JSIM: A JAVA-BASED SIMULATION AND ANIMATION ENVIRONMENT

John A. Miller
Rajesh S. Nair
Zhiwei Zhang
Hongwei Zhao
Department of Computer Science
University of Georgia
Athens, Ga 30602-7404, U.S.A.

Abstract

In this paper, we present JSIM, a Java-based simulation and animation environment being developed at the University of Georgia. The JSIM library includes many Java classes to make developing simulation models easy. A model is constructed by building a graph with nodes (e.g. Queues and ServerBays) and edges (e.g. Transports). A model of moderate complexity can be constructed with minimal effort. Furthermore, in a future release of JSIM, much of the code will be created by a graphical designer. Using the built-in features of JSIM, the design diagram may be animated when the simulation is run. A novel aspect of the environment is that it follows the principles of Query Driven Simulation for the purpose of storing and retrieving simulation models and results in databases.

1. Introduction

Java is a very elegant yet powerful language with several built-in capabilities that make it quite useful for developing simulations. A couple of important advantages of using Java are that simulation models become easily available on the Web and Java is a clean object-oriented programming language which facilitates a high level of code reuse. With Java, it is possible to have "executables" on the Web. This provides users with a dynamic environment as opposed to a largely static, text-driven environment that was existent prior to Java. Simulation users can finally view results over the Web in a dynamic environment complete with animation, sound and textual information. In [10], the rationale for using Java to build a simulation environment has been discussed in detail. We summarize these arguments below.

Java is a programming platform that brings a high level of dynamism to Web applications. This feature makes it particularly suitable to represent simulation events using animation on the Web. Java also provides many features aimed at making animation easier. Because Java applets run on Web browsers, they can be widely distributed very easily. Furthermore, the concept of architecture independent portability is realized with Java. Java also provides simple and powerful thread programming support making it easier for us to implement a process interaction approach to simulation. Currently, Java has some limitations, due to security restrictions, which prevent it from providing persistence to data. We have dealt with this problem by storing information in databases.

In later sections, we will introduce JSIM and also discuss the JSIM library in greater detail. JSIM is a comprehensive library providing packages for both process interaction and event scheduling approaches to simulation. JSIM also provides animation inherently because the simulation classes have animation routines built in. Simulation results are generally obtained through computationally intensive procedures. Hence, it is essential that these results be stored for future use. JSIM incorporates database connectivity with simulation classes for this purpose. The underlying concept for this integration is Query Driven Simulation as defined in [5]. We have also shown a simple M/M/1 Bank simulation model coded using the JSIM library. Finally, we will present some conclusions and discuss future work.

2. JSIM: Concepts and theory

JSIM provides the user with a simulation and animation environment in which s/he can build simulation models, execute the models, watch the animation of a static design diagram, embellish the animation with special icons, and generate statistics.

JSIM was designed to simplify the work of simulationists and simulation model users. It accomplishes this task by providing the model builder with a graphical design environment and a flexible array of classes. Hence, the model builder is able to construct a simulation model with minimal coding. The model user is helped by the animation which depicts simulation activities in an easy to understand manner.

An underlying concept in the design of the JSIM library is graph theory. Simulation models are represented as directed graphs with different types of nodes connected by a single edge type. There are several advantages to using graph theory as a conceptual basis. One can use proven algorithms to test for deadlock, suitability of design, etc. It is also conceptually easier to reuse parts of models since they can be thought of as subgraphs.

JSIM provides the model builder with an environment in which s/he can design simulation models using point and click operations. Essentially, the model builder constructs a static representation of the simulation model. JSIM provides for the animation of this static representation through incorporating animation facilities within each simulation component.

One of the key concepts in the development of JSIM is Query Driven Simulation (QDS) ([5], [6], [7], [8], [9]). QDS is based on the tenet that simulation analysts as well as naive users should see a system based upon QDS as a sophisticated information system which will be able to store or generate information about the behavior of systems the user wishes to study. QDS suggests using a database to store simulation results, which might have been generated at great computational costs, as well as simulation models, which might be queried by a user. When a user queries a simulation system based on QDS, the system first tries to locate the required information in the database, since it might have been stored as the result of an earlier model execution. If the required data is present, it is simply retrieved and presented to the user. If it is not present, the QDS system instantiates the relevant model (or models), executes it (or them) and shows the results of the execution to the user.

Another component of the JSIM package is the graphical designer. Currently, the graphical designer facilitates the development of a simulation model. The trend is to move towards Visual Programming [1] wherein the model designer will be able to generate code while building a model using the graphical designer. We will discuss the graphical designer in greater detail later in the paper.

3. The JSIM library

In this section, we explain the JSIM library in detail. We will also endeavor to justify the design of the JSIM class library. JSIM is intended to make model creation an easy task. It is also intended that the model user be able to understand the model as well as the simulation easily. This is achieved through providing animation capabilities to the simulation classes. We believe that the JSIM library is easy to use and that even a fairly complex simulation model can be built with minimal coding. Presently, the JSIM library is organized into five packages. They are:

1. *queue* - The *queue* package includes four different types of queues including FIFO, LIFO, Priority, and Temporal queues.
2. *statistic* - The *statistic* package provides classes for collection and analysis of data generated by simulation.
3. *variate* - The *variate* package has random number generators and several commonly used random variate generators.
4. *process* - This package includes classes used to implement the process interaction approach to simulation.
5. *event* - This package provides classes for the event scheduling approach to simulation.

We now discuss each of these packages in greater detail. The signatures for the public methods for the principal base class for each package as well as for some other important classes will be given. For brevity, if a class has multiple constructors, only one is shown. The full description of all classes is available as HTML documents generated by *javadoc*.

