Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

FreeSML: DELIVERING ON THE OPEN-SOURCE SIMULATION LANGUAGE PROMISE

John J. DiLeo

Department of Engineering Management
and Systems Engineering
The George Washington University
Washington, DC 20052, U.S.A.

ABSTRACT

FreeSML is a Java-based simulation language, providing support for process-oriented and event-oriented simulation, along with limited support for continuous-variable simulation. The core simulation engine is indirectly derived from that of Silk 1.3, and the language's public interface is based heavily on those of Silk and SSJ. Unlike earlier languages, FreeSML was developed with the specific intent that it be released as an open-source package, and has been released under the Free Software Foundation's Lesser General Public License.

1 INTRODUCTION

1.1 Motivating Application

Beginning in 2001, analysts at the MITRE Corporation's Center for Advanced Aviation System Development (CAASD) undertook the development of a new discrete-event simulation of air traffic control (ATC), focusing on the actions and interactions of controllers managing en-route traffic in the National Airspace System (NAS). Of primary interest were the effects of changes in operating procedures and information awareness—at the level of individual en-route controllers and controller teams—on the overall performance of the NAS.

In the initial version of this new ATC simulation, each air traffic controller (or more correctly, each *ControlTeam*) was associated with a sector of en-route airspace, and possessed a *Resource* instance representing their *attention*. Tasks requiring *attention* from the *ControlTeam* were placed in a priority queue, and the highest-priority waiting task was removed from the queue to be processed when the *attention* became available.

As the new simulation took shape, the development team recognized a problem with this approach. In extant simulation languages, the ordering of elements in priority queues is determined based on the priority of each element *at the time it is added to the queue*. The priorities of waiting tasks, however, are not static and can vary with changes in system state.

To better reflect these inherent priority variations, it was determined that another approach was required. The most natural approach (from a simulation developer's standpoint) would be to utilize priority queues where the ordering of entries would be automatically updated, based on the *current* priorities reported by the queue's elements. Vaucher (1971) mentioned such an approach in passing, but no record exists in the literature of any previous implementation.

This approach was not, however, used in the new ATC simulation, due to lack of management support for the required development effort. This author's involvement with the ATC simulation project ended in 2003, and we elected to develop this new approach independently, in conjunction with our doctoral research (DiLeo 2005b).

1.2 Modifications to Silk 1.3

At the beginning of the new ATC simulation effort at CAASD, the development team selected Silk (Healy and Kilgore 1997) as the implementation language for the new simulation. Version 1.3 of the language had been released in early 2001, and was adopted (in conjunction with the Java 2 Software Development Kit, version 1.3) as the development platform.

Deployment of the new simulation was anticipated on several platforms, including Web-based execution. Unfortunately, Silk 1.3 as released did not function correctly on all platforms; reliable execution was assured only on Windows 2000 systems. As part of a cooperative agreement between CAASD and Richard Kilgore (then of ThreadTec, Inc.), this author developed a variant of the Silk 1.3 executive (dubbed "Silk 1.4-MITRE") which executes correctly on all Java 2 platforms on which it has been tested.

Under the terms of CAASD's agreement with Dr. Kilgore, it was stipulated that the changes made by this author (using public funds) must eventually be incorporated into an open-source language, for use by the wider simulation community.

1.3 Design Considerations

As a result of the parallel developments described in the preceding sections, we found ourselves in possession of two "chunks" of software: a simulation executive derived from Silk 1.3 (i.e., "Silk 1.4-MITRE") and a collection of data structures and algorithms supporting state-dependent and stochastic "reordering" of event sets and priority queues. By themselves, these "chunks" of software did not constitute a complete language; rather, they implemented specific elements that could be useful when incorporated in a complete language implementation.

In the absence of an active open-source simulation language effort to which these elements could be contributed, we elected to develop a complete language incorporating them. We have named this new language FreeSML, to emphasize its implementation as free software, released under the GNU Lesser General Public license (LGPL).

It was never intended that an entirely new language be developed; rather, we sought to incorporate our new contributions into an established, familiar framework. For this reason, we designed the FreeSML language to be similar to existing object-oriented simulation languages, and to adopt existing metaphors and modeling constructs whenever feasible. The greatest influences on the design of FreeSML were Silk 1.3 (Healy and Kilgore 1997) and SSJ (L'Ecuyer, Meliani, and Vaucher 2002).

Both Silk 1.3 and SSJ are implemented as Java packages, as is FreeSML. The core of the public interface to FreeSML is an integration of the interfaces published for these two languages. Additional features and metaphors were adopted from other languages, including SLX (Henriksen 1993) and SIMSCRIPT II.5 (Russell 1994), where they represented concepts not present in either Java-based language (e.g., *waitUntil* and time-ordered event lists).

2 IMPLEMENTING CORE LANGUAGE FEATURES IN FreeSML

Kiviat (1969) enumerated six fundamental features a general-pupose simulation language must possess to be of significant utility:

- Representation of simulated time;
- Management of simulated entities, including:
  - Creation and destruction of entity instances;
  - Data structures for managing collections of entities; and
  - Management of entity states;
- Generation of uniform pseudorandom numbers;
- Generation of non-uniform random variates;
- Statistical data collection; and
- Reporting facilities, for summary and/or detailed performance data.

