

JAPROSIM: A Java framework for Process Interaction Discrete Event Simulation.

Bourouis Abdelhabib, University Larbi Ben M'Hidi, Oum El Bouaghi 4000 Algeria.

Belattar Brahim, University Colonel El Hadj Lakhdar, Batna 5000, Algeria.

Abstract

In this paper, we discuss various aspects of the design, implementation, and use of JAPROSIM which is a general purpose discrete event simulation framework based on the Java programming language. JAPROSIM is an open source project developed for both academic and industrial purposes. It also merges process-interaction modeling structures with powerful java features in an intelligent way that encourages model simplicity, reusability and automatic statistics collection. Further motivations and aims are discussed. Java multithreading is a powerful built-in mechanism used to coordinate different entities in a coroutine-like mode. The main body of the paper is devoted for explaining the design of the framework in the context of Object Oriented Simulation. Finally, a summary of the proposed framework together with suggestions for improvements are given.

1 INTRODUCTION

Simulation models can be implemented in a variety of languages. We distinguish different periods in the history of simulation software. Early simulation tools were basically collections of routines written in general-purpose programming languages like FORTRAN. The forerunners are dedicated simulation languages like SIMSCRIPT. Predecessors are application-specific tools like NETWORK and ARENA. Actually there are integrated environments with point-click-drag-drop graphical interfaces like eM-Plant and Extend. This is in fact the result of reducing the effort deployed at the modelling stage of simulation projects, so expressiveness is weakened in spite of effectiveness and productivity. However, simulation still requires programming, even with direct access to a powerful programming language.

The opportunity to extend features of existing commercial simulation languages is limited due to the separation of the user from the base languages by offering pre-specified functionalities; thus deep access is reserved only to vendors. Separation has not eliminated the need for programming in simulation model building. In fact, successful industrial modellers are those who overcome separation by “programming” around the limitations caused by separation. Separation is also an obstruction to the

long-term model development and maintenance because this programming skill is outside of the mainstream of information systems training in academia and within the enterprise. Some simulation languages allow for certain programming-like expressions or statements, which are inherently limited. Others allow the insertion of procedural routines written in other general-purpose programming languages. Even when that is possible, the task of the user is complicated. It has to learn and master a new language. It must also deal with creation, insertion and update of statistical variables which is a source of several errors.

Object Oriented Modelling (OOM) is an excellent approach that deals with large and complex systems through abstraction, modularity, encapsulation, layering and reuse. OOM was born with SIMULA, the first object-oriented programming language. It introduced the object-oriented programming paradigm.

Conceptual model is obtained by decomposing a real system in a set of objects in interaction. Each object represents a real world entity that encapsulates state and behaviour. A class is a template for creating objects that share common related characteristics. Guidelines are, in particular, identification of object classes that make up a system with their interfaces and implementations.

Model conceptualization is one of the early steps in a simulation study for which OOM is suitable. Hence Object Oriented Simulation (OOS) benefits from all the powerful features of the OOM. It is, in addition, based on entities, events and simulation time that make the main differences, see [Joines & Roberts 98].

The use of the Unified Modelling Language (UML) in OOS seems to be even more appropriate since its version 2.0 offers significant improvements in dynamic behaviour modelling, which is a key aspect in any discrete event simulation, see [Page & Kreutzer 05]. The generation of executable code from static and dynamic UML models provides an important means for narrowing the gap between conceptual and computer models in simulation. Some research projects have already focused on an automatic generation of simulation programs from UML specifications; see [Arief & Speirs 00].

This paper presents the JAPROSIM framework design, implementation and use, for developing object-oriented simulations. The framework is documented using the UML and is divided into packages to organize the collection of classes into important functional areas. The main purposes of the framework are the easy to conceive, implement, use, reuse, understand and maintain discrete event simulation models, in addition to automatic collection of statistics.

The framework is implemented in Java programming language allowing deep access to its powerful features. It can serve as a kernel for the development of dedicated object-oriented simulation environments. In addition, since Java has been widely adopted as a teaching language in Computer Science curricula, it may also serve as an academic material for teaching discrete event modelling and simulation.



2 PROCESS INTERACTION AND SIMULATION IN JAVA

Process-interaction simulation denotes a particular world-view used to model the dynamics of discrete-event systems. The origins of this approach can be traced to the authors of SIMULA. It provides a way to represent a system's behaviour from the active entities point of view. As in SIMULA, active entities are transient entities moving through the system (dynamic entities). A process-oriented model is a description of the sequence of processing steps these entities experience as they flow through the system. This approach has significant intuitive appeal and is the predominant modelling worldview supported by commercial simulation software tools. Transaction flow is a special case of the more general process interaction worldview.

A system is modelled as a set of active entities in interaction. Interaction is a consequence of competition and/or cooperation for the acquisition of critical resources. Each active entity's life cycle consists of a sequence of events, activities and delays. A routine implementing an active entity requires special mechanisms for interrupting, suspending and resuming its execution at a later simulated time under the control of an internal event scheduler. This can be achieved using special programming languages that offer at least a SIMULA's coroutine like mechanism, thus programming languages offering multithreading like Java are suitable.

An entity's life cycle is a sequence of active and passive phases. On one hand, an active phase is characterized by the execution of the relevant process. Normally this corresponds to the events during which system state changes without progression of simulation time. On the other hand, passive phases are characterized by activities and delays. So the relevant process is suspended while simulation time advances. Events are the criterion of scheduling which explain the use of a future event list (FEL). After a process is suspended, the scheduler resumes and decides of which is the next process to reactivate according to the system state and the FEL. The scheduler is a special process that coordinates the execution of a simulation model.