3.1. The queue package

In this section, we discuss the queue package in more detail. The Queue class is defined as an abstract base class, from which FIFO_Queue (First-In, First-Out queue), LIFO_Queue (Last-In, First-Out queue), PriorityQueue, and TemporalQueue are derived. The class hierarchy of the *queue* package is shown in figure 1. In the class hierarchy diagrams, foundation, JSIM and model specific classes are represented by dashed, regular and bold rectangles respectively. Interfaces follow the same pattern but are represented as rounded rectangles instead.

```
public abstract class Queue {
   public boolean empty ();
   public boolean full ();
   public int length ();
   public abstract Object front ();
   public abstract void clear ();
   public abstract void enqueue (
      Object item);
   public abstract Object dequeue ();
```

Most of the methods of the Queue class are abstract and have to be implemented in subclasses. The *empty* method determines whether the queue is empty, while the *full* method

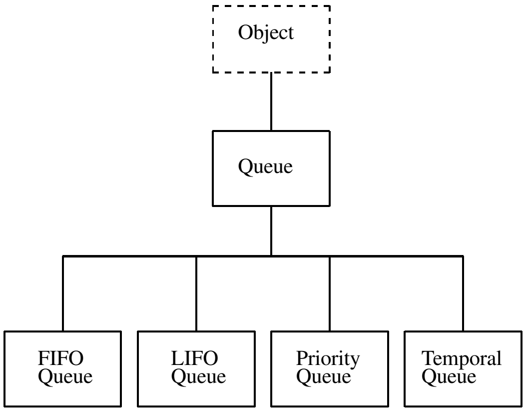

Figure 1. Class hierarchy diagram for the queue package

determines whether the queue is full. The *length* method returns the number of elements in the queue. The *clear* method clears the entire queue by removing all its elements. *front* peeks at the element at the front of the queue but does not dequeue it. The *enqueue* method inserts a single element into the queue. There may be multiple *enqueue* methods in subclasses of Queue. For instance, the priority queue class implements a second *enqueue* method shown below.

```
public void enqueue (Object item,
   int priority)
   throws FullQueueException;
```

The *dequeue* method removes and returns the entity at the front of the queue to the invoking object. It returns an entity of type *Object*. One might need to cast the returned object suitably in order to retrieve useful information from it. We use splay trees, based on [12], to implement priority and temporal queues, and simple lists to implement the FIFO and LIFO queues.

Each of these queues has, by default, infinite capacity. However, users have the freedom to set the capacity when constructing a queue object. In addition, we have defined two exceptions in the queue package: EmptyQueueException and FullQueueException. An EmptyQueueException is thrown when an attempt is made to *dequeue* an item from an empty queue, while a FullQueueException is thrown when an attempt is made to *enqueue* an item into an already full queue.

Priority and temporal queues may be used for lines ordered by priority as well as to implement Future Event Lists and time advance mechanisms. We first insert an incoming process (or event for event scheduling) into the Future Event List (FEL), implemented as a temporal queue, based on the process activation time (ties are broken by priority) and then invoke the process at the front of the FEL and update the clock to the time of the next process.

3.2. The statistic package

The *statistic* package contains classes to collect statistical information. The *Statistic* class is an abstract base class and contains methods to analyze statistical data and aid in outputting simulation results. Figure 2 shows the class hierarchy of the *statistic* package. *Format* is used for formatted output similar to the *printf* function in the C language. It was developed by Cay Horstmann for the Core Java (Book/CD-ROM) published by SunSoft Press/Prentice Hall.

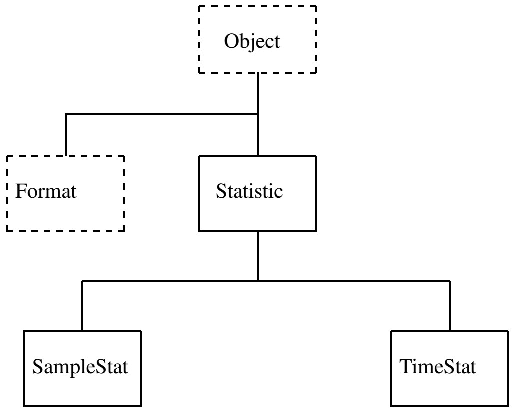

Figure 2. Class hierarchy diagram for statistic package

```
public abstract class Statistic {
   public int samples ();
   public double min ();
   public double max ();
   public double mean ();
   public double variance ();
   public double stdDev ();
   public double rms ();
   double confidence (double level);
   public void printStat (String name);
   public static void printLabel ();
   public void resetStat (double time);
```

The *SampleStat* and *TimeStat* classes extends the base *Statistic* class. *SampleStat* is used for collecting sample statistical data (via its *tally* method), while the *TimeStat* class is used to gather time-persistent statistics (via its *accumulate* method). The *Statistic* class has the ability to calculate minimums, maximums, means, variances, standard deviations, root mean squares and confidence interval half widths. The methods *sample, reset, printStat,* and *printLabel* return the number of observations, resets the statistical counters, and prints the statistical summaries and labels respectively.

3.3. The variate package

The *variate* package provides a wide variety of random variates. The class *Variate* is a class which is extended by all other variates. The *Variate* class uses JSIM's own Linear Congruential Generator or *LCGRandom* as it is called. But, this can be changed very easily to use Java's *Random* class by modifying the code of the *Variate* class. From the code shown below, it can be seen that the modifications needed to swap random number generators are very minor. It is also a simple matter to install yet another random generator. The *LCGRandom* number generator has a very long period and has been statistically proven to provide good inter-sample independence. The *gen* method returns the next random number. For the derived classes, the random number returned depends on the type of distribution used.

```
public class Variate {
    private LCGRandom rand;
    //private Random rand;

    protected Variate(int i) {
        rand = new LCGRandom (i);
        //rand = new Random (i);
    }; // Variate

    public double gen() {
        return rand.gen ();
        //return rand.nextDouble ();
    }; // random
```

We have implemented 14 continuous random variate generators and 8 discrete random variate generators. The discrete random variates available in JSIM's variate package are *Bernoulli, Binomial, DiscreteProb, Geometric, HyperGeometric, NegativeBinomial, Poisson,* and *Randi*. The continuous random variates available are *Beta, Cauchy, ChiSquare, Erlang, Exponential, F_Distribution, Gamma, HyperExponential, LogNormal, Normal, StudentT, Triangular, Uniform*, and *Weibull*. The class hierarchy diagram of the *variate* package is shown in figure 3. The algorithms for these generators may be found in [3] and [11].

