Fornax: Web-based Distributed Discrete Event Simulation in Java

Berry A. W. van Halderen   Benno J. Overeinder

Department of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, the Netherlands
{berry,bjo}@wins.uva.nl

Abstract

In this paper we present an environment for the development of Java-based discrete event simulations. An integrated approach is taken that combines discrete event semantics with Java and exceeds the expressiveness of discrete event libraries. The environment supports advanced techniques including distributed evaluation with conservative or optimistic event execution using the Time Warp protocol. We present some performance results and discuss the difficulties encountered with Parallel Discrete Event Simulation.
Keywords: Java, Discrete Event Simulation, Distributed Computing.

1 Introduction

The large interest in Java over the last years is primarily motivated by its role in interactive World Wide Web presentations and network services. However, the various features of Java as a language for network computing makes Java a candidate for writing portable resource-intensive distributed or parallel applications in general. The hardware and software platform independence of Java, and the support of concurrent objects match the requirements for seamless metacomputing environments that offer coordinated networks of distributed heterogeneous resources.

Notwithstanding the potential of Java in scientific computing, the use of the language in this field is still quite limited. In this respect, the use of Java in scientific computing is primarily to coordinate Web-based access to repositories or computing resources. The main obstacle hampering the use of Java to implement scientific computation kernels is the fact that Java is not yet a high performance language. Currently, the main application of Java in scientific simulation has been in the area of visualization and animation of simulation results, where the computational intensity of the simulations is small compared to the visualization time.

A distinct application area in scientific computing is Discrete Event Simulation (DES). In recent years the observation that components in many physical systems behave and interact in an asynchronous manner has reinforced the importance of discrete event simulation. The protocol formulated by the DES scheme is able to capture the asynchronous characteristics that are essential to the physical model. Discrete event simulations are known to be computationally intensive, where the control of the computational process is responsible for a significant amount of the complexity and execution time. In different studies that evaluate the potential of Java as a high-performance computing language, it is indicated that certain improvements are necessary for successful usage [2, 3]. On the other hand, Java offers an efficient and flexible simulation engine for DES with its complex control of the computational process.

In general, simulation studies and experiments are pursued with specific simulation languages such as MODSIM III, SIMULA 67 or Pearl [7], or with general purpose languages such as C or Fortran that are extended with simulation libraries. Both approaches have their pros and cons. Simulation languages automatically provide most of the features needed in programming a simulation model, and by the design of the simulation language, it promotes a natural framework for simulation modeling. General purpose programming languages have the benefit of being generally available and familiar to programmers. Furthermore they support greater programming flexibility.

Extending Java to a discrete event simulation language combines the best of both worlds. Its popularity makes it generally available and a vast amount of programmers is familiar with Java. The object-oriented concepts of Java allow for a natural extension of the language to incorporate primitives for discrete event simulation and to promote a framework for simulation modeling. Apart from the modeling flexibility and the ease of model development, Java provides versatile primitives for animation to monitor the dynamics of a simulation. The suitability of Java for discrete event simulation has been recognized by a number of projects, which resulted in various simulation environments [1, 5, 6]. Our approach differs from the other initiatives in its aim to support an integral conceptual framework for discrete event modeling, while the simulation engine efficiently manages and executes the events in the simulation.

In this article we will first describe the conceptual modeling framework for DES as used within the Pearl simulation language [7]. Next, we show how this framework is incorporated by a natural extension of the Java language. Here, we also discuss the aspects of the different levels of concurrency in the Java execution model, and the potential exploitation of concurrency by parallel and distributed execution of the simulation. In section 5 the most important implementation aspects of the simulation engine are presented. Some performance results and comparisons with other systems are made in section 6 and we conclude with some discussion in section 7.

2 The Discrete Event Model

In a discrete event simulation model, the real-world system is modeled by set of objects which are usually known as *entities*. These entities are all discrete objects, each being separate from all the others. Entities possess *attributes* which are properties that affect the behaviour of the entities within the model. An *activity* represents a time period of a specified length. Entities may be in one of two states: either they are busy, engaged in some activity, or they are idle, doing nothing but waiting for the arrival of an event in a *queue*. An *event* is defined as an instantaneous occurrence that may change the state of the system.