In the following sections, FreeSML's support for each of these required capabilities will be addressed, along with other significant language features.

2.1 Simulated Time

In FreeSML, the simulated time is maintained by the *clock* field of the *SimManager* class. The value of the simulation clock is maintained in a double-precision floating-point attribute (a Java `double`) within a *DoubleStateVar* instance. The underlying design of the *DoubleStateVar* class is adopted from Silk (along with *IntStateVar* and its abstract parent *StateVar*), and provides a mechanism for implementing conditional delays based on changes in the variable's value. State variables, and their use in implementing conditional delays, are addressed in section 2.4.

The value stored in the *clock* field represents the number of "time units" that have elapsed since the beginning of the simulation. All time-related operations within the simulation executive (e.g., inter-event delays) are expressed in terms of time units. The relationship between time units and seconds of simulated time can be customized by the simulation developer, using the static *setTimeUnitsPerSecond* method in the *SimTime* class. In many cases, the default 1:1 ratio is appropriate. The *clock* value is advanced using a conventional next-event mechanism, managed by the *Executive*.

The *SimTime* class provides several useful features for working with dates and times in FreeSML. *SimTime* retains a static *baseDate* field (an instance of *java.util.Date*), whose value represents the date and time corresponding to a clock value of 0.0. By default, *baseDate* is initialized to midnight on January 1 of the current year. The *SimTime* class provides static methods for converting *clock* values to dates and times, and vice versa; these methods make use of the value of *baseDate*, in conjunction with the time-unit:second ratio.

Instances of the *SimTime* class can be used to store individual time values. The *SimTime* class extends the abstract *Number* class (defined in the *java.lang* package) and, as with other *Number* types (e.g., *java.lang.Double*), *SimTime* instances are immutable. As a *Number* subclass, *SimTime* also implements the *Comparable* interface. Several convenience methods are provided to support time-related comparisons (e.g., *isBefore, isAfter,* and *timeUntil*).

2.2 Management of Simulated Entities

The representation of simulated entities in FreeSML is based on a combination of those used in Silk 1.3 and SSJ. In Silk 1.3, all simulated entity types are implemented as subclasses of the abstract *Entity* class, and support is provided only for process-oriented entities. In SSJ, process-oriented entities are implemented using subclasses of *Process*, while events are implemented using subclasses of

Event. In FreeSML, the abstract *Entity* class has two (abstract) children—*ProcessEntity* and *EventEntity.*

ProcessEntity instances represent entities within the simulated system that have a "lifetime": they are created, perform a sequence of actions representing their behavior with respect to the system, and they eventually depart. While in the simulation, *ProcessEntity* instances can incur delays, including conditional delays while competing with other instances for constrained resources.

Each process-oriented entity type in a simulation must be defined by the programer as a subclass of *ProcessEntity*, and its behavior must be specified by overriding the abstract *process* method inherited from the parent class

EventEntity instances represent "events" that cause instantaneous changes in the system state. Because their execution is intended to span zero time, an *EventEntity* cannot incur delays. Each event type in the simulation must be defined by the programer as a subclass of *EventEntity*, and its behavior must be specified by overriding the abstract *actions* method inherited from the parent class.

2.2.1 Creation and Destruction of Entity Instances

Instances of all Entity subclasses are "recycled" by the simulation engine, rather than created directly using the `new` operator and garbage collected after one use. This behavior is essential to the management of *ProcessEntity* instances, due to high overhead and virtual machine restrictions associated with creation and destruction of threads.

The *EntityManager* keeps track of all *Entity* instances that have been freed and, when a new instance of an *Entity* class is requested (using the *getEntity* method), satisfies the request with a recycled instance if possible. If no "recycled" instances are available, a new instance created.

In either case, the entity's *init* method is invoked to place the instance in a programmer-defined initial state. This procedure (adopted from Silk 1.3) is used for object initialization because constructor code is invoked by the Java virtual machine (JVM) only when the instance is first created. Code contained in the *init* method is executed every time an entity instance is "created" whether or not it was "recycled."

Invocation of *init* methods is not automatically "chained" like that of object constructors, so the developer is responsible for invoking *super.init* when appropriate. The *Entity* class contains an empty *init* method, so no initialization is performed by default.

Each *Entity* instance can create new instances of the same entity type using the *createAnotherIn* method. This method accepts a `double` argument specifying a delay (in time units) before creating the new instance..

Entity instances automatically terminate upon reaching the end of their *process* or *actions* method, and the simulation executive invokes *freeEntity* to recycle them. It is also possible to explicitly invoke the *terminate* method on a *ProcessEntity*, to force its termination regardless of its current state. The *cancel* method can be invoked to cancel a scheduled *EventEntity* activation.

2.2.2 Management of Collections of Objects

Because FreeSML utilizes the Java 2 Platform, collections of generic objects can be managed using the Java Collections API. In addition to these facilities, FreeSML includes a set of collection classes whose states (i.e., current length and capacity) can be monitored.