3.4. The process package

The *process* package provides classes that are used to create simulation models using the process interaction paradigm of simulation. As mentioned before, simulation models built using JSIM can be conceptually viewed as graphs composed of nodes and edges. The *process* package can be subdivided into subclasses of *Node*, classes representing the edges and additional classes. The nodes include the classes *ServerBay*, *QueueManager*, *Facility*, *Source* and *Sink*. The edges are formed using the *Transport* class. In addition, *process* contains *VirtualScheduler*, *Clock*, *Model*, *SimObject*, and *LineSegment*. The *LineSegment* class will not be discussed any further since it is a trivial class used only for the purpose of animation. Figure 4 shows the class hierarchy diagram of the *process* package.

We also provide for both pseudo-real-time simulation and virtual-time simulation. Virtual-time simulation is suitable for fast simulations where the user does not wish to be constrained by the speed of the real time clock. A virtual clock is used instead of a real time clock. Virtual-time simulations are also suitable when the simulation model requires substantial computation. If pseudo-real-time simulation is used in a high computation simulation model, it may slow down simulation due to animation activities. Pseudo-real-time simulations are more suitable when the simulationist wishes to see the results of simulation as represented by animation and where speed is not a crucial issue. We call this pseudo-real-time since the clock advance rate is adjusted to make the animations as visually informative as possible (e.g., bank customer service times of a few seconds rather than a few minutes). A model builder may also use pseudo-real-time simulation coupled with animation to test the validity of a model. Conversion of a pseudo-real-time model to a virtual-time model can be done quite easily. The details are given in the subsection on *Clock*.

The classes in the *process* package are easy to use and are highly flexible. It is intended that the model builder should not have to write much code in order to design a model. The classes are designed such that it does not require great effort to create a model by hand coding. This also facilitates automatic code generation.

3.4.1. VirtualScheduler

The *VirtualScheduler* is used by JSIM's classes to implement virtual time simulation.

```
class VirtualScheduler {
```

This class and its methods will not be visible to the simulation model builder (it is only available to other classes within the process package). The important methods of *VirtualScheduler* include *vSleep*, the counterpart to Java's *Thread.sleep* method, and *vCurrentTime*, the counterpart to Java's *System.currentTimeMillis* method.

3.4.2. Clock

The *Clock* class is the time keeper of the simulation model for both pseudo-real-time and virtual-time simula-

