Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

COMPONENT-BASED SIMULATION ENVIRONMENTS: JSIM AS A CASE STUDY USING JAVA BEANS

John A. Miller
Yongfu Ge
Junxiu Tao

Computer Science Department
415 GSRC
University of Georgia
Athens, GA 30602-7404, U.S.A.

ABSTRACT

Component-based software can be used to develop highly modular simulation environments supporting high reusability of software components. This paper examines a case study in which JSIM, a Java-based simulation environment, is extended using Java Beans technology. This allows simulation models to be treated as components that can be dynamically assembled to build more complex models. It also allows simulation inputs and outputs to be dynamically linked to database systems.

1 INTRODUCTION

This paper provides an overview of the latest version of JSIM, a Java-based simulation and animation environment (Nair et al., 1996, Miller et al., 1997). This latest version encorporates the newest approach to software development, component-based technology. If component-based technology succeeds, the long hoped for gains in software development productivity may finally be realized. Complex software systems will be built by assembling components, minimizing the amount of low-level coding required. Systems built with components should be more extensible and often able to run in distributed and heterogeneous environments. Typically, component-based software will also run on the Web (e.g., Java Beans and Active X). Since JSIM is built using Java and Java Beans, it can be used for Web-based simulation (Fishwick, 1996, Fishwick, 1998).

JSIM has three foundation packages, `queue`, `statistics` and `variates`, that are generally useful, in and out of simulation. Built on top of these are two packages implementing the two most popular simulation world views. The `event` package support the construction of event-scheduling type models, while the `process` package support the construction of process-interaction type models. The `jmodel` package provides a visual designer for the `process` package. The design model may be animated when the simulation is run. Finally, the `qds` package provides a convenient means for accessing/generating simulation data. This package allows simulation inputs and outputs to be stored in databases, and simulation models to be launched as a part of query processing.

2 COMPONENT-BASED SOFTWARE

During the 1990's, simulation software has utilized the advantages of Object-Oriented Programming (OOP). The next step in software development is Component Software. Component software begins where OOP left off and adds capabilities to maximize software reuse and facilitate rapid development through assembling components rather than traditional coding. Graphical/visual design tools are typically used in this assembly process.

Component-based software development systems may support some or all the following capabilities.

- Object-Oriented Programming. Under OOP, software is developed as objects consisting of attributes (data members) and methods (member functions). The advantages of encapsulation, inheritance and polymorphism have been well documented.
- Persistence. Mechanisms are provided to save and restore the state of executing objects with little or no programming. Traditionally, these operations required a substantial amount of custom coding.
- Introspection. By following certain coding conventions (design patterns) and by providing supplementary

classes, different software components can be made to interact without any custom coding. For example, one class may inquire about properties, methods or events of other classes. Properties are appearance or behavioral attributes that are exposed (e.g., by get/set methods) to other classes or visual tools.

- Distribution. Although not a requirement, it is useful to able to have components work together even if they are not executing on the same machine. This is facilitated by providing high-level mechanisms for distributed object to object communication, typically through remote method calls or remote handling of events.

- Platform Independence. While "Write-Once, Run Anywhere" is not a requirement, it certainly simplifies the developer's job. If this is fully supported, any object can be downloaded to any machine and executed without recoding or even recompiling.

At present, software components may be developed with either ActiveX or Java Beans. Java Beans in our opinion has the advantages of platform independence and a simplier programming environment (full object-orientation, automatic garbage collection, safe use of memory and a straightforward approach to multithreading). For these reasons, we have chosen to develop JSIM in Java and utilize Java Beans technology. Speed disadvantages of Java versus C++ are becoming less of an issue as Just-In-Time (JIT) and native-code compilers are becoming available.

3 JSIM PACKAGES

Before focusing on the use of software components, the JSIM packages are briefly discussed.

