Using Java for Discrete Event Simulation

R. McNab and F.W. Howell*

September 30, 1996

Abstract

A discrete event simulation library has been written in the Java language, based on the SIM++ library for C++. This allows live simulations to be incorporated into web pages and run remotely. This paper presents a performance comparison with the equivalent C++ simulator and discusses advantages and disadvantages of Java as a simulation language.

1 Motivation

The primary purpose of writing simulations in the Java language was to allow "live diagrams" to be incorporated into documents describing the behaviour of computer architectures.

Using Java, other people can experiment with a working simulation model by clicking on a web link. This contrasts with using a traditional simulation language written in Simula or C++, where exporting simulation code requires recompilation and installation on each different machine.

Java incorporates the language features necessary for simulation, notably objects and threads. Current Java implementations compile down to an intermediate byte code, which is interpreted. Thus the main disadvantage of using Java is expected to be longer simulation run times compared with a native C++ compiler. This penalty is quantified in section 6.

SIM++, a discrete event simulation library for C++ written by Jade Simulations Inc [5] has been used for computer architecture simulations as

*Department of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ. Email: {rmcn,fwh}@dcs.ed.ac.uk

part of the HASE project [4, 3] for several years. To allow running simulations on architectures not supported by Jade (such as Linux and the Cray T3D), the library was reimplemented using C++ and standard threading libraries, to produce a new library called HASE++ [1]. HASE++ was used as the basis for the Java simulator [2].

The most attractive part of the HASE environment is the support of animated models of simulations. This feature has also been incorporated into the Java simulator, with the simulation linked in with a graphical model, displayed using a Web browser such as Netscape.

2 A Brief overview of SIM++/HASE++

A HASE++ simulation is a collection of C++ objects (*sim_entities*) each of which runs in parallel in its own lightweight thread. The *sim_system* object controls all the threads, advances the simulation time and maintains the event queues.

Entities communicate and synchronise by passing *sim_event* objects. The primitives of the language are:

- `sim_schedule(sim_port port, double delay, int tag)` sends a message to the entity connected to the port after simulation time delay with the given tag.
- `sim_wait(sim_event &ev)` waits for an event sent using `sim_schedule`.
- `sim_hold(double t)` blocks for t simulation time units.
- `sim_trace(int level, char* ...)` adds a line to the trace file.

This simulation model is well suited to modelling hardware and to modelling parallel software. DEMOS-style process interactions, using `resources` and `waitqs` have been layered atop this model.

3 The Java version

Java is superficially very similar to C++; the class and expression syntax is very similar. The Java documentation even states that if a feature hasn't been fully explained, it can be assumed to be the same as C++. The sections below describe how the differences between the languages affected the simulator development.

3.1 Interface classes

Separating interface from implementation actually requires more effort in C++ than it does with C. “Interface classes” are the standard C++ solution to this problem. These contain an opaque pointer to another class which defines the representation, so any code using the class depends solely on the interface and not the representation. This technique was used in HASE++ to reduce the compilation dependencies; it works, but obfuscates the code.

In Java dependencies are handled more neatly. Linking is done during the initialisation phase of the runtime system, changing the symbolic names to numeric offsets to preserve speed. The storage layout of objects in memory is also deferred until runtime, instead of being determined by the compiler. This means that members can be added to a class without the need for recompilation of classes which depend on it. Java allows access control of class members at the library[1] level. These two points enabled the Java version of the simulation library to do away with the confusing interface classes, leading to a clearer object oriented design.

3.2 Collections

The Java language provides no pointer arithmetic; it was thought that this would be a problem in porting HASE++ to Java, however it proved to be a minor point. As an example, HASE++ makes extensive use of pointers in the way it implements the event queues using a dynamic array class, which was implemented from scratch. The Java version was able to make use of the `Vector` class provided in the standard API packages. The Java programmer can use the arrays built into the language when pointer arithmetic style indexing is needed.

3.3 Threads support

In C++, threads are the responsibility of the underlying operating system and are not part of the language. Various threading libraries are available for different machines; HASE++ uses a `class thread` with separate implementations for Solaris, Linux, and the Cray T3D threading libraries. Semaphores and mutexes are also provided by the threading library.

In contrast, Java incorporates threads into the language. A new thread can be created by deriving an object from `class Thread`, which provides methods for all the standard operations such as starting, stopping, and changing priority.

[1] “package level” in Java terminology

Where Java threads stand out, however, is in their synchronisation primitives. The runtime system ensures that methods in a class which are marked as `synchronised`, do not run concurrently for each instance of that class. Using this mechanism it is simple to make classes "thread-safe", i.e. ensure that they do not show an inconsistent state in a multi-threaded environment. All of Java's standard API packages are written to be thread-safe.