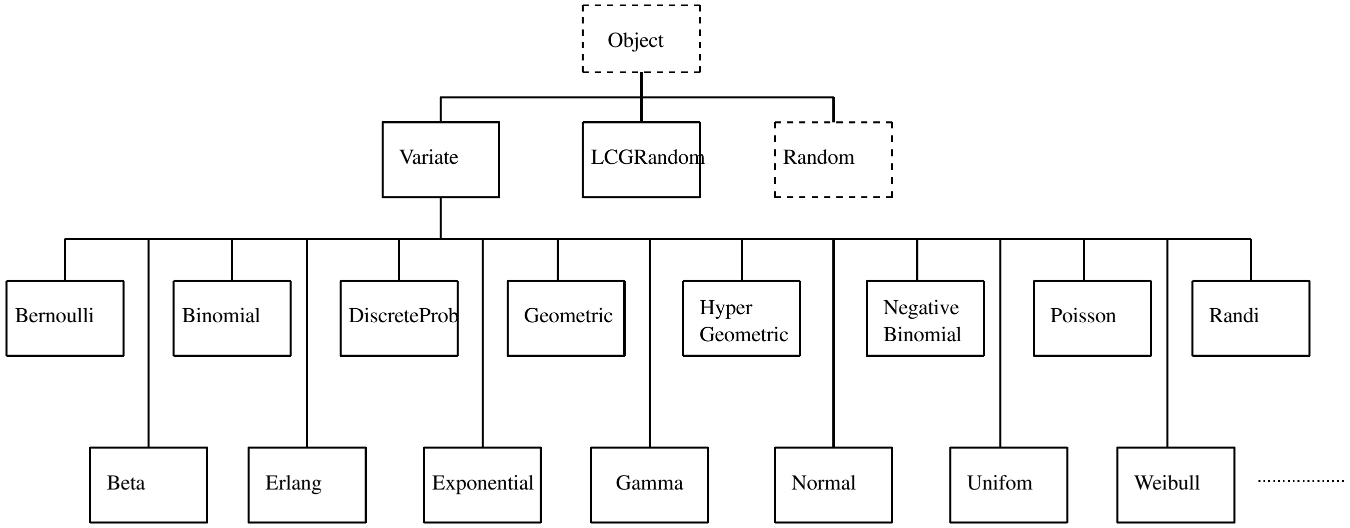

Figure 3. Class hierarchy diagram for variate package

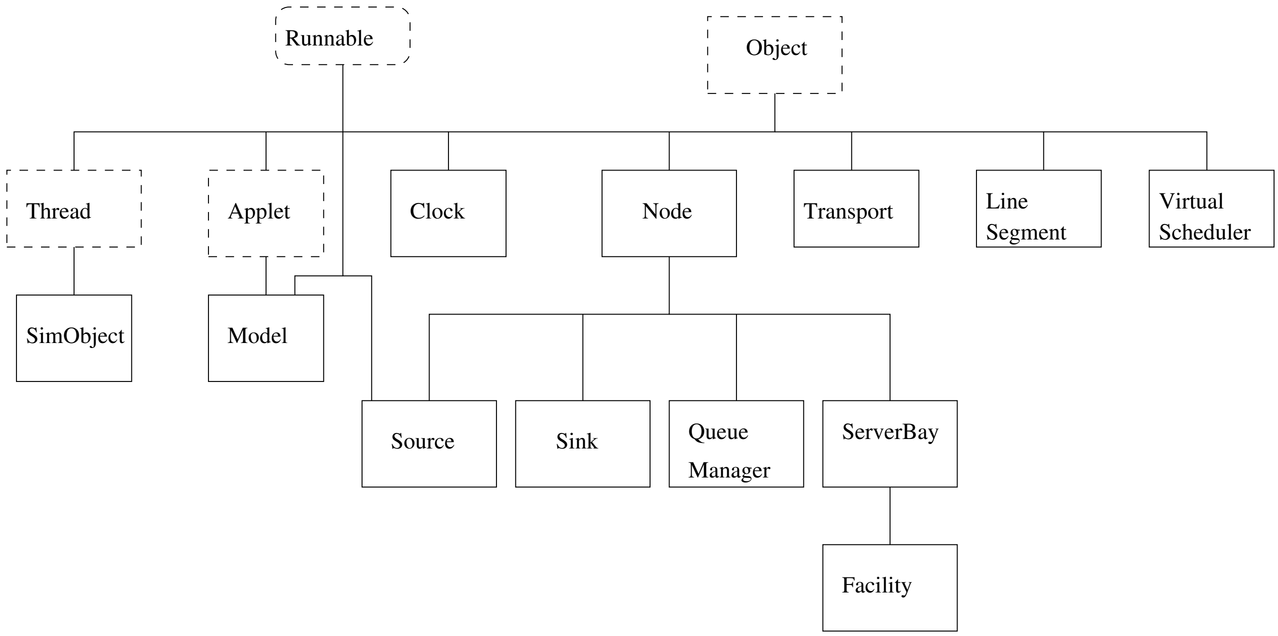

Figure 4. Class hierarchy diagram for the process package

tions.

```
public class Clock extends Object {
   public static void toggle
      (boolean isReal);
   public static void delay
      (double sleepTime, Thread thr);
   public static double currentTime ();
```

A model is switched from pseudo-real-time simulation to virtual-time simulation using the *toggle* method. The boolean value *isReal* decides whether the simulation is pseudo-real-time or virtual-time. The *delay* method is used by a simulation entity which wishes to sleep for some period of time. In pseudo-real-time simulation, the thread, *thr* actually sleeps for a time duration *sleepTime* since the *delay* method invokes the Java supplied *sleep* method. However, in virtual-time simulation, JSIM's own *VirtualScheduler.vSleep (double sleepTime, Thread currentProcess)* method is invoked by the *delay* method. Similarly, the *currentTime* method uses Java's *System.currentTimeMillis ()* for pseudo-real-time simulation and JSIM's *VirtualScheduler.vCurrentTime ()* for virtual-time simulation.

3.4.3. SimObject

An instance of the *SimObject* class represents a single simulation entity. The *SimObject* class encompasses the basic functionality of a simulation entity. The simulation model builder should extend *SimObject* to create a useful simulation entity.

```
public class SimObject extends Thread {
   public SimObject ();
   public long simId ();
   public long simPriority ();
   public double simStartTime ();
   public void draw (Graphics offScreen);
```

A SimObject has data members *xpos* and *ypos* which represent its position during animation. They are friendly (neither private nor public) data members since they have to be modified by other classes in the *process* package. *SimObject* also has data members used to keep track of temporal data such as start time, end time, wait time, etc. *SimObject* extends the *Thread* class provided by Java. Hence, every entity in a JSIM process interaction model is a separate thread. The *SimObject*'s logic (behavior during its lifetime) has to be defined by the model builder by coding its *run* method (see the example in section 6).

3.4.4. Transport

The *Transport* class forms the edges of the simulation model graph. A *Transport* connects two nodes. Simulation entities or *SimObject*s travel along *Transport*s while moving from one node to another. A *Transport*, generally has a constant speed. *Transport*s have been implemented as piecewise linear segments. So, the model builder can specify any path that s/he wants between two nodes.

```
public class Transport {
   public Transport (Vector pointVector,
      Model env);
   public double move (SimObject entity);
   public boolean atEnd (
      SimObject entity);
   public void draw (Graphics offScreen);
```

*SimObject*s move along *Transport*s using the *move* method. The *move* method causes the *SimObject* to *join* the *Transport*, take several *step*s on the *Transport* until it reaches the end of the *Transport* (signified by the *atEnd* method returning true), and *leave*s the *Transport*. The *join*, *step*, and *leave* methods are protected methods of the *Transport* class. The *atEnd* method returns true or false depending on whether the *SimObject* is at the end of the *Transport* or not.

3.4.5. Node

Node is an abstract class that encapsulates the features common to the classes that appear as nodes in a JSIM model, namely, *Source*, *QueueManager*, *ServerBay*, *Facility* and *Sink*.

```
public abstract class Node {
    public Node (String name,
       Point topLeft, int width,
       int height, Model env);
    public void printStatistic ();
    public abstract void draw
       (Graphics offScreen);
```

The *printStatistic* method takes care of printing the statistics collected by the *Node* during simulation. Every node collects two different types of statistics namely duration/time data and occupancy/usage data. Suitable labels are created using the *Node*'s name for printing purposes. The *draw* method is an abstract method that must be implemented in every subclass of *Node*. This method is used to implement the look of the node during animation.

3.4.6. Source

Source is a generator/creator of *SimObject*s. It creates *SimObject*s depending on defined parameters such as inter-arrival time or the number of entities to create. In the JSIM library, *Source* has been designed as an abstract class. This is because different simulation models will require different types of entities to be created. Hence, the model builder is required to extend *Source* to create a specialized entity generator. This is a very simple process and involves only the implementation of the abstract *makeEntity* method. This is because the decision as to which entities need to be created has to be made by the simulation model builder. The simulation model builder must extend *Source* and implement the abstract method *makeEntity* to return whatever entity is required. The *run* method implements the lifetime of the *Source* class. It has been implemented to create an entity periodically according to the inter-arrival distribution.

```
public abstract class Source extends
      Node implements Runnable {
   public Source (String name,
      Point topleft, int numStop,
      Variate arrivalTime, Model env);
   public abstract SimObject makeEntity ();
   public void run();
   public void draw (Graphics offScreen);
```

3.4.7. ServerBay

A *ServerBay* acts as a service provider. It initially creates a certain number of servers as defined by the model builder, thus providing servers to *SimObjects* requesting service. *SimObject*s obtain service by *request*ing a server, *us*ing the server and then *releas*ing it. The *ServerBay* also maintains statistics regarding the usage of the servers and the service time.

```
public class ServerBay extends Node {
   public ServerBay (String name,
      Point topLeft, int numServers,
      Variate serviceTime, Model env);
   public void request (SimObject entity);
   public void use (SimObject entity);
   public void release (SimObject entity);
   public boolean busy ();
   public int numBusy ();
   public void draw (Graphics offScreen);
```

Currently, the servers have been designed as static data items. However, it is intended in the future to have an option where servers resemble simulation entities in that they retain the freedom to move within the model. The *busy* method is used to check if all the servers in a *ServerBay* are busy or not. The *numBusy* method is used to determine the number of busy servers in the *ServerBay*.