Three different approaches are in vogue for the modeling of discrete event systems, the so-called *world views*. The differences between the world views lie in the way in which manner the model processes the events. The differentiation among world views is best captured using the concept of locality [8]:

Event scheduling provides locality in time: each event routine in a model specification describes related actions that may all occur in a single instant.

Activity scanning provides locality of state: each activity routine in a model specification describes all actions that must occur due to the model assuming a particular state.

Process interaction provides locality of object: each process routine in a model specification describes the entire action sequence of a particular model object.

2.1 The Pearl Discrete Event Model

The Pearl discrete event model adopts the process interaction world view. The entities own attribute data exclusively, i.e., data is not shared among other entities, and interact with each other by exchanging event messages. Event messages can be sent in two modes: *synchronous* and *asynchronous*. When an entity sends a synchronous event message, the entity immediately waits after the send until it receives a result back. With asynchronous event messages, the entity continues directly after sending the event message, so it does *not* wait for a result to be returned.

The progress of simulation time, or *virtual time* is explicitly modeled in the Pearl model. This means that an activity that represents a time period is modeled by blocking the entity for the specified time. The explicit modeling of elapsed virtual time also has consequences for synchronous event messages. If the entity that has received the synchronous event message, increases its virtual time before it replies, then the virtual time of the other object is also increased with the same amount because it was waiting for the reply.

In the Pearl discrete event model there is a natural mapping of the entities in the model to instances of classes in object-oriented languages. The events in the system are modeled by *method invocations*, and are therefore also called event methods. The exchange of events in the Pearl model is an extension of the object-oriented method invocation. Progress of virtual time is realized by a primitive in the entity which models the elapse of a specified time.

In the process interaction model each entity is an active object where the entities perform operations concurrently. This means that each entity has its own thread of control, which is either processing the activity related to an event or waiting for events. In the Pearl model, only the thread of control of the entity itself will process events one at a time and only if the event is expected.

2.2 Parallel Discrete Event Simulation

The Pearl discrete event simulation model possesses a certain amount of concurrency, originating from the fact that each entity is processing concurrently using its own thread of control. This concurrency can be exploited in three different ways.

Using one central simulation clock. The parallelism can be exploited within one single program on a multiprocessor using multiple Light Weight Processes (LWPs). The simulation time is controlled by a single instance of the simulation clock.

With one distributed clock. Instead of on a single shared memory multiprocessor, a coarser granularity of parallelism can be exploited by dividing the set of entities over two programs on different machines. The simulation is governed by instances of a distributed simulation clock present on each machine, which amongst them implement *one* single simulation clock.

Asynchronous distributed clocks. In this case the distributed simulation clocks independently advance the virtual time. Instead of a global notion of a virtual time, the distributed clocks can advance independently, which increases the potential to exploit the available concurrency.

Parallel discrete event simulation (PDES) is essentially concerned with the correct ordering or scheduling of the asynchronous execution of events over distributed or parallel systems. The essential problem PDES must solve is the *local causality constraint*: a discrete event simulation, consisting of processes that interact exclusively by exchanging event messages, obeys the local causality constraint if and only if each process executes events in nondecreasing simulation time order. There are basically two approaches to impose the correct temporal order of the asynchronous event execution: *conservative* and *optimistic* methods [4].

The basic problem conservative mechanisms must address is to determine which event is safe to process. Processes that do not contain any safe event must block. This behavior can result in deadlock situations if no appropriate precautions are taken (e.g., deadlock avoidance or detection and recovery). Optimistic methods do not strictly adhere to the local causality constraint, but rather use a detection and recovery mechanism. Whenever the incorrect temporal order of events is detected, a rollback mechanism is invoked to recover from the erroneous computation.

The Pearl model of process-interaction oriented discrete event simulation is a simple and elegant method of modeling entities. Here, the event interaction between entities is modeled by events using method calls without sharing data. Both these aspects facilitate concurrent evaluation of events, however careful coordination to preserve causal constraints is required. The pure object-oriented approach of the Pearl model using, active processes for each entity, makes it a good candidate for implementation in Java.

