Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

SIMFONE′: AN OBJECT-ORIENTED SIMULATION FRAMEWORK

Manuel D. Rossetti

Department of Industrial Engineering
4207 Bell Engineering Center
University of Arkansas
Fayetteville, AR 72701, U.S.A.

Ben Aylor
Ryan Jacoby
Alyson Prorock
Antoine White

Department of Systems Engineering
Thornton Hall
University of Virginia
Charlottesville, VA 22903, U.S.A.

ABSTRACT

This paper presents an overview of a software design framework for the development of object-oriented simulations. The framework is documented using the Unified Modeling Language (UML) and is divided into packages to organize the collection of classes into important functional areas. The purpose of the framework is two-fold. First, the framework is useful in understanding the concepts and abstractions within simulation modeling and languages. Secondly, the framework can serve as the basis for the development of object-oriented simulation libraries. We illustrate the latter through a Java implementation.

1 INTRODUCTION

According to Booch et al. (1999), a software design framework is "an architectural pattern that provides an extensible template for applications within a domain." A framework provides a set of abstract and concrete classes that can be extended via sub-classing or used directly to solve a particular problem within a particular domain. This paper discusses a software design framework for object-oriented simulations. A framework can ease the work of researchers, educators, and practitioners. An object-oriented simulation framework can provide a better understanding of key abstractions within simulation modeling and can provide a blueprint for the development of object-oriented simulation libraries.

Numerous organizations and individuals have developed object-oriented simulations tools. These tools have been implemented using languages such as C, C++, and, recently, Java; however, there is very little documentation in the literature describing the common abstractions used within these tools. For example, Hirata and Paul (1998) describe a method for developing object-oriented designs for simulation. The article does not emphasize the underlying framework, but instead, focuses on the process of design. Other object-oriented simulation tools or packages include SIMEX (1998), a set of C++ classes for performing dynamic population simulations, Schwetman's (1986) CSIM++ an extensive set of classes for performing process and event based simulations in C++, and Joines and Roberts' YANSL (1996) which supports the development of C++ simulations using the network modeling view. In addition, Healy and Kilgore (1997) describe Silk, a comprehensive extension of the Java simulation language for developing process-oriented simulations. Finally, MODSIM, Mullarney (1997), is a complete simulation language developed explicitly for the object-oriented modeling of complex, dynamic systems.

Our emphasis in this paper is to propose and present a framework that can serve as the basis for the development of simulation libraries in any language. At this point our framework is not intended to be the definitive framework for object-oriented simulation modeling; however, our purpose is to spur further development and documentation of such a framework so that the duplication of effort that occurs in every simulation language development effort can be mitigated and that some standardization can take place between these efforts.

Since the framework is presented as a set of Unified Modeling Language diagrams, we begin with a basic introduction to reading these diagrams. The paper then discusses the key packages within the framework. This will primarily consist of an explanation of the abstractions, their intended behaviors, and how they interact with each other. Finally, to make the framework more concrete, we present a limited implementation of the framework in Java with examples of event and process modeling.

2 UNIFIED MODELING LANGUAGE

The Unified Modeling Language (UML) is a modeling language for the conceptual and physical representation of object-oriented systems. The language is general enough to include both software intensive systems as well as general systems modeling. The UML contains a set of graphical symbols (notation) and a well defined set of semantics for specifying precise object-oriented models. An object-oriented model contains such abstractions as classes, attributes, operations, objects, associations, states, etc. The UML is based on a detailed meta-model of these abstractions that precisely defines their meanings and how they relate to each other. For further details of the UML's meta-model, we refer the interested reader to the Object Management Group's UML version 1.3 specification at `<www.omg.org/cgi-bin/doc?ad/99-06-08.pdf>`.

The UML allows the development of various models of a system in the form of diagrams to represent key view points just as different blueprints of a house examine important aspects such as the electrical, water, and structural components. The primary modeling view used in this paper is the structural view. We primarily use class diagrams to show a set of classes and their relationships. UML also contains a rich set of behavioral diagrams to specify the dynamic aspects of a system.

The static structure of a system can be visualized in the form of a class diagram. A class describes a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. A class diagram illustrates the relationships between classes through associations. An association describes a group of links with common structure and semantics. A link is simply a physical or conceptual connection between object instances.

Figure 1 presents a generic class diagram. Each class is indicated with a rectangle divided into three areas for the class name, attributes and operations. An object attribute is a named property of a class that describes a value held by each object of the class. Each class can have some operations. An operation is the implementation of a service that can be requested from any object of the class. Operations affect the behavior of an object instance. Associations are indicated by an adorned line between classes.

