World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:7, No:2, 2013

A Java Based Discrete Event Simulation Library

Brahim Belattar, Abdelhabib Bourouis

***Abstract*—**This paper describes important features of JAPROSIM, a free and open source simulation library implemented in Java programming language. It provides a framework for building discrete event simulation models. The process interaction world view adopted by JAPROSIM is discussed. We present the architecture and major components of the simulation library. A pedagogical example is given in order to illustrate how to use JAPROSIM for building discrete event simulation models. Further motivations are discussed and suggestions for improving our work are given.

***Keywords*—**Discrete Event Simulation, Object-Oriented Simulation, JAPROSIM, Process Interaction Worldview, Java-based modeling and simulation.

I. Introduction

FROM an external point of view, the principal component of simulation software is the simulation language (SL) which allows description of simulation models and their dynamic behavior. Such languages are descendants of programming languages like FORTRAN, or ALGOL. Part of this heritage includes the batch-programming environment. To generate a program, the user had to create a source file, compile it, link it and then execute it. The user detects syntax errors in the compilation phase, and run time errors in the execution phase. To correct any errors, all phases have to be repeated. Such a procedure presents an extremely cumbersome interface to the user and is very time-consuming. Actual trends are in favor of integrated simulation and modeling environments where graphical user interfaces (GUI) play a great deal. This had led to the development and marketing of a huge amount of such environments from a multitude of sources.

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. Furthermore, separation has not eliminated the need for programming in simulation model building. In fact, successful industrial modelers are those who overcome separation by "programming" around the limitations caused by separation. Separation is also an obstacle 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, see [1].

Today, Object Oriented Modeling (OOM) is largely recognized as an excellent approach that deals with large and complex systems through abstraction, modularity, encapsulation, layering and reuse. A 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 behavior. A class is a template for creating objects that share common related characteristics. Object Oriented Simulation (OOS) benefits from all the powerful features of the OOM especially model conceptualization which is one of the early steps in a simulation study.

JAPROSIM is an object oriented simulation library, free and open source that adopts the popular process interaction worldview. Its design is simple and easy to understand. The library is implemented in Java programming language allowing deep access to its powerful features. 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. These features justify the choice of Java as an implementation language for the JAPROSIM library.

The library is documented using the UML and is divided into packages to organize the collection of classes into important functional areas. It is easy to build discrete event simulation models using JAPROSIM, either for experimented programmers in Java or for simulation experts with elementary programming knowledge. JAPROSIM can serve as a basis for the development of dedicated object-oriented simulation environments. Furthermore, since Java has been commonly adopted as a teaching language in Computer Science area, JAPROSIM may also serves as an academic material for teaching discrete event modeling and simulation.

The rest of the paper is organized as follows. We begin by reviewing related works. In Section III we describe the process interaction world view adopted by JAPROSIM. In Section IV we present the major components of the simulation library. A simple example is given in Section V in order to illustrate how to use JAPROSIM for building discrete event simulation models. Section VI summaries the paper and provides suggestions for future improvements of our work.

II. Related Works

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

Brahim Belattar is with the Department of computer Science, University Colonel El Hadj Lakhdar, Batna 05000, Algeria (phone: 213-06-96-96-92-19; fax: 213-33-86-89-42; e-mail: brahim.belattar@univ-batna.dz).

Abdelhabib Bourouis is with the Department of computer Science, University Larbi Ben M'Hidi, Oum El Bouaghi 04000, Algeria (e-mail: habib.bourouis@hotmail.com).

example, both of CSIM++ [2] and YANSL [3] are based on C++, while PsimJ [4], JSIM [5] are based on Java. Discrete Event Simulation tools written in Java, like PsimJ and SSJ [6] are well designed freeware libraries but not open source. Silk from Threadtec [1] and [7] 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 [8] is a set of Java packages for building discrete event process-based simulation, similar to that in Simula and C++SIM.
- JSIM [5] is a Java-based simulation and animation environment supporting Web-Based Simulation.
- Simjava [9] is a process based discrete event simulation package for Java, similar to Jade's Sim++, with animation facilities.
- jDisco [10] is a Java package for the simulation of systems that contains both continuous and discrete-event processes.
- DESMO-J [11] is a framework which supports both event and process worldviews.
- SimKit [12] is a component framework for discrete event simulation, influenced by MODSIM II and based on the event graph modeling.