3 Discrete Event in Java

We have implemented the Pearl discrete event simulation model, as described in the previous section, using Java. The complete Java package consists purely of Java class files and has been named *Fornax*, a constellation which stands for "the furnace." Our package forms the furnace in which simulations can be prepared. The design objectives are to match the DES process interaction programming model as closely as possible.

Java provides the object-oriented programming structures to implement entity, event and simulation time control objects. To add discrete event simulation semantics we need to add the following functionalities:

Generating events. When an event method of an entity is called the appropriate event should be generated, instead of the direct execution of the actual method body.

Event handling. The handling of an event should be driven by the internal simulator, and the same method that generated the event should be executed.

Waiting for events. An entity must be able to block for an event to be handled or for a virtual time period to pass.

Simulation time. Events for a future simulation time need to be scheduled and the virtual time needs to be advanced when no events for the current simulation time are left.

To express these functionalities in a Java application, additional semantics on a language level are required to offer the same conceptual framework as simulation languages do. This goes beyond traditional library implementations for DES in Java. Using the standard Java language offers the advantages of portability, Web-based simulation, integration with other packages (e.g., visualization), and that the programmer does not have to familiarize with another language.

In the next paragraphs we describe how the extra functionalities are incorporated in Java, effectively augmenting the semantics of the language without changing the actual language.

Entities and events. By extending from a predefined class within the Fornax package, the programmer defines DES entities. The predefined class adds the functionality to wait for events to occur, wait for the virtual time to advance and query some other state. Furthermore it implements a basic framework in which the DES entity will run. The interaction between entities by event methods requires that the method call (scheduling an event) and the actual method execution (handling an event) must be decoupled. The arguments in the event method calls must be passed by value in order to prevent shared data between entities.

The decoupling of the method call and the execution is realized by introducing an encapsulating class, that overrides each event handler method. When a method is called by another entity, the associated event is generated and the execution of the original method body is scheduled. When the time has come to process the event, the thread belonging to the entity will call the underlying overridden method. The approach of decoupling the method call and the execution enables the generation of events using a normal method class of the target entity, without changing the language.

The encapsulating classes are generated in two compilation phases. In the first phase, classes that represent entities (defined by the user programmer) are compiled into class files using a Java compiler. In the second phase, the resulting class files are examined. This examination, based purely on "Java core reflection," results in the generation of the encapsulating class definition for each entity extending from the original.

This two step compilation is somewhat similar to the generation of code by the Remote Method Invocation (RMI) compiler. Both the RMI and the Fornax technique marshal method arguments into a message to be delivered at the target. The main difference is that no server code needs to be generated but instead additional code for generating and scheduling events is needed. Furthermore, the definition of a separate interface for the server is unnecessary.

Associated with each entity is a thread which is spawned by the system when the simulation is started. This thread processes events sent to the entity when the user code explicitly waits for such an event. The instruction to wait for an event to occur is provided by an inherited method in the entity's superclass.

Simulation time. The simulation time is represented by an object controlling the entire simulation, called the *simulation time control object*. Every event is passed to this control object, which delivers the event at the indicated simulated virtual time to the proper entity event queue. The sequential bottleneck that the simulation time control object forms is tackled by using PDES. When all entities are waiting for events, the thread of the simulation time control object is activated to advance the virtual time. Entities can also wait for a period of simulated time to pass. A predefined queue in the superclass of each entity, for which an event is scheduled for the indicated period, implements waiting on simulation time.

4 An Example

In the previous two sections we have described the concepts and basics of the Fornax discrete event system in Java. In this section we will put these ideas together and construct a working example of a simple simulation of a client-server entity pair. The implementation details of the simulation kernel are described in the next section, here we will focus mainly on the semantics used in the system.

During the construction of the entities, the attributes of the entity are passed as arguments together with a reference to the simulation time control object. The superclass constructor performs the necessary administration to register the entity to the set of entities.

