remote, slaved simulator. The threads are scheduled by a central, master scheduler.
ThreadedCMBSimulator	A multi-threaded simulator implemented using the distributed, conservative synchronization method due to Chandy, Misra, and Bryant.
DistributedCMBSimulator	A distributed, multi-threaded simulator in which each logical processes runs as a separate thread in a remote, slaved simulator. The threads are scheduled using the distributed, conservative synchronization method due to Chandy, Misra, and Bryant.
ThreadedTWSimulator	A multi-threaded simulator implemented using the distributed, optimistic synchronization method known as Time Warp[13]. This simulator does completely transparent copy state saving (checkpointing) and rollback error recovery.
DistributedTWSimulator	A distributed, multi-threaded simulator in which each logical processes runs as a separate thread in a remote, slaved simulator. The threads are scheduled using using the distributed, optimistic synchronization method known as Time Warp.

of the message handler is invoked for every received message. In this case, the run method is empty.

```
class Sink extends AbstractModel
{
   Sink ()
   {  super(1, 0);
      setMessageHandler(new ArrivalHandler());
   }

   class ArrivalHandler
      implements MessageHandler
   {
      public void run(Message message) {}
   }
}
```

6.3 Queue-and-Server Model

The specification of the queue-and-server model is given below. The model shown implements a first-in, first-out queue and a single-server that provides non-preemptive service with exponentially distributed service times.

The queue-and-server model has one input and one output. The model encapsulates two inner classes:

ArrivalHandler A message handler invoked each time a customer arrives.

Departure A customer departure event invoked when the server finishes serving a customer.

A message is received by the queue-and-server model represents the arrival of a customer. The behaviour of the arrival handler is determined by the state of the model. If the server is busy when a customer arrives, the customer joins the queue. Otherwise, a new customer departure event is scheduled.

The behaviour of the departure event is also determined by the state of the model. Invoking the run method of a departure event sends a message which represents a customer to the sink model. If there are still more customers in the queue, a new customer departure event is created scheduled.

```
class QueueAndServer extends AbstractModel
{
   RandomVariable serviceTime;
   int numberInQueue = 0;
   boolean serverBusy = false;

   QueueAndServer (long mean)
   {  super(1, 1);
      serviceTime = new ExponentialRV(mean);
      setMessageHandler(new ArrivalHandler());
   }

   class ArrivalHandler
      implements MessageHandler
   {
      public void run (Message message)
      {  if (serverBusy) ++numberInQueue;
         else
         {  serverBusy = true;
            schedule(new Departure(
               Math.round(getTime() +
               serviceTime.nextDouble())));
         }
      }
   }

   class Departure extends AbstractEvent
   {
      Departure (long time) { super(time); }

      public void run ()
      {   send(new VoidMessage(getTime()));
         if (numberInQueue == 0)
            serverBusy = false;
         else
         {  --numberInQueue;
            schedule(new Departure(
               Math.round(getTime() +
               serviceTime.nextDouble())));
         }
      }
   }
}
```

6.4 System Model

In Parsimony we define a *simulation* class to represent the system to be simulated. A simulation is a runnable object (i.e., it java.lang.Runnable). The run method of a simulation is expected to instantiate the logical processes that comprise a simulation as well as the communication channels that connect the processes.

The code for the class Queueing given below describes the queueing network shown in Figure 4. The system comprises two communication channels, chan0 and chan1, and three logical (simulation) processes. The run method creates the channels and processes and then invokes the simulate method in order to commence the simulation.

```
class Queueing extends AbstractSimulation
{
   public void run ()
   {  Channel chan0 = createChannel();
      Channel chan1 = createChannel();
      createProcess(new Source(1000),
         new ChannelHead[] {},
         new ChannelTail[] { chan0 });
      createProcess(new QueueAndServer(1000),
         new ChannelHead[] { chan0 },
         new ChannelTail[] { chan1 });
      createProcess(new Sink(),
         new ChannelHead[] { chan1 },
         new ChannelTail [] {});
      simulate();
   }
}
```

7 Summary and Conclusions

The Parsimony Project is a vehicle research in network-centric distributed simulation. In this paper we have briefly described the goals of the project and the current project status. The principal contribution of this paper is the identification of the requirements of distributed, discrete-event simulation with respect to the underlying implementation technologies. In addition, we show how features of the Java language and the Java virtual machine directly support these requirements.