JAPROSIM is not a java version of any existing simulation language as Simjava or JavaSim. There are, however, unique aspects in JAPROSIM that lead to fundamental distinctions between our work and others. For example, JAPROSIM embeds a hidden mechanism for automatic collection of statistics. This approach 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.

III. The Process Interaction Worldview in Japrosim

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 behavior 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 modeling worldview supported by commercial simulation software tools. Transaction flow is a special case of the more general process interaction worldview.

A system is modeled 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.

IV. The Japrosim Library

The JAva PRocess Oriented SIMulation (JAPROSIM) library is part of an ongoing project that aims at providing an advanced visual interactive simulation and modeling environment for Discrete Event Systems (DES). The library 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 results presentation.
- Utilities: a set of useful classes for express model development.

We will focus on the simulation kernel, random, statistics and utilities packages.

A. The Kernel Package

The kernel package is at the heart of JAPROSIM. A UML class diagram of the kernel is given below.

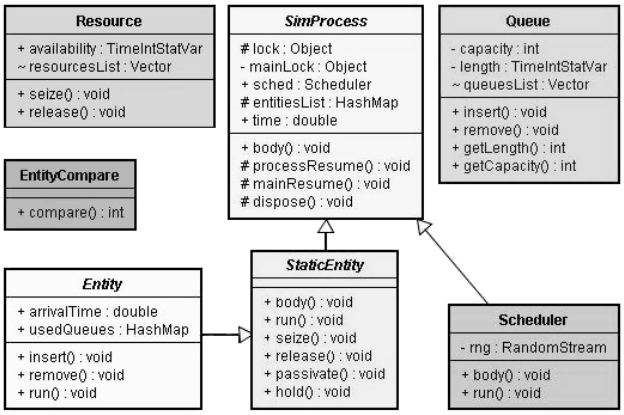

Fig. 1 The Kernel class diagram

As we can see, the kernel package is made up of classes dealing with active entities, scheduler, queues and resources.

The coroutine like mechanism is implemented trough 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. 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. 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.

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 cannot 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 from the FEL and reactivates the corresponding thread. These two phases are repeated as long as the simulation experiment termination condition isn't verified. 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 is 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. There is no implicit conditional status delay logic associated with queues, which means that 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 waiting 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 the 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 using the insert(Entity e) and remove(Entity e) methods defined in the Queue class.

B. The Random and Statistics Packages

Random number generators (RNGs) are the basic tools of stochastic modeling. 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]. JAPROSIM provides a set of well known good RNGs see [13] and [14], as Park-Miller, McLaren-Marsaglia and RandMrg in which the backbone generator is the combined multiple recursive generator (CMRG) proposed in [15]. 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 to be used in discrete event simulation. However, the user could supply it with additional RVGs.

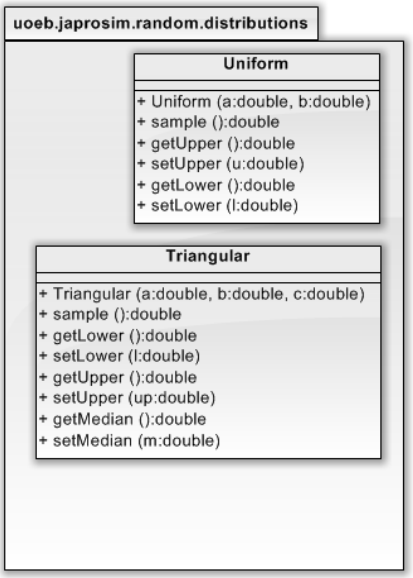

Fig. 2 The distribution sub-package

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 measures.

C. 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.

V. Simulation using JAPROSIM

A. Example Description

Let us give an example which illustrates a simplified simulation model of a TVs inspection and adjustment process as described in [16]. In this model, an arriving TV is first inspected at an inspection station. If a TV is found to be functioning improperly, it is routed to an adjustment station. After adjustment, the TV is sent back to the inspection station where it is again inspected. TVs passing inspection whether to the first time or after one or more routings through the

adjustment station, are sent to a packing area. A probabilistic branching is used when a TV passes the inspection station. It specifies that 15% of the TVs inspected are sent to the adjustment station and 85% are sent to the packing area. The inter-arrival time between TVs to the system, the inspection delay and the adjustment delay are all modeled as uniform variates (See the source code in Fig. 5).