Java is a general purpose language for creating safe, portable, robust, object-oriented, multithreaded and interactive programs for theoretically any area of application. It provides several extensive class libraries for developing graphical user interfaces, network and distributed applications with capabilities for web-based computing. It also has a utility package that contains useful classes that implement vectors, arrays, linked lists, hash tables...etc. It has been commonly adopted as a teaching language in Computer Science area. These features justify the choice of Java as an implementation language.

3 RELATED WORK

The idea of building process-oriented simulations using a general purpose object-oriented programming language is not original and several tools were developed in this way. For example, both of CSIM++ [Schwetman 95] and YANSL [Joines & Roberts 96] are based on C++, while PsimJ [Garrido 01], JSIM [Miller & al 98] are

based on Java. There are, however, unique aspects in JAPROSIM framework that lead to fundamental distinctions between our work and others.

Discrete Event Simulation tools written in Java, like PsimJ [Garrido 01] and SSJ [L'Ecuyer & al 02] are well designed and freeware libraries but not open source. Silk from Threadtec [Healy & Kilgore 97] and [Kilgore 00] is also well designed but is a commercial tool.

There is also a large collection of free open source libraries, we may consider for instance:

- JavaSim developed at the university of Newcastle upon Tyne [Little 99] is a set of Java packages for building discrete event process-based simulation, similar to that in Simula and C++SIM.
- JSIM [Miller & al 98] is a Java-based simulation and animation environment supporting Web-Based Simulation.
- simjava [Howell & McNab 98] is a process based discrete event simulation package for Java, similar to Jade's Sim++, with animation facilities.
- jDisco [Helsgaun 00] a Java package for the simulation of systems that contains both continuous and discrete-event processes.
- DESMO-J [Page & al 00] is a framework which supports both event and process worldviews.
- SimKit [Buss 02] is a component framework for discrete event simulation, influenced by MODSIM II and based on the event graph modeling.

JAPROSIM is also a well designed library, free and open source that adopts the popular process interaction worldview. Its design is simple and easy to understand. It is easy to build discrete event simulation using JAPROSIM, either for experimented programmers in Java or for simulation experts with elementary programming knowledge. JAPROSIM is not a java version of any existing simulation language as simjava or JavaSim.

In addition, JAPROSIM embeds a hidden mechanism for automatic collection of statistics. This approach also enables a clean separation between implementing the dynamics of the model and gathering data, so traditional performance measurements are automatically computed. The model can thus be created without any concern over which statistics are to be estimated, and the model classes themselves will not contain any code involved with statistics. This leads in more code source clarity. Nevertheless, users could, if needed, implement specific statistics collection using different classes offered by the JAPROSIM statistics package.

This feature makes the key difference between JAPROSIM and the other discrete event simulation libraries written in Java. Exception is made for SimKit which already offers this possibility, but which uses a different modeling approach based on event graphs.

4 JAPROSIM DESIGN AND PACKAGES

The JAva PRocess Oriented SIMulation (JAPROSIM) is a framework for developing discrete event object oriented simulation models. It is currently divided into six main packages:



-
- kernel: a set of classes dealing with active entities, scheduler, queues and resources.
 - random: contains classes for uniform random stream generation.
 - distributions: contains a rich set of classes for useful probability distributions.
 - statistics: contains classes representing intelligent statistical variables.
 - gui: a set of graphical user interface classes to use for project parameterization, trace and simulation result presentation.
 - utilities: a set of useful classes for express model development.

We will focus on the simulation *kernel*, *random*, *statistics* and *utilities* packages. We will briefly discuss the other packages to highlight their main structures in a queuing network scenario. UML will serve as the conceptual language to describe packages, classes and simulation models.

JAPROSIM kernel package:

The coroutine like mechanism is implemented through *SimProcess*, *Scheduler*, *StaticEntity* and *Entity* classes. A coroutine program is a collection of coroutines which run in quasi-parallel with one another. Each coroutine is an object with its own execution state, so that it may be suspended and resumed. In contrast of the JavaSimulation package [Helsingaun 00] which offers a Coroutine class identical to SIMULA's one, our aim in the design of JAPROSIM was putting a great emphasis into following the semantic of SIMULA but the design itself is not close to it. The advantage of this approach is that design is simpler without explicit coroutine class support and the semantics of facilities that are well-known and thoroughly tested through many years use of SIMULA are completely supported.

Native support for multithreaded execution is a fundamental aspect to the implementation of a natural process-oriented modeling capability in Java. Every active entity's life cycle is executed in a single separate thread.

In a process oriented worldview, simulation processes are placed into the FEL with respect to chronology (increasing simulation time) and managed by a scheduler. Processes are executed in pseudo-parallel and only one (which has the imminent simulation time) is running at any instance of real time. Simulation processes may execute concurrently at any instance of simulation time. Hence the scheduler executes in alternation with other simulation processes. This shared behavior is modeled through the *SimProcess* abstract class which extends the Java *Thread* class. The method *processResume(Entity e)* is called by the scheduler to reactivate a simulation process and *mainResume()* is called by a simulation process to reactivate the scheduler. Each simulation process has its own *lock* object. The scheduler has the *mainLock* object. Locks are used in combination with *wait()* and *notify()* to synchronize implementation threads instead of the Java deprecated methods *suspend()* and *resume()*. A thread which calls any of the previous methods will block on its own *lock* after notifying the appropriate one. *schedule(Entity e)* is a synchronized method offered by the *SimProcess* class which could be called by the scheduler or by a newly created simulation process for an appropriate insertion into the FEL.

At the end of its life cycle, a simulation process calls automatically the *dispose()* method to reactivate the scheduler without blocking itself. So the corresponding thread could be terminated. This leads to free occupied memory and improve simulation performance. Otherwise this may cause a Java runtime error “*java.lang.OutOfMemoryError: unable to create new native thread*” as we experienced with an academic version of the commercial package Silk [Bourouis & Belattar 06].