3.4.8. QueueManager

QueueManager is a node which houses and controls one of the queues provided by JSIM's *queue* package. *SimObject*s can *enqueue* themselves in queues while they are waiting for service within a simulation model.

```
public class QueueManager extends Node {
   public QueueManager (String name,
      Point topleft, Queue line,
      Model env);
   public boolean empty ();
   public boolean full ();
   public int length ();
   public void enqueue (SimObject entity);
   public void enqueue (SimObject entity,
      int priority);
   public void enqueue (SimObject entity,
      double time, int priority );
   public SimObject dequeue ();
   public SimObject front ();
   public void clear ();
   public void draw (Graphics offScreen);
```

The *QueueManager* implements all the methods implemented in the *Queue* class of the *queue* package and some additional methods. This is because *QueueManager* is designed to wrap around *Queue* and manage the queue while collecting statistics and providing animation. The additional methods are *enqueue*s for the derived classes as well as a *draw* method. The *QueueManager* collects statistics regarding the queue length and the waiting times of the simulation entities in the queue. Any method that modifies the queue, *enqueue*, *dequeue* and *clear* must also collect statistics for the QueueManager. The *draw* method, as in other simulation classes, implements the look of the *QueueManager* during animation.

3.4.9. Facility

A *Facility* is derived from *ServerBay* since it is most similar to this class. It encapsulates a *QueueManager* and a *Transport* as private data members. The encapsulated *Transport* connects the *ServerBay* with the *QueueManager*. In some models, simulation entities will only be exposed to the functionality of *Facilit*ies and not to that of the *ServerBay* and *QueueManager*. It is easier to develop models using this higher level component.

```
public class Facility extends ServerBay {
   public Facility (String name,
      Point topLeft, int numServers,
      Queue line, Variate serviceTime,
      Model env);
   public void request (SimObject entity);
   public void use (SimObject entity);
   public void release (SimObject entity);
   public void queueLength ();
   public void printStatistic ();
   public void draw (Graphics offScreen);
```

Simulation entities (*SimObject*s) obtain service by requesting the *Facility* using the *request* method. If the *Facility* is not busy, the simulation entity acquires a server and *use*s it. However, if the *Facility* is busy, the simulation entity is enqueued by the *Facility*'s *QueueManager*. When the simulation entity finishes its work, it must release the server using the *release* method. The *queueLength* method returns the length of the queue within the *Facility*. *Facility* also overrides the *Node*'s *printStatistic* method since it needs to print the statistics collected by its *QueueManager* and *ServerBay* in addition to its own statistical data.

3.4.10. Sink

A *Sink* is, conceptually, the opposite of a *Source* in that a *Sink* phases out or destroys *SimObject*s created by a *Source*. *SimObject*s go to a *Sink* when they complete their lifetime. *SimObject*s are eliminated at *Sink*s using the *capture* method, prior to which their statistics are collected.

```
public class Sink extends Node {
   public Sink (String name,
      Point topLeft, Model env);
   public void capture (SimObject entity);
   public int numEntitiesCaptured ();
   public void draw (Graphics offScreen);
```

The *numEntitiesCaptured ()* method returns a count of the number of simulation entities that have been captured by the *Sink*. As before, the *draw* method defines the look of the *Sink* during model animation.

3.4.11. Model

The *Model* class encapsulates the basic features of a simulation model applet. It saves the model builder considerable effort in terms of coding for both simulation and animation.

```
public abstract class Model extends
       Applet implements Runnable {
    public void init();
    public void start();
    public void run();
    public void paint();
    public void update();
```

This class overrides three of the *Applet* methods, namely *init, paint,* and *update*. It also overrides two methods of the *Runnable* interface, namely *start* and *run*. The *init* method initializes the variables required for double buffering (explained in the section on animation) and also sets up the main applet thread which does the display. The *paint* and *update* methods are standard implementations as required by the double buffering technique. The *start* method starts the main applet thread. The *run* method implements the sleep-draw-sleep cycle of the main applet thread. It is not required that the *run* method be overridden by the model builder since it has been completely coded in *Model*. The simulation scene (animation) is redrawn periodically by the *run* method.

3.5. The event package

We also provide an *event* package which can be used to build event-scheduling simulation models. The *event* package is composed of the classes *Event*, *Entity* and *Scheduler*. The scheduler uses a FEL to implement event scheduling. However, we will not be discussing this package further in this paper.

4. Animation

JSIM provides for animation of a simulation model built by a model designer. Each of the simulation classes in the *process* package has animation built into it. Each of the nodes, the *Transport* and the *SimObject* classes implement a *draw* method. The *draw* method receives an off-screen graphics buffer as a parameter. Each simulation class draws into this off-screen buffer. The *draw* method implements the look of the simulation object. The main model applet initially creates an off-screen graphics buffer of the same size as its actual on-screen graphics buffer. It then invokes the *draw* method of each of the simulation objects defined within the model, passing the off-screen graphics buffer as a parameter. It does this in the implementation of the *drawComponents* method of the *Model* class of the *process* package. After this, the main model applet draws the off-screen buffer onto the screen by copying it onto the on-screen graphics buffer. This technique, referred to as double buffering, is generally used to reduce flicker in animation. We have, however, also used this technique to enable multiple threads to represent their status on screen. In other words, we used the double buffering method to make multi-threaded animation possible. An added advantage is smooth animation. Furthermore, all this can be achieved at minimum cost of coding to the model builder. All that the model builder needs to do is to extend the *Model* class of the *process* package while defining the simulation model.