3.1 The `queue` Package

The `queue` package is rooted by the *Queue* class which is defined as an abstract base class, from which *FIFO_Queue* (First-In, First-Out), *LIFO_Queue* (Last-In, First-Out), *PriorityQueue* and *TemporalQueue* are derived. Splay trees are used to implement priority and temporal queues, and simple lists are used 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.

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 `statistics` 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. *Format* is used for formatted output and has methods 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.

The *SampleStat* and *TimeStat* classes extend 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 package also contains a *BatchStatistic* class which is derived from *SampleStat*. *BatchStatistic* is used to collect batch statistics. The batch size is a parameter to the constructor of *BatchStatistic*. Finally, histograms can be produced using the *Histogram* class.

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 called *LCGRandom*, but this can be changed very easily to use Java's *Random* class by modifying the code of the *Variate* class. 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.

JSIM has implementations of fourteen continuous random variate generators and eight 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 algorithms for these random variate generators may be found in (Law and Kelton, 1982, Pritsker, 1986). The `variate` package also contains two classes used to test

the random number generators, namely *LCG_Test* and *KS_Test*.

3.4 The `event` Package

JSIM provides 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 *Event* class is used to code event routines, e.g., what happens at an arrival event. The *Entity* class is used to maintain information about entities (e.g., customers) in the simulation. Finally, the *Scheduler* class is used to schedule future events by putting them into the Future Event List.

3.5 The `process` Package

The `process` package provides classes that are used to create simulation models following the process-interaction paradigm. A simulation model is encapsulated as a Java bean. Such bean objects contain several *DynamicNode*s. Currently, *Server*, *Facility*, *Signal*, *Source* and *Sink* are provided as types of *DynamicNode*s. These nodes are connected with edges which *Transport* entities (*SimObject*s) between the nodes. A *Model* object is used to control the simulation by starting all of the *Source*s as well as stopping the simulation. If animation is to be performed, the model creates a *ModelCanvas* object in which it displays the animation.

3.6 The `jmodel` Package

JSIM provides a visual model designer implemented using the Java `awt` package. It is a GUI-based model builder that supplies simulationists with more direct, intuitive means to build a model. It allows users to position a simulation object on a model-builder canvas by selecting a button from the tool-bar and then clicking on a location on the canvas to place the object (e.g., Server node).

3.7 The `qds` Package

Query Driven Simulation (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. One that uniformly enables the retrieval or generation of information about the behavior of systems under study (Miller and Weyrich, 1989, Miller et al., 1990, Miller et al., 1991). This means that the user must be provided with an easy to use environment where s/he may trigger complex actions by entering a simple query, for example, on a form.

QDS suggests using a database to store simulation results as well as simulation models, which might be queried by a user. Simulation is often a computationally intensive and costly exercise. Hence, it only makes sense that simulation results be stored for future use. Database Management Systems (DBMSs) provide well defined means of storage, manipulation and retrieval of large amounts of data. JSIM can be linked to a variety of DBMSs because of its reliance on Java Database Connectivity (JDBC).

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.

In the rest of this paper, we focus on the `process` and `jmodel` packages and will cover them in more detail in the next two sections. These along with the `qds` package make the most use of component-based software technology.

4 PROCESS-ORIENTED SIMULATION

The JSIM library provides 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 a real time clock. Consequently, a virtual clock is used instead of a real time clock. Virtual time simulations are also suitable when the simulation model requires substantial computation.

Pseudo-real time simulations are useful when the simulationist wishes to see the results of simulation as represented by animation. 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 help test the validity of a model. Each entity/process is implemented as a Java *Thread*. Conversion between a pseudo-real time model and a virtual time model can be done quite easily, as they are unified by the *Clock* class which includes a method to toggle between these alternatives.

The *VirtualScheduler* class is a virtual time simulation manager. It manages the scheduling of entities generating events on the basis of the time at which the entity is supposed to cause the event. *VirtualScheduler* has a FEL, implemented using a *TemporalQueue* which it uses to schedule entities. The scheduling of entities on the FEL is done on the basis of each entity's/process's activation time. The activation time of an entity is the time at which it is supposed to generate its next event. To do this, the *VirtualScheduler* has methods that are counterparts to

Java's *Thread* methods, namely *vStart, vStop, vSuspend, vResume* and *vSleep* instead of the *start, stop, suspend, resume* and *sleep* methods of the *Thread* class. The state transition diagram for threads in pseudo-real time animation is shown in figure 1, while the state transition diagram for virtual time animation is shown in figure 2.