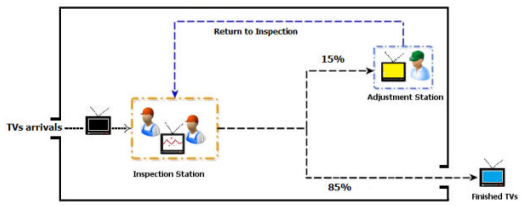

Fig. 3 The TVs Inspection example

From the description given, we can easily identify two resources which represent the two stations of the system modeled. The first resource represents the inspector and has a capacity of two units. The second resource represents the adjustor and has a capacity of one unit. Since we have one input arrivals, we distinguish one active entity in the model.

B. The JAPROSIM Simulation Model

In JAPROSIM we can model each active entity in a separate class derived from the Entity class. A class diagram of the JAPROSIM simulation model for this example is shown below:

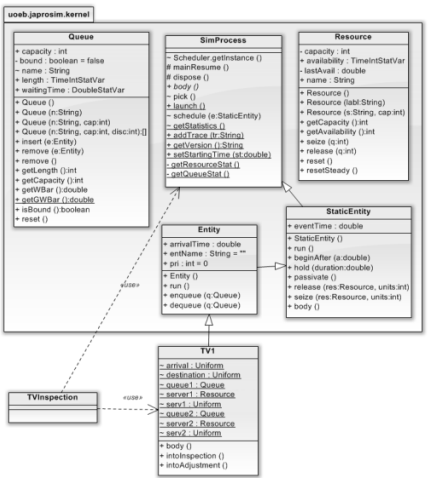

Fig. 4 A class diagram of the simulation model

From Fig. 4, it appears that the JAPROSIM simulation model of the example uses two classes: TVInspection and TV1. The source code of each class is given below.

```
1.  import uoeb.japrosim.kernel.*;
2.  import uoeb.japrosim.random.distributions.*;
3.  public class TV1 extends Entity {
4.  static Uniform arrival = new Uniform(3.5, 7.5);
5.  static Uniform destination = new Uniform(0.0, 1.0);
6.  static Queue queue1 = new Queue("INSP QUEUE");
7.  static Resource server1 = new Resource("Inspection", 2);
8.  static Uniform inspectdelay = new Uniform(6, 12);
9.  static Queue queue2 = new Queue("ADJT QUEUE");
10. static Resource server2 = new Resource("Adjustment", 1);
11. static Uniform adujstdelay = new Uniform(20, 40);
12. public void body() {
13. new TV1().beginAfter(arrival.sample());
14. intoInspection();
15. }
16. public void intoInspection() {
17. queue1.insert(this);
18. while (server1.getAvailability() < 1) {
19. passivate();
20. }
21. seize(server1, 1);
22. queue1.remove(this);
23. hold(inspectdelay.sample());
24. release(server1, 1);
25. if (destination.sample()<=0.15) {
26. intoAdjustment();
27. }
28. }
29. public void intoAdjustment() {
30. queue2.insert(this);
31. while (server2.getAvailability() < 1) {
32. passivate();
33. }
34. seize(server2, 1);
35. queue2.remove(this);
36. hold(adujstdelay.sample());
37. release(server2, 1);
38. intoInspection();
39. }
40. }
```

Fig. 5 Source code of The TV1 class

We can easily distinguish four parts in the source code of The TV1 class. The first part (from line 4 to line 11) serves to set the parameters of the model. We can see that the inspection delay, the adjustment delay and the inter-arrival time are defined as uniform variates with specific arguments. We have also to define the inspector and adjustor resources and their associated queues. The variable *destination* is defined as a uniform variate and is used when deciding if a TV just inspected is to be routed to the adjustment station or to exit the system.

The second part (from line 12 to line 15) serves to route the active entity to the inspection station and to create next TVs arrivals with respect to the inter-arrival time between TVs. The third part (from line 16 to line 28) represents the classical scheme of resource allocation. A TV arriving at the inspection station is inserted in the associated queue. When a resource unit is free, it is allocated to a waiting TV with respect to the queue priority. An inspection delay associated to this TV is sampled, and the TV will hold the resource unit seized until the associated delay is elapsed. The resource unit is then released and can be allocated to other waiting TVs. Line 28 serves to decide if the TV just inspected is to be routed to the adjustment station or to exit the system.