When all simulation entities are created, an initialization routine provided by the superclass of the entity is executed. An initial topology of entities is created by specifying the neighboring entities as the argument to the initialization routine. Separating the construction of the entities and the creation of a topology supports a more flexible topology specification and removes inter-entity dependency. After initialization, the simulation is started by the simulation time control object that causes the threads associated to each entity to start executing the main loop of that entity.

```
public Server(Fornax fornax, double speed) {
   super(fornax);
   ...
}
...
Queue Read, Write, Prefetch, Cancel;
public int Read(int nbytes) {
   waitfor( f1(nbytes) );
   return 0;
}
public void Write(int nbytes) {
   waitfor( f2(nbytes) );
}
public void Prefetch(int nbytes, Client peer) {
   long t;
   t = waitfor( f1(nbytes), Cancel );
   if(t == f1(nbytes) )
      peer.Deliver(f3(nbytes), nbytes);
}
public void Cancel() { /* no-op */
}
public void main(Entity peers[]) {
   for(;;) /* forever */
      waitfor(Read, Write, Prefetch,
              Cancel);
}
```

```
public abstract class Client extends Entity {
...
Queue Deliver;
public void Deliver(int nbytes) {
   ++counter;
}
abstract void Deliver(long t, int nbytes);
public void main(Entity peers[]) {
   Server svr = peers[0];
   long t = (new Random()).nextInt();
   for(int i=0; i<niter; i++) {
      svr.Read( bytes[i][0] );
      svr.Write( bytes[i][1] );
      svr.Prefetch( bytes[i][2], this );
      waitfor( t2 , Deliver);
      svr.Cancel();
   }
}
public void main(String args[]) {
   Fornax fornax = new Fornax();
   Client clt = new Client(fornax, 10);
   Server svr = new Server(fornax, 1.0);
   Entity peers[] = new Entity[0];
   svr.start(peers);
   peers = new Entity[1]; peers[0] = svr;
   clt.start(peers);
   fornax.start();
   System.out.println(fornax.getTime());
}
```

Figure 1: Example code for two entities and the main program to create and start the simulation. The Fornax object is the simulation time control object.
The Server entity contains four separate event queues. In the main loop, the server simply waits and processes any type of event. The `Read` event is synchronous, forcing the generator of the event (who calls the `Read` method) to wait until a value is returned by the return statement. In this case the simulation time will advance. In the other event handlers, the peer entity is allowed to continue, and does so within the same simulated time frame. The `Prefetch` event handler waits for a certain simulation time to pass or until the `Cancel` event handler has been invoked. If the Cancel event did not occur (waitfor returns the time waiting on one of the events) a Deliver event is sent to the peer, that was passed to it within the prefetch event parameter list.
The Client entity performs a number of Read, Write and Prefetch requests to the Server entity, to which it is connected by the neighbor list. It counts the number of Prefetch requests which could be satisfied within a period `t2` from the start of the request.

In the example, one Client and one Server entity objects were created. A fragment of the Server object code is shown in figure 1. Event queues, which can be used to wait for an event to occur, are defined by declaring `Queue` fields for each event method using the same name. Asynchronous event handlers are declared using methods returning "void." The synchronous event handlers, which return a value, cause the calling entity to block until the event has been processed.

In our model an event is processed by the thread of control belonging to the entity itself, and only if the entity is waiting for that specific event to occur. All waiting in the simulation is done using the `waitfor` method, provided in the superclass definition of each entity by the simulation package. The waitfor method is a blocking call, which waits for one of the specified events to occur, or for the simulation time to advance for a specific amount of time. For example, the statement `waitfor(20,E)` waits for event `E` to occur, or for the simulation time to advance 20 units, whichever happens first. If event E arrives before the 20 units have passed, the event handler method is called and only after this method returns the waitfor statement completes (returning the simulation time spend waiting for the event to occur). The event method is allowed to further advance the simulation time.

The way in which the Server entity calls the Client entity Deliver event method shows how an event can be generated that has to be delivered on a future simulation time. The first parameter indicates the amount of simulation time that has to elapse before the event is scheduled for execution. Compare this with the implementation of the Client. The called event handling method is declared as an abstract method with an additional parameter indicating the simulation time after which the event should occur. The implementation for this method is automatically generated in the compilation phase. The resulting implementation of the method enqueues an event that is eventually handled by the original Deliver event method.

