The JSIM Web-Based Simulation Environment

John A. Miller

Andrew F. Seila

Xuewei Xiang

Computer Science Department

415 GSRC

University of Georgia

Athens, GA 30602-7404

Abstract

In its ideal form, Web-based simulation should allow simulation models as well as simulation results to be as readily distributable and composable as today's Web documents. The rapid advances in Web technology, most notably Java, are helping to make this a possibility. Support for executable Web content, universal portability, component technology, and standard high-level packages for accessing databases and producing graphical user interfaces are important enablers of Web-based simulation. Component-based software can be used to develop highly modular simulation environments supporting high reusability of software components. Because of the potentially large scope of Web-based simulation, greater demands are placed on simulation environments. They should support rapid visual model development, access to local and remote databases, techniques for executing models or federations of models in a variety of ways, and embedding of simulation within larger systems. The use of component technology in the JSIM Web-based simulation environment allows simulation models to be treated as components that can be dynamically assembled to build model federations. It also allows simulation inputs and outputs to be dynamically linked to database systems, making storage of simulation results easy and flexible.

1 Introduction

Web-based simulation is an important new direction for simulation research and development. In its ideal form, Web-based simulation should allow simulation models as well as simulation results to be as readily distributable and composable as today's Web documents. Clearly, this is easier said than done. It is easy to hyperlink documents even if authored by different people. Hyperlinking

simulation models is not so easy. Easier than hyperlinking models is making models widely and easily accessible over the Web. Executable content (e.g., Java applets) facilitates this type of client side execution. Expensive or long-running simulations can still be executed on powerful severs. This can done by having a thin client running on a browser communicate with a server where the models run as, for example, Java servlets or applications.

Because of the larger scope and scale of Web-based simulation, provision for appropriate simulation environments takes on a higher level of importance. With the potential of having models executing all over the world producing simulation results, it would be wasteful not to store these results. Java Database Connectivity (JDBC) makes it easy to store simulation results in an organized fashion in relational or object-relational databases. Although JDBC takes care of low level database connectivity issues, many higher level issues remain. Java also facilitates the construction of friendly Graphical User Interfaces (GUIs) to present results from databases. Consequently, users need not know detailed formatting specifications in order to access simulation results stored in databases all over the Web. For rapid model development, the environment should provide visual tools for designing simulation models.

Just as a Web document may be hyperlinked to several other Web documents, models may be linked together to form a model federation. A loosely coupled approach to hyperlinking simulation models into a federation is suggested in this work. (At present, JSIM is not compliant with the Department of Defense's High Level Architecture (HLA) [Page, 1998, Dahmann et al., 1998], so that JSIM models will not interoperate with HLA federates.) Using component based software, in this case Java Beans, one model is able to inject entities into another model. Using the forthcoming Enterprise Java Beans, these two models may be located anywhere on the Web.

This chapter examines several issues pertaining to simulation environments for Web-based simulation. These issues are illustrated by examining the functional modules in the JSIM Web-based simulation environment [Nair et al., 1996, Miller et al., 1997, Miller et al., 1998]. The current version has evolved to encorporate 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). JSIM, built using Java and Java Beans, can provide many of the advantages of Web-based simula-

tion [Fishwick, 1996, Fishwick, 1998b].

In the rest of this chapter, a perspective on research and development in Web-based simulation is given. A discussion of the importance of software component technology is also presented. To illustrate these issues, some parts of the JSIM Web-based simulation environment are examined in detail. JSIM consists of three layered groups of Java packages. The first group, JSIM foundation packages, has little to do with Web-based simulation, but is necessary for any simulation system. Its constituent packages are briefly described for completeness. The second group, JSIM engine packages, are more relevant since simulation models can be manifest as Java beans for distribution or assembly into model federations. The third group, JSIM environment packages, facilitates the creation of simulation models, the connection to databases, the controlled execution of models or model federations, and query driven simulation. This third group will be examined in greater detail.

2 Web-Based Simulation

After starting in 1996 [Fishwick, 1996, Buss and Stork, 1996, Nair et al., 1996], research and development work on Web-based simulation has exploded. Users on the Web are familiar with clicking on hyperlinks to find useful information or run small Web-enabled applications (e.g, applets). It is only natural that the same could be done for simulation models. While this can be done, for example, by using Java applets, this is only the beginning step along new dimensions of research and development for the simulation community. At least three dimensions can be being explored: (1) Models are being enriched by linking them to Web documents and multimedia (e.g., text to describe the model and images to connect the models with reality) [Fishwick, 1998a]. (2) Simulations can be developed by assembling model elements from multiple sites on the Web. They can even be developed by teams at geographically remote sites. Full exploitation of this requires not only the use of software component technology, but also collaborative development tools and designers, model repositories, and mechanisms for data interchange and interoperability. (3) Simulation models can now be executed on the Web in a variety of ways. The list below considers increasingly complex ways in which models can be executed on the Web.

- **Web-Based Execution**. The simplest way to allow simulation models to be executed over the Web is to include executable content within Web pages. This allows a user to download and run a simulation model from anywhere in the world. Typically, this done with a Java applet. Java provides "universal portability" allowing downloaded applets to run anywhere.