Specific behavior of a simulation process is normally described using the dedicated abstract method *body()*. It must be rewritten to be an ordered sequence of method invocations terminated by an implicit automatic call to *dispose()*. The behavior of the scheduler is also described using this method.

Since *SimProcess* is abstract, it is intended to be extended. A new class is created to model simulation processes. The *Entity* class provides the basis for defining classes that obey to the process-oriented simulation worldview. This class is declared to be abstract, so instances of *Entity* can not be created directly. Instead, modelers define their own classes that extend *Entity* and describe the dynamic behavior of the corresponding system components in terms of the process-oriented methods inherited in particular from those classes.

Each class derived from *Entity* runs in its own thread of execution, a capability inherited from *SimProcess*. The *Entity* class provides the implementation of the *run()* method which in turn invokes *body()*. The user is required to supply the *body ()* method. Four remarkable methods are offered, *insert()*, *remove()*, *seize()*, *hold()* and *release()*. They could be used to model familiar queuing scenarios. The *passivate()* method is used to wait until a specific system state is reached (ex: waiting for a resource to be free). Since the thread will be suspended and inserted into the passive list (PL) after a call to *passivate()*, this call is typically used within a *while()* loop. Each time the scheduler takes control; it starts reactivating suspended threads in the PL first, then dealing with the FEL. So such a reactivated thread would have the opportunity to return back to the PL, if there is no expected evolution in the system state.

The abstract class *StaticEntity* is used to model the behavior of active entities that have not the ability to move. Typical examples of those entities are “intelligent resources”. *StaticEntity* derives directly from *SimProcess*. Since The *Entity* class is used to model dynamic entities, it derives from *StaticEntity* and defines two new methods *insert()* and *remove()*. The other methods: *seize()*, *hold()*, *release()* and *passivate()* discussed previously are defined in the *StaticEntity* and hence inherited by *Entity*.

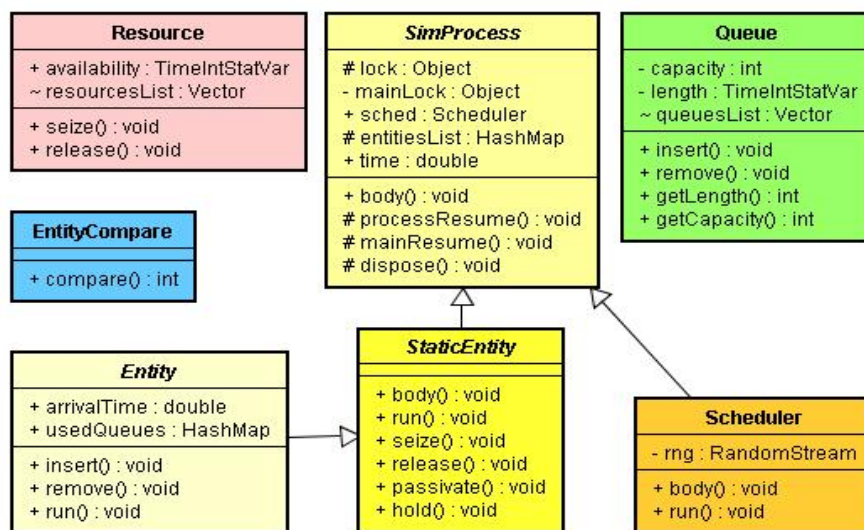
The scheduler proceeds in two phases. First, it reactivates each thread in the PL. So the reactivated thread checks for expected changes in the system state and may return back to the PL as it may continue executing the rest of its operations. Secondly, the scheduler picks the imminent simulation process notice from the FEL and reactivates the corresponding thread. These two phases are repeated as long as the simulation experiment termination condition isn't verified. Generally the termination condition is expressed in the form of pre-specified time duration.



The Scheduler class has an attribute *rng* which is an instance of a random number generator and could be customized by the user. The *EntityCompare* class implements the Java Comparator interface and used to implement priority queuing mechanism.

The *Resource* class represents a passive entity characterized by a capacity. Generally, a simulation process seizes some units of a resource to accomplish a service and releases them later. The *hold()* method of the *StaticEntity* class is used to specify the service duration.

The *Queue* class models a space for waiting which may be limited. It provides an ordered list where entities (or other user-defined types) can reside. Typically, an entity is inserted into a queue by having it activate the *insert(Queue q)* method of the *Entity* class.



Kernel class diagram

There is no implicit conditional status delay logic associated with queues, which means the entity's thread of execution is not suspended pending some system status evolution. Modeling conditional status delays is the realm of the *while()* and *passivate()* constructs. As a consequence, an entity can reside simultaneously in any number of queues. This feature can be particularly convenient in collecting certain types of system statistics related to wait times or queue lengths.

Another important distinction is that the removal of an entity from a queue could be independent of the ordering of the queue at the time of removal. Users are required to explicitly identify the entity to be removed at that time of removal. Typically this is accomplished by having the corresponding entity activate the *remove(Queue q)* method of the *Entity* class. While entities are generally inserted and removed from queues using the *insert(Queue q)* and *remove(Queue q)* methods of the *Entity* class, the same tasks can be accomplished by directly accessing the *insert(Entity e)* and *remove(Entity e)* methods defined in the *Queue* class.

The random and statistics packages

Random number generators (RNGs) are the basic tools of stochastic modeling. As any other craftsman, the modeler has to know his tools. Bad random number generators

may ruin a simulation and there are several pitfalls to be avoided. The *random* package provides the *RandomStream* interface which represents a base reference for creating Random Number Generators. Each RNG must rewrite the *RandU01()* method which normally returns a uniformly distributed number (a Java double) in the interval [0,1]. The *java.util.Random* could be used indirectly. Nevertheless, JAPROSIM provides yet a set of well known good RNGs see [L'Ecuyer 98] and [L'Ecuyer & Panneton 05], as Park-Miller, McLaren-Marsaglia and RandMrg in which the backbone generator is the combined multiple recursive generator (CMRG) Mrg32k3a proposed in [L'Ecuyer 99]. The *setSeed(long[] seed)* method is used to specify seeds instead of default values.