5 Implementation

5.1 The sequential simulation kernel

Generating Events. A consequence of event methods in the Pearl model is the decoupling of method call and execution. When the thread of an entity calls a method of another entity, an event needs to be queued instead of calling that method directly. We solve this by overriding each event handler method defined by the programmer. In the compilation process, each entity class is subclassed by a new class which will take the role of the original entity class (and bears the original entity class name). This process is completely transparent to the application programmer.

```
public class Server extends ServerEntity {
   public void Prefetch(int nbytes, Client peer) {
      if(externalUse()) {
         Object args[] = new Object[2];
         args[0] = new Integer(nbytes);
         args[1] = peer;
         send(Prefetch, args);
      } else
         super.Prefetch(nbytes, peer);
   }
```

Figure 2: Generated code for an overridden method as produced in the second phase compilation. The `ServerEntity` class is the original class written by the programmer, which is renamed in the two phase compilation.

Figure 2 shows the code generated for one of the methods in the Server entity in section 4. If the Client entity calls a Server event method then the objects are marshalled into an array. The `send` method (provided by the super class of each entity) then creates an event, cloning non-primitive and non entity class events to ensure parameters are passed by value, and schedules the event in the specified queue using the simulation time control object. The event is processed by calling the overridden original method.

Scheduling Events. To implement the scheduling of events we need to be able to: (i) wait for an event, (ii) indicate event arrival, and (iii) manage simulation time.

Java has incorporated multi-threaded support, consisting of abstraction of threads using the Thread and Runnable classes and using monitors, which are highly suited to implement the deliverance of events and to control the virtual time clock. A monitor is a critical section of program code, in which only one thread of control is allowed to enter at the same time. In Java this is designated by a so-called synchronized program block, which is guarded by a object specified by the programmer.

The monitor facility (synchronized sections) can be used to implement the deliverance of events. When an entity instructs to wait for an event to occur, the thread synchronizes (i.e., enters a monitor) with the queue (obtaining a lock for it). It checks whether an event has already been delivered, if not, it performs a wait on the queue until the event has been delivered (releasing the lock temporarily). When an event is delivered to a queue, a notify instruction on the queue object will wake up the possible waiting thread.

All events that are generated during the simulation process are queued using the simulation time control object. Events sent with a simulation time larger than the current simulation time are stored by the controlling object. If an entity needs to wait for a virtual time to pass, it simply queues an event for this virtual time with as target a queue dedicated for this purpose, which is defined within each entity's superclass. Waiting for simulation time then is the same as receiving an event.

The simulation time is advanced when all entities have processed the events they are waiting for and each entity is blocked. A count of the number of entities blocked is kept collectively. The counter is decremented whenever an entity is about to block on the arrival of an event. The counter is incremented when an event is inserted in a queue on which an entity is blocked. Once this counter reaches zero, indicating that all entities are blocked, the simulation time control object is signaled. The thread associated with that object will retrieve the event with the smallest simulation time and schedule the event for execution.

From the implementation aspects described above, it is apparent that Java provides powerful multi-threading primitives which enables an implementation of the Fornax DES simulation event handling. The creation, scheduling and invocation of the handler method of events is hidden from the programmer in the superclass of the entities.

5.2 PDES using Java

A Java implementation of the Pearl model for DES allows for the exploitation of all three levels of concurrency as presented in section 2.2. Since each entity is implemented using a thread, a Java Virtual Machine (JVM) which uses multiple LWPs can use the most simple form of parallelism. The exploitation of LWP parallelism comes essentially free in Java when using threads.

One Distributed Clock. To efficiently use the second level of concurrency, where entities are distributed over multiple processes running on different machines, we can make use of Java's networking capabilities. The distributed implementation of DES with one distributed global simulation clock requires: (i) a distributed implementation of the simulation time control object, (ii) distribution of the entities, and (iii) the ability of entities to transparently send events to non-local entities.

A distributed implementation of a simulation clock is straightforward when one single virtual time is maintained. The process that starts the simulation maintains the master clock and the other clocks forward requests to advance simulation time to the master. We can use Java's object serializability capabilities to implement the distribution of the entities. All entities are generated on one machine and can, before the simulation is started, be written to a stream to other machines. This requires no additional programming effort on the application programmer site, since object serializability is transparent for the entity class.

In the design of the DES model, the only interaction between entities is through events. This improves not only the modeling quality, but also enforces the separation of the entities into objects which can be distributed. Since all communication between entities is scheduled by the simulation time control object, only this object has to be implemented in a distributed manner.