HASE++ uses counting semaphores for two purposes: to protect common data, such as the event queues, from reaching an inconsistent state by being updated by two threads simultaneously, and to control when the entities in the simulation run. The `Vector` class, used in the Java simulation for the event queues, was already thread-safe, however a counting semaphore class still had to be written to control the entities.

3.4 Documentation

One of the bugbears of programming is maintaining up to date documentation of programs. Java includes a very useful tool, `javadoc` for helping with this. It parses a Java program, and automatically generates an `html` document describing the public interface to the classes and their methods. Extra information may be included by using specially formatted comments in the program source. The tool was used to produce the technical documentation [6] of the SimJava library. An extract is shown in figure 1.

3.5 I/O

Java supports the I/O required in a simulation for reading data files and generating trace files using a set of `streams` classes similar to C++. There are severe restrictions, however, on the file I/O when running a Java program within Netscape. Security restrictions mean that reading files may only be done using a complete URL, and writing files is not possible directly. This is not really a problem for the small demonstration simulations, where the tracefile output would be piped straight into another display applet, rather than to a file.

A workaround to enable file saving from Netscape uses the fact that only remote classes are subject to the security restrictions. All the file I/O would be done through a local class which the user previously downloaded and put in their `CLASSPATH` for Netscape to find. This is a messy solution which breaks the built in security mechanisms, and is not really for the novice or casual browser.

public class **Evqueue**
extends Vector

This class implements an event queue used by the Sim_system to manage the list of future and deferred Sim_events. It works like a normal FIFO queue, but during insertion events are kept in order from the smallest time stamp to the largest. This means the next event to occur will be at the top of the queue.

The current implementation is uses a Vector to store the queue and is inefficient for popping and inserting elements because the rest of the array has to be moved down one space. A better method would be to use a circular array.

See Also:
Sim_system

Constructor Index

- **Evqueue**()
  Allocates a new Evqueue object.
- **Evqueue**(int)
  Allocates a new Evqueue object, with an initial capacity.

Method Index

- **add**(Sim_event)
  Add a new event to the queue, preserving the temporal order of the events in the queue.

Figure 1: An extract from automatically generated technical documentation.

3.6 Statistics

HASE++ handles statistics gathering and random number generation by using a set of statistics classes. These were straightforward to implement in Java, using the standard maths library. A surprising find was that it had a normally distributed random number generator as part of the library.

3.7 Java run-time systems

Sun's Java Development Kit was used for development; this provides a standalone applet viewer for running programs. Applets may also be run from within a Web browser such as Netscape. This introduces an initial "load time" as the classes are loaded on demand across the network before the simulation starts, but subsequent runs start instantly.

As it is an interpretted system, Java provides better run time error handling than C++, including array bounds checking and null pointer exceptions, both reported with the source file line number where the exception occured. This makes it a more forgiving development environment than C++.

4 Integration with C++ tools

Figure 2 shows how the Java simulator may be linked with existing C++ based tools.

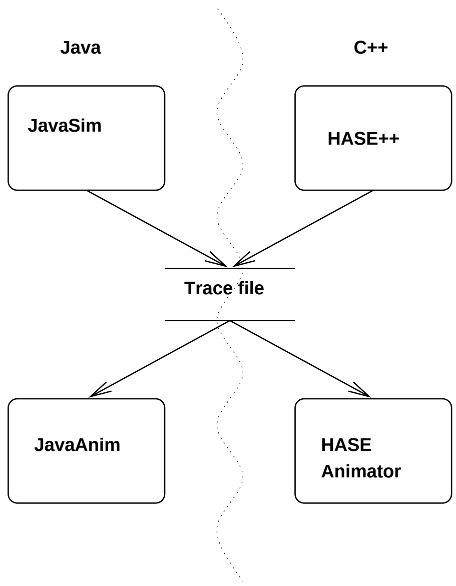

Figure 2: Linking Java simulations with SIM++ simulations

Because there is a standard trace format, traces produced from a HASE++ simulation may be read into a Java display, and vice-versa. Thus the Java simulation environment may be used for displaying results from a long running HASE++ simulation.

All of the HASE++ functions have been ported to Java, and in the large part their usage is identical. Naming remains the same except for classes, which follow the Java convention of capitalising the first letter. The major changes each illustrate a language difference between Java and C++, and are detailed below:

- Java does not allow global objects external to classes, so all the members of the controlling `Sim_system` class had to be made static and an `initialise()` method added.

- HASE++ allows data to be sent between entities, by passing the `sim_schedule()` method a `(void *)` pointer and a data size. In Java, however, there is no `sizeof()` operator, so the data must be enclosed in a class, then passed as a generic `Object` reference.