ProcessEntity instances can reside in *Queues*. Two types of *Queue* are defined in FreeSML: *FifoQueue* and *PriorityQueue*. The determination of priority ordering for a *PriorityQueue* instance can be made when a new item is added to the queue (insertion-ordered) or when the "first" item is requested *from* the queue (removal-ordered). Removal-ordered queues are discussed further in section 3.

To permit monitoring of changes in the occupancy of a *Queue* instance, the *length* attribute of each queue is defined as an *IntStateVar*. Using this mechanism, conditional delays can be defined for *ProcessEntity* instances, based on the state of one or more *Queue* instances (e.g., a processor remaining idle while its input queue remains empty).

Support for occupancy monitoring of generic collections is provided by a set of three "wrapper" classes: *MonitoredList, MonitoredSet,* and *MonitoredSortedSet*. Each of these classes implements the corresponding Collections API interface; to permit monitoring, the occupancy of the collection is retained in an *IntStateVar* attribute.

2.2.3 Management of Entity States

Schriber and Brunner (2001) describe five states in which a simulated entity can reside within a simulation: Active, Ready, Time-Delayed, Condition-Delayed, and Passive. To incorporate the entire lifetime of Java objects used to represent entities, we have found it useful to add two "external" states: Created and Terminated. Objects in the latter two states exist within the simulation program, but not within the simulated system. Figure 1 depicts the relationships (transitions) among these seven states supported by FreeSML.

Because events are defined to occur instantaneously and cannot incur delays, *EventEntity* instances are permitted to exist in only a subset of the listed states—they cannot occupy the Condition-Delayed or Passive states.

The *ProcessEntity* and *EventEntity* classes provide methods to effect each of the permitted state transitions; for example, the *delay* method in *ProcessEntity* moves the entity from the Active to the Time-Delayed state.

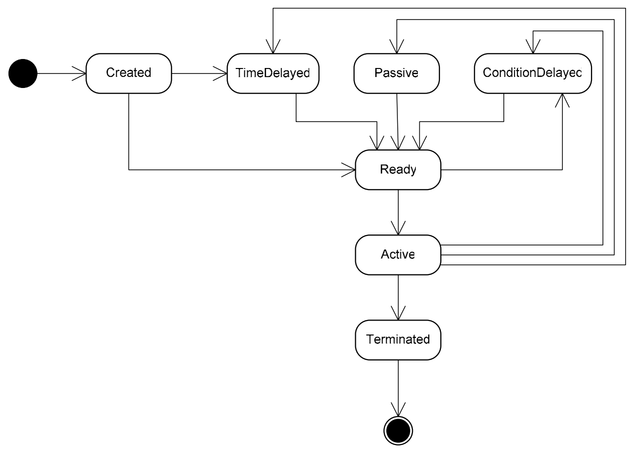

Figure 1: States of Simulated Entities

The sets of entities residing in the various states are maintained using several data structures. The set of Time-Delayed entities is maintained in the simulation's future events set (FES). The FES is an instance of an *EventSet* subclass. *EventSet* is defined as an abstract class, and several concrete time-ordering implementations are provided in the current FreeSML release:

- Doubly-linked linear list;
- Doubly-linked linear list with median pointer (McCormack 1979);
- Splay tree (Sleator and Tarjan 1985);
- Two-list data structure (Blackstone, Hogg, and Phillips 1981a, 1981b); and
- Array-based implicit binary heap (Gonnet 1976, McCormack 1979, Bentley 1985).

Regardless of the data structure used to place events in increasing time-order, each entry in the FES is itself a collection—an instance of *NoticeQueue*. Each *NoticeQueue* contains one or more *EventNotice* instances. This two-level data structure (a time-ordered collection of collections) permits the simulation developer to separately modify the algorithm used for time-ordering, and the ordering scheme used for time-tied events.

Like the *Queue* class, *NoticeQueue* is abstract, and three implementations are provided: *FifoOrdered*, *InsertionOrdered*, and *RemovalOrdered*. The same underlying data structures are used as for *Queues* (see section 3).

The set of Ready entities is maintained in the current events set (CES), which is an instance of *NoticeQueue.* At each clock advance, the clock's value is advanced to the time associated with the first *NoticeQueue* in the FES. That *NoticeQueue* is removed from the FES to become the CES.

References to Condition-Delayed entities are stored in *ValueListenerQueues* maintained by the state variables whose states they are monitoring. *ValueListenerQueue* instances support the three ordering methods discussed earlier. Created and Passive entities are not automatically tracked by the simulation executive; they must be managed by the simulation developer. Terminated entities are stored in the free lists maintained by the *EntityManager*.

2.3 Queues and Resources

As noted earlier, the *Queue* class provides a mechanism for retaining ordered lists (in either FIFO or priority order) of *ProcessEntity* instances. Methods provided in the *ProcessEntity* class for interacting with queues include:

```
protected final int enqueue( Queue toJoin );
protected final int enqueueConditional( Queue
toJoin );
protected final boolean dequeue( Queue
toLeave );
```

The *enqueue* method will add the invoking *ProcessEntity* to the specified *Queue*, whether or not that *Queue* is at its specified *capacity*, while the *enqueueConditional* method blocks entry into the *Queue* if it is currently full. The first form should be used for uncapacitated *Queues*, or after a conditional wait involving a complex blocking condition that includes a check on *Queue* capacity.