There are three techniques to ensure that all the necessary files are available for execution: (i) install a simulation system at each site, (ii) adjust linking (CLASSPATH) information to include the download site, or (iii) form each model into a self-contained archive file (jar file).

- **Thin-to-Thick Clients**. Although executable content (e.g., Java applets) provides the most common approach for model execution in Web-based simulation, it represents one extreme point on the dimension of client thickness (i.e., Java applets are thick or heavyweight clients). This is particularly useful if simulation and animation capabilities are tightly coupled. One could make the client thinner by having the animation run on the client and the simulation run on the server (e.g., as a Java servlet). If animation is not used, the client can be made very thin by simply providing a mechanism for users to define inputs and view output results. In this case, almost all of the computation is performed on the server.

- **Distributed Execution**. So far, the discussion has addressed the ability to execute models at appropriate locations on the Web (i.e., anywhere, but at one particular place). The ever-growing aggregate computational capacity of the Web argues for more. The execution of models can be distributed in four ways: (i) a user may simultaneously execute multiple models or scenarios, (ii) multiple replications of a model under a particular scenario may be simultaneously executed, (iii) loosely-coupled components of a model federation may be executed simultaneously, or (iv) techniques from parallel simulation may be applied to partition and execute a model in parallel. (In JSIM, the `qdsAgency` provides the first, while the second and third are provided by JSIM's `runAgency`. For discussions of the final option see [Ferscha and Richter, 1997, Page et al., 1997]).

- **Execution of Dynamic Model Federations**. Because of the enormous scope of the Web, the concept of an individual simulation model is too small scale. Although large and sophisticated models may be built hierarchically, the standard laws of causality apply resulting in the need for global time and synchronization. One approach to allow a large number of models to interact in a loosely-coupled fashion is to link models through less frequent external events which are not synchronized with internal events. This is easy to do using Java beans component technology. If models are implemented as Java beans, they can be easily linked using a Java visual development tool. These tools allow this linkage to be done dynamically. Models do not need to be recoded or recompiled to be included in a federation. A model federation may be assembled graphically and immediately executed. A developer can quickly

assemble a model federation by browsing one or more model repositories to extract existing models. The elemental models making up the federation could be drawn from sites all over the world.

- **Embedded Execution**. Frequently, simulation users are not principally interested in running simulation models. Rather simulation models assist them in making decisions. In addition, simulations may be useful in tuning automated or semi-automated systems. In such cases, it may be useful to have the simulations simply run behind the scenes. One type of embedding to be discussed later is Query Driven Simulation (QDS) [Miller and Weyrich, 1989]. In this case, simulation is embedded in an information system. Another example, is embedding simulation within workflow technology for the purpose of design improvement and operational tuning [Miller et al., 1995]. Although embedded simulation is generally useful, a Web-based and component-based approach makes it easier to assemble and modify the overall system.

- **Agent-Based Execution**. As is apparent from the previous discussion, the execution environment for Web-based simulation may be anywhere from straightforward to enormously complex. If it is not straightforward, then users (e.g., simulation analysts or business decision makers) require help to execute the models. Rather than having a fixed way to execute models, it would be better to have a dialog with the user to determine what s/he wants done. Once this is known, resources should be enlisted to accomplish what the user wants. Agent technology facilitates flexible solutions to such complex problems making it particularly useful for Web-based simulation [Campos and Hill, 1998]. One agent may interact with the user, and then based upon profiles of other specialized agents, contract with them to perform the work. The agents collaborate to solve the overall problem.

2.1 Component-Based Simulation Environments

Although Web-based simulation is useful and viable without software component technology, utilization of components increases the potential of Web-based simulation.

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 be 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, Web-enabled software components may be developed with either ActiveX or Java Beans. Java Beans have the advantages of platform independence and a simpler programming environment (full object-orientation, automatic garbage collection, safe use of memory and a straightforward approach to multithreading).

Java and Java Beans technology provide an excellent foundation for Web-based simulation. Java's only clear disadvantage, its speed compared to compiled languages like C++ is becoming less of an issue as Just-In-Time (JIT) and native-code compilers are becoming available.

Much of the current research in Web-based simulation involves the development of simulation systems and environments implemented in Java:

- JSIM [Nair et al., 1996, Miller et al., 1997, Miller et al., 1998],
- SimKit [Buss and Stork, 1996],
- SimJava [McNab and Howell, 1996, Page et al., 1997],
- Silk [Healy and Kilgore, 1997], and
- JUST [Pidd and Cassel, 1998].

3 Overview of JSIM

JSIM consists of three layered groups of packages: foundation packages, engine packages and environment packages (see figure 1). Each of these layers is briefly described below.