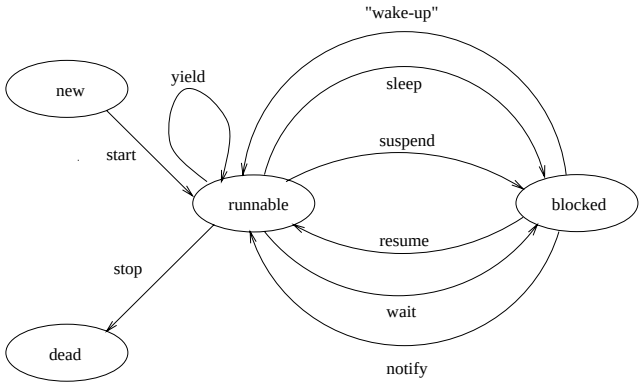

Figure 1: State Transition Diagram: Pseudo-Real Time Simulation

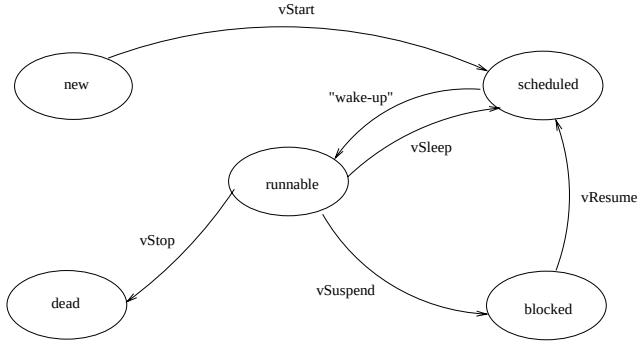

Figure 2: State Transition Diagram: Virtual Time Simulation

The *vStart* method is invoked by the *Source* class during virtual time simulation to schedule a newly created entity in the *VirtualScheduler*. During virtual time simulation, we need to ensure that only one thread is runnable at any given point in time. This is done by making *VirtualScheduler* the singular point of control in the simulation system. Every time one of the above mentioned *VirtualScheduler* methods is invoked, control is handed over to the *VirtualScheduler* which decides on the next entity to be activated based on activation time. The *vStop* method is invoked by an entity/process that has completed its life-cycle in the simulation and is ready to be terminated. The *vSuspend* and *vResume* methods are used during virtual time simulation instead of *suspend* and *resume* methods respectively, in order to ensure that the *VirtualScheduler* controls all scheduling decisions. The *vCurrentTime* method provides the current value of the virtual clock and is the virtual time counterpart to Java's *System.currentTimeMillis* method that is used during the pseudo-real time simulation.

The *VirtualScheduler* class as well as Java's *Thread* class provide the foundation for process-oriented simulations. JSIM builds several classes on top of these, facilitating concise coding of simulation models as well as automatic code generation. JSIM process-oriented models may be viewed as directed graphs (digraphs) with nodes connected by edges and entities flowing through the graph. We discuss JSIM models from this point of view in the subsequent subsections.

4.1 SimObject Class

A simulation model based on the process-interaction paradigm needs to define the entities and their life-cycle within the simulation model. An instance of the *SimObject* class represents a single simulation entity or process. The simulation model builder should extend *SimObject* to create a useful simulation entity. Precisely, the simulation model builder needs to specify the functioning or life-cycle of the simulation entity as is required by the model. *SimObject* extends the *Thread* class provided by Java. Hence, every entity in a JSIM process-interaction model is a separate thread. A *SimObject*'s logic (behavior during its lifetime) is defined by the model builder by coding its *run* method.