In the figure, SubClass1 specializes the more general ParentClass as indicated with the arrow between SubClass1 and ParentClass. The generalization/specialization relationship is typically implemented using the inheritance features of a language. An association exists between Class1 and Class2. This is indicated in the figure by the line labeled Association between Class1 and Class2. Each end of an association is adorned with the multiplicity of the relationship. Multiplicity indicates the number of instances of one class that may relate to a single instance of an associated class. When reading an association, the class you begin with is called the source. The class you traverse to is called the target. The association between Class1 and Class2 would be read as follows. Each instance of Class1 is associated with zero or more (0..*) instances of Class2. Each instance of Class2 is associated with exactly 1 instance of Class1. Another association exists between SubClass1 and Class2. In this case, this association has attributes and operations. The class AssociationClass describes the set of links between SubClass1 and Class2. The instances of an association class are objects but they derive their identity (existence) from instances of the participating classes. Aggregation is a form of association that is used to represent the "part of" association. The whole is called the assembly, aggregate, or composite. The parts are called parts or components. An aggregation is indicated with a diamond on the end of the associate that is the aggregate. Identifying a relationship as an aggregation has implications during the implementation phase of design. Further detail can be added to the model through the use of an attached note as indicated in Figure 1.

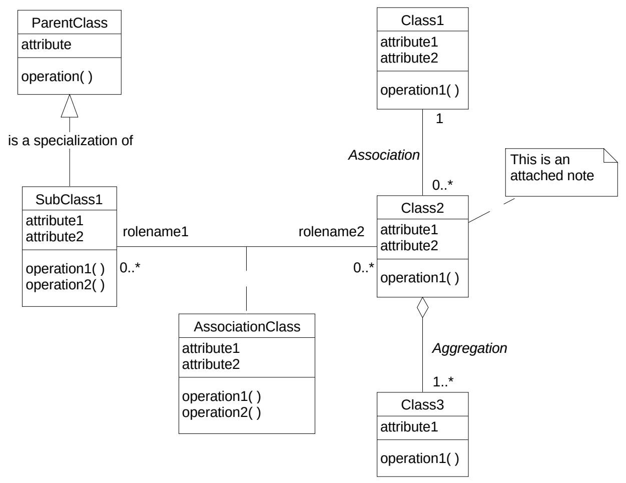

Figure 1: Generic Class Diagram

We have only touched on parts of the UML specification. The UML is the recommended standard for specifying object-oriented systems. As such, it will be increasingly important that simulationists understand the language. The UML can be used to model simulations, enterprise information systems, real-time systems, and other software applications. The UML can be used for modeling, documentation, visualizing, specifying and with code generation even constructing software programs. The power of UML is in achieving a standardized communication medium. A complete description of the UML is beyond the scope of this paper. We refer the interested reader to Larman (1998) or Booch et al. (1999) for further information on the UML.

3 SIMFONE′ PACKAGES

The SIMFONE′ (*sim*ulation *f*ramework *one*) represents a proposed framework for the development of object-oriented simulation libraries. SIMFONE′ (pronounced symphony) is currently divided into six main packages pertaining to:

- Simulation management
- Simulation executive
- Processes
- Resources and queues
- Random number generation
- Statistics

Additional utility and support packages are also necessary when developing a specific language implementation. This paper will concentrate on the simulation management, simulation executive, and the process package. We will briefly discuss the other packages to highlight their main structures.

3.1 Simulation Management Package

The simulation management package contains classes that are used in the development of the simulation including modeling and experimentation. The class Simulation provides for the overall coordination of the simulation run. The Simulation class requires an experiment, a model, and a scheduler to perform its work. The Simulation class acts as a mediator between the Experiment, Model, and SchedulerIfc classes. In addition, it also controls the reporting of replication results, final summary results, and the start, stopping, and pausing of the simulation run.

A key class in this package is SimObject. The SimObject class represents those objects within a simulation that can participate in event scheduling and that can be included in a model. Instances of SimObject can schedule and cancel events and can be added or deleted from a model. They not only schedule events but also process the events when the time of the event occurs. They can be setup at the beginning of an experiment, initialized at the beginning of each replication, and warmed up individually. They can also react to the beginning of a replication and to the end of a replication.

The Model class is a subclass of SimObject. This allows a model to schedule and respond to events. A Model is a representation of a system with the intention to predict the system’s behavior when certain events occur. A Model manages the addition and/or removal of SimObjects; therefore, the Model can be composed of one or more object of type SimObject. An instance of Model is necessary for an object of type Simulation to be created.

The class Experiment represents the information necessary for the control of a replication of an experiment. An experiment has a specified number of replications, the starting and ending times for the replications, and a warm up time to be applied to each replication. A set of zero or more response variables can be attached to an experiment. A response variable is a variable of interest within the experiment that can be recorded across replications. A response variable is a subclass of SimObject. This allows response variables to schedule and handle events, especially those events that represent the beginning and ending of the simulation replication. Although not shown in this diagram, an experiment can also have a set of input factors attached to it.