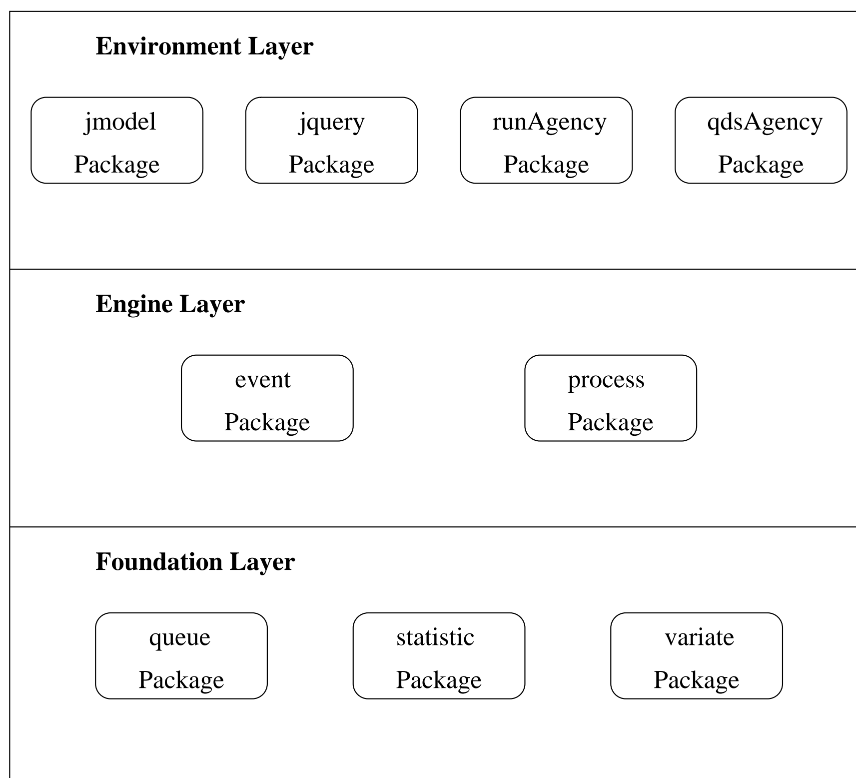

Figure 1: Layers of Packages

1. **Three Foundation Packages**. This bottom layer consists of the `queue`, `statistic` and `variate` packages. These packages are generally useful in simulation as well as in other related application domains.

- The `queue` package is rooted by the *Queue* class which is defined as an abstract base class, from which *FIFO_Queue*, *LIFO_Queue*, *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. Priority and temporal queues may be used for lines ordered by priority as well as to implement Future Event Lists and time advance mechanisms.

- 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. 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 derived from *SampleStat* which is used to collect batch statistics. Finally, histograms can be produced using the *Histogram* class.

- The `variate` package provides a wide variety of random variates. The *Variate* class is a base class from which all other variates are extended. 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 number generator. 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].

2. **Two Engine Packages**. This middle layer supports the two most popular simulation world views or paradigms. The `event` package supports the construction of event-scheduling type models, while the `process` package supports the construction of process-interaction type

models.

- The `event` package 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.
- The `process` package provides classes that are used to create simulation models following the process-interaction paradigm. A simulation model may be 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. **Four Environment Packages**. This top layer provides a flexible environment for Web-based simulation.

   - The `jmodel` package provides a visual designer for the `process` package. The designer's rendering of the model may be animated when the simulation is run.
   - The `jquery` package consists of general purpose code enabling models to access databases. It consists of beans to access databases utilizing JDBC.
   - The `runAgency` package controls the execution of one or more models. The beans in this package spare users many details of executing models or model federations.
   - The `qdsAgency` package provides a convenient means for accessing/generating simulation data (i.e., for query driven simulation [Miller and Weyrich, 1989]). This package allows simulation inputs and outputs to be stored in databases and simulation models to be launched as a part of query processing, thus providing a simple and unified view of simulation results whether they exist or not.

   The last two packages are referred to as agencies since they include Java beans capable of collaborating in order to make decisions (i.e., they include agents).

4 Process-Oriented Simulation and Animation in JSIM

Process-oriented simulations typically are implemented by representing the life-cycle of a simulation entity as a thread or coroutine. Since Java has built-in support for threads, such implementations are simplified. Threads modify the simulation state, carry out actions over time, and wait for resources. The advancement of time as well as the control of execution (which thread(s) gets to execute), is done most straightforwardly by using (i) a clock keeping track of simulated (or virtual) time and (ii) a Future Event List (FEL) to determine the order of thread activations. In the JSIM implementation, at most three threads will be active simultaneously (the virtual scheduler thread, the animation display thread, and the current simulation thread). Simulation threads may be a simulation entity (e.g., an ER patient), an entity source or a signal. Since only one simulation thread can be active at a time, race conditions as well as complex synchronization code can be avoided. The virtual scheduler loops until the FEL is empty, pulling the next thread activation off the FEL. Since the virtual scheduler runs at the lowest priority, the activated thread will take control until it finishes. In addition, the animation display thread wakes up every so many milliseconds, reads the state of the simulation to display it graphically. This thread operates relatively independently and in particular never modifies the simulation state. Indeed, the animation could be placed in a separate package or bean, without too much of an adverse effect on performance.

Animation may be turned off to maximize the speed of the simulation. If animation is used, virtual time simulation is still too fast for humans to follow the animation in detail. Consequently, the simulation/animation may be incrementally slowed down to the speed most desirable to the user. This is done by placing a `Thread.sleep` method in the main loop of the virtual scheduler. This Java method will produce a real time sleep of a given number of milliseconds.

It is also possible to use the built-in methods provided by Java (e.g., `Thread.start` and `Thread.sleep`) to control the activation of threads, as opposed to using the FEL. JSIM provides this option by using Java's built-in (real time) clock `System.currentTimeMillis`. Using this approach, several simulation threads may be active simultaneously. Unfortunately, complex synchronization code must be added to eliminate race condition (such as entities disappearing for no reason). Conversion between a virtual time and real time simulation can be done quite easily, as they are unified by the *Clock* class which includes a method to toggle between these alternatives.