Constrained resources (e.g., bank tellers) can be represented in FreeSML using instances of the *Resource* class. This class represents a passive resource with *n* identical units. *ProcessEntity* instances use the conventional request/release mechanism to obtain units of the resource as needed. The following methods are provided in FreeSML:

```
protected final void request(Resource toUse);
protected final void seize(Resource toUse);
protected final void release(Resource toF-
ree);
```

The methods list request, seize, or release one unit of the specified *Resource,* respectively. A second version of each method is available that accepts a second integer argument, specifying the number of units. The *request* method is used to obtain units of the *Resource*, as soon as sufficient units become available; a conditional wait occurs if enough units are not yet available.

The *seize* method grabs units of the *Resource* without first checking availability; a runtime exception is generated if insufficient units exist. The latter should be used only after a conditional wait involving a complex blocking condition that includes availability of the *Resource.*

2.4 State Variables and Conditional Delays

In FreeSML, support for conditional delays is provided by *StateVar* instances. The functionality provided in FreeSML is based on that found in Silk, and is similar to that provided by `control` variables in SLX. All conditions on which a simulation developer wishes to base a conditional delay must be expressed in terms of the values of one or more *StateVar* instances.

FreeSML provides four *StateVar* subclasses: *Condition, IntStateVar, DoubleStateVar,* and *ContinuousStateVar. IntStateVar* and *DoubleStateVar*, as their names imply, store `int` and `double` values, respectively; *Condition* instance store `boolean` values. Each *StateVar* has an associated *ValueListenerQueue*, used to maintain references to the *ProcessEntity* instances currently monitoring changes in that *StateVar*'s value.

The value contained in a *StateVar* instance can be accessed in one of two ways: using the *getValue* method, which causes no simulation side-effects; or using the *checkValue* method, which *does* generate side-effects, and whose invocation must be associated with a *ProcessEntity* instance. Updates to the value contained in a *StateVar* are made by invoking the *setValue* method, with an argument of the appropriate type.

A *ProcessEntity* instance enters a conditional delay by invoking its *condition* method. To generate a conditional delay, this method must be passed a Boolean argument that (directly or indirectly) invokes the *checkValue* method of at least one *StateVar*. If the Boolean argument evaluates to `true`, the invoking *ProcessEntity* enters the Condition-Delayed state until the value of a referenced *StateVar* changes.

When the value of any referenced *StateVar* changes, the *ProcessEntity* is notified and returns to the Ready state, awaiting subsequent activation. On the entity's return to the Active state, the *condition* method immediately exits, and the invoking entity should re-check the condition to determine if it has in fact become `false`. In practice, calls to the *condition* method are "wrapped" in an empty `while` loop, to permit repeated checks of the Boolean condition:

```
while( condition( <Boolean expression> ) );
```

This construct, taken as a whole, is essentially a "wait while" statement, semantically equivalent to the `wait until` statement provided by many languages, including SLX. In addition to this construct, the *ProcessEntity* class provides two *waitUntil* methods for interacting with *Condition* instances:

```
protected final void waitUntil( Condition
toWaitFor );
protected final void waitUntil( Condition
toWaitFor, boolean state );
```

In the first form, the *ProcessEntity* waits until the specified *Condition* instance reports a value of `true`; in the second, the value to await is specified by the caller.

Selected attributes of several FreeSML classes are retained in *StateVar* instances, and those classes provide "pass-through" methods for accessing their values. These classes, their *StateVar*-based attributes, and the attributes' types are listed in Table 1.

The *ProcessEntity* class defines a set of convenience methods for interacting with frequently referenced state variables. Each of these methods simply makes a pass-through call to the corresponding *checkValue* method for the appropriate attribute (see Table 1). For example, the definition of the *checkLength* method is as follows:

```
protected int checkLength( Queue toCheck ) {
    if( toCheck == null ) {
        throw new IllegalArgumentException( );
    }
    return toCheck.length.checkValue( this );
}
```

Though these methods are not strictly necessary, they provide a means of expressing entity behavior from the active entity's "point of view." All of these methods are declared to be `protected`, as they are intended for use only by *ProcessEntity* instances "acting on their own behalf."

2.5 Uniform Pseudorandom Number Generation

The behavior of FreeSML objects implementing pseudorandom number generators is specified by the *RandomStream* interface. This interface is based on the like-named interface in SSJ, and incorporates several of the convenience methods defined in the class *java.util.Random*. The primary method defined in this interface is *nextDouble*, which generates a double-precision floating-point value uniformly distributed on the interval [0.0, 1.0) each time it is invoked. The procedure used to generate these pseudorandom values is determined by the implementing class.

The current FreeSML release includes a default generator (*DefaultRandomStream*) implemented as an extension of the linear congruential generator in Java's *Random* class. A second class (*MersenneRandomStream*) incorporates a 32-bit integer implementation of the Mersenne Twister (Matsumoto and Nishimura 1998). Continuing efforts include the development of additional *RandomStream* implementations.