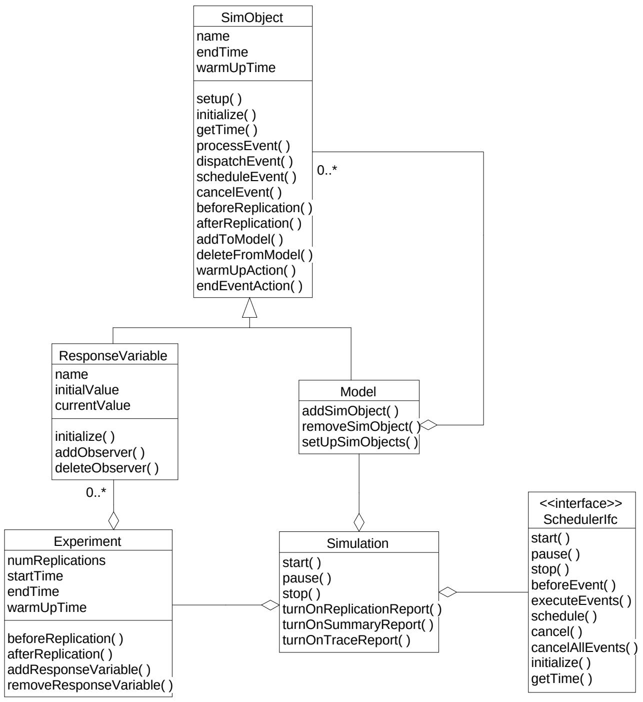

Figure 2: Simulation Management Class Diagram

The `SchedulerIfc` interface defines the behaviors expected of the simulation executive. The primary purpose of this class is the time ordered execution of events. The Simulation class delegates to an instance of a class that implements the SchedulerIfc interface to control the events during a replication.

3.2 Simulation Executive Package

The simulation executive package, shown in Figure 3, manages the execution of events. When a simulation object wishes to have some action occur at some point in the simulated future, the object will notify the scheduler of the time the action should occur and the type of the action. The Scheduler Package manages these future events in chronological order and performs the requested actions at the specified times.

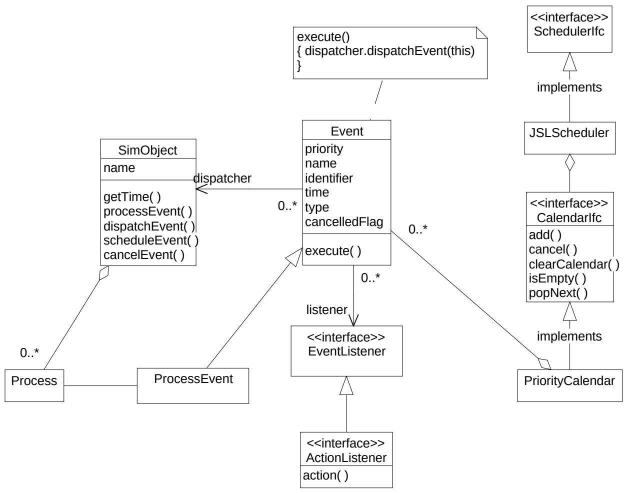

Figure 3: Simulation Executive Package

The Simfone′ framework has been designed with the goal of allowing multiple implementations of simulation schedulers. This will enable simulation scheduling algorithms to be easily evaluated. Any new scheduling algorithm simply must implement the behavior contained in the ScheduleIfc class. The class JSLScheduler is an example of this for a Java implementation. In addition, an interface to represent the simulation calendar has been defined such that different calendar implementations can be easily interchanged as long as the CalendarIfc interface is implemented. The class PriorityCalendar is a concrete class that implements a simulation calendar within our Java implementation. It is based on the use of data structures available within the Java Generic Library.

The class Event represents a simulated event. A simulated event can have a priority, a type, a time, and a unique identifier. While time is the primary means by which events can be ordered, the current definition allows ordering via priority, type, and order of creation. These attributes are useful for the implementation of general scheduling algorithms.

The framework uses a delegation-based event processing model. Objects of type SimObject have the ability to schedule events. This creates an instance of type Event and places the event on the schedule. When the event's execute method is called the event calls back to a SimObject that is acting as its dispatcher. The dispatcher processes the event by sending the event to an event listener that has been registered to handle that type of event. The event listener then invokes the appropriate action for the event.

3.3 Process Package

The process package, shown in Figure 5, consists of those classes necessary to implement the process view of simulation. The process package interacts primarily with the Simulation Management package. A process can be used to represent the life of a simulated object. The abstract class Process has methods that allow transitions between states that are mapped to the life of the object.