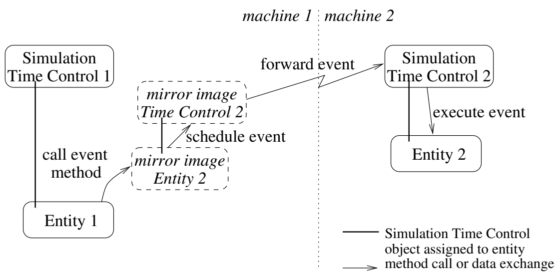

Figure 3: Forwarding an event to non-local entities by mirror stub-entities and simulation clocks.

Figure 3 shows how an entity residing on one machine that sends events to a remote entity. The DES runtime system maintains a list of all entities that can be referenced. The local DES entities run normally, but for non-local entities a mirror of the remote entity is created, which only operates as a stub interface. Events are always scheduled through the simulation time control object, which reference is kept by each entity. By assigning a different simulation time control object to non-local entities, the event can be communicated over the network to the machine with the actual entity's implementation.

Asynchronous Distributed Clocks. The third level of concurrency can be obtained by using asynchronous clock updates. In the optimistic evaluation of events, the distributed clocks advance their virtual time independently. This can cause the "out of simulation order" execution of events. The different distributed instances of the simulation time control object need to be able to detect when they made a wrong assumption in advancing the simulation time. They can do this by simply comparing their own simulation clock against incoming event time stamps and from this determine which objects have violated the local causality constraint and need to be "rolled-back" in simulated time.

A rollback in simulated time is achieved by restoring the state of the entities that have changed state (i.e., processed an event) since that simulation time. The local simulation time control object can detect which entities need to be rolled-back to which simulation time frame.

The role of an entity in optimistic PDES is to save its state each time an event is about to be processed and to be able to restore this state. The state of an entity comprises the data in the object as well as the thread associated with the entity. The data in the object can be stored and restored using Java's serialization and object streams. Thread status however, cannot be saved up to now.

Instead of saving a thread status, we need to ensure that when saving or restoring the state of an entity, the entity's thread is in the same state. The general approach in PDES is to ensure that the entity's thread of control is in a well defined program state at which the entity's state can be both saved and restored. This can be achieved by restricting the main loop to contain a regular visited point in which the entity waits for events. This restriction still allows to express most scientific simulations but harms the conceptual modeling framework.

6 A Qualitative and Quantitative Evaluation

Discrete event simulation is a powerful simulation method to rapidly experiment with different models and parameter settings in the simulation application. Both libraries and special purpose language have been developed in the past to implement structures common to all discrete event simulations and to provide a better programming framework for the application programmer by implementing the DES paradigm in a programming language. In this section we compare the performance of our system related to other DES systems and point out the differences in approach.

While Java's performance has not grown to maturity, comparison with a commercial DES language like MODSIM III shows that Java is able to compete where complexity in event handling dominates over normal computing. In figure 4 the runtime of a simple—asynchronous event intensive—application is shown. The benchmark performs hardly any computing action. It is clear that for any substantial computing, Java would not yet be able to compete with the compiled MODSIM code. The performance of the Java implementation is also very sensitive to the behavior of the thread scheduling process in the JVM, with large performance variations for different JVM platforms. Still from these results we can see that Java can play a clear role in discrete event simulation, although scheduling performance leaves room for improvement.