The user can define its own RNG by implementing the *RandomStream* interface. To be used with JAPROSIM, an instance of the user-defined RNG must be assigned to the *Scheduler*'s static public attribute *rng*.

A prosperous set of discrete and continuous Random Variate Generators (RVGs) is offered by the *distribution* sub-package. This set covers typically most practical distributions in discrete event simulation. However, the user could supply it with additional RVGs.

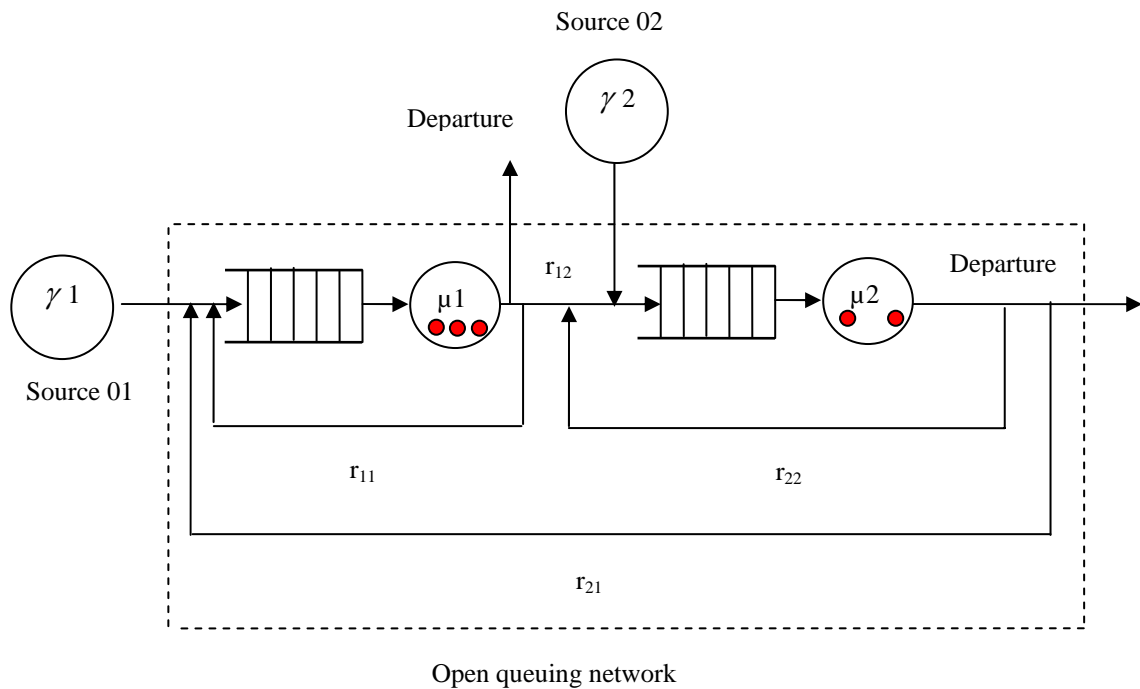
The statistics package provides two useful classes. *DoubleStatVar* class dealing with time-independent statistical variables (having double values) as response time and waiting time in a queue. It implements the mechanisms for keeping track of observational-based statistics and must be updated every time its value change using the *update()* method. *TimeIntStatVar* class is used for time-dependent statistics (with integer values) such as a queue length or number of customers in a system. Typically, the user instantiates the desired class, then puts and updates it in the appropriate code locations. The placement of statistical variables and their update is a source of several pitfalls. For this reason we have enhanced automatic placement and update of those variables for the most known and useful performance measurements, as we will see later through a queuing network scenario.

The utilities package

This package offers pre-specified entities with specific behavior. The *SimpleServiceStation* is used to model intelligent servers which are able to take decisions like "batch servers". The *SymetricServiceStation* models a service station with identical servers while *AsymetricServiceStation* models a service station with multiple heterogeneous servers. The homogeneity/heterogeneity of servers here comes from service distributions.

5 A SIMPLE QUEUING NETWORK SCENARIO

In order to show the JAPROSIM capabilities, let's compute an example of a simple queuing network, depicted in the figure bellow.



The network contains two service stations; each of them has an unlimited FIFO queue where transactions awaiting to be served are put. The transactions (coming from two independent exogenous sources) go into the system by means of two input points and may leave it also using two output points, after being served. We assume exponentially distributed random arrival times in the input streams of transactions and exponentially distributed random service time of both servers. The corresponding parameters of the simulation model are:

- $\gamma_1=3.57$ and $\gamma_2=4.82$, where γ_i is the exogenous arrival rate of the input source number i (and the parameter of the exponential distribution of arrival time).
- $\mu_1=4.15$, $\mu_2=5.96$, where μ_i is the parameter of the exponential distribution of a server of the service station number i .
- $c_1 = 3$, $c_2 = 2$, where c_i is the number of identical parallel servers at the i th service station.
- $r_{11} = 0.17$, $r_{12} = 0.33$, $r_{21} = 0.23$, $r_{22} = 0.18$, where r_{ij} is the probability that a transaction moves from station i to the station j .