4.2 DynamicNode Class

DynamicNode is an abstract class that encapsulates the features common to the classes that appear as nodes in a JSIM model, currently, *Server*, *Facility*, *Signal*, *Source* and *Sink*. Every such 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 display purposes.

4.2.1 Server Class

A *Server* acts as a service provider. It initially creates a certain number of service units as defined by the model builder, thus providing servers to *SimObject*s requesting service. *SimObject*s obtain service by *request*ing a server and then *u*sing the server. If all the service units are busy, the client entity will be lost. Service may be preempted by invoking the *preempt* method. Each server also maintains statistics regarding the usage of its service units and its clients' service times.

4.2.2 Facility Class

A *Facility* is derived from *Server* since it is most similar to this class. It encapsulates a *Queue* as a private data member. Simulation entities (*SimObject*s) obtain service

by requesting a 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 in the facility's *Queue*. When the simulation entity finishes its work, it releases the server. The *queueLength* method returns the length of the queue within the facility.

4.2.3 Signal Class

A *Signal* affects the behavior of servers, by alternatively, increasing or decreasing the number of service units. For example, a Signal may be used as a traffic light in a simulation of an intersection of streets. When the signal turns on/green, servers/facilities (representing traffic lanes) in its control list will have a service unit added (using the Server *expand* method) so that traffic can flow. Conversely, when the signal turns off/red, a service unit will be removed (using the Server *contract* method) to stop the traffic flow.

4.2.4 Source Class

A *Source* is a generator or 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 time distribution.

4.2.5 Sink Class

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 lifetimes. *SimObject*s are eliminated at sinks using the *capture* method.

4.3 Transport Class

Objects from the *Transport* class form the edges of the simulation model graph, with each connecting two nodes. Simulation entities or *SimObject*s travel along transports while moving from one node to another. A transport has a default constant speed which may be changed using the *adjustSpeed* method. After *join*ing a transport, an entity moves along the transport using the *move* method. The *move* method returns false when the end of the transport is reached. Transports have been implemented as quad curves, so that the model builder can flexibly specify the edge connecting two nodes. Quad curves are part of the Java 2D API and specify a curve as a quadratic function of x and y coordinates.

4.4 Model Class

The *Model* class is derived from (extends) *Frame*, allowing multiple models to be run simultaneously, each in a separate window frame. It also implements the *Runnable* interface and its *run* method starts off all *Source* objects. Then until the simulation is over, it periodically wakes up to repaint the animation canvas. When the simulation is over, it displays statistical summary results.

5 VISUAL MODEL DEVELOPMENT

Although JSIM is designed to allow models to be rapidly hand coded utilizing JSIM's extensive class library, the easiest way to create a model is to use JSIM's visual designer, JMODEL.

JMODEL provides several buttons to control the construction of a model on a canvas. The following buttons are currently provided.

- Server. A server node provides service to client entities arriving at the node. The default number of service units (e.g., tellers) is one, but may be easily changed using the Update button.
- Facility. A facility node inherits from server and adds a queue to hold waiting client entities.
- Signal. A signal node may alter the number of service units in a server(s).
- Source. A source node produces entities with an inter-arrival time given by a random variate.
- Sink. A sink node consumes entities and records statistics about them.
- Transport. A transport edge connects two nodes (from, to) together. An entity leaving a node probabilistically chooses one of the node's outgoing edges to transport it to the next node.
- Move. Nodes may be repositioned at any time by clicking on this button. Edges automatically follow along with any node movement.

- Delete. Both nodes and edges may be deleted simply by selecting this button and clicking either on a node or edge.

- Update. The properties of nodes may be viewed and/or changed at any time by selecting this button.

- Generate. Once the visual design is complete, Java code implementing the model can be produced by clicking on this button and then anywhere on the canvas.

Models are built visually by clicking on a button to set the designer mode. Then when the mouse is clicked, an action will be performed at its location in the drawing canvas. For example, if in "Facility" mode, a new facility will be drawn at the location (see Figure 3). To connect two nodes with a transport, enter "Transport" mode and then click on the two nodes. This will cause a straight line to be drawn. To produce a curve, click on a point outside the nodes to serve as a control point.