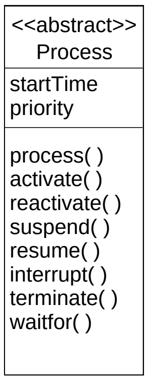

Figure 4: Process Class

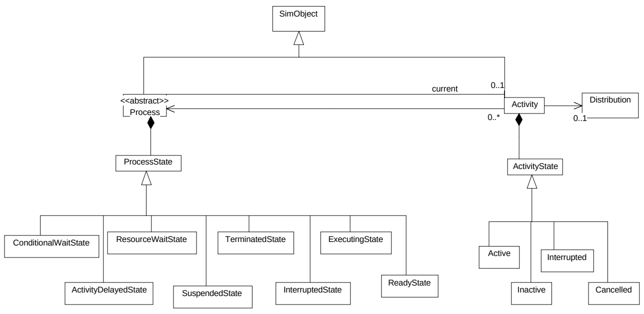

Figure 5: Process Package

A Process allows the simulation of elapsed time within its `process()` method. The `process()` method contains calls that may interact with resources and activities to model the life or sequence of activities associated with a simulated object. A process may `waitfor()` a resource to become available, for an activity to be completed, and for a condition to become true. A process interacts with the Scheduler through its methods inherited from SimObject.

Within a process view implementation of this design, pseudo-parallelism, i.e. the appearance of concurrent multiple simulated objects interacting through time, must be considered. A variety of techniques exist to implement the process view. Traditional simulation languages such as SIMAN and SLAM present a process view essentially by defining functions for every language construct, even if-then statements. A process is defined by a set of function calls. The state of the process essentially becomes the current function call. This has disadvantages in terms of the lack of local variables and is inherently not an object-oriented approach. Other languages such as Simscript II.5 and MODSIM use a co-routine facility to save the state of the process. This is dependent upon a compiler implementation of these facilities. CSIM++ relies on a similar implementation through the use of preprocessing. If the underlying language supports multi-threading either directly as in Java or indirectly through a thread package (e.g. POSIX) then the process view can be implemented via threads. The point is the mechanism by which the pseudo-parallelism is implemented should be presented as layer of abstraction between a simulation modeling framework and its implementation. Although our framework is not dependent upon a thread-based implementation our example in Java assumes the availability of a thread framework.

The ProcessState class and its subclasses (SuspendedState, TerminatedState, ActivityDelayedState, ResourceWait State, ConditionalWaitState, ExecutingState, ReadyState) model the states that a process may be in during its life via the State pattern. See Gamma et al. (1995) for additional information on patterns in software design. The states that a process can be in are illustrated in Figure 6. The ReadyState represents a process that has been initialized and is ready to begin its life. The ActivityDelayedState represents when the process is delayed waiting for an activity to complete. The InterruptedState represents when a process has been interrupted during an activity delay. The SuspendedState represents when the activity has suspended itself waiting for a reactivation from a different process. The ConditionalWaitState represents when an activity is waiting for a condition in the model to become true. The ResourceWaitState represents when a process is blocked waiting for a resource to be available. The ResourceWaitState is actually a special case of a ConditionalWaitState; however, since waiting for a resource is such a common formalism in simulation, we decided to present it as a separate class to allow for potential implementation optimizations. The ExecutingState represents when a process is not blocked or delayed. The TerminatedState represents when a process's life cycle has completed.