Table 1: *StateVar*-Based Attributes in FreeSML Classes

FreeSML Class	Attribute	Type
Executive	*clock*	*DoubleStateVar*
SimManager	*replication*	*IntStateVar*
EventSet	*size*	*IntStateVar*
Queue	*length*	*IntStateVar*
	capacity	*IntStateVar*
Resource	*availability*	*IntStateVar*
	numBusy	*IntStateVar*
	numActive	*IntStateVar*
	capacity	*IntStateVar*
StatObject	*count*	*IntStateVar*
	minimum	*DoubleStateVar*
	maximum	*DoubleStateVar*
	mean	*DoubleStateVar*
	variance	*DoubleStateVar*
MonitoredList *MonitoredSet* *MonitoredSortedSet*	*size*	*IntStateVar*

2.6 Non-Uniform Random Variate Generation

In FreeSML, the simulation developer has access to 20 pre-defined distributions (defined as subclasses of *Random-Variable*), along with facilities for defining empirical discrete (*RandomStep*) or continuous (*RandomLinear*) distributions.

To introduce a random variable into the simulation, the developer creates an instance of the desired class, providing appropriate values for the required parameters. Samples from the specified distribution are obtained by invoking the *sample* method on that instance. Each *Random-Variable* instance is associated with a *RandomStream*; any number of *RandomStreams* can be defined in a simulation.

To permit the user to specify distributions for model elements at run time, the *RandomVariable* class includes a static *createInstance* method, permitting the user to specify the distribution type and parameters as system properties. The *createInstance* method requires the base name for a set of properties as an argument, as in:

```
RandomVariable.createInstance("svcTimes");
```

The base name specified is used to obtain up to five properties from the *SimManager*. First the '*<baseName>.distType*' parameter is requested, and an attempt is made to create an instance of the specified type; the '*distType*' property must contain the name of a valid *RandomVariable* subclass (e.g., "Exponential").

Once created, the new instance is queried for the number of required parameters using its *getParameterCount* method. For each parameter required, the *SimManager* is queried for the value of the '*<baseName>.distParamX*' property ('X' is between 1 and the number of parameters).

2.7 Statistical Data Collection

The current version of FreeSML provides a collection of four Java classes (defined in the *org.freesml.stat* package) to support statistical data collection: *StatObject, Observational, Weighted,* and *TimeWeighted.* The *Weighted* class is unique to FreeSML, providing a general capability to compute statistics on system state information using arbitrary weighting schemes.

The *Observational* and *TimeWeighted* classes implement the weightings most frequently used in simulations. In the *Observational* class, all weights are defined to be unity; for *TimeWeighted* instances, the weights used are the proportions of simulated time for which each particular value is retained. Fields and methods common to all three object types are defined in the abstract *StatObject* class, along with several utility methods.

All *StateVar* instances (including those in Table 1) provide built-in support for the collection of *TimeWeighted* statistics on their values. This mechanism is used in FreeSML to automate collection of commonly-requested statistics, such as queue occupancy and resource usages.

The values of the observation count, mean, minimum, maximum, and variance of each *StatObject* are stored in *StateVar* instances, to support conditional delays based on collected statistics (e.g., terminating the warmup period when the variance of a measure reaches a threshold value).

2.8 Reporting Facilities

Report generation in FreeSML is handled by the classes of the *org.freesml.logging* package. Interaction with the logging facilities takes place using `public` methods of the *LogManager* class:

```
public static void systemMessage(String msg);
public static void outputMessage(String msg);
public static void errorMessage(String msg);
public static void traceMessage(Level level,
String msg);
```

For each of the four message types listed, the *Log-Manager* maintains a collection of *Handler* instances (defined as part of the *java.util.logging* package), including a default instance created for each type (the console is the default destination for system messages only). If a graphical user interface is available, the default destination for trace, output, and error messages will be instances of the *WindowHandler* class, also defined in the logging package. The simulation developer is free to attach any number of *Handler* instances to each of the loggers.

By default, each message is time-stamped with the current value of the simulation clock, converted to an elapsed time in "DD:HH:MM:SS.d" format, using a customized *LogRecord* subclass, *SimLogRecord* (the default behavior of *LogRecord* is to use the system clock). A second version of each method is provided that accepts a scond, Boolean argument. Invoking one of these methods with a value of `false` in the second argument suppresses output of the timestamp value.

3 REMOVAL-ORDERED PRIORITY QUEUES

FreeSML incorporates one feature that is unique among simulation languages: built-in support for the transparent "reordering" of priority queues in response to changes in entity priorities. This support is included in the implementations of *NoticeQueue, Queue,* and *ValueListenerQueue*, the classes used to manage future events, programmer-specified queueing, and conditional waits, respectively.

In FreeSML, support is provided for three user-selectable orderings: insertion-ordered priority-based (the "traditional" method), removal-ordered priority-based, or first in-first out (FIFO). In response to a user request, support for last in-first out (LIFO) ordering will be added in a subsequent release. The data structures and algorithms used to implement "removal-ordered" queues are detailed in a separate publication (DiLeo 2005a). A brief description of each will be given herein.

FIFO-ordered queues are maintained as doubly-linked lists. Insertion of new entries occurs at the tail of the list, and removals are always from the head.

Insertion-ordered priority-based queues are implemented using array-based implicit binary heaps (Gonnet 1976, Bentley 1985), as modified by McCormack (1979).

