Discrete-Event Simulation in Java – a Practitioner's Experience

D. H. King
Manager, Simulation Modeling
Ausenco Sandwell
855 Homer Street
Vancouver, BC V6B 2W2
Canada
harry.king@ausencosandwell.com

Harvey S. Harrison
Simulation Analyst
Ausenco Sandwell
855 Homer Street
Vancouver, BC V6B 2W2
Canada
harvey.harrison@ausencosandwell.com

Keywords: Java, discrete-event simulation, supply chain, parallel processing, practitioner.

Abstract
The experience of a simulation practitioner with development of a new Java simulation engine and its application to a large simulation model is described. Our simulation engine is implemented as a simple extension of the Java programming language and features interactive 3-D graphics and parallel processing. We believe that this article will be of interest to researchers because it shows how some of the concepts from simulation research have been applied "in the field" to model complex supply chains.

1. INTRODUCTION

Ausenco Sandwell has been using discrete-event simulation modeling as part of our engineering consulting services for more than 30 years and has completed more than 300 simulation projects during this time. At present we employ 16 professionals working full-time in simulation.

After many years of using commercial simulation software such as GPSS and SLAM, we began using Java for our simulation models starting in 2002. This article reports on the decisions behind this choice, the development of our new Java simulation engine, and our success with developing a very large simulation application in Java [1]. We also describe our on-going research to apply parallel processing to the complex supply chains modeled by our software.

2. WHY SIMULATE IN JAVA?

Commercial simulation software with graphical user interfaces for model building are excellent for creating models with moderate complexity and where the system to be modeled is a good fit with the tools provided. For very complex models or ones that are a poor fit with simulation tools provided, then commercial software can bring more restrictions and problems than advantages.

The problems with commercial simulation software start when the model builder must modify or supplement the logic provided with the built-in objects or create entirely new objects. A programming interface is provided for this purpose by the commercial software, but this programming environment is by its nature quite limited. The programming language will be a standard one such as C or Java, or worse, a subset of a standard language, but in either case the model builder will have to write this code without the use of all the programming and debugging tools provided with modern programming languages. This situation may be acceptable if only a few hundred lines of code are to be written. However, if thousands of lines are required to capture the desired level of detail in the system, then the advantages of the built-in objects and time-keeping provided by the commercial software start to seem very minor compared to the difficulties of developing complex software without modern programming tools. This was the case for our application, which is described later in this article.

The alternative to using commercial simulation software is to write your own discrete-event simulation engine in a general-purpose programming language, preferably an object-oriented one such as Java or C++, or to adapt an open-source engine provided by a researcher. When we considered adapting open-source simulation software, we found that the software available to us in 2002 was still in the early days of development and in many cases was restricted to non-commercial use, which eliminated consultants like ourselves. For these reasons, we chose to write our own simulation engine as part of our model building effort. The difficulty of writing a simulation engine should not be underestimated. In our case, a total of approximately one man-year has been spent since 2002 in perfecting our engine.

The use of Java or another general-purpose programming language for simulation has other advantages. Limitations and restrictions imposed by the commercial software, many of which will not be apparent at the start of the project, are avoided. A model written in a general-purpose language can be interfaced more easily with databases and control systems. Custom user interfaces can be written to make the model more accessible to non-specialists. Licensing costs and legal restrictions are avoided.

Any threaded programming language can be used to write a discrete-event engine. We chose Java because it has good thread handling routines, promotes readable code, has excellent development tools, and includes a 3-dimensional graphics package (Java 3D).

3. DEVELOPMENT OF THE JAVA SIMULATION ENGINE

There are many ways to create a discrete-event simulation engine in Java as a brief survey of the literature reveals. An up to date listing of all the available simulation software is given in [2] and [3]. Of the ones listed, a total of 10 are based on Java in one form or another: CSIM [4], DESMO-J [5], DEUS [6], DSOL [7], JavaSim [8], JiST [9], JSL [10], SimJava [11], Simkit [12], and Tortuga [13].

Some of the approaches used in these engines are quite sophisticated, with the objective of maximizing execution speed, but they restrict the way in which the user must design the simulation model. Others seek to create the engine with the least complication, but restrict the practical size of the models that can be created. Still others are designed to reproduce the commands and structure of previous-generation simulation languages.