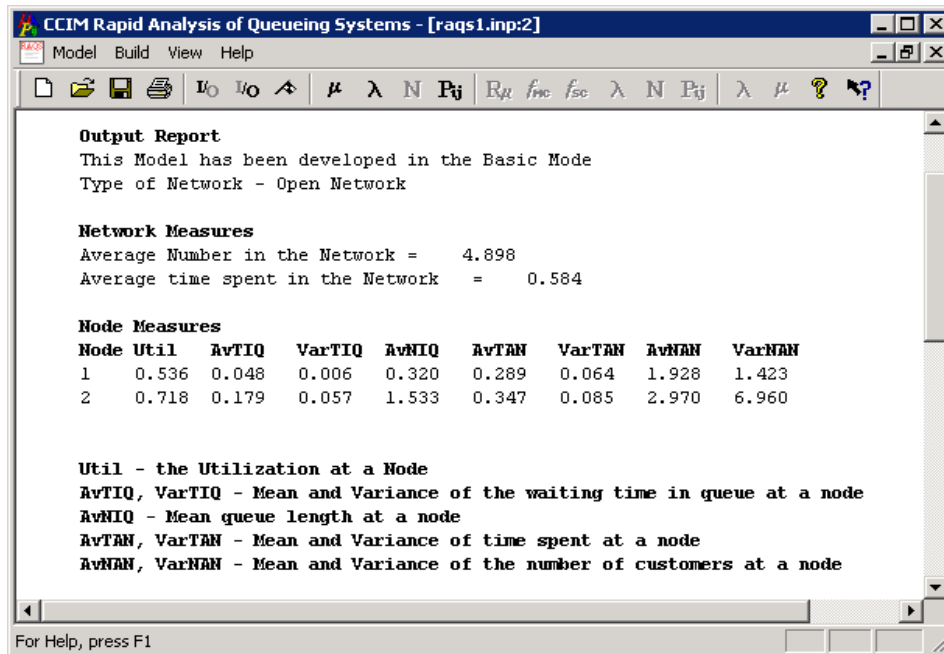
Analytical solution

This is a single-class open network with FCFS multiserver nodes, unlimited waiting rooms, reliable servers and probabilistic routing. Thus:

- $$\begin{cases} \lambda_1 = \gamma_1 + r_{11} \cdot \lambda_1 + r_{21} \cdot \lambda_2 \\ \lambda_2 = \gamma_2 + r_{12} \cdot \lambda_1 + r_{22} \cdot \lambda_2 \end{cases} \quad \text{Where } \lambda_i \text{ is the effective rate at node } i.$$

The solution for the previous equation system is: $\lambda_1= 6.2041$ and $\lambda_2= 6.8669$. Stations utilization: $\rho_1=0.5360$, $\rho_2=0.7184$, which means that the two stations are stable and hence the whole network. We can now compute the steady state network

performances. We obtained the results shown bellow in the figure, from the RAQS software developed at the Center for Computer Integrated Manufacturing (CCIM) Oklahoma State University. This software performs performance analysis using the algorithm of two-moments:



RAQS output window


Simulation using JAPROSIM

We can easily identify two resources which represent the two stations of the network. The first resource has a capacity of 3 and the second has a capacity of 2. Since we have two input arrivals, we must distinguish between two active entities with distinct life cycles.

In JAPROSIM we can model each active entity in a separate class derived from the *Entity* class, as we can expect a unique class in which the distinction between the inputs is made in the *body()* method.

```

01 import uoeb.japrosim.random.distributions.*;
02 import uoeb.japrosim.kernel.*;
03 public class Transaction extends Entity{
04     static Exponential arrival1 = new Exponential(3.57),
        arrival2 = new Exponential(4.82),
        serv1     = new Exponential(4.15),
        serv2     = new Exponential(5.96);
05     static Queue queue1 = new Queue("Queue 01"),
        queue2 = new Queue("Queue 02");
06     static Resource server1 = new Resource("Station
1",3),
        server2  = new Resource("Station
2",2);
07     static Uniform selection = new Uniform(0.0, 1.0);
    
```



```

08     int trID;
09     double choice;
10     public Transaction(int id) {
11         trID = id;
12     }
13     public void body() {
14         if(trID == 1) {
15             Transaction(1).beginAfter(arrival1.sample());
16             intoStation1(); } else {
17             Transaction(2).beginAfter(arrival2.sample());
18             intoStation2(); }
19     }
20     public void intoStation1() {
21         queue1.insert(this);
22         while(server1.getAvailability() < 1) {
23             passivate(); }
24         seize(server1, 1);
25         queue1.remove(this);
26         hold(serv1.sample());
27         release(server1, 1);
28         choice = selection.sample();
29         if(choice <= 0.17) { intoStation1(); }
30         else { if (choice <= 0.5) { intoStation2(); }
31             }
32     }
33     public void intoStation2() {
34         queue2.insert(this);
35         while(server2.getAvailability() < 1) {
36             passivate(); }
37         seize(server2, 1);
38         queue2.remove(this);
39         hold(serv2.sample());
40         release(server2, 1);
41         choice = selection.sample();
42         if(choice <= 0.18) { intoStation2(); }
43         else { if (choice <= 0.41) { intoStation1(); }
44             }
45     }
46 }

```

Transaction class

We sustain the last alternative given that all transactions have the same priority. So we obtained the code source for the unique needed Transaction class shown above.

The class structure consists of the data declarations (lines 4-9) which will define the characteristics of the simulation entities created from this class. The *body()* method (line 12-17) that will modify those entity's characteristics as the state of the system changes. Each instance of this class is assigned a set of static, user-defined attribute identifiers *arrival1* and *arrival2* for arrival distributions, *serv1* and *serv2* for service distributions, *queue1* and *queue2* are the queues for waiting. So *server1* and *server2* are service stations (resources). The *selection* represents a CDF function values when *choice* is the inverse transform function to get the destination according

to the routing probabilities. The source of the transaction is identified by *trID*. In the *body()* method, distinction is made between transactions according to their *trIDs*. They have a little difference in their life cycles. The *intoStation1()* and *intoStation2()* methods model the behavior of an entity in the corresponding station.