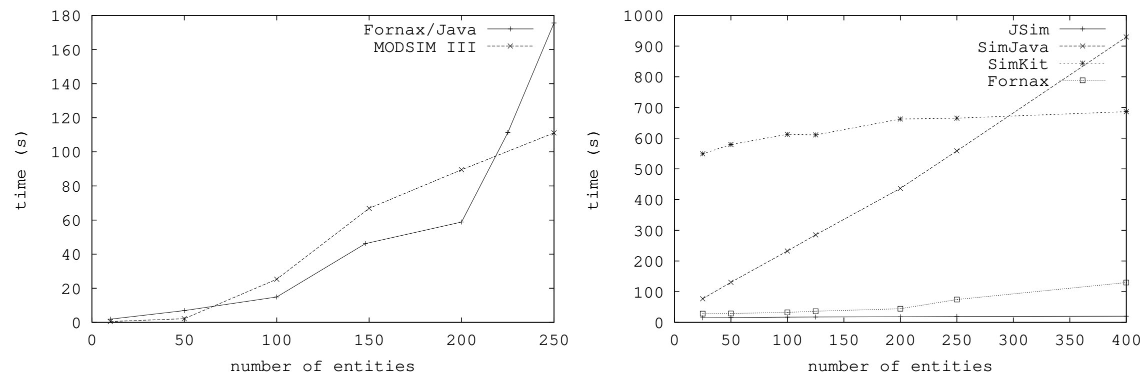

Figure 4: The left figure shows a performance comparison of our DES Java implementation Fornax with MODSIM III on a Sun Ultra II. The benchmark application consists of n entities in a ring topology. Each entity sends an event to its successor in the topology and then waits for a random amount of simulation time. During this time it may receive an event from its predecessor. Each entity repeats this process a 1000 times.
The right figure shows a comparison of the Fornax simulation environment with other Java based DES packages. The application used is an Ising spin simulation, which is an application within the computational science field that simulate ferromagnetic behaviour. In this simulation the gridpoints in a lattice have a weighted random change (based upon the values of their neighbours) to change state after a random time period.

When we compare Fornax with systems like SimJava, Simkit and JSim it immediately becomes clear that the ease of programming used our integrated approach is much better. In addition, the model allows automatic parallel and distributed evaluation of the simulation. The strength of Fornax is its integrated approach in combining the DES semantics in the object-oriented programming model. Because of the other facilities in Java, such as serialization, reflection and language-level support for multi-threading, this is possible in a way which is not feasible within languages such as C++.

There are however also performance drawbacks in the Java oriented implementation of the simulation model. If we would compare the performance against the Pearl implementation from which we have taken the model, we see that Pearl vastly outperforms a Java program. The special purpose language, which has special runtime optimizations for the evaluation of discrete event simulations, runs approximately ten times faster.

A more prominent problem, which can be observed from the results in figure 4, is that if the number of entities increases above a threshold, the performance breaks down. The reason behind this problem is fundamental to our simulation model. The simulation model is based on the ability of each entity to wait for an event to arrive when processing another event, which requires a multi-threaded implementation. The association of a thread to each entity means that a large amount of overhead memory (two memory pages at least for each thread) is needed. For applications which hardly perform computation this means that when the cache memory limit has been reached, the memory bottleneck penalty has to be paid.

Most applications do however not require the powerful expressiveness in this model and in other applications not all threads will be active at the same time. In this case you can have threads that service multiple entities. This is the same type of limited (but not restrictive) model needed for parallel discrete event simulation using asynchronous virtual time as described in section 5.2.

In the figure we see the evaluation time for increasing number of entities, which associated threads are distributed by the JVM over the processors. Because of the random distribution of the threads over the processors, a near perfect speedup cannot be expected. The good initial speedup can be ascribed to the simulation clock thread where the concurrent processing of the entity and simulation clock immediately pays off.

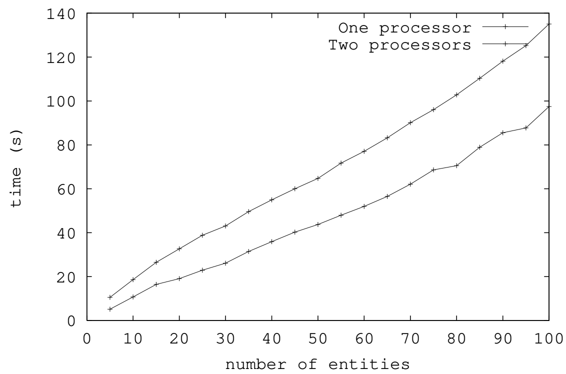

Figure 5: The performance of a parallel evaluation of a simulation model. The application used was the same as in the comparison with MODSIM in figure 4.

7 Discussion

The advantage of a pure Java implementation of the simulation environment is that the user can write the application in an already well established language. The programming support facilitated by the Java compiler and runtime classes can be fully exploited by the integration of the event generation and handling within Java. Instead of a library approach, events have become an integral part of the language through an implementation using event methods, effectively augmenting Java with DES semantics.