Our objective was to write models with very complex logic and hundreds of model entities that would be re-used for many projects over 10 or more years. Therefore, in our case, ease of use and simplicity were the most important features for our engine, rather than extreme efficiency. With this in mind, we set the following requirements for our engine:

- Use the standard Java virtual machine
- Use the standard Java programming tools
- Impose no artificial program structure
- Promote readable code for simulation models
- Allow both event- and process-oriented model design

We now give a brief description of our simulation engine that is intended to provide the reader with just enough information to understand our approach and the engine's capabilities. Table 1 lists the basic object classes that were created to implement the discrete-event logic within Java.

Table 1. Basic Simulation Objects

Object	**Description**
Simulation	The overall simulation model. Controls reading of input data, starting of the simulation run, termination of the simulation run, and printing of output reports.
EventManager	Maintains simulated time and the list of future and conditional events to be executed.
Entity	The basic object for the simulation. Can be permanent or temporary during the simulation run and can be either active or passive in the model logic.
Process	A sub-class of thread that allows an entity to execute a series of methods in simulated time while other entities execute their own series of methods.

Note that we have separated the two objects Entity and Process. When an Entity is playing an active role in the simulation, it will have one or more Processes underway. When it is playing an inactive role or is temporarily dormant, it will have no Process underway.

Entities can start processes and schedule activities using the basic methods in Table 2.

Table 2. Basic Methods for Entity

Method	**Description**
scheduleWait(*dur*, *pri*)	Stops the execution of the current method for the given duration in simulated time. Can be placed anywhere within a method and can be used multiple times within a method.
startProcess(*method*, *arg1*, *arg2*, *…*)	Calls the given method with the specified arguments, i.e. this.method(arg1, arg2, …); However, it differs from a simple method call in that a new process is created, allowing the method to be executed in parallel to the original method, rather than in series.
scheduleProcess(*dur*, *method*, *arg1*, *arg2*, *…*)	The same function as startProcess except that the given method is called after the specified delay. The new process is created at the end of the delay, minimizing the number of active processes.
while(*condition*) { waitUntil() ; } waitUntilEnded()	Code structure used to create conditional waits. Two new methods waitUntil() and waitUntilEnded() are used. Stops execution of the current method until the given condition is false.

Method	Description
scheduleLast()	Stops execution of the current method until all other events scheduled for the present simulated time have been executed.
getProcess()	Returns the active process executing the method. The methods interruptProcess and killProcess are the only ones that require a process to be identified.
interruptProcess(*processName*)	Interrupts the given process and causes its next event to be executed immediately.
killProcess(*processName*)	Interrupts the given process and terminates it.

The simulation engine allows both the process- and event-orientation to be used freely in the construction of a simulation model:

- The method scheduleWait is the key one for writing a process-orientated simulation model. Unlike traditional programming languages for writing process-oriented simulations, we have made the distinction between starting a new method, which is done in series with the original method, and starting a new process (using the method startProcess), which is done in parallel with the original method. Traditional simulation languages return control to the original method if the called method is halted by a wait, which is equivalent to starting a new process each time a method is called.

- The method scheduleProcess is the key one for writing an event-oriented simulation model. It is somewhat more efficient than using the methods startProcess and scheduleWait because it delays the creation of a new process until the end of the wait.

With a simulation engine of this type, discrete-event simulation becomes a simple extension of the Java programming language.

4. 3-D ANIMATION

In addition to preparing our Java simulation engine, we created a visualization tool for our simulation models using the Java 3D graphics package [14]. The three-dimensional animation runs in real time and the windows showing the animation can be open and closed at any time during a simulation run.

Real time animation is an indispensible tool for testing, verifying, and debugging a simulation model. The simulation model can be paused at any time and the individual Entities queried by clicking on their representations on the computer screen. The user can check model inputs and the present values of the state variables. If necessary, the values of all the internal variables (properties) of the Entity can be displayed. If execution is resumed while these tools are open, the entries are updated automatically as events are executed.

5. APPLICATION TO MINE-TO-PORT SUPPLY CHAINS

The main application of our simulation engine is a generalized model for mine-to-port and port-to-port supply chains called the Transportation Logistics Simulator (TLS). Objects such as ships, trains, stockpiles, berths, shipping channels, shipping routes, conveyors, weather conditions, tides, etc. are included in TLS. Typical products simulated are coal, iron ore, crude oil, refined products, and liquefied natural gas (LNG).