While each *Transaction* instance will have these unique attribute identifiers, all of its instances will share common static class variables representing either Java or JAPROSIM objects.

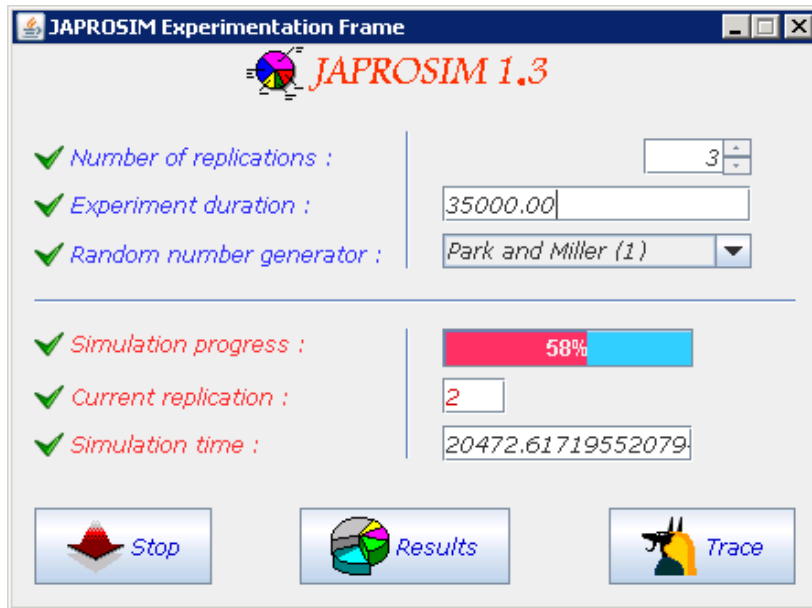
In this example, each *Transaction* creates (lines 14/16) the next arrival using a sample from a JAPROSIM *Exponential* random variable object defined in the data declaration. The *hold()* method is used to model a service with the specified time duration. The *delay* parameter is then assigned a sample value from the appropriate service time distribution (lines 24/35). More complex models would likely have different distributions for different arrivals and services. We can examine the use of *passivate()* inside a *while()* loop to insure waiting until the condition being wrong (lines 20, 21, 31 and 32).

To run a JAPROSIM simulation, we need another class which constitutes a starting point for any Java program. This class contains a the *main()* method for stand alone programs or the *init()* method for browser-based applets. It is where simulation model would be initialized, and the scheduler started. In our example, this class is called *OpenNetwork*:

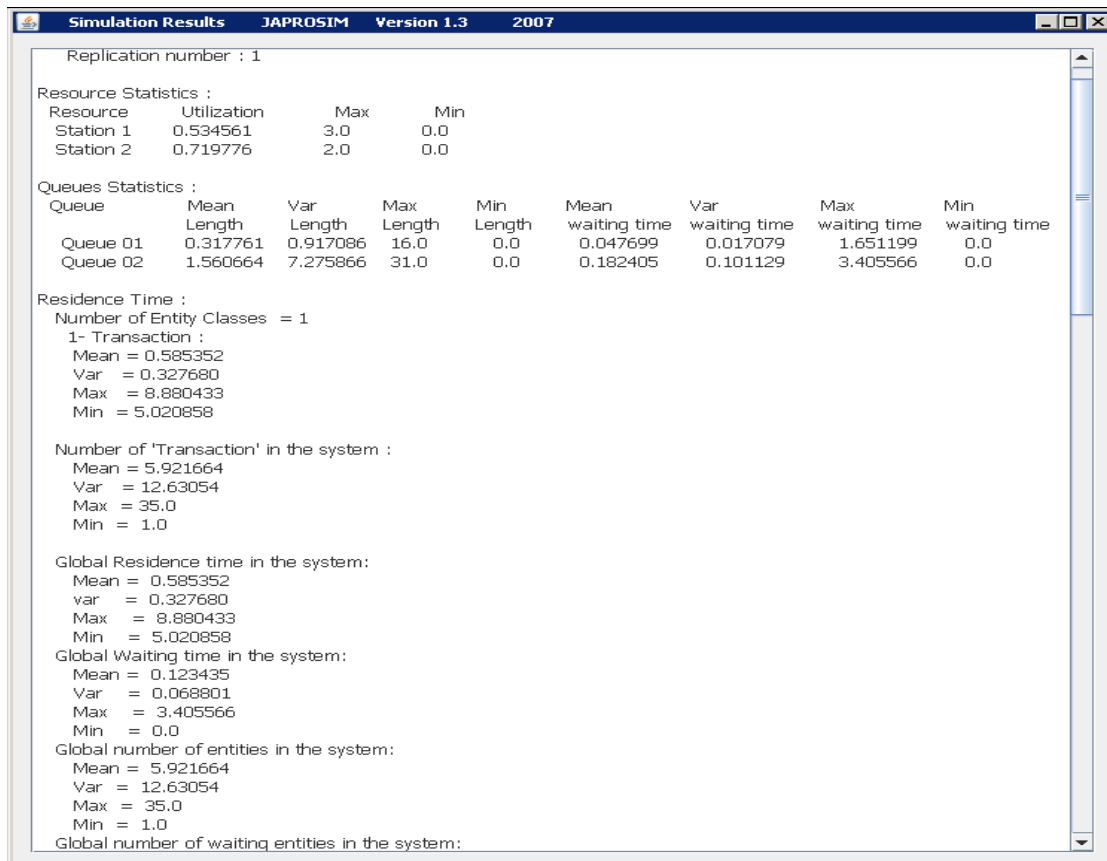
```
import uoeb.japrosim.kernel.*;
public class OpenNetwork {
    public static void main(String[] args) {
        new Transaction(1).beginAfter(0.0);
        new Transaction(2).beginAfter(0.0);
        SimProcess.sched.time = 0.0;
        SimProcess.sched.start();
    }
}
```