JSIM's *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 useful simulation entity types (e.g., Customer or Patient). Precisely, the simulation model builder needs to specify the functioning or life-cycle of the simulation entity as 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 life-time) is defined by the model builder by coding its *run* method.

4.2 DynamicNode Class and Its Subclasses

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.

- **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 *us*ing 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.

- **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.

- **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.

- **Source Class**. A *Source* is a generator or creator of entities (*SimObject*s). It creates *SimObject*s depending on defined parameters such as inter-arrival time or the number of entities to create. A *Source* must be created for each entity type. 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.

- **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 life-times. *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.

4.5 Animation

In Java, animations are relatively easy to create, since Java provides high-level graphics/GUI packages and APIs (e.g., awt, swing, 2D API and 3D API). 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 consist of entities flowing through graphs (or networks). Each node and edge has fixed coordinates determined by the JMODEL visual designer. (Alternatively, 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. 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. Currently, JSIM uses Java's 2D API to draw/paint shapes onto the screen.

5 The JSIM Simulation Environment

As previously discussed, an environment to support Web-based simulation has greater demands on it because of the potentially huge scale and scope. In addition, Web users are not inclined to work with systems that are not easy to use. To deal with this situation, the JSIM simulation environment currently consist of the four packages: the `jmodel`, `jquery`, `runAgency` and `qdsAgency` packages.

5.1 The `jmodel` Package

JSIM provides a visual model designer implemented using the Java `swing` 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). 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 control buttons currently provided are shown in table 1.

Server	provides service to entities arriving at the node
Facility	inherits from Server and adds a queue to hold waiting entities
Signal	alters the number of service units in a server(s)
Source	produces entities with random inter-arrival times
Sink	consumes entities and records statistics about them
Transport	connects two nodes (from, to) together
Move	relocates nodes to new positions in the canvas (edges follow)
Delete	deletes nodes or edges by clicking on them
Update	views/changes the properties of selected nodes
Generate	emits Java code implementing the designed model

Table 1: JMODEL's Control Buttons

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 5). 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.

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.

5.2 The `jquery` Package

The `jquery` package contains classes and beans making it easy for other components to access relational (e.g., Mini SQL) and object-relational (e.g., Oracle 8) databases. JSIM can be linked to

a variety of database management systems because of its reliance on Java Database Connectivity (JDBC). Typically, this is as easy as changing the connection specification (e.g., to change the connection specification from Mini SQL to Oracle simply change the url as shown below).

```
url = "jdbc:msql://orion.cs.uga.edu:1888/jsim";
url = "jdbc:oracle:thin://orion.cs.uga.edu:1521";
```

In the worst case, the JDBC driver may need to be changed, but this only effects a few lines of code since the JDBC API is the same in any case.

A couple of important beans in this package are the following.

- `DbQuery` Bean. The `DbQuery` bean responds to events that embed SQL queries. Using JDBC it sends a query to the designated database and places the results in an AbstractTableModel allowing the results to be readily sent back to the requester or displayed.

- `DbUpdate` Bean. The `DbUpdate` bean similarly responds to events that embed SQL updates. It may be used by model beans in order to store their results in databases. This makes it easy to run models with or without a database, or switch databases since the association between the beans is established dynamically.

5.3 The `runAgency` Package

In JSIM, models may be formed as Java beans so that model federations may be dynamically formed out of simpler models and immediately executed. In addition, models may be dynamically associated with beans from the environment. Model beans may be either atomic or composite, and may be assembled into model federations.

5.3.1 Atomic/Composite Models

The simplest type of model is an 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.

A composite model consists of two or more 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.

Composite models may be formed by combining two or more simpler existing models. This is done by loading the two models into the `jmodel` designer and allowing the user to link them as appropriate. New code is then generated for this composite model. Note that synthesizing a new composite model is less dynamic than assembling a new model federation. This is because elements are more tightly coupled in a composite model (e.g., they share a common Future Event List).

A simple illustration of a composite model is the *FoodCourt* example. 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 fed by individual queues (as is done at McDonald's), while the bottom digraph represents an establishment in which the multiple clerks are fed by a single queue (as is done at Wendy's). A screenshot of the animation of this model is shown in figure 2. This is a classic comparison of a G/G/2 queue versus two G/G/1 queues.