Animation is a useful tool in checking the correctness of a model. The simulation model builder can track the movements of simulation entities through the model during its lifetime. It is also useful for a model user since it is easier to understand the simulation by looking at the animation.

We shall now discuss the *draw* method of the *ServerBay* class as an example of the implementation of animation. The *draw* methods are easy to code because of the capabilities provided by the Java AWT package.

```
public void draw (Graphics offScreen) {
   offScreen.setColor (Color.green);
   offScreen.drawRoundRect (xpos, ypos,
      width, height, arcW, arcH);
   offScreen.drawString (name, xpos,
      ypos - yOffset);
   offScreen.setColor (Color.red);
   for (i = 0; i < numServers; i++) {
      Point pos = serverPosition[i];
      offScreen.fillOval(pos.x, pos.y,
         size, size);
      if (serverBusy[i]) {
         servedGuy[i].draw (offScreen);
      }; // if
   }; // for
}; // draw
```

The first three lines of code in the *draw* method implement the look of the *ServerBay* which is that of a rounded rectangle. The color of the *ServerBay* is green. This method also draws the name of the *ServerBay* just above the rectangle. It then draws the servers as ovals at points called server positions. While drawing the servers, it also checks if a server is busy in which case it invokes the *draw* method of the simulation entity or *SimObject* which is being served at this server position. It should be noted here that a *SimObject*'s *draw* method is invoked by the node or edge it is contained in.

Figure 6 shows a screen shot of an animation designed using the graphical designer. Currently, code generation is not fully functional. More details about the graphical designer can be found in the following section.

5. Graphical designer

The graphical model designer, implemented using the Java AWT package, is a GUI-based model builder that supplies simulationists with more direct, intuitive means to build a model. It allows users to draw all the simulation objects on a model-builder canvas by selecting buttons from the tool-bar followed by drag and drop operations.

The properties of the simulation objects are defined by properly populating the property window which is popped up every time a new object is drawn on the canvas. Properties can also be checked and redefined by double clicking on the object and refilling the property window for the object. The deletion of an object is accomplished by selecting delete from the edit menu followed by clicking on the object to be deleted. Help information is available in a Help window.

All the information about simulation objects is stored in data structures and is intended to be used later to accomplish code generation.

6. An example model

In this section, we consider the simple Bank simulation example that we have been referring to in the previous sections. For this simple model, a customer may enter the bank and wait in a queue or approach a teller depending on whether the teller is busy or not. After approaching the teller, the customer may transact some business which may take a finite amount of time following which s/he leaves the bank. The class hierarchy diagram of the JSIM classes utilized in this example is shown in figure 5.

Let us discuss the coding requirements for the *Bank* model. To develop a model, one simply needs to code three derived classes. Static components of a model are specified in a class derived from *Model* (e.g., *Bank*). Only the *init* method needs to be coded. The creation of dynamic components (*SimObjects*) is specified in a class derived from *Source* (e.g., *CustomerSource*). Only the constructor and *makeEntity* method need to be coded. Finally, the behavior of dynamic components are specified in a class derived from *SimObject* (e.g., *Customer*). Only the constructor and *run* method need to be coded. Of course, one may wish to have multiple types of dynamic components and generators for them, in which case additional derived classes will be necessary.

Models are instantiated or made executable by defining a scenario (values for input parameters). The *Bank* model gets its parameters at runtime from the enclosing HTML file using the *getParameter* method. Alternatively, a model may query the user for the parameters. When the QDS component is finished, scenarios will be built on-the-fly.

Models such as *Bank* are derived from the *Model* class, which itself is derived from *Applet*. As stated before, *Model* has five methods *init, start, run, update,* and *paint*. Of these, only the *init* method need be overridden when coding models such as *Bank*. The *init* method sets up the working variables for the applet and is the first method to be executed. The *init* method has been partially implemented in the base class, *Model*. Hence, it should be invoked using *super.init ()*.

Let us now discuss the code required to implement the *Bank* model's *init* method in more detail.

- CustomerSource: We need to create a generator of entities at position (100, 100) and set the number of entities to be created to *numStop* (the number of entities it must create before stopping). It is also required that the *SimObjects* created by the *Source* know that they exist within the *Bank*. Hence, we supply *this*, a reference to the *Bank*, as a parameter to the constructor of *CustomerSource*, the customer generator described later in this section. We also provide the *Variate* for customer inter-arrival time as a parameter to the constructor of the customer source. In our example, the inter-arrival time uses an *Exponential* distribution with a mean value specified by the parameter *meanArrival*.

- Facility: We also need to create a *Facility* which represents the service center within the *Bank*. It is required that we supply the constructor of the *Facility* with certain parameters such as the number of servers, an *Exponential Variate* for service time, and some drawing parameters such as top left corner. The *Facility* sets up both a *QueueManager* and a *ServerBay* within itself. Hence, we need to supply the *Facility*'s constructor with the queue to manage. In the code shown below, the *Facility* created is called "Teller Facility" and is located at point (200, 100) (top left corner), manages a FIFO queue, and its *ServerBay* has *numTellers* tellers.

- Sink: A *Sink* has to be created for capturing simulation entities when they are finished. In our example, a sink is created at the location (700, 200) (top left corner). The code below illustrates how the model components described above are defined.