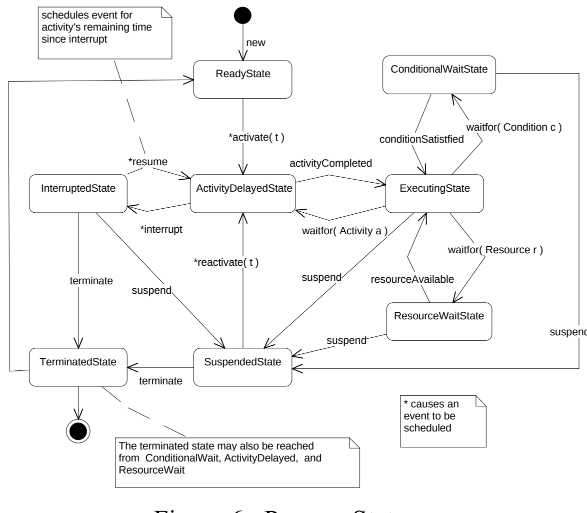

Figure 6: Process States

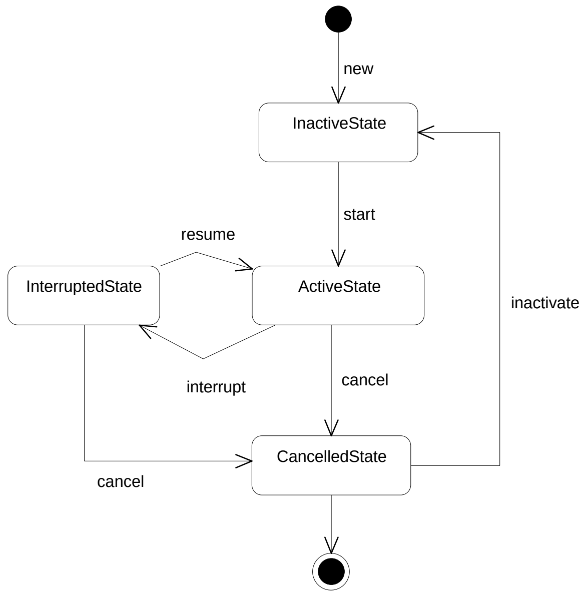

Figure 7: Activity States

3.4 Resource and Queue Packages

The Resource Package, shown in Figure 8, is designed to model the general allocation of resources in complex systems. Specifically, this package provides a default design for the modeling of reusable system capacity through Resources, Requests, and Queues. The Resource class handles all service requests and allocation of capacity. The Request class stores information pertinent to an individual request for a Resource, such as the amount of the Resource requested and the current state of the Request. Classes that want to make requests for resources must implement the RequestorIfc. The Resource class allocates capacity to requests by placing them in a service queue or if no capacity exists into a waiting queue. The Resource class works with classes that implement the QueueIfc. This allows different queueing disciplines to be implemented in concrete classes such as the FIFOQueue. The QueueIfc works with classes that implement the QObjectIfc such as the QObject. Objects of type Request are sub-classed from QObject so that requests can be held in queues.

An Activity represents a duration of simulated time that a process may experience. The states of an activity (Inactive, Active, Interrupted, Cancelled) are also modeled with the State pattern via subclasses of ActivityState.

3.5 Random Number Package

A simulation framework would be incomplete without a random number package. An example hierarchy for distributions is given in Figure 9. The random number generation functionality should be factored out of the distribution hierarchy via delegation. The distributions can delegate random number generation to a class that implements the random number generator interface (`RNGIfc`) when calling their individually implemented `sample()` methods. In this way, different random number generators can be used as long as they implement the `RNGIfc` such as done by `JSLRNG`.

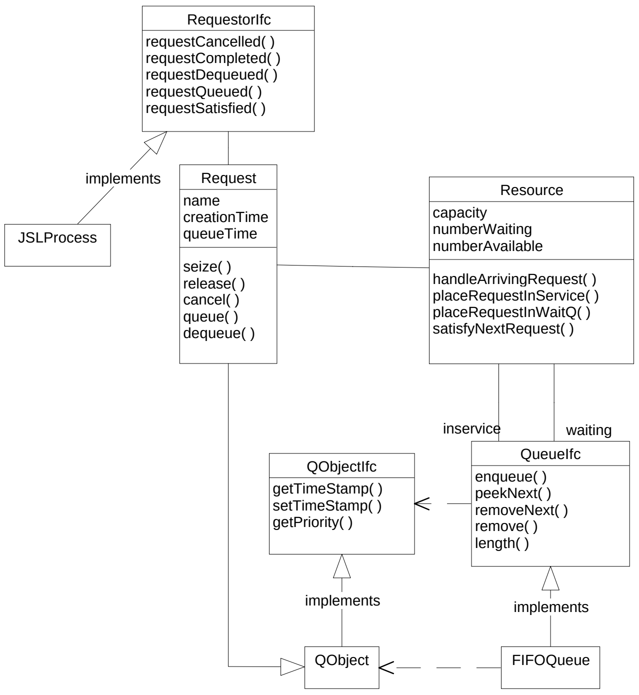

Figure 8: Resource Package

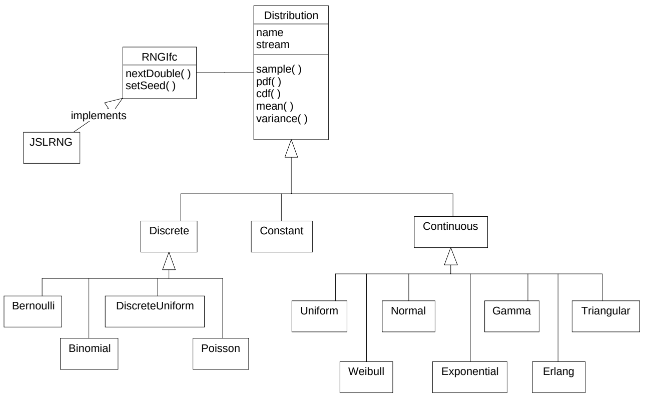

Figure 9: Random Package

3.6 Statistics Package

The collection of statistics is accomplished in the statistics package, shown in Figure 10. Because of the concept of response variables, our statistics package is minimal. The ResponseVariable class is sub-classed into count based responses (Counter), observational responses (Tally), and time persistent responses (TimeWeighted). These classes appropriately override the `setValue()` method. The `setValue()` method performs the updating of the value of the response variable and its weight. In addition, it is responsible for notifying any observers of changes to the response variable. An instance of type Statistic can be an observer of a response variable since it extends the abstract class RVObserver. Additional observers can be defined such as dynamic charts. The Statistics class can be further subclassed to implement specialized estimators such as batch means and time series.