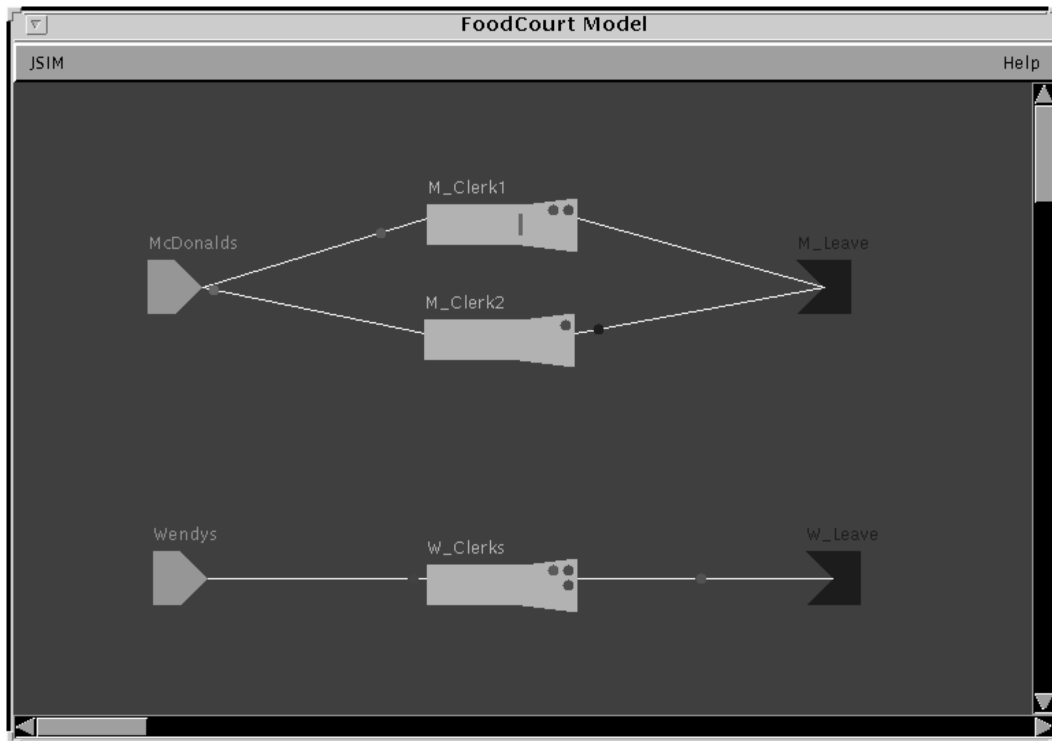

Figure 2: Screenshot of FoodCourt Animation

5.3.2 Assembling Models into a Model Federation

Models can be assembled to form a model federation without requiring any traditional programming. Each model may be represented by an icon and implemented as a Java bean. A designer can select from existing models to dynamically build a model federation. Individual models in the federation are linked via Java bean events. When an entity in one model reaches a sink, an external event can be triggered which will be handled in another model. Handling the external 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.

Consider the following situation (the *FoodAndHealth* model federation) as one to connect the two models. Some of the customers leaving the *FoodCourt* find that they are feeling ill, so with probability *triggerProb* they choose to go to a hospital Emergency Room (ER). Patients enter the ER and wait to see the Triage Nurse, who will decide whether the patient will see the Physician's Assistant or one of the ER Doctors. Before this step, the patient must register with the Admit Clerk. A screenshot of the animation of this model is shown in figure 5. The *Signal* is used to periodically change the number of ER Doctors on duty.

The *FoodCourt* and *ER* models can be easily assembled into a model federation. For example, using the BDK bean box one would carry out the following steps:

1. Select the *FoodCourt* model/bean from the tool box and drag-and-drop it into the bean box.

2. Adjust any of properties using the BDK property editor.

3. Do the same with the *ER* model/bean.

4. Edit the *FoodCourt*'s events and connect its *EntityEvent*s to the *ER* by clicking on the *ER*'s icon in the bean box. This will enable the *FoodCourt* model to feed the *ER* model.

5. Select the appropriate method from the event target dialog box which pops up.

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 *ER* bean to create and start (i.e., inject) a new entity. Basically, *Sink.capture* calls *Model.triggerEntityEvent* which constructs and broadcasts an *EntityEvent* to all targets (specified or linked dynamically and graphically using the BeanBox.) The adaptor class (which is automatically generated at event linkage time) listens for any *EntityEvent*s. When it hears one, it will call *ER.injectEntity* (also established at linkage time). This method then calls *Source.startEntity* which injects a new entity into the *ER* in response to this external stimulus. These injected entities are in addition to any that the *Source* would normally (internally) produce.

When the model federation is executed, each model is animated in a separate frame. The models may run independently, but typically will interact through external events created by one model being handled by another. 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.

A JSIM model federation is run (executed) by beans from the runAgency. The principal agents controlling the execution are described in the following subsections.

- **`ModelAgent` Beans**. A `ModelAgent` is responsible for executing a JSIM model. It will start up the model for a certain amount of time/work. The model will report back the results of this run/batch. The model agent will then, based on the selected output methodology [Law and Kelton, 1982, Banks et al., 1996, Fishman and Yarberry, 1997], either terminate or continue the execution (more runs or batchs) of the model. The JSIM approach is very modular and flexible allowing new model agents to be added at any time, so long as they follow the rules of the game. Currently, two types of model agents have been implemented - one implementing the Method of Independent Replications and the other implementing the Method of Batch Means.

- **`ScenarioAgent` Beans** The overall control of the execution of a model federation is the responsibility of the `ScenarioAgent`. After collaborating with other agents to choose an approach to executing the model federation, the scenario agent will inform the model agents to use an appropriate output analysis methodology (e.g., method of independent replication, method of batch means) as well as appropriate goals (e.g., confidence levels and relative precisions).