To obtain a relatively efficient implementation for removal-ordered queues, a simplifying constraint was placed on the behavior of entities in FreeSML: the priority values reported by entities must remain constant for a given value of the simulation clock. This behavior is accomplished using two methods: *getPriority* and *computePriority*.

The *getPriority* method is defined in the *Entity* class, and is the method invoked when requesting an entity's current priority. In this method, the value of the simulation clock is checked, and a check is made to determine if the entity's priority has been computed *at this time* already. If so, the previously computed value is simply returned; if not, the *computePriority* method is invoked, and the computed value is stored locally and returned to the caller.

The *computePriority* method has a simple default implementation in the *Entity* class: a constant value (stored in the *priorityConstant* field) is returned. This method can be overridden by subclasses to define more complex priority behavior. The implementation of *computePriority* can make use of any information on the current system state, including the current time, and may incorporate one or more stochastic components as well.

In the current FreeSML implementation, removal-ordered priority queues alternate between two states: an unordered array-based list, and an array-based implicit binary heap (like that used for insertion-ordered queues).

As elements are first added to the queue, they are simply added to the end of the array. When a request is made for the "first" element in the queue, the contents of the array are converted in-place into an array-based implicit binary heap, and the first entry is removed. Until the simulation clock advances, the queue is maintained as a heap; the ordering of the entries remains stable because priority values are not permitted to change.

Once time advances, entity priorities are again permitted to change. The queue is again treated as an unordered list, and subsequent additions are made to its tail. On a subsequent "remove first element" request, the ordering is regenerated, and the queue is again treated as a heap.

The behavior defined for removal-ordered queues is optimized for operation of the *NoticeQueue* instances comprising an *EventSet*. All *EventNotices* in each *NoticeQueue*, by definition, have the same associated event time, and they are processed by the *Executive* "all at once."

4 EXAMPLE MODELS IN FreeSML

When introducing a simulation language to a new audience, the simplest initial example is the classic single-server exponential (*M/M/1*) queueing model. As FreeSML supports model expression in both process-oriented and event-oriented forms, both forms will be presented.

The models presented herein are nearly identical to those given (in SSJ) by L'Ecuyer, Meliani, and Vaucher (2002); they have been "ported" to FreeSML. We chose this approach for our example models to demonstrate the similarities between models defined in FreeSML and those defined in other Java-based languages (such as SSJ).

4.1 Process-Oriented Single-Server Model

In the process-oriented model, the majority of the work is performed by entities representing arriving customers. They arrive according to a specified Exponential inter-arrival distribution, await service from the server, and depart the system immediately after completing service.

The customers in this model are implemented as a subsclass of *ProcessEntity* named *Customer*, and the server is represented by a *Resource* instance with a single unit initially available. Statistics are collected on system occupancy (L), queue occupancy (L_q), sojourn times (W), queue waiting times (W_q), and server availability (p_0).

The mean customer inter-arrival time is 10.0 time units, and the mean service time is 8.0 time units. The simulation is executed for 10 replications, with a length of 11,000 time units each, and a 1,000 time unit warmup period.The FreeSML source for the main application program (*SingleServer_Process*) is given in Figure 2, and that for the *Customer* class is given in Figure 3.

4.2 Event-Oriented Single-Server Model

In the event-oriented version of this model, we have two significant events to handle: arrivals and departures. Rather than an active player in the simulation, each customer is now a "token" being maneuvered through the system by the actions performed in each event's logic. As such, the class representing a customer in the system is simply a subclass of *Object*, and is named *CustomerObj*. The FreeSML source for *CustomerObj* is given in Figure 4.

Arrivals and departures are represented by the *Arrival* and *Departure* classes, respectively. Each class extends the *EventEntity* class. The FreeSML source for these classes is given in Figures 5 and 6.

The *SingleServer_Event* class (Figure 7) provides the main program entry point, implements the *Replication-Handler* and *ExperimentHandler* interfaces, and defines required system queues and resources.

```
public class SingleServer_Process implements ReplicationHandler, ExperimentHandler {
    static Resource server = new Resource( "Server", true, true, false );
    static Queue serverQueue = new FifoQueue( "Server Queue", true );
    public SingleServer_Process( ) {
        SimManager.setReplicationHandler( this );
        SimManager.setExperimentHandler( this );
    } // End of constructor
    public static void main( String[ ] args ) {
        SimManager.setModelName( "SingleServer -- Process-Oriented" );
        SingleServer_Process program = new SingleServer_Process( );
        SimManager.setRunLength( 11000.0 );
        SimManager.setWarmupPeriod( 1000.0 );
        SimManager.setReplicationCount( 10 );
        SimManager.startSimulation( );
    } // End of main( ) method
    // ReplicationHandler and ExperimentHandler methods omitted
} // End of SingleServer_Process class definition
```