- Transports: We need to create two *Transport*s to connect the nodes created, one leading from the *CustomerSource* to the *Facility* and another leading away from the *Facility* connecting the *Facility* to the *Sink*. For linear *Transport*s, a simple constructor, which supplies the coordinates of the beginning and ending points, can be used.

```
import java.awt.*;
import jsim.queue.*;
import jsim.process.*;
import jsim.variates.*;

public class Bank extends Model {
   private CustomerSource custSource;
           Transport      toTeller;
           Facility       tellerFac;
           Transport      fromTeller;
           Sink           custSink;

   public void init () {
      super.init ();
      int numTellers = getParameter (
         "NUM_TELLERS");
      double meanArrival = getParameter (
         "MEAN_ARRIVAL");
      double meanService = getParameter (
         "MEAN_SERVICE");
      int numStop = getParameter (
         "NUM_STOP");
      custSource = new CustomerSource (
         "Customer Source",
         new Point (100, 100), numStop,
         new Exponential(meanArrival, 1),
         this);
      toTeller = new Transport (
         new Point (150, 125),
         new Point (200, 125), this);
      tellerFac = new Facility (
         "Teller Facility",
         new Point(200, 100), numTellers,
         new FIFO_Queue(), new Exponential
         (meanService, 2), this);
      fromTeller = new Transport (
         new Point (600, 125),
         new Point (700, 225), this);
      custSink = new Sink (
         "Customer Sink",
         new Point(700,200), this);
```

It should be noted that parameters such as number of servers, number of entities to be created, mean inter-arrival time and mean service time are obtained from the HTML file that runs this applet. These parameters are provided within the HTML file using the <param name=..... value=...> tag following the <applet> tag. A sample segment of the HTML file for the *Bank* model is shown below.

```
<applet code=Bank.class width = 800
                        height = 800>
<param name = NUM_TELLERS  value = 1>
<param name = MEAN_ARRIVAL value = 6000>
<param name = MEAN_SERVICE value = 5500>
<param name = NUM_STOP     value = 100>
</applet>
```

Now, we need to create a customer source or generator by extending the *Source* class provided in the JSIM library. This only involves implementing the abstract method *makeEntity*. This can be done as shown below.

```
import java.awt.Point;
import jsim.process.*;
import jsim.variate.Variate;

public class CustomerSource extends Source {
   private Bank env;

   public CustomerSource (String name,
      Point topLeft, int numStop,
      Variate arrivalTime, Model env) {
      super (name, topLeft, numStop,
         arrivalTime, env);
      this.env = env;
   }; // CustomerSource
```

```
   public SimObject makeEntity() {
      return new Customer (env);
   }; // makeEntity
}; // CustomerSource
```

Note that simple *Source*s can be created easily by substituting some other entity type for *Customer* and another model for *Bank* in the code above. We plan to investigate the use of reflection available in JDK 1.1 to make it unnecessary to even code *Source* classes.

Let us now proceed to code the *Customer* class. The *Customer* needs to implement the lifetime of a bank customer. This is quite simple and can be done as follows.

```
import jsim.process.SimObject;

public class Customer extends SimObject{
   private Bank env;

   public Customer (Bank env) {
      this.env = env;
   }; // Customer

   public void run () {
      env.toTeller.move (this);
      env.tellerFac.request (this);
      env.tellerFac.use (this);
      env.tellerFac.release (this);
      env.fromTeller.move (this);
      env.custSink.capture (this);
   }; // run
}; // Customer
```

7. Conclusions and future work

We believe that Web-based simulation will become an important technology for the future. With the emergence of Java, highly interactive and dynamic simulations/animations may be distributed over the Web. JSIM is an ongoing project to develop sophisticated capabilities for Web-based simulation. The JSIM library provides a widely available and easy-to-use facility for developing Web-based simulations and animations. Although the library is smaller, it provides many of the capabilities found in its C++ cousins ([2], [3]). The environment combines simulation, animation and database access exploiting the strengths of Java as well as minimizing its weaknesses (e.g., current lack of data persistence in applets). Please visit *http://lsdis.cs.uga.edu/∼jam/JSIM* for more information and demos. The complete library along with documentation generated using javadoc is stored in jsim.tar for easy down-loading.

As an ongoing project, we still have much work to do. The following are the major areas of interest for future work. We are currently incorporating advanced features such as fission/fusion (splitting an entity and combining several entities into one). The graphical model designer, which has limited capabilities as of now (can only be used to design a model), is intended to be a GUI-based model builder that would do much of the code generation that has to be currently done manually. Another area of focus is Query Driven Simulation. We will initially use an mSQL (Mini-SQL) RDBMS with JDBC as our database component.

References

[1] O. Balci, A. I. Bertelrud, C. M. Esterbrook and R. E. Nance. A Picture-Based Object-Oriented Visual Simulation Environment, *Proceedings of the 1995 Winter Simulation Conference,*IEEE:1333-1340, Arlington, VA, 1995.

[2] P. Fishwick. SIMPACK: getting started with simulation programming in C and C++. *Proceedings of the 1992 Winter Simulation Conference*:154-162, Arlington, VA, 1992.

[3] D. C. Grunwald. AWESIME–II: A Widely Extensible Simulation Environment, *AWESIME/DUDE Distribution,*, Boulder, Colorado, 1993.

[4] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, Inc., NY, 1982.

[5] J. A. Miller and O. R. Weyrich. Query driven simulation using SIMODULA. *Proceedings of the 22nd Annual Simulation Symposium*:167-181, Tampa, FL, 1989.

[6] J. A. Miller, W. D. Potter, K. J. Kochut, A. Keskin and E. Ucar. The active KDL object-oriented database system and its application to simulation support. *Journal of Object-Oriented Programming,* 4(4):30-45, 1991.

[7] J. A. Miller, K. J. Kochut, W. D. Potter, E. Ucar and A. Keskin. Query-driven simulation using active KDL: A functional object-oriented database system. *International Journal in Computer Simulation,* 1(1):1-30, 1991.

[8] J. A. Miller, W. D. Potter, K. J. Kochut and D. Ramesh. Object-oriented simulation languages and environment: a four-level architecture. *Object-oriented Simulation.* Zobrist-Leonard, IEEE Press:53-88, 1997.

[9] J. A. Miller, O. R. Weyrich, W. D. Potter and V. C. Kessler. The SIMODULA/OBJECTR query driven simulation support environment. *Progress in Simulation,* Vol. 3. Leonard-Zobrist, Editors, 1997 (to appear).