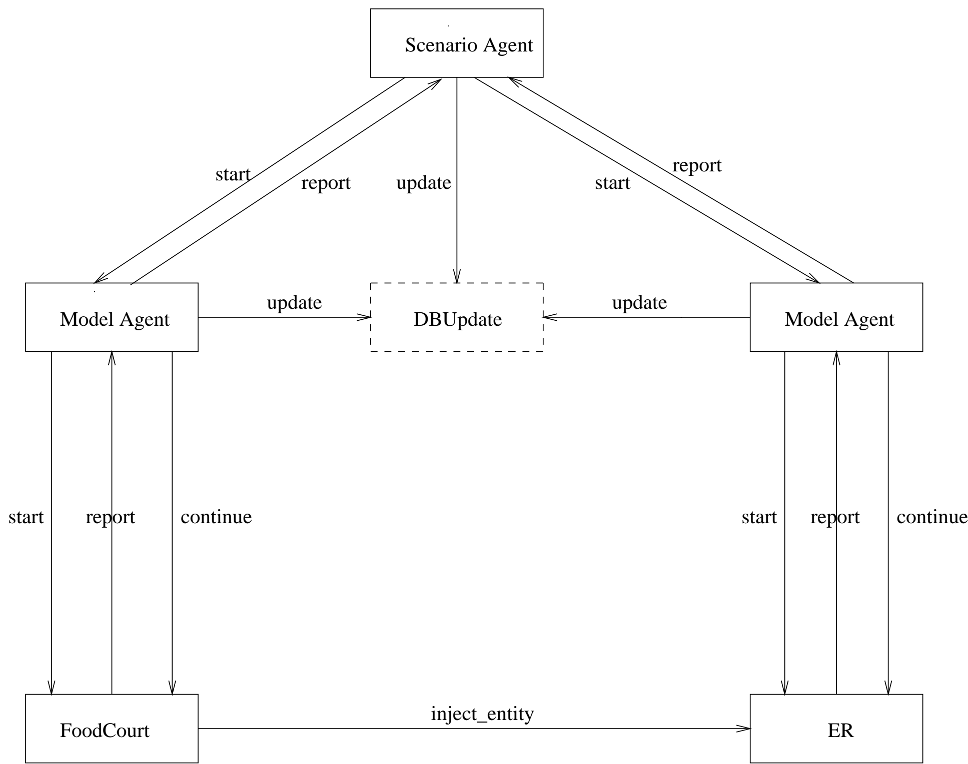

Figure 3: Beans in the runAgency

An example of the execution framework for FoodAndHealth model federation is shown in figure 3. It depicts the arrangement of the *FoodCourt* and *ER* beans as well as their relationships in the bean box.

5.4 The `qdsAgency` Package

Simulation can be embedded in other systems in several ways. One such approach, referred to as 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.

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 provide efficient techniques for the storage, manipulation and retrieval of large amounts of data. Consequently,

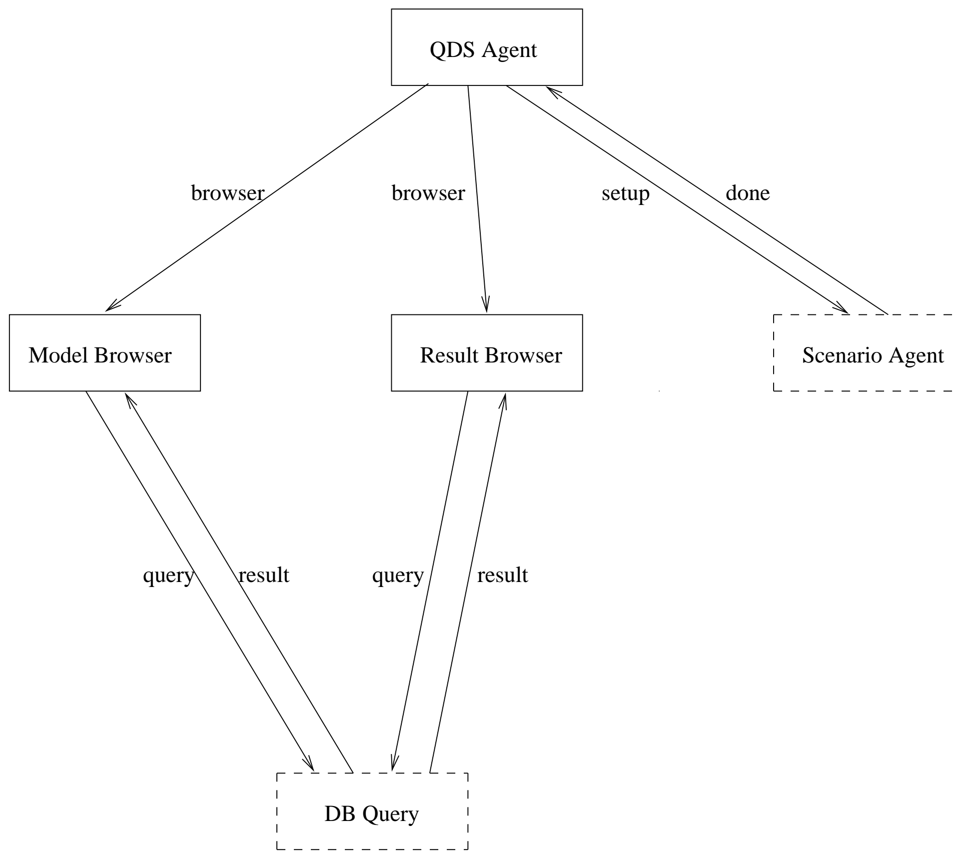

Figure 4: Beans in the qdsAgency

a QDS system should use databases to store simulation results as well as simulation models. In addition, other information useful for decision making would be stored and associated with the simulation data.