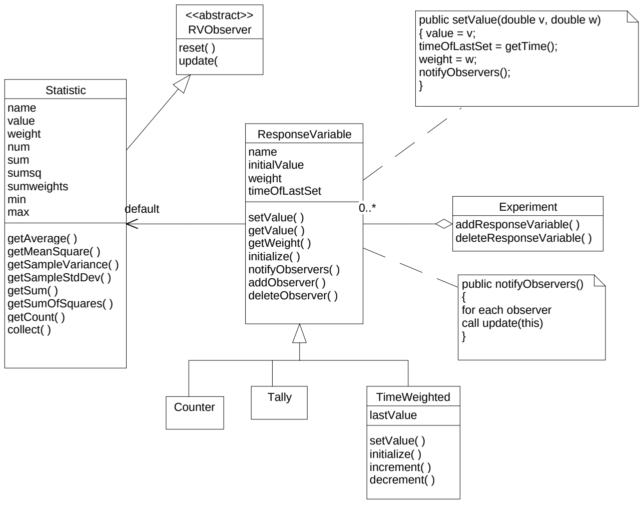

Figure 10: Statistics Package

4 JAVA IMPLEMENTATION EXAMPLE

In this section, we present an implementation of a simple G/G/c queueing simulation in Java. The Simfone′ framework was implemented into a class library to support simulation in Java. The library is called the Java Simulation Library (JSL), pronounced “jissle”. A complete discussion of the implementation of the JSL is beyond the scope of this paper; however, its functionality is based on the already discussed Simfone′ framework. Our intention is to provide enough detail so that the reader can make the above concepts more concrete.

We will first illustrate how to model a M/M/1 queue with the event view within JSL and then illustrate how the same system can be modeled using the process view. To begin the event view model, we must extend a class from the Model class. In this example, we have created a general model of a G/G/c queue. The constructor takes in a parameter for the number of servers, the arrival distribution, and the service distribution. The default constructor makes a M/M/1 model with the mean time between arrivals equal to 1 and the mean service time equal to 0.5. The Exponential class is used to create the appropriate distributions that are passed into the model.

```
public class GGcEventModel extends Model
{
    public GGcEventModel()
    {
        this(1,new Exponential(1.0), new Expo-
nential(0.5));
    }

    public    GGcEventModel(int    numServers,
Distribution arrivals, Distribution service)
    {
// constructor logic here
    }
// method and variable definitions here
}
```

In a main application class, the user must create the model.

```
public class Application
{
    public static void main(String args[])
    {
        double arrivalMean = 1.0;
        double serviceMean = 0.5;
        int numServers = 1;
        int numReps = 5;
        double simLength = 1000.0;

        Project    project    =    new    Project
("Rossetti", "GGc Test");
        Experiment    exp    =    new    Experiment
(numReps, simLength);
        GGcEventModel  model  =  new  GgcEvent-
Model(numServers,        new        Exponential
(arrivalMean), new Exponential(serviceMean));

//      GGcProcessModel model = new
GGcProcessModel(numServers, new
Exponential(arrivalMean), new
Exponential(serviceMean));

Simulation sim =
project.makeSimulation(model, exp);

        sim.start();
    }
    }
```

In the application, a project is created, an experiment is created and then the model is created. The project is then told to make a simulation given the model and the experiment. This ensures that the model, project, simulation, and experiment are properly allocated and their associations are properly created. Then the simulation is told to `start()`.

The `GGcEventModel` begins by accepting the parameters of the model and then builds the event listeners for events. Every model has two methods for initializing its state. The method `setup()` is called before the first replication and allocates the response variables (`queueTimeRV` and `queueLengthRV`) and adds them to the experiment. The method `initialize()` is called before each replication and can be used to set the state of the system before each replication.

```
public class GGcEventModel extends Model
{
    public GGcEventModel()
    {
        this(1,new    Exponential(1.0),    new
Exponential(0.5));
    }

    public    GGcEventModel(int    numServers,
Distribution arrivals, Distribution service)
    {
        setNumberOfServers(numServers);
        setServiceDistribution(service);
        setArrivalDistribution(arrivals);
        setName("GGcEventModel");
        myArrivalListener    =    new    Arrival
Listener();
        myBeginServiceListener    =    new    Begin
ServiceListener();
        myEndServiceListener  =  new  EndService
Listener();
        myNumBusyServers = 0;
    }

    // get/set methods would be here

    public void setUp()
    {
        queueTimeRV    =    new    ResponseVariable
("Queue Time");
        queueLengthRV    =    new    TimeWeighted
("Queue Length");
        queueTimeRV.addToExperiment();
        queueLengthRV.addToExperiment();

        myWaitingQ  =  new  FIFOQueue("GGC  Q");

        myWaitingQ.addToExperiment();
    }

    public void initialize()
    {
        // empty and idle
        myWaitingQ.clear();
        myNumBusyServers = 0;
        // start the arrivals
        scheduleArrival();
    }
```