Instead of redeveloping a new simulation model, we have based the simulation system on an well-understood type of DES simulation model. In the same manner that using an well established programming language as an basis, we feel that not redeveloping a simulation model provides the preferred access for application programmers. Moreover, the chosen DES simulation model needs no extensions, since it is powerful enough to cover the widest set of DES simulation applications and can be implemented efficiently and in a distributed manner.

The novelty of our approach lies in the way we have added the semantics to the Java language. Using Java reflection and other object oriented features we created a programming framework in which the application programmer is not burdened anymore with programming tedious and obscure routines for generating and accepting events, instead we reuse the method call semantics and replace it with discrete event semantics.

Main feature of our approach is to use standard Java environments. Nor the Java language, the Java Virtual Machine, or standard Java class files were changed. The result is a distributed simulation environment, which runs on any Java platform. A pure Java approach to simulation environments is necessary for Web-based simulations. A simulation system should furthermore be flexible to allow the cooperation with other libraries or packages, such as visualization or information gathering such as statistics and report generation. Our approach fulfills these design considerations, while providing a more integrated solution by using event methods.

The model also enables a transparent manner of distributed computing. This is useful for both Web-based computing, where a simulation is designed on the users' machine and run on compute servers, and parallel evaluation using

PDES. Although the JVM definition in principle makes it possible, current implementations do not support a checkpoint and restart facility for threads. This limits the conceptual modeling framework for PDES using the Time Warp mechanism. Development of *serializable* threads will broaden the programming paradigm that can be used for migratable active code (agents) to our model. Serializable threads will not only make it possible to use optimistic DES scheduling more freely, but also enables the migration of entities from one machine to another. This can be used to implement load balancing scheduling policies.

Performance improvements in Java have made it a viably option for programming complex simulations. A more intelligent and flexible use of threading can improve the expressiveness and performance in our implementation, where the performance degrades due to excessive memory usage related to the threads needed per entity. Shielding the application programmer from the difficulties encountered when writing parallel programs (both in multi-threading as message passing programs) by presenting a inherent distributed model aids not only the ease of programming. It also allows the runtime system to perform internal optimizations, such as using threads more efficiently, or performing platform dependent load balancing. We find that programming frameworks in which concurrent evaluation is implicitly embedded in the model, such as within the proper PDES model, will be the preferred method of programming in the future.

Acknowledgments We gratefully acknowledge P.M.A. Sloot for the valuable discussion and feedback during our current research.

More information about Fornax can be found at HTTP://WWW.WINS.UVA.NL/~BERRY/PROJECTS/FORNAX/.

References

[1] BUSS, A. H., AND STORK, K. A. Discrete event simulation on the world wide web using Java. In *Proceedings of the 1996 Winter Simulation Conference* (Coronado, CA, Dec. 1996), pp. 780–785.

[2] CARPENTER, B., CHANG, Y.-J., FOX, G., LESKIW, D., AND LI, X. Experiments with 'HP Java'. *Concurrency: Practice and Experience 9*, 6 (June 1997), 633–648.

[3] FLYNN HUMMEL, S., NGO, T., AND SRINIVASAN, H. SPMD programming in Java. *Concurrency: Practice and Experience 9*, 6 (June 1997), 621–631.

[4] FUJIMOTO, R. M. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (Oct. 1990), 30–53.

[5] MCNAB, R., AND HOWELL, F. W. Using Java for discrete event simulation. In *Proceedings of the Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW)* (Edinburgh, UK, 1996), pp. 219–228.

[6] MILLER, J. A., NAIR, R. S., ZHANG, Z., AND ZHAO, H. JSIM: a Java-based simulation and animation environment. In *Proceedings of the 30th Annual Simulation Symposium* (Atlanta, Georgia, Apr. 1997).

[7] MULLER, H. *Simulating Computer Architectures*. PhD thesis, Department of Computer Science, University of Amsterdam, Amsterdam, The Netherlands, Feb. 1993.

[8] NANCE, R. E. A history of discrete event simulation programming languages. *ACM SIGPLAN Notices 28*, 3 (Mar. 1993), 149–175.