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.

The principal agent that facilitates a user's interaction with the QDS system is the `QdsAgent` bean. It is assisted by the `ModelBrowser` and `ResultBrowser` beans. A typical session involving a user and the QDS system can be explained as follows (see figure 4).

1. Start the `QdsAgent`.

2. Browse or query the available models or model federations. This work is done by the

ModelBrowser at the request of the QdsAgent.

3. Once the user selects a model or model federation of interest, the QdsAgent allows the user to browse the corresponding simulation results using the ResultBrowser. Alternatively, the user may define a scenario of interest for the model or model federation. This is facilitated by the QdsAgent obtaining the list of model parameters from the database (including default values).

4. After parameter specification, the QdsAgent checks the database to see if information exists regarding the prior execution of the selected model or model federation with the given parameter values. If the simulation results exist, the QdsAgent retrieves them from the database via the ResultBrowser. If not, the QdsAgent collaborates with a ScenarioAgent to execute a model or model federation with the given parameters. Execution will cause the results to be stored in a database. The results can now be shown using the ResultBrowser.

6 Conclusions

The JSIM Web-based simulation environment provides the following advantages over conventional simulation environments:

- JSIM supports the distribution of both simulation results and simulation models, since they can be accessed over the Web. This provides the potential to make simulation analysis much more widely available.

- JSIM supports universal portability in that models can be run essentially anywhere. The use of the Java language along with its comprehensive class libraries make this possible. The use of JDBC makes it easy to switch between several database management systems.

- JSIM supports the dynamic assembly of model federations. 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. Model beans can be linked to form model federations and may be linked to environment beans to control their execution and save their results in databases.

- JSIM provides simple and uniform access to simulation results based on the notion of query driven simulation. The users simply ask for information and it is up to the system to figure

out how to get it, e.g., by accessing databases and/or running simulation models.

Besides JSIM's use as a research testbed for Web-based simulation, 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. JSIM is freely available for download at

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

Future releases of JSIM will exploit the capabilities of new Java APIs, in order to enrich the visual appearance, the distributed capabilities and the interoperability 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 distributed systems on heterogeneous platforms. In addition to EJB, Remote Method Invocation (RMI), Java IDL (CORBA) and Servlet technology will be explored.

- XML for Data Interchange. The eXtensible Markup Language (XML) promises to not only become the new improved HTML, but will likely become the standard for data interchange between a variety of systems, tools and applications, thus simplifying interoperability. As one example, JSIM now uses Java io serialization to save `jmodel` designs. This binary data may either be stored in files or databases (e.g., as Binary Large Objects (BLOBs)). Utilizing work in currently ongoing projects to serialize Java objects as XML documents, JSIM objects can be directly stored in object-oriented or object-relational databases without any custom coding.

References

[Banks et al., 1996] Banks, J., Carson, J., and Nelson, B. (1996). *Discrete-Event System Simulation, 2nd Ed.* Prentice-Hall, Inc., Upper Saddle River, NJ.

[Buss and Stork, 1996] Buss, A. and Stork, K. (1996). Discrete Event Simulation and World-Wide Web Using Java. In *Proceedings of the 1996 Winter Simulation Conference*, pages 780–785, Coronado, California.

[Campos and Hill, 1998] Campos, A. and Hill, D. (1998). Web-Based Simulation of Agent Behaviors. In *Proceedings of the 1998 WEBSIM Conference*, San Diego, California.

[Dahmann et al., 1998] Dahmann, J., Fujimoto, R., and Weatherly, R. (1998). The DoD High Level Architecture: An Update. In *Proceedings of the 1998 Winter Simulation Conference*, pages 797–804, Washington, DC.

[Ferscha and Richter, 1997] Ferscha, A. and Richter, M. (1997). Java Based Cooperative Distributed Simulation. In *Proceedings of the 1997 Winter Simulation Conference*, pages 381–388, Atlanta, Georgia.

[Fishman and Yarberry, 1997] Fishman, G. and Yarberry, S. (1997). An Implementation of the Batch Means Method. *INFORMS Journal on Computing*, 9(3):296–310.

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

[Fishwick, 1998a] Fishwick, P. (1998a). An Architectural Design for Digital Objects. In *Proceedings of the 1998 Winter Simulation Conference*, pages 359–365, Washington, DC.

[Fishwick, 1998b] Fishwick, P., editor (1998b). *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.

[Healy and Kilgore, 1997] Healy, K. and Kilgore, R. (1997). Silk: a Java-Based Process Simulation Language. In *Proceedings of the 1997 Winter Simulation Conference*, pages 475–482, Altanta, Georgia.

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

[McNab and Howell, 1996] McNab, R. and Howell, F. (1996). Using Java for Discrete Event Simulation. In *Proceedings of the Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW)*, pages 219–228, Univ. of Edinburgh.