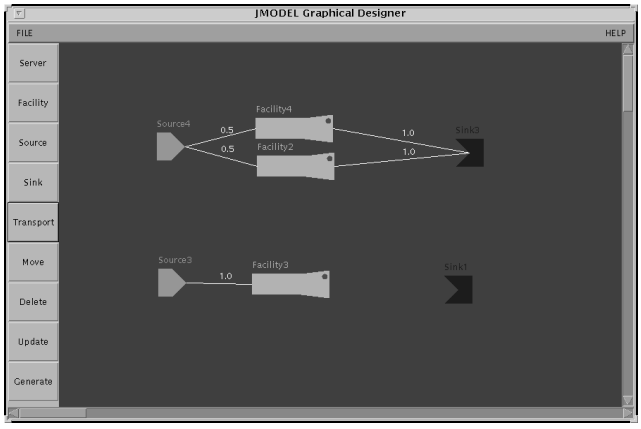

Figure 3: Screenshot of Visual Designer

The code in the `jmodel` package was created to be easily extensible. Each node in the graph is a polygon, so that adding new shapes to represent new types of nodes is easy. Similarly, each edge in the graph is a quad curves allowing flexible pathways between nodes.

6 JSIM MODELS AS BEANS

6.1 Atomic Model

An atomic model is built as a connected digraph with a single source and a single sink. The source is the producer of entities (e.g., customers) that flow through the graph to be consumed by the sink. All other nodes must have both incoming and outgoing edges.

6.2 Model

A model consists of one or more atomic models sharing a common environment and display frame. Since general models may have multiple sources, different types of entities can be created (e.g., McDonald's customers and Wendy's customers). These flows may be independent (no shared nodes), competing (shared nodes), or interacting (one playing client role and other playing server role). Allowing digraphs with multiple sources and sinks introduces complex issues of well-formedness.

6.3 Model Complex

Models can be assembled to form a model complex without requiring any traditional programming. Complex models may be built hierarchically and even animated hierarchically. Each model is represented by an icon and is implemented as a Java Bean. A designer can select from existing models to dynamically build a model complex. Individual models in the complex are linked via Java Beans events. When an entity in one model reaches a sink, an event can be triggered which will be handled in another model. Handling the event will cause an entity to be created by a source in that model. Effectively, one model is able to interact with another model (by injecting an entity). (Currently, we assume that entity injection occurs "now" in the target model with appropriate method synchronization to prevent race conditions. If a time, *t*, is given for entity injection, it is possible *t* could be in the past for the target model, so techniques for parallel simulation would be needed.)

Such interactions require that sinks in one model be linked to sources in another model. This linkage is not designed or coded in, but rather established dynamically using the facilities of visual development tools like the BeanBox in the Beans Development Kit (BDK), Java Studio or Visual Cafe, etc.

6.4 Example Model I: `Bank`

The first example model is a simple bank simulation with a single *Facility* feed by a single *Source*. The Facility has a *FIFO_Queue* and one or more service units (tellers). After receiving service, customers are captured by the *Sink*. A screenshot of the animation of this model is shown in Figure 4.

6.5 Example Model II: `FoodCourt`

The second example model is a little more complicated and non-atomic. It contains two *Source*s, three *Facilities* and two *Sink*s forming two digraphs. The top digraph represents a fast food establishment in which multiple clerks are feed by individual queues (as is done at McDonald's),