OpenNetwork class

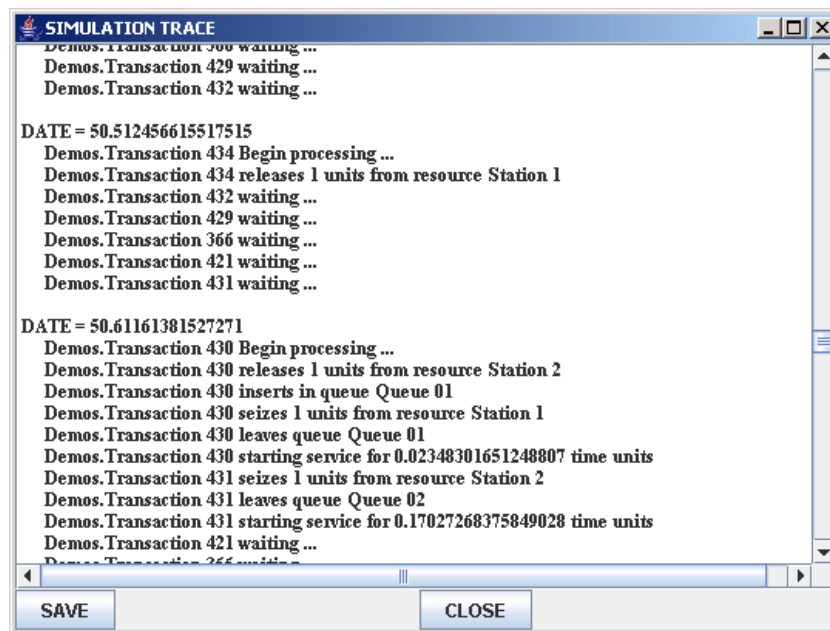
When executing this Java program, the JAPROSIM window occurs. It consists of an experimental frame where simulation parameters are set. Parameters like the number of replications, the experiment duration, the RNG used are to be specified by the user. A button *Run/Stop/Continue* allows user to start simulation, stop and resume it at any time during execution. Two other buttons are used for presentation of statistics and simulation trace. JAPROSIM model of this example was executed for 35000 time units as shown in the figure below. The high accuracy of results could be compared with those of analytical tools as seen above with RAQS.



JAPROSIM experimentation frame



Simulation results window



```

SIMULATION TRACE
Demos.Transaction 300 waiting ...
Demos.Transaction 429 waiting ...
Demos.Transaction 432 waiting ...

DATE = 50.512456615517515
Demos.Transaction 434 Begin processing ...
Demos.Transaction 434 releases 1 units from resource Station 1
Demos.Transaction 432 waiting ...
Demos.Transaction 429 waiting ...
Demos.Transaction 366 waiting ...
Demos.Transaction 421 waiting ...
Demos.Transaction 431 waiting ...

DATE = 50.61161381527271
Demos.Transaction 430 Begin processing ...
Demos.Transaction 430 releases 1 units from resource Station 2
Demos.Transaction 430 inserts in queue Queue 01
Demos.Transaction 430 seizes 1 units from resource Station 1
Demos.Transaction 430 leaves queue Queue 01
Demos.Transaction 430 starting service for 0.02348301651248807 time units
Demos.Transaction 431 seizes 1 units from resource Station 2
Demos.Transaction 431 leaves queue Queue 02
Demos.Transaction 431 starting service for 0.17027268375849028 time units
Demos.Transaction 421 waiting ...
Demos.Transaction 366 waiting ...
  
```

JAPROSIM trace window

6 AUTOMATIC STATISTICS COLLECTION

The first thing we can observe in the code source of the two classes, used to implement the above example in JAPROSIM, is that no class of the statistics package is explicitly used. In addition, no Java constructs are clearly used to do so. This is the key feature of JAPROSIM that all well known and useful performance measurements are implicitly and automatically handled. This ease to use of JAPROSIM is reflected by the user comfort in coding simulation models. The user doesn't worry about how many, or what kind of statistical variables to use, nor where to place and update them. Explicit statistical variable handling by the user may lead to undetectable programming errors and pitfalls. It could ruin simulation programs since the accuracy of simulation results is crucial. This is why JAPROSIM is said to be easy and safe to use for all users, including those who aren't qualified Java programmers.

This mechanism is embedded in the library. The *SimProcess* class declares a protected static *entitiesList* which is a Java *HashMap* to collect the residence time of each simulation entity class (a Java class that extends the JAPROSIM *Entity* class). The key for the *HashMap* is the class name and values are *DoubleStatVar*. In the *Entity* constructor, each time a new entity class is created, the above *HashMap* is updated. In the *run()* method of the *Entity* Class and after the call to the *body()* method, the residence time is updated using the simulation time and the *arrivalTime* attributes.

If in our previous example we used different *Transaction* classes, defined each one in its own Java class (extending the *Entity* JAPROSIM class), we would obtain different elements for *entitiesList*. In addition, each instance has another Java



HashMap to register all used queues. Hence, calls to *insert()/remove()* methods update automatically waiting time in the specified queues.

Each *Queue* object possesses a statistical variable to hold waiting time in it. This variable is updated through *insert()/remove()* methods. The number of entities in a queue is handled by a length time-dependent statistical variable. The resource *availability* is also a time-dependant variable. It is used to compute resource utilization. The *Queue* class has a static Java *Vector* to register all queues used in the simulation model. In the same way, the *Resource* class also has an analogous list to keep track of all used resources. Those lists have a package visibility; hence they could be accessed by all the simulation processes. They are updated each time a new resource or queue instance is created.

Nevertheless, the user is free to use JAPROSIM statistics package classes in his simulation code. There may be complex systems or situations that need specific statistics not covered by JAPROSIM.