Virtually any supply chain for bulk materials can be simulated using TLS. A given system is specified by a "configuration file" that defines the individual objects in the system, provides input data for each object, and links them together. The size, shape, position, and other graphical attributes of each object are also specified in the file. The configuration file contains a complete description of the system to be modeled and of the simulation run to be executed.

TLS has been developed over more than 10 years of consulting assignments and represents approximately 250,000 lines of Java code. The Java simulation engine, graphics package, and model building tools involve a further 40,000 lines.

In addition to our own consulting projects, TLS has been licensed by four major companies in the oil and gas industry who use it to carry out their own studies.

6. SIMULATING AN ACTUAL SUPPLY CHAIN

An example of a TLS application is the Goonyella coal supply chain[1] in Queensland, Australia. In this system, coal is sent by train on a just-in-time basis from 18 mines to 2 marine terminals. The design throughput capacity is 129 million tonnes per year (Mt/y) of coal exports. Approximately 1,500 ship loads and 13,000 train cycles are required to handle this volume each year. Figure 1 shows an overview of the model. A complete display of the model requires several computer screens and it is necessary to

[1] We thank our client DBCT Management for permission to use their project as an example for this article. DBCTM is the owner of the Dalrymple Bay Coal Terminal, the larger of the two marine terminals in the Goonyella system.

zoom in on individual portions of the model to follow the activities in detail.