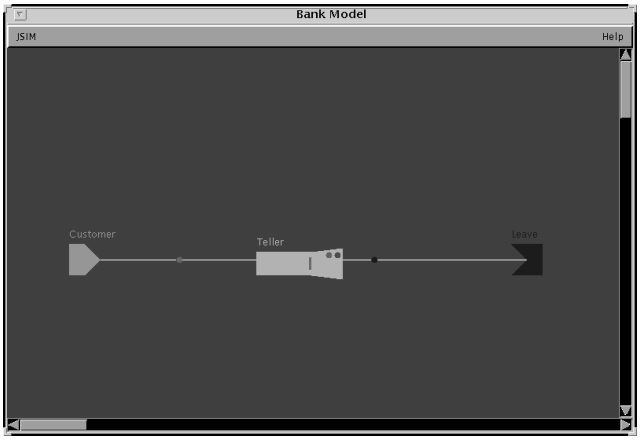

Figure 4: Screenshot of Bank Animation

while the bottom digraph represents an establishment in which the multiple clerks are feed by a single queue (as is done at Wendy's). A screenshot of the animation of this model is shown in Figure 5. This is a classic comparison of a G/G/2 queue versus two G/G/1 queues.

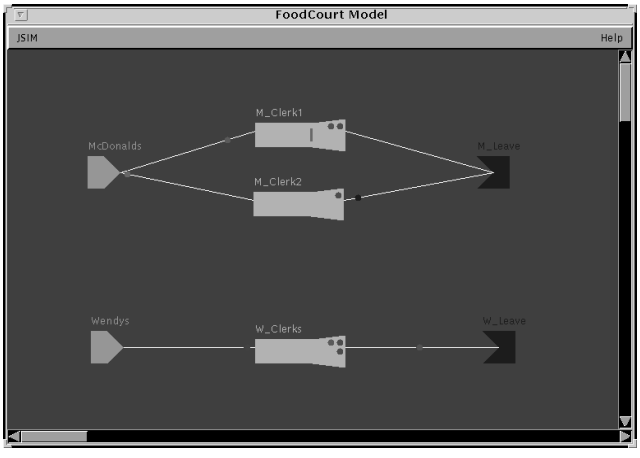

Figure 5: Screenshot of FoodCourt Animation

6.6 Assembling Models I and II

As a simple example, we can assemble the *FoodCourt* and *Bank* models into a model complex. Some of the customers leaving the food-court find that they are in need of money, so with probability *triggerProb* they chose to enter the bank.

Models I (*Bank*) and II (*FoodCourt*) can be assembled into a model complex, for example, using the BDK bean box: Select the *FoodCourt* bean from the tool box and drag-and-drop it into the bean box. Adjust any of properties using the BDK property editor. Do the same with the *Bank* bean. Since Model II will feed Model I, the linkage is made graphically by editing Model II's events. *EntityEvent*s from Model II (*FoodCourt*) are connected to Model I (*Bank*) by clicking on Model I's icon in the bean box. This brings an event target dialog box which lists target methods to select from. The selected method will be executed when these events occur. The events will be constructed and broadcast when an entity is captured by a *Sink* in the *FoodCourt* bean. This will cause the *Source* in the *Bank* bean to create and start (i.e., inject) a new entity. Figure 6 shows the mechanisms by which this happens.

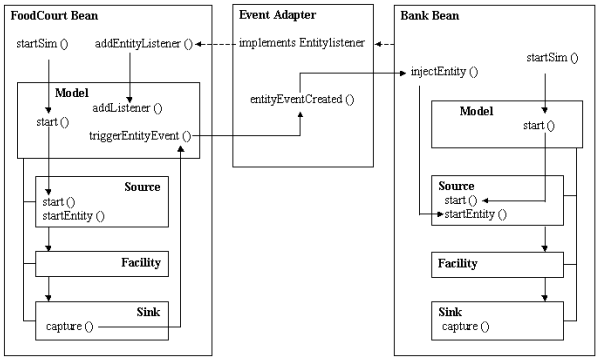

Figure 6: Model Interaction: Entity Injection

Basically, *Sink.capture* calls *Model.triggerEntity-Event* which constructs and broadcasts an *EntityEvent* to all targets. (A target had been specified (linked) dynamically and graphically using the BeanBox.) The adaptor class (which was automatically generated at event linkage time) listens for any *EntityEvent*s. When it hears one, it will call *Bank.injectEntity* (also established at linkage time). This method then calls *Source.startEntity* which injects a new entity into the bank in response to this external stimulus. These injected entities are in addition to any that the *Source* would normally (internally) produce.