In the following code snippet, the ArrivalListener class's action method creates an object of type Customer. The Customer class is simply a class to represent customers with the primary purpose of tagging their arrival times. For brevity the class is not shown here. The ArrivalListener enqueues the customer, adjusts the response variable, and checks to see if the customer can begin service. In addition, it schedules the next arrival. The BeginServiceListener increments the number of busy servers, removes the customer from the waiting queue, adjusts the values of the response variables, and schedules the end of service. The EndServiceListener decrements the number of busy servers and checks to see if any additional customers are waiting. If so, the customer is scheduled to begin service.

```
    class ArrivalListener implements JSLAction
Listener
    {
        public void action(JSLEvent event)
        {
            Customer c = new Customer();

            myWaitingQ.enqueue(c);

    queueLengthRV.setValue(myWaitingQ.size());

            if   (myNumBusyServers   <   myNum
Servers)
                scheduleBeginService(c);

            scheduleArrival();

        }
    }

    class BeginServiceListener implements JSL
ActionListener
    {
        public void action(JSLEvent event)
        {
            myNumBusyServers++;

            Customer c =
            (Customer)myWaitingQ.removeNext();

    queueLengthRV.setValue(myWaitingQ.size());
            queueTimeRV.setValue(getTime()    -
c.getArrivalTime());
            scheduleEndService(c);
        }
    }

    class EndServiceListener  implements  JSL
ActionListener
    {
        public void action(JSLEvent event)
        {
            myNumBusyServers--;
            if (myWaitingQ.size() > 0 )
            {
                Customer nc =
             (Customer)myWaitingQ.getNext();
                scheduleBeginService(nc);
            }
        }
    }
```

After construction of the event listeners within the GGcEventModel constructor, the delegation event mechanism requires that event listeners know which events to process. This can be accomplished when the event is scheduled. For example, the methods `scheduleArrival()`, `scheduleBeginService()`, and `scheduleEndService()` construct events and assign the appropriate listeners.

```
    private void scheduleArrival()
    {
        double               t              =
myArrivalDistribution.sample();
        JSLEvent           temp             =
scheduleEvent(myArrivalListener, t);
        temp.setName("Arrival");
    }

    private void scheduleBeginService(Customer
c)
    {
        JSLEvent  temp  =  scheduleEvent(myBegin
ServiceListener,     0.0,     JSLEvent.DEFAULT_
PRIORITY, c);

        temp.setName("Begin Service");
    }

    private void scheduleEndService(Customer c)
    {
        double   t   =   myServiceDistribution.
sample();
        JSLEvent   temp   =   scheduleEvent(myEnd
ServiceListener,                           t,
JSLEvent.DEFAULT_PRIORITY, c);
        temp.setName("End Service");
    }
```

As one can see, the use of Java here is not conceptually any different from the use of other languages. The JSL automatically collects statistics, manages the event list, and allows for random number generation. In support of the event view paradigm, the JSL allows the events to be encapsulated within the class of interest (e.g. GGcEventModel).

The advantages for using Java for the implementation of the process view of simulation have already been exploited by a variety of other tools; see Healy and Kilgore (1997). In this example, we illustrate how the implementation of the Simfone′ framework in the JSL allows easy modeling via processes and resources. Again in this case the user must develop a class to model the system. In the `setup()` method, the resource is allocated and a call to the static method of the GGcCustomer class is made to initialize statistical collection. The `initialize()` method begins the scheduling of arrivals. In the ArrivalListener class a GGcCustomer is created and activated. In addition, the next arrival is scheduled. This arrival creation and scheduling logic could have also been easily incorporated into the GGcCustomer class.

```
public class GGcProcessModel extends Model
{
    public GGcProcessModel()
    {
        this(1,new    Exponential(1.0),    new
Exponential(0.9));
    }

    protected void setUp(Experiment exp)
    {
        myResource    =    new    Resource("GGC
Resource", myNumServers);
        myResource.addToExperiment();
        myResource.addToModel();
        GGcCustomer.setUpStats();
    }

    protected void initialize(Experiment exp)
    {
        GGcCustomer.resetCount();
        scheduleArrival();
    }

    private void scheduleArrival()
    {
        double              t              =
myArrivalDistribution.sample();
        scheduleEvent(myArrivalListener, t);
    }

    class ArrivalListener implements JSLAction
Listener
    {
        public void action(JSLEvent event)
        {
            GGcCustomer  c  =  new  GgcCustomer
(myServiceDistribution, myResource);

            c.activate();
            scheduleArrival();
        }
    }
}
```