7.1 Online Materials

More information about the Parsimony project is available from the Parsimony website[1]. In particular, the queueing network simulation described in Section 6 is available as a Java applet[2].

References

[1] B. R. Preiss. The Parsimony Project Website[3], 1998.

[2] Y.-B. Lin and B. R. Preiss. Optimal memory management for Time Warp parallel simulation[4]. *ACM Trans. on Modeling and Computer Simulation*, 1(4):283–307, October 1991. (Accepted May 1992. Published September 1992.).

[3] Y.-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the checkpoint interval in Time Warp parallel simulation[5]. In *Proc. 1993 Workshop on Parallel and Distributed Simulation*, pages 3–10, San Diego, CA, May 1993. Institute of Electrical and Electronics Engineers, Inc.

[4] B. R. Preiss and W. M. Loucks. Memory management techniques for Time Warp on a distributed memory machine[6]. In *Proc. 1995 Workshop on Parallel and Distributed Simulation*, pages 30–39, Lake Placid, NY, June 1995. Institute of Electrical and Electronics Engineers, Inc.

[5] B. R. Preiss, I. D. MacIntyre, and W. M. Loucks. On the trade-off between time and space in optimistic parallel discrete-event simulation[7]. In *Proc. 1992 Workshop on Parallel and Distributed Simulation*, pages 33–42, Newport Beach, CA, January 1992. Society for Computer Simulation.

[6] B. R. Preiss. The Yaddes distributed discrete event simulation specification language and execution environments[8]. In *Proc. SCS Multiconf. on Distributed Simulation*, pages 139–144, Tampa, FL, March 1989. Society for Computer Simulation.

[7] B. R. Preiss and I. D. MacIntyre. YADDES—Yet Another Distributed Discrete Event Simulator: User manual[9]. CCNG Technical Report E-197, Department of Electrical and Computer Engineering and Computer Communications Networks Group, University of Waterloo, 1990.

[8] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–66, March 1986.

[9] D. Lea. *Concurrent Programming in Java^{TM}*[10]. The Java^{TM} Series. Addison-Wesley, Reading, MA, 1996.

[10] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*[11]. The Java^{TM} Series. Addison-Wesley, Reading, MA, 1996.

[11] Sun Microsystems, Inc. *Java^{TM} Object Serialization Specification*[12], 1998.

[12] Sun Microsystems, Inc. *Java^{TM} Remote Method Invocation Specification*[13], 1998.

[13] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and the Time Warp Operating System. In *Proc. 12th SIGOPS—Symposium on Operating Systems Principles*, pages 77–93, 1987.

[1] http://www.pads.uwaterloo.ca/Bruno.Preiss/parsimony
[2] http://www.pads.uwaterloo.ca/Bruno.Preiss/parsimony/queueing-demo.html
[3] http://www.pads.uwaterloo.ca/Bruno.Preiss/parsimony
[4] http://www.acm.org/pubs/citations/journals/tomacs/1991-1-4/p283-lin/
[5] http://www.brpreiss.com/papers/published/1993/pads/paper.pdf
[6] http://www.brpreiss.com/papers/published/1995/pads2/paper.pdf
[7] http://www.brpreiss.com/papers/published/1992/pads/paper.pdf
[8] http://www.brpreiss.com/papers/published/1989/emc/paper.pdf
[9] http://www.brpreiss.com/reports/ccng/E-197/report.ps
[10] http://www.awl.com/cseng/titles/0-201-69581-2/
[11] http://www.awl.com/cseng/titles/0-201-63452-X/
[12] ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.ps
[13] ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.ps