[10] R. S. Nair, J. A. Miller, and Z. Zhang. Java-Based Query Driven Simulation Environment. *Proceedings of the 1996 Winter Simulation Conference*:786-793, Coronado, CA, 1996.

[11] A. Alan B. Pritsker. *Introduction to Simulation and SLAM II, 3rd Ed.* John Wiley & Sons, NY, 1986.

[12] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the Association for Computing Machinery,* 32(3):652-686, 1985.

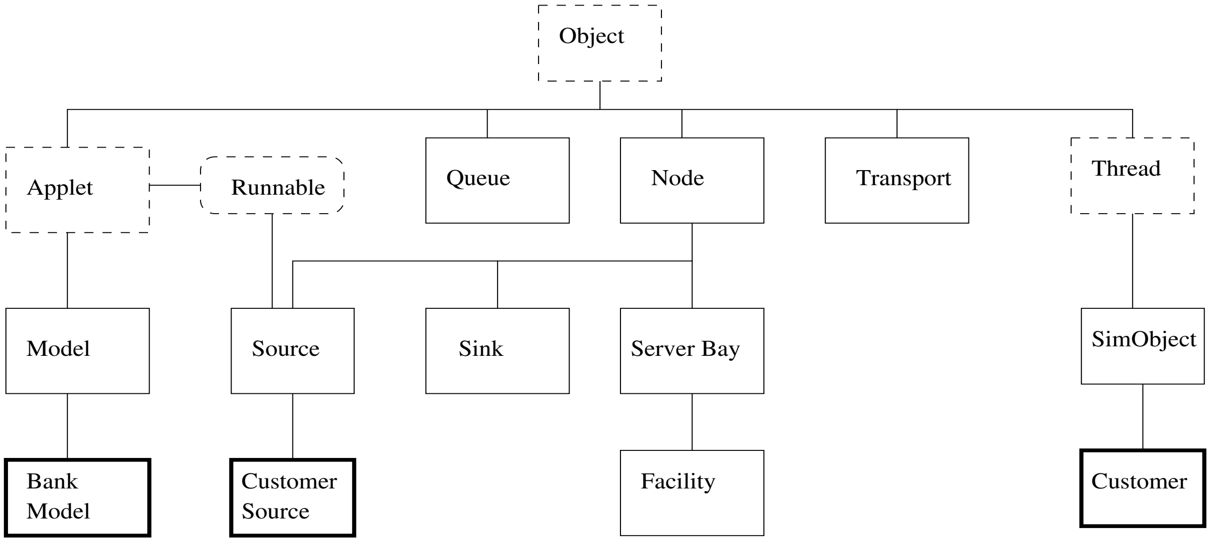

Figure 5. Classes used in the Bank Model example

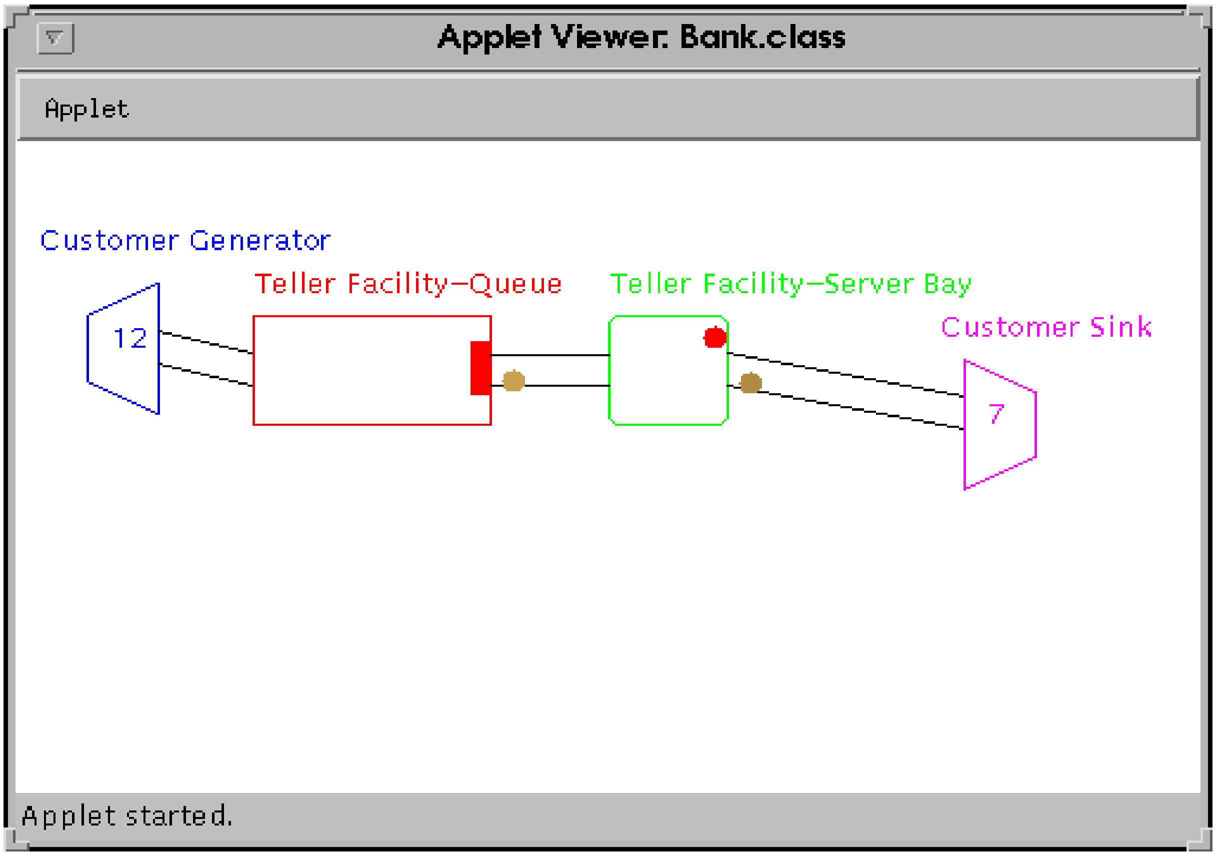

Figure 6. Snapshot of animation of the Bank Model