[Miller et al., 1998] Miller, J., Ge, Y., and Tao, J. (1998). Component-Based Simulation Environments: JSIM as a Case Study Using Java Beans. In *Proceedings of the 1998 Winter Simulation Conference*, pages 373–381, Washington, DC.

[Miller et al., 1991] 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 et al., 1997] 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 et al., 1990] 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, Nashville, Tennessee.

[Miller et al., 1995] Miller, J., Sheth, A., Kochut, K., Wang, X., and Murugan, A. (1995). Simulation Modeling Within Workflow Technology. In *Proceedings of the 1995 Winter Simulation Conference*, Arlington, VA.

[Miller and Weyrich, 1989] 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 et al., 1996] 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.

[Page, 1998] Page, E. (1998). The Rise of Web-Based Simulation: Implications for the High Level Architecrure. In *Proceedings of the 1998 Winter Simulation Conference*, pages 1663–1668, Washington, DC.

[Page et al., 1997] Page, E., Moose, R., and Griffin, S. (1997). Web-Based Simulation in SimJava Using Remote Method Invocation. In *Proceedings of the 1997 Winter Simulation Conference*, pages 468–474, Atlanta, Georgia.

[Pidd and Cassel, 1998] Pidd, M. and Cassel, R. (1998). Three Phase Simulation in Java. In *Proceedings of the 1998 Winter Simulation Conference*, pages 367–371, Washington, DC.

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

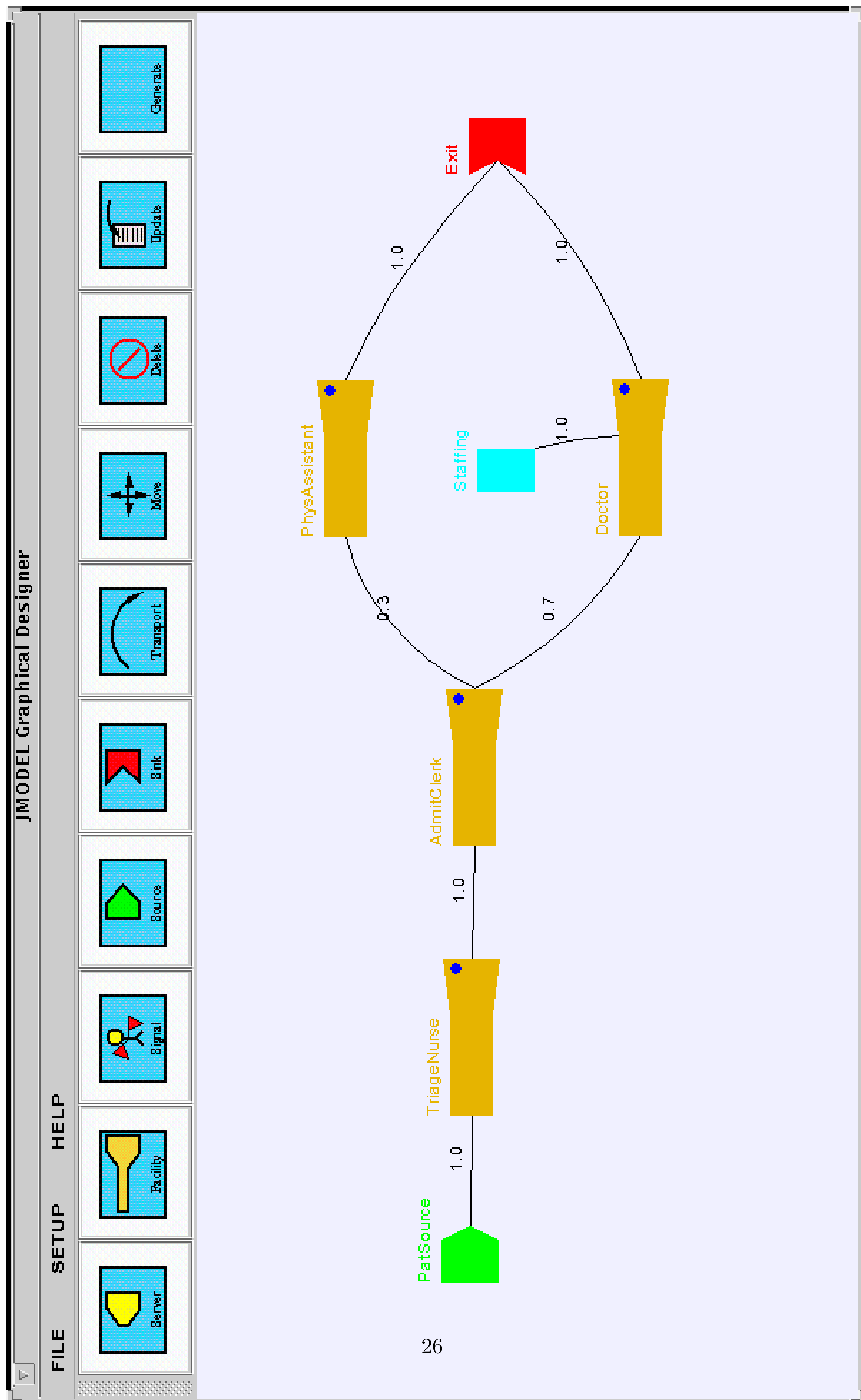

Figure 5: Screenshot of ER Model