- HASE++ uses `printf()` and `varargs.h` for its `sim_trace()` method to build up traceline strings. Java allows numeric variables to be concatenated onto strings using the '+' operator. So the kludgy
`sim_trace("There were %d counts in %f msecs", count, time);`
becomes the neater
`sim_trace("There were "+count+" counts in "+time+" msecs");`

Unfortunately porting a HASE++ simulation to Java is not a mechanical process, in the same way as porting HASE++ itself was not. In the best cases, however, where the simulation makes little use of C++ language features and consists mostly of calls to HASE++ functions, porting is simply a matter of following the syntax changes detailed above.

5 Graphics

The main aim of producing a Java based simulator was to allow inclusion of live simulation models into web documentation (with the same advantages over plain diagrams that Science Museum exhibits with buttons have over static displays).

The current animation facilities are illustrated in figure 3, which shows an animation of a task farm simulation as it would appear in a page viewed with a Java enabled web browser. Text boxes and buttons allow the user to control the simulation and change initial parameters. Entities and ports have their own icons loaded from GIF files. The icon can be changed to represent the current state of the entity, and other entity parameters can be displayed as text. Messages passed between entities are displayed as squares which travel along the connecting lines, the number attached to the square is the message tag.

6 Performance

6.1 Simulation

To compare the performance of the Java and C++ versions of the library, a simple simulation was written in both languages and the execution time measured. The Java version was run as a stand-alone application, as Netscape applet on a Sun SPARCstation 5 under Solaris, and as a Netscape applet on a Pentium 133 under Windows NT. The simulation contains two entities which pass 200 messages between them, a simple example for comparative purposes. The results are shown in table 1.