Figure 7 depicts the arrangement of the *FoodCourt* and *Bank* beans as well as their relationships in the bean box. A control button is present to initiate each model (bean). Clicking on a button will cause the frame for the model to pop up. The animation actually starts when the start option in the File menu is selected. To not miss any events triggered by the *FoodCourt*, the *Bank* animation should be started first.

6.7 Multi-Frame Animation

In Java, animations are very easy to create. JSIM designs can be animated by reusing much of the code used for the visual designer. Animation brings the design diagram to life. For JSIM, models consists of entities flowing through graphs (or networks). Each node and

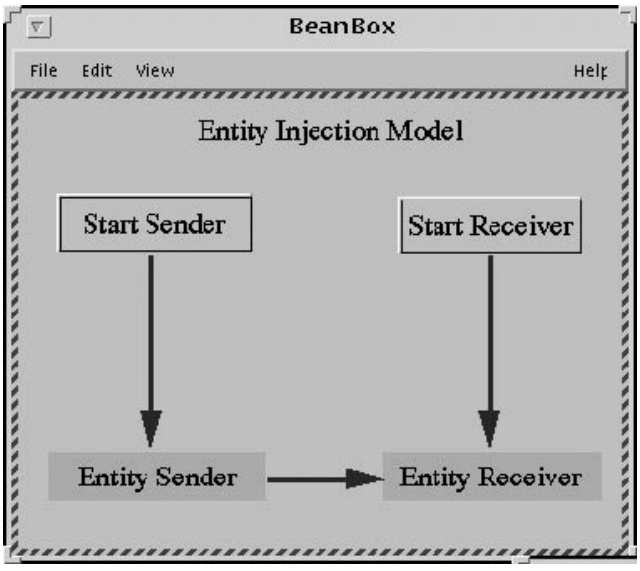

Figure 7: Model Interaction: Entity Injection

edge has fixed coordinates determined by the JMODEL visual designer. (Alternative, coordinates can be given in hand-coded constructor calls.) As entities flow, their coordinates are repeatedly updated. The *Model* class has a *displayThread* that wakes up periodically to repaint all the nodes, edges and entities. Smooth motion is obtained by updating the coordinates of entities sufficiently often.

A *Model* object initially creates an off-screen graphics buffer of the same size as its actual on-screen graphics buffer. It then paints this off-screen graphics buffer. After this, it paints the off-screen buffer onto the screen by copying it onto the on-screen graphics buffer. This technique, referred to as *double buffering*, is a good way to reduce flicker in animation.

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 clients as well as model builders, since it is often easiest to understand the simulation model by looking at animations.

Each model is animated in a single frame. At any time, several model animations may be running simultaneously, each in a different frame. They may run independently or interact through events created by one model being handled by another model. The animation of a very complex model is hard to watch in its totality and make any sense of it. With multi-frame animation, one can easily focus in on parts of the simulation by bringing the relevant window frames into the foreground of the screen. Furthermore, these multi-frame animations can be quickly reconfigured using a Java visual development tool, since models are encapsulated in Java beans.

Currently, JSIM uses Java's `awt` package and the Java 2D API to draw/paint shapes onto the screen.

7 CONCLUSIONS

JSIM is an easy-to-use Java-based simulation and animation environment that can be used as a testbed for Web-based simulation as well component-based technology. By utilizing component-based technology, in this case Java Beans, the environment is built up from reusable software components that can be dynamically assembled using visual development tools. JSIM is available for download at

http://orion.cs.uga.edu:5080/~jam/jsim.

Besides its use as a research testbed, it can also be used to teach simulation (it has been used in the CS 421/621 simulation course at the University of Georgia). In addition, it has been coded in a very modular and straightforward way, so that it can be readily extended by others.