Figure 2: The *SingleServer_Process* Class

```
public class Customer extends ProcessEntity {
    static Observational iaTimes = new Observational( "Inter-Arrival times" );
    static Observational services = new Observational( "Service times" );
    static Observational queueWaits = new Observational( "Customer queue times" );
    static Observational sojournTimes = new Observational( "Customer sojourn times" );
    double svcTime;
//  Constructor
    public Customer( ) { }
    public void init( ) { this.svcTime = svcTimes.sample( ); }
    public void process( ) {
        double delay = arrTimes.sample( );
        createAnotherIn( delay );
        double arrivalTime = Executive.getTime( );
        enqueue( SingleServer_Process.serverQueue );
        request( SingleServer_Process.server );
        dequeue( SingleServer_Process.serverQueue );
        queueWaits.update( Executive.getTime( ) - arrivalTime );
        delay( this.svcTime );
        release( SingleServer_Process.server );
        sojournTimes.update( Executive.getTime( ) - arrivalTime );
    } // End of process( ) method
    static void scheduleFirst( ) {
        Customer firstCust = (Customer) EntityManager.getEntity( Customer.class );
        LogManager.systemMessage( "Scheduling start of first Customer" );
        firstCust.startAt( arrTimes.sample( ) );
    } // End of scheduleFirst( ) method
} // End of Customer class definition
```

Figure 3: The *Customer* Class (Process-Oriented Single-Server Model)

```
public class CustomerObj {
    static RandomVariable arrTimes = new Exponential( 10.0 );
    static RandomVariable svcTimes = new Exponential(  8.0 );
    static int instanceCount = 0;
    String name;
    double arrivalTime;
    double serviceTime;
    public CustomerObj( ) {
        this.serviceTime = svcTimes.sample( );
        this.name = "Customer #" + ++instanceCount;
    }
    public String toString( ) { return this.name; }
} // End of CustomerObj class definition
```

Figure 4: The *CustomerObj* Class (Event-Oriented Single-Server Model)

```
public class Arrival extends EventEntity {
    public void actions( ) {
        Arrival next = (Arrival) EntityManager.getEntity( Arrival.class );
        double iaTime = CustomerObj.arrTimes.sample( );
        next.scheduleIn( iaTime );
        CustomerObj currCust = new CustomerObj( );
        LogManager.traceMessage( LogManager.TRACE_PROCS, "--> Processing arrival for " + currCust );
        currCust.arrivalTime = Executive.getTime( );
        if( SimEnv.serviceList.size( ) > 0 ) {
            LogManager.traceMessage( LogManager.TRACE_PROCS,
                "    Server is busy..." + currCust + " joining server Queue" );
            SingleServer_Event.serverQueue.add( currCust );
        }
        else {
            SingleServer_Event.serviceList.add( currCust );
            LogManager.traceMessage( LogManager.TRACE_PROCS,
                "    Server is idle..." + currCust + " starting service" );
            Departure departure = (Departure) EntityManager.getEntity( Departure.class );
            departure.scheduleIn( currCust.serviceTime );
            SingleServer_Event.queueWaits.update( 0.0 );
        }
    } // End of actions( ) method
    static void scheduleFirst( ) {
        Arrival firstArrival = (Arrival) EntityManager.getEntity( Arrival.class );
        LogManager.systemMessage( "Scheduling arrival of first Customer" );
        double iaTime = SimEnv.arrTimes.sample( );
        firstArrival.scheduleAt( iaTime );
    } // End of scheduleFirst( ) method
} // End of Arrival class definition
```

Figure 5: The *Arrival* Class (Event-Oriented Single-Server Model)

```
public class Departure extends EventEntity {
    public void actions( ) {
        CustomerObj cust = (CustomerObj) SingleServer_Event.serviceList.remove( 0 );
        SingleServer_Event.systemList.remove( cust );
        LogManager.traceMessage( LogManager.TRACE_PROCS, "<-- Processing Departure for" + cust );
        SingleServer_Event.sojournTimes.update( Executive.getTime( ) - cust.arrivalTime );
        if( SingleServer_Event.serverQueue.size( ) > 0 ) {
            cust = (CustomerObj) SimEnv.serverQueue.remove( 0 );
            LogManager.traceMessage( LogManager.TRACE_PROCS,
                "    " + cust + " departed server Queue and entering service" );
            SingleServer_Event.queueWaits.update( Executive.getTime( ) - cust.arrivalTime );
            SingleServer_Event.serviceList.add( cust );
            Departure departure = (Departure) EntityManager.getEntity( Departure.class );
            departure.scheduleIn( cust.serviceTime );
        }
    } // End of actions( ) method
} // End of Departure class definition
```

Figure 6: The *Departure* Class (Event-Oriented Single-Server Model)

5 SUMMARY

We have developed a new Java-based simulation language, similar to existing languages, but incorporating a number of unique features. The development of FreeSML took place not merely for the sake of developing another language, but as a vehicle for making these new features available to the simulation community.

FreeSML has been released as free, open-source software under the Free Software Foundation's Lesser General Public License (LGPL). As such, members of the simulation community are free to adapt and improve upon the language, with the concommittant obligation to contribute their improvements to the development community.

It is sincerely hoped that practitioners will find FreeSML useful, and that sufficient interest will be generated to sustain the language as a community-supported open-source initiative. In other arenas, many open-source applications are considered to be the best available; with sufficient interest and support in the simulation community, FreeSML could eventually fill that role in the discrete-event simulation arena.