In the process view of simulation, one traces the path of the life of an entity of interest through the system. The GGcCustomer class does that using the constructs available within the Process class. The following code fragment illustrates how to do this using the JSL. The response variables have been defined a static variable because they represent statistics across all the instances of the class GGcCustomer. The key methods within the process class are indicated in bold. A customer may wait for a request to be fulfilled from a resource. If the request is not immediately fulfilled, then the thread of control for this process stops and the JSL event scheduling mechanism takes over to find the next event. Once the request can be fulfilled, the request is returned to the customer. The `waitfor()` method is again used to implement a wait for the completion of an activity. After the activity is completed, the release method of the resource is called with the appropriate request. This implementation is similar in some respects to the waitfor method found in MODSIM. In addition, the functionality of the process class is essentially equivalent to that available within the industrial strength Silk environment presented by Healy and Kilgore (1997). The JSL again manages the execution of the simulation including the appropriate use of Java's threads.

```
public   class   GGcCustomer   extends   Process
Adapter
{
    public GGcCustomer(Distribution d, Resource
r)
    {
        super("Customer" + count);
        count++;
        setServiceActivity(d);
        setResource(r);
        timeOfArrival = getTime();
    }

    public   void   process()   throws   Process
TerminatedException
    {
        double arriveTime;
        arriveTime = getTime();

        queueLengthRV.increment();
        Request r = waitfor(myResource);

        queueLengthRV.decrement();
        queueTimeRV.setValue(getTime()-
arriveTime);
        waitfor(myServiceActivity);
        myResource.release(r);
    }

// setup methods etc.

    private double timeOfArrival;

    private Activity myServiceActivity;
    private Resource myResource;
    private static int count = 1;

    public static ResponseVariable queueTimeRV;
    public static TimeWeighted queueLengthRV;
}
```

5 CONCLUSIONS

In this paper, we have presented an overview of an object-oriented simulation framework called Simfone′. The Simfone′ framework was developed to gain a better understanding of the requirements of a generic object-oriented framework for simulation modeling. Through a detailed object-oriented analysis and design effort, we hope that a framework can be established to allow the development of simulation libraries in any object-oriented language. Work is continuing on this effort, including a more detailed examination of the dynamic modeling requirements for such a framework. This would include an explanation through state diagrams and interaction diagrams of the key dynamic issues needed for simulation modeling. In addition, we are continuing our evaluation of the static structures within the framework. A key emphasis of this analysis is the use of interfaces and patterns to provide flexibility to the modeling constructs.

In addition to the Simfone′ framework, we briefly illustrated how simple simulation models can be developed using a Java implementation of the framework. Complete versions of the above models have been implemented, verified, and tested. Additional development work is in progress on the JSL. We are currently examining the efficiency issues related to thread management within the process view. After additional testing, the JSL will be released to the public domain via open software foundation licensing. The JSL represents just one instantiation of the Simfone′ framework. As the work continues on the framework, we expect that implementations in other languages will be possible. We see this effort as an iterative process and invite the interested reader to assist us in establishing a well-documented and comprehensive object-oriented framework for simulation.

ACKNOWLEDGMENTS

This work was originally supported through a grant for undergraduate research projects from the USENIX Association at <www.usenix.org>.

REFERENCES

Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The Unified Modeling Language User Guide*, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Massachusetts: Addison-Wesley Publishing Company, Inc.

Healy, K. J. and R. A. Kilgore. 1997. Silk™: A Java-Based Process Simulation Language. *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson, 475-482, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Hirata, C.M. and Paul, R. J. 1996. Object-Oriented Programming Architecture for Simulation Modeling, *International Journal in Computer Simulation*, 6: 269-287.

Joines, J.A. and S. D. Roberts. 1996. Design of object-oriented simulations in C++. In *Proceedings of the 1996 Winter Simulation Conference*, ed. J. Charnes, D. Morrice, D. Brunner, and J. Swain, 65-72. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Larman, C. 1998. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Upper Saddler River, New Jersey: Prentice-Hall, Inc.

Mullarney, A., West, J., Belanger, R., & Rice, S. 1997. ModSim Tutorial. La Jolla, CA: CACI Products Company.

Nair, R.S., Miller, J.A., and Zhang, Z. 1996. A Java-Based Query Driven Simulation Environment. *Proceedings of the 1996 Winter Simulation Conference*, ed. D. Morrice and J. Charnes, 786-793, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

SIMEX, 1998. National Micropopulation Simulation Resource, SIMEX Package, Minneapolis, Minnesota: Division of Health Computer Sciences, University of Minnesota, <www.nmsr.umn.edu/nmsr/>.

Schwetman, H., 1986. CSIM++: A C-Based, Process-Oriented Simulation Language. *Proceedings of the 1986 Winter Simulation Conference*, ed. J. Wilson, J. Henrikson, S. Roberts, 386-396, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

AUTHOR BIOGRAPHIES

MANUEL D. ROSSETTI is an Assistant Professor in the Industrial Engineering Department at the University of Arkansas. He received his Ph.D. in Industrial and Systems Engineering from The Ohio State University. His research interests include the design, analysis, and optimization of manufacturing, health care, and transportation systems using stochastic modeling, computer simulation, and artificial intelligence techniques. Dr. Rossetti is an Associate Member of the Institute of Industrial Engineers and a member of the IIE OR Division. Dr. Rossetti is also a member of INFORMS and SCS. His email and web addresses are <rossetti@comp.uark.edu> and <www.uark.edu/~rossetti>.

BEN AYLOR, RYAN JACOBY, ALYSON PROROCK, AND ANTOINE WHITE are former undergraduate Systems Engineering students supported by the USENIX grant. Ben made contributions to the scheduling package. Ryan made contributions to the resource and queue package. Alyson made contributions to the random number and statistic packages and Antoine made contributions to the simulation management packages.