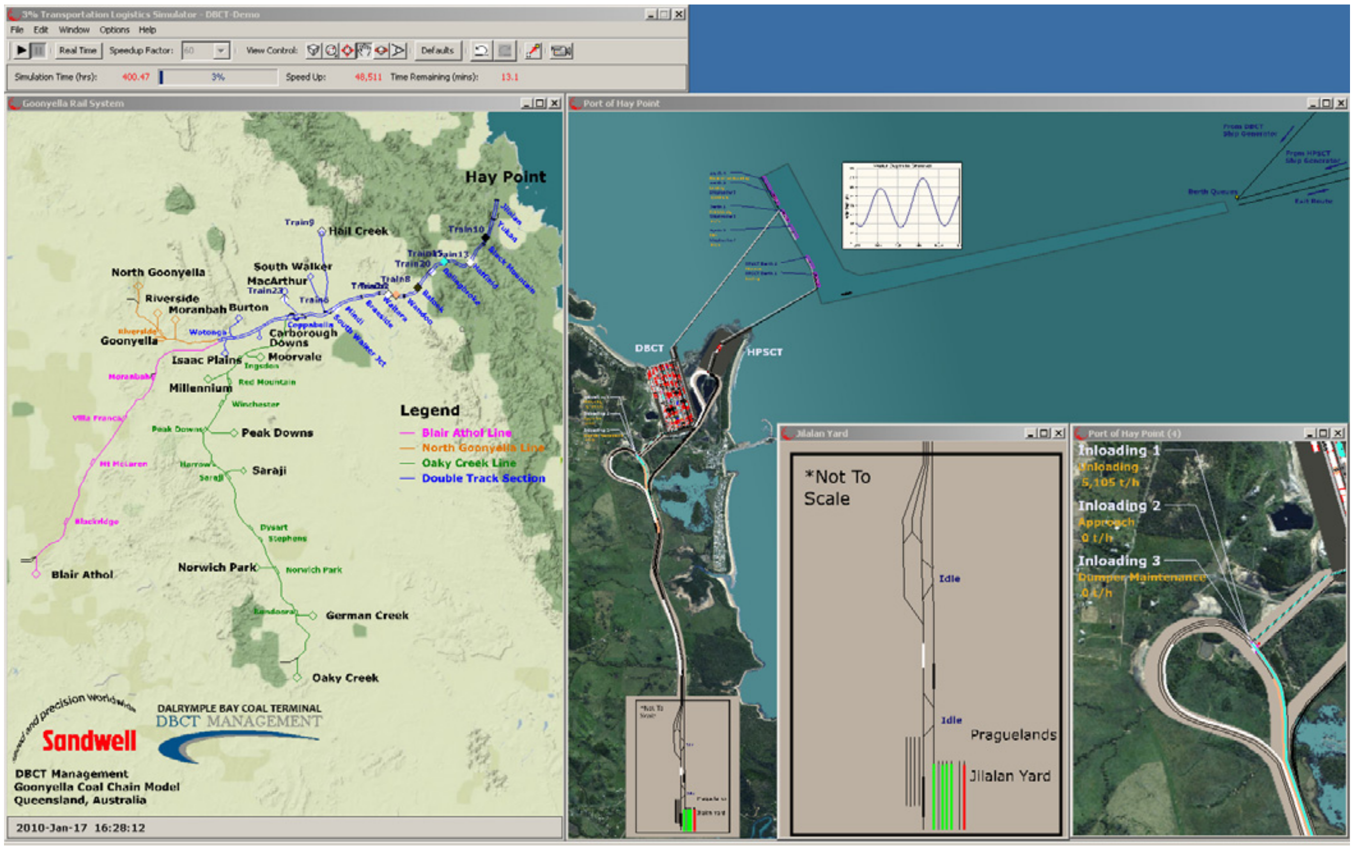

Figure 1. Goonyella Coal Supply Chain Model

The system is modeled to a high level of detail, with approximately 700 permanent entities such as rail signal blocks (258), stockpiles (132), conveyors (39), trains (32), etc. Labels, graphs, and other bystanders in the model are not included in this total.

A one-year simulation run at 129 Mt/y requires the execution of approximately 8 million events and requires approximately 45 minutes of processing time on an Intel i7 operating at 3.0 GHz. It should be noted that event processing by the simulation engine was not the bottleneck for these simulations. Most of the computer's time is spent on complicated routing and logistics decisions.

7. PARALLEL PROCESSING – RESEARCH IN PROGRESS

A supply chain model is a natural candidate for parallel processing: each component of the chain interacts only with its upstream and downstream neighbors. Figure 2 shows the structure of the Goonyella system.

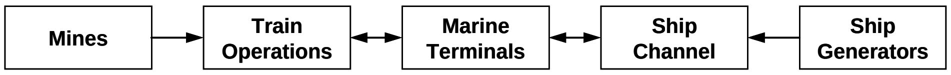

Figure 2. Supply Chain Components

In this example, the 18 mines have no direct interaction with the 2 marine terminals so it should be possible to execute the events for these 20 components in parallel with one another.

To exploit this structure, we experimented with various ways to implement parallel processing in our discrete-event engine. The use of Java threads makes parallel processing a natural extension of our event processing logic. It is just a case of allowing the virtual machine to execute more than one event thread at a time. The challenge was to manage the execution of the parallel processes in such a way that the results were unchanged from the non-parallel case.

After some trial and error, we re-organized the components into the parent/child structure shown in Figure 3, and assigned separate EventManager objects to each component.

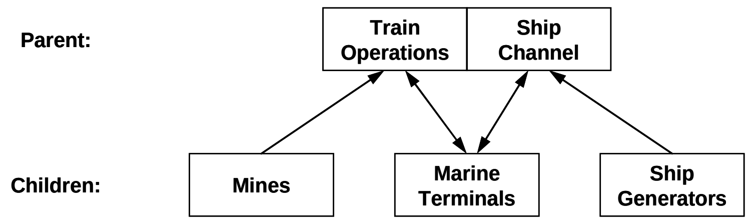

Figure 3. Supply Chain Re-Organized as a Parent/Child Structure

The children components in the lower portion of the diagram have no interactions with each other, and only interact with the parent components in the upper level. Children were restricted to having only one parent, so it was necessary to combine the train operations and ship channel components for the purpose of parallel processing.

One EventManager was assigned to handle both the train operations and ship channel components, while one EventManager was assigned to each of the 18 mines, 2 marine terminals, and 2 ship generators. This arrangement gave a structure with one parent and 22 children.

Each EventManager maintained its own future event list and simulated time. Simulated time was synchronized between parent and child EventManagers whenever it was necessary for an object controlled by one EventManager to communicate with an object controlled by another EventManager. The program logic allowed for the EventManagers that were children of a common parent to be executed in parallel. EventManagers that were parents/children of one another were executed in series so that simulated time could be synchronized between them whenever communication took place.

The logic for executing events was written so that the model results for a multi-processor run were identical to those for a single-processor run. For this to occur, portions of TLS had to be re-written to avoid direct communications between components that were not a parent or child of one another. These forbidden communications were usually the result of shortfalls in the object oriented design or the use of a global variable, and were easy to fix once they were found. To find them, we developed a tool to compare the sequence of events executed during a multi-processor run to those for a single-processor run and to flag the first occurrence of a non-trivial difference.