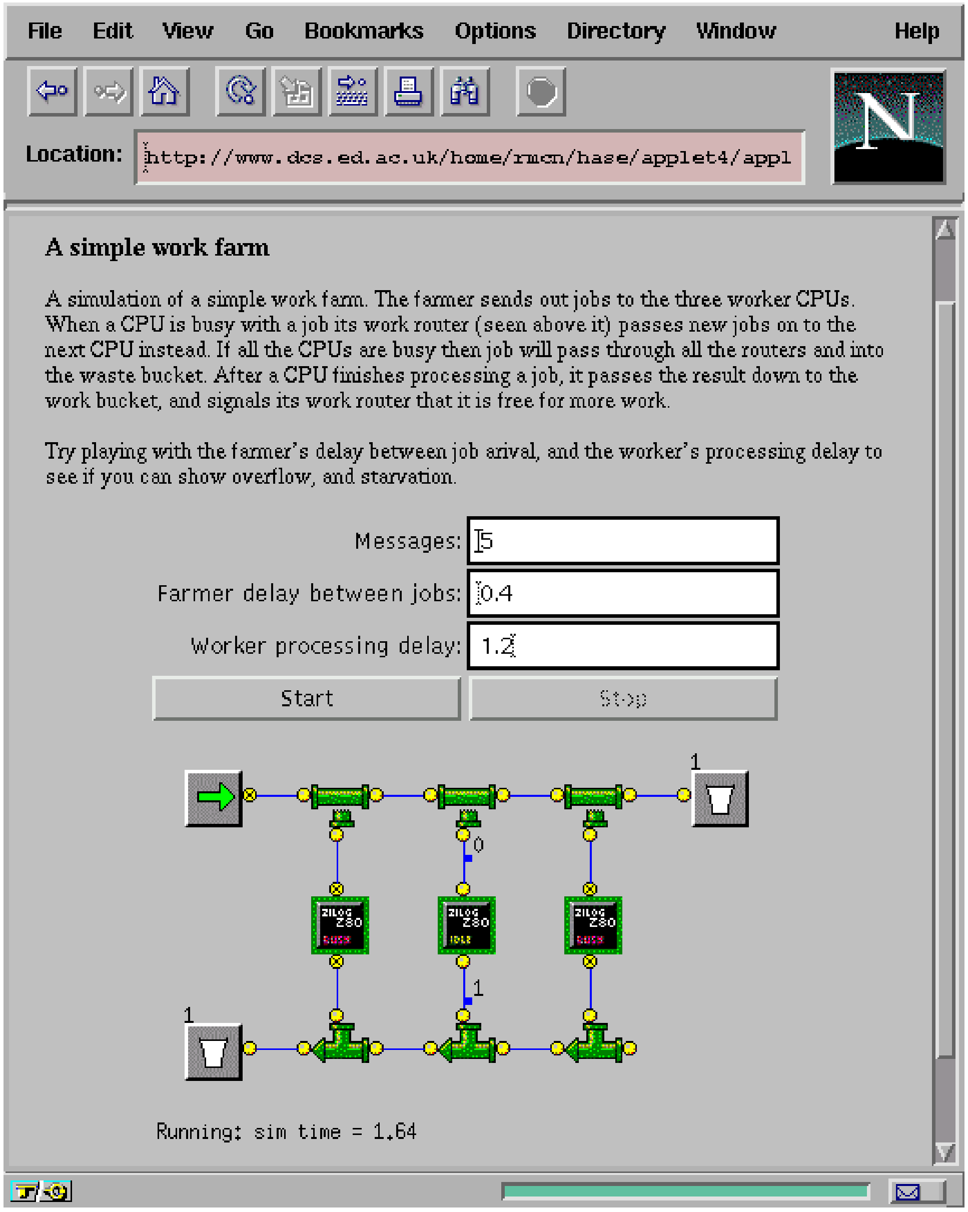

Figure 3: Animation of a task farm simulation

Platform	Average execution time over 5 runs
Solaris C++	1538 ms
Solaris Stand-alone Java	12910 ms
Solaris Netscape	11214 ms
Windows NT Netscape	9341 ms

Table 1: Simulation execution times

The results show that the simulation ran around ten times faster under C++, and the stand-alone Java and Netscape were roughly equal.

6.2 Threads

Each simulation object runs in its own thread, so the performance of the underlying threading system is important for large simulations. The Java runtime system uses the underlying operating system for thread support, or its own software emulation if the OS does not support threads. The performance of a threaded Java program running under Solaris is shown in figure 4. The program launches N concurrent threads, each of which runs a computation loop. The graph shows the total time for N threads to complete divided by the number of threads. Optimal thread performance occurs at 128 threads, and a sensible maximum is 2000, which is consistent with native Solaris threads.

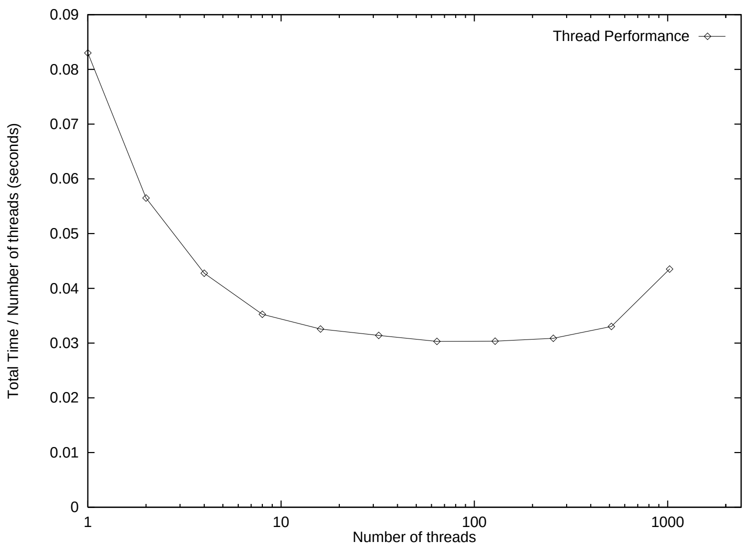

Figure 4: Thread performance

7 Conclusions and Further Work

The portability of Java makes it an attractive simulation language. The factor-of-ten performance hit compared to the C++ version is the price paid, so Java is a realistic option for simulations where visualisation is as important as speed. Java is a very young language, and faster implementations are likely in the future.

As an accessible environment for displaying simulation results, Java is much more convenient than using native applications. It allows experimentation with animated simulation models from a standard web browser.

Future plans include implementing more Java modules for displaying graphs and timing diagrams, and providing 3D models using VRML-2.

References

[1] F.W. Howell. Hase++: a discrete event simulation library for C++. Available from `http://www.dcs.ed.ac.uk/home/fwh/hase++/hase++.html`, Feb 1996.

[2] R. McNab. SimJava: a discrete event simulation library for Java. Available from `http://www.dcs.ed.ac.uk/home/rmcn/simjava`, July 1996.

[3] F.W. Howell and R.N. Ibbett. *STATE-OF-THE ART IN PERFORMANCE MODELING AND SIMULATION Modeling and Simulation of Advanced Computer Systems: Techniques, Tools and Tutorials, edited by Kallol Bagchi*, chapter 1:Hierarchical Architecture Simulation Environment, pages 1–18. Gordon and Breach, 1996.

[4] R.N. Ibbett, P.E. Heywood, and F.W. Howell. Hase: A flexible toolset for computer architects. *The Computer Journal*, 38(10):755–764, 1995.

[5] Jade Simulations International Corp., Calgary, Canada. *SIM++ User Manual*, 1992.

[6] R. McNab. SimJava package documentation, Available from `http://www.dcs.ed.ac.uk/home/rmcn/hase/doc/Package-hase.html`, July 1996