In future releases of JSIM, our main focus will be to exploit the capabilities of new Java APIs, in order to enrich the visual appearance as well as the distributed capabilities of the system.

- Java 3D. This API will allow models to have increased visual richness. At the same time, we will introduce additional types of nodes as well as conditional branching between nodes. The visual designer currently only supports probabilistic branching.

- Enterprise Java Beans (EJB). This API will allow interacting models to be run on multiple machines just as easily as they are now run on a single machine. Furthermore, complete JSIM simulation environments will be run as a distributed system on heterogeneous platforms. In addition to EJB, Remote Method Invocation (RMI), Java IDL (CORBA) and Servlet technology will be explored.

REFERENCES

Fishwick, P. (1996). Web-Based Simulation: Some Personal Observations. In *Proceedings of the 1996 Winter Simulation Conference*, Coronado, California.

Fishwick, P., editor (1998). *Proceedings of the 1998 International Conference on Web-Based Simulation and Modeling*. Society for Computer Simulation, San Diego, CA.
http://www.cise.ufl.edu/ fishwick/webconf/full.

Law, A. and Kelton, W. (1982). *Simulation Modeling and Analysis*. McGraw-Hill, Inc., New York, NY.

Miller, J., Kochut, K., Potter, W., Ucar, E., and Keskin, A. (1991). Query-Driven Simulation Using Active KDL: A Functional Object-Oriented Database System. *International Journal in Computer Simulation*, 1(1):1–30.

Miller, J., Nair, R., Zhang, Z., and Zhao, H. (1997). JSIM: A Java-Based Simulation and Animation Environment. In *Proceedings of the 30th Annual Simulation Symposium*, pages 31–42, Atlanta, Georgia.

Miller, J., Potter, W., Kochut, K., and Weyrich, O. (1990). Model Instantiation for Query Driven Simulation in Active KDL. In *Proceedings of the 23nd Annual Simulation Symposium*, pages 15–32, Nash-ville, Tennessee.

Miller, J. and Weyrich, O. (1989). Query Driven Simulation Using SIMODULA. In *Proceedings of the 22nd Annual Simulation Symposium*, pages 167–181, Tampa, Florida.

Nair, R., Miller, J., and Zhang, Z. (1996). A Java-Based Query Driven Simulation Environment. In *Proceedings of the 1996 Winter Simulation Conference*, pages 786–793, Coronado, California.

Pritsker, A. (1986). *Introduction to Simulation with SLAM II*. Wiley, New York, NY, 3rd edition.

AUTHOR BIOGRAPHIES

JOHN A. MILLER is an Associate Professor and the Graduate Coordinator in the Department of Computer Science at the University of Georgia. His research interests include Database Systems, Simulation and Workflow as well as Parallel and Distributed Systems. Dr. Miller received the BS degree in Applied Mathematics from Northwestern University in 1980, and the MS and PhD in Information and Computer Science from the Georgia Institute of Technology in 1982 and 1986, respectively. During his undergraduate education, he worked as a programmer at the Princeton Plasma Physics Laboratory. Dr. Miller is the author of over 45 technical papers in the areas of Database, Simulation and Workflow. He has been active in the organizational structures of research conferences in all these three areas. He has served in positions from Track Coordinator to Publications Chair to General Chair of the following conferences: Annual Simulation Symposium (ASSP), Winter Simulation Conference (WSC), Workshop on Research Issues in Data Engineering (RIDE), Workshop on Workflow and Process Automation in Information Systems, and Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems (IEA/ AIE). He has also been a Guest Editor for the International Journal in Computer Simulation.

YONGFU GE is a graduate student in the MS program in the Department of Computer Science at the University of Georgia. He has already earned a PhD degree in Crop and Soil Sciences at the University of Georgia. His research interests include Simulation, Component Software and Database Systems.

JUNXIU TAO is a graduate student in the MS program in the Department of Computer Science at the University of Georgia. Her research interests include Simulation, Graphics, Distributed Systems and Database Systems.