An increase in execution speed of about 20% was achieved with the tree structure given in Figure 4. The improvement was relatively modest because the train operations and one of the marine terminals were the most complex parts of the model, and these were executed in series, not parallel, because of their parent/child relationship. The small speed-up was the result of executing the mines, ship generators, and the second marine terminal in parallel with the first marine terminal. Although many components were involved, each was relatively simple compared to the rail operations and first marine terminal.

More substantial gains in performance could be achieved by breaking up the rail operations into a double-track mainline and three single-track branch lines. The revised parent/child structure for this structure is shown in Figure 4.

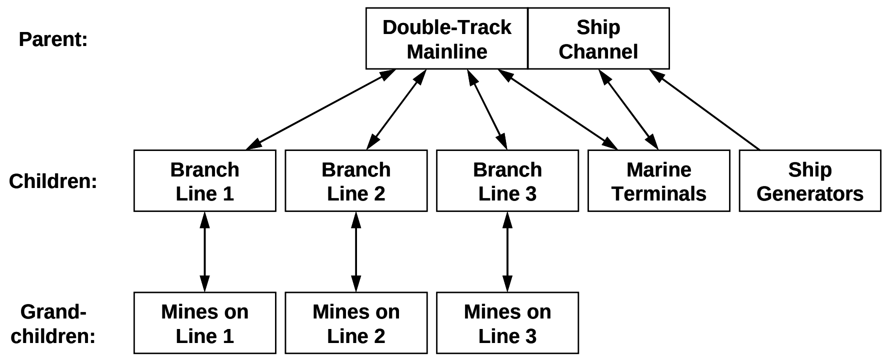

Figure 4. More-Refined Child/Parent Structure

Further work on our TLS software is required to enforce the correct parent-child communication when trains pass between the single-track and double-track portions of the model before we can test this structure.

8. CONCLUSIONS

Java simulation is a viable alternative to commercial simulation software. We believe that it is the preferred choice for applications that are very complex or that are a poor fit to the simulation tools provided by commercial software. These applications are the ones that require a significant amount of programming to be done, and for these cases, the advantages of using standard Java development tools outweigh the features provided by commercial software.

Eight years of consulting experience with a large team of engineers using the software on a daily basis has resulted in a stable and powerful simulation engine. Although many Java simulation engines have been discussed in the literature, we are not aware of any that have been used as intensively or that have undergone as much development.

We have provided parallel processing capability in our engine and have demonstrated that with good application design it is possible to utilize this capability in large-scale models.

If there is sufficient interest, Ausenco Sandwell would consider making our simulation engine available on an open-source basis. We invite you to contact us to discuss opportunities for collaboration.

References

[1] Oracle Corporation. Sun Developer Network. http://java.sun.com/

[2] A.E. Rizzoli. A Collection of Modelling and Simulation Resources on the Internet. http://www.idsia.ch/~andrea/sim/simtools.html

[3] Wikipedia. List of discrete event simulation software. http://en.wikipedia.org/wiki/List_of_discrete_event_simulation_software

[4] Mesquite Software. CSIM. http://www.mesquite.com/

[5] University of Hamburg. DESMO-J.

[6] University of Parma. DEUS. http://code.google.com/p/deus/

[7] Delft University of Technology. DSOL. http://sk-3.tbm.tudelft.nl/simulation/index.php

[8] JavaSim. http://javasim.codehaus.org/

[9] R. Barr. JiST – Java in Simulation Time, User Guide. http://jist.ece.cornell.edu/

[10] M. D. Rossetti. JSL. http://www.uark.edu/~rossetti/research/research_interests/simulation/java_simulation_library_jsl/

[11] University of Edinburgh. SimJava. http://www.dcs.ed.ac.uk/home/hase/simjava/

[12] A. Buss. Simkit. http://diana.nps.edu/Simkit/

[13] Tortuga. http://code.google.com/p/tortugades/

[14] Oracle Corporation. Java 3D. https://java3d.dev.java.net/

Biography

D. H. King is the Manager of the Simulation Department at Ausenco Sandwell. He holds a Ph.D. in Theoretical Physics from the University of Texas at Austin and has devoted his career to simulation modeling since 1979.

Harvey Harrison is a simulation analyst with experience in many types of modeling including transportation demand models, population change models, vehicle micro-simulation and toll revenue models. He has an interest in systems programming and has many contributions to the Linux kernel.