ACKNOWLEDGMENTS

The original motivation for this work arose while the author was employed by the MITRE Corporation's Center for Advanced Aviation System Development. The author would like to thank his former MITRE/CAASD col-

```
public class SingleServer_Event implements ReplicationHandler, ExperimentHandler {
    static MonitoredList serverQueue = new MonitoredList( new Vector( ), "Server queue" );
    static MonitoredList serviceList = new MonitoredList( new Vector( ), "In-Service List" );
    static MonitoredList systemList = new MonitoredList( new Vector( ), "In-System List" );
    static Observational queueWaits = new Observational( "Customer queue times" );
    static Observational sojournTimes = new Observational( "Customer sojourn times" );
    public SingleServer_Event( ) {
        SimManager.setReplicationHandler( this );
        SimManager.setExperimentHandler( this );
    } // End of constructor
    public static void main( String[ ] args ) {
        SimManager.setModelName( "SingleServer -- Event-Oriented" );
        LogManager.systemMessage( "Constructing single-server simulation", false );
        SingleServer_Event program = new SingleServer_Event( );
        SimManager.setRunLength( 11000.0 );
        SimManager.setWarmupPeriod( 1000.0 );
        SimManager.setReplicationCount( 10 );
        SimManager.startSimulation( );
    } // End of main( ) method
    // ReplicationHandler and ExperimentHandler methods omitted
} // End of SingleServer_Event class definition
```

Figure 7: The *SingleServer_Event* Class

leagues, Carla Gladstone and Duane Small, for the lively and thought-provoking discussions which inspired the concepts presented herein.

This work was completed as part of the author's doctoral studies at The George Washington University, in the Department of Engineering Management and Systems Engineering. The author would like to thank his research advisor, Prof. Gideon Frieder, and the members of his Research Committee for their guidance and encouragement in the completion of his studies.

APPENDIX: OBTAINING AND USING FreeSML

The first release of FreeSML (version 1.0) is available via SourceForge `<www.sourceforge.net>`. Interested parties may obtain FreeSML there, or by contacting the author. Javadoc-generated API documentation is included with the software. A User's Manual is in preparation, with publication anticipated by January 2006.

To use FreeSML, a Java 2 Software Development Kit (SDK), version 1.4 or later, is required, as the source code contains `assert` statements. FreeSML is deployed as a single Java Archive (JAR) file, *FreeSML.jar*. This JAR file must be included in the CLASSPATH for both the compiler and the virtual machine. A *build.xml* file is included, for use with the Ant build management tool.

REFERENCES

Bentley, J.L. 1985. Thanks, heaps [Programming pearls]. *CACM* 28 (3): 245 – 250.

DiLeo, J.J. 2005a. Adding support for state-dependent and stochastic priorities to a discrete-event simulation language. Submitted for publication.

DiLeo, J.J. 2005b. Discrete-event simulation with state-dependent and stochastic process priorities. D.Sc. diss., The George Washington University.

Gonnet, G.H. 1976. Heaps applied to event driven mechanisms. *CACM* 19 (7): 417 – 418.

Healy, K.J., and R.A. Kilgore 1997. Silk: A Java-based process simulation language. In *Proceedings of the 1997 Winter Simulation Conference1*, D.J. Medeiros, E.F. Watson, J.S. Carson, and M.S. Manivannan, eds. 475-482. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Henriksen, J.O. 1993. SLX, the successor to GPSS/H. In *Proceedings of the 1993 Winter Simulation Conference*, G.W. Evans, M. Mollaghasemi, E.C. Russell, and W.E. Biles, eds. 263–268. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Kiviat, P.J. 1969. Digital computer simulation: Computer programming languages. Santa Monica, CA: The RAND Corporation. RAND doc. no. RM-5883-PR.

L'Ecuyer, P., L. Meliani, and J. Vaucher. 2002. SSJ: A framework for stochastic simulation in Java. In *Proc. 2002 WSC*, E. Yücesan, C.-H. Chen, J.L. Snowdon, and J.M. Charnes, eds. 234 – 242.

Matsumoto, M., and T. Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM TOMACS* 8 (1): 3 – 30.

McCormack, W.M. 1979. Analysis of future event set algorithms for discrete event simulation. Ph.D. diss., Syracuse University.

Russell, E.C. 1994. *SIMSCRIPT II.5 programming language*. La Jolla, CA: CACI Products Company.

Sleator, D.D., and R.E. Tarjan. 1985. Self-adjusting binary search trees. *JACM* 32 (3): 652 – 686.

Vaucher, J.G. 1971. Simulation data structures using Simula 67. In *Proceedings of the 1971 Winter Simulation Conference*, 255–260. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

AUTHOR BIOGRAPHY

JOHN J. DiLEO received his D.Sc. in Operations Research from The George Washington University in May 2005. He received his B.S.E.E. from The Johns Hopkins University, his M.S. in Operations Research from The George Washington University, and his M.S.A. from Central Michigan University. Formerly, he worked as a simulation software engineer at the U.S. Army Materiel Systems Analysis Activity, and the MITRE Corporation's Center for Advanced Aviation System Development. In August 2005, he joined the faculty of ECPI College of Technology's Greenville campus, as an instructor of computer technology, information systems, electronics, and mathematics. His e-mail address is <dileo@direcway.com>.