7 SUMMARY

In this article, some basic facts about JAPROSIM have been presented, including its theoretical background. Being written in Java, a powerful easy-to-learn language, its ease to use and results accuracy are proven. It has the major key future of automatic and implicit collection of statistics over other similar frameworks. These are no doubt great advantages. In the distribution package, there are included source texts, compiled classes, documentation and many demonstration examples.

Today, JAPROSIM is a fully functional library which has been tested thoroughly. It could be used even for academic purposes as it is yet in our universities or for industrial purposes. Being a consistent kernel for general purpose discrete event simulation, it provides also a basis for building application-specific environments.

Future improvements will focus on increasing the JAPROSIM performances, integrating a graphical model building using JavaBeans, providing animations of simulation models, using xml standards for web-based simulation and ontologies to give more semantic to modeling and simulation.

REFERENCES

- [Arief & Speirs 00] Arief L. B, Speirs N. A: “A UML Tool for an Automatic Generation of Simulation Programs”, *the 2nd International Workshop on Software Performance (WOSP 2000)*, September 2000
- [Bourouis & Belattar 06] Bourouis Abdelhabib, Belattar Brahim: “Impact du choix de l’entité active sur les performances d’une simulation orientée processus multithreds basée sur Java”, *Proceedings of the CIAA 06*, Saida, Algeria, 15-16 Mai 2006.
- [Buss 02] Buss. A : “Component Based Simulation Modeling with SimKit”, *Proceedings of the 2002 Winter Simulation Conference*, pages 243-249.

- [Garrido 01] Garrido J. M : “Object-oriented Discrete Event Simulation with Java”. Kluwer/Plenum, NY, September 2001.
- [Helsgaun 00] Helsgaun Keld: “Discret Event Simulation in Java”, DATALOGISK SKRIFTER (writings on computer science), Roskilde University, 2000.
- [Healy & Kilgore 97] Kevin J. Healy, Richard A. Kilgore: “SilkTM : A Java-Based Process Simulation Language”, *Proceedings of the 1997 Winter Simulation Conference*, pp. 475-482, December 7-10, 1997.
- [Howell & McNab 98] Howell Fred and McNab Ross : "simjava: a discrete event simulation package for Java with applications in computer systems modelling", First International Conference on Web-based Modelling and Simulation, San Diego CA, Society for Computer Simulation, Jan 1998.
- [Joines & Roberts 98] Jeffrey A. Joines and Stephen D. Roberts: “Fundamentals of Object –Oriented Simulation”, *Proceedings of the 1998 Winter Simulation Conference*, pp. 141-149, December 13-16, 1998.
- [Joines & Roberts 96] Jeffrey A. Joines and Stephen D. Roberts: “Design of object oriented simulations in C++”. *Proceedings of the 1996 Winter Simulation Conference*, pp. 65-72. December 1996.
- [Kilgore 00] Kilgore Richard: “Silk, Java and Object-Oriented simulation”, *Proceedings of the 2000 Winter Simulation Conference*, pp 246-252. December 2000.
- [L’Ecuyer 98] L’ecuyer Pierre: “Uniform Random Number Generator” , *Proceedings of the 1998 Winter Simulation Conference*, pp 97-104. December 1998.
- [L’Ecuyer 99] L’ecuyer Pierre: “Good parameters and implementations for combined multiple recursive random number generators”. *Operations Research*, vol 47(1), pp 159–164. 1999.
- [L’Ecuyer & al 02] L’Ecuyer Pierre, Melian. L and Vaucher. J: “SSJ: A framework for stochastic simulation in Java”. *Proceedings of the 2002 Winter Simulation Conference*, pages 234–242. IEEE Press, 2002.
- [L’Ecuyer & Panneton 05] L’ecuyer Pierre, Panneton François: “Fast Random Number Generators Based on Linear Recurrences Modulo 2: Overview and Comparaison” , *Proceedings of the 2005 Winter Simulation Conference*, pp 110-119. December 2005.
- [Little 99] Little, M, C: “The JavaSim User's Manual”, Department of Computing Science, University of Newcastle upon Tyne, 1999.
- [Miller & al 98] John A. Miller, Y. Ge and J. Tao : “Component Based Simulation Environments: JSIM as a Case Study Using Java Beans”, *Proceedings of the 1998 Winter Simulation Conference*, pages 373-381, Washington DC.
- [Page & al 00] Page. B, Lechler. T and Claassen. S : “Objektorientierte Simulation in Java mitdem Framework DESMO-J” (“Object-Oriented Simulation in Java with the Framework DESMO-J”, in German). Libri Book on Demand, Hamburg, 2000. University of Hamburg, Faculty of Informatics. DESMO-J, 2004. [Online] <http://www.desmoj.de> (in December 2006).



[Page & Kreutzer 05] Bernd Page, Wolfgang Kreutzer: *The Java Simulation Handbook : Simulating Discrete Event Systems with UML and Java*, Shaker Publishing, November 2005.

[Schwetman 95] Schwetman, H: "Object-Oriented simulation modeling with C++/CSIM17". *Proceedings of the 1995 Winter Simulation Conference*, pp. 529-533. December 1995.

About the authors



BOUROUIS Abdelhabib received his BS degree in Computer science from the University of Constantine in 1999 and his MS degree from the University of Batna in 2003 where he is preparing a PhD degree. He is a lecturer at the University of Oum el Bouaghi since 2003. His research interests include Artificial intelligence, performance evaluation, parallel and distributed simulation.



BELATTAR Brahim is a professor at the University of Batna since 1992. He has also taught at the University of Constantine from 1982 to 1985. He received his BS degree in Computer science from the University of Constantine in 1981 and his MS and PhD degrees from the University Claude Bernard of Lyon (French) respectively in 1986 and 1991. His research interests include simulation, databases, semantic web and AI.