The fourth part (from line 29 to line 39) models the adjustor resource allocation scheme. A TV arriving at the adjustment station is inserted in the associated queue. When the adjustor resource is free, it is allocated to a waiting TV with respect to the queue priority. An adjustment delay associated to this TV is sampled, and the TV will hold the adjustor resource seized until the associated delay is elapsed. The adjustor resource is then released and the TV is sent back to the inspection station.

To run a JAPROSIM simulation model, we need another class which constitutes a starting point for any Java program. This class contains the main() method for standalone 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 *TVInspection*. The source code is as follows:

```
1  import uoeb.japrosim.kernel.*;
2  import uoeb.japrosim.random.distributions.*;
3
4  public class TVInspection {
5      public static void main(String[] args) {
6          SimProcess.time = 0.0;
7          SimProcess.sched.start();
8          new TV1().beginAfter(0.0);
9      }
10 }
```

Fig. 6 Source code of the TVInspection class

C. Running the Simulation Model

When running the simulation model, the JAPROSIM window is first displayed. It consists of an experimentation frame where simulation parameters are to be set. Parameters like the number of replications, the simulation duration, the RNG used must be specified here by the user. A button Run/Stop allows user to start simulation, stop and resume it at any time during execution. Two other buttons are used for presentation of simulation results and trace execution.

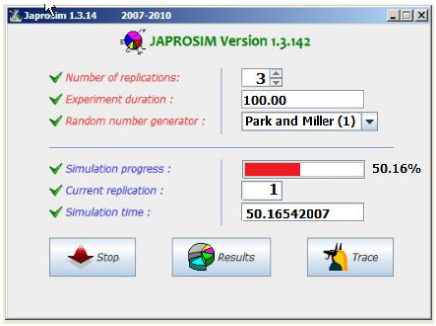

Fig. 7 JAPROSIM experimentation frame

At the end of each simulation run, the simulation results can be viewed in a textual form or in a graphical one.

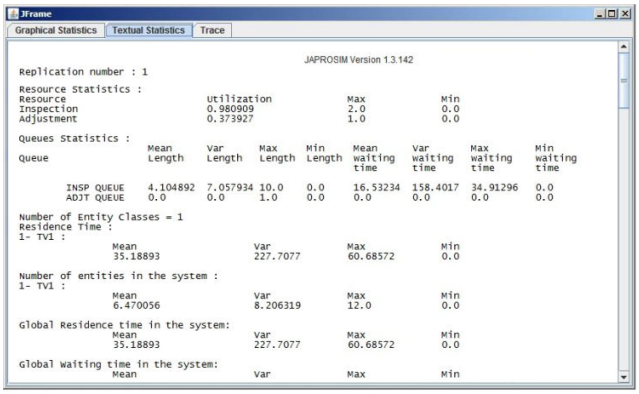

Fig. 8 Textual Simulation results

As we can see, the textual simulation results are expressed as statistical quantities which resume resources and queues utilization during a run. On the other hand, the graphical form uses plots, bar charts or pie charts. For example, Fig. 9 shows the utilization of the two resources used in the simulation model during each replication.

Fig. 9 graphical Simulation results

D. Automatic Statistics Collection

The first thing we can observe in the source code of the two classes used in the JAPROSIM simulation model, 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 measures are implicitly and automatically handled. 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.

Each Queue object possesses a statistical variable to hold waiting time in it. This variable is updated trough 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. It is clear that in practice, there may be complex systems or situations that need

specific statistics not covered by JAPROSIM.

VI. Conclusion

In this paper we have presented the JAPROSIM library for developing object-oriented simulations. It is written in Java and was deliberately kept simple, easy to use and extensible. The example presented reveals many advantages of the object-orientation of JAPROSIM and the process interaction worldview adopted. The relationship between the simulation model and the real system is more obvious and therefore easier to teach and to understand. The java source code of the simulation model is easy to understand and users can learn far more than if they have to experiment with sophisticated commercial simulation packages in which important details of the simulation implementation are hidden and thus never understood. 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. JAPROSIM is distributed since several years as an Open Source project (http://sourceforge.net/projects/japrosim/). The source code is available freely along with some documentation. Future improvements will focus on increasing the JAPROSIM performances, integrating a graphical model building facility, providing animations of simulation models and using xml standards for web-based simulation.

References

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

[2] H. Schwetman, "Object-Oriented simulation modeling with C++/CSIM17", *In Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, pp. 529-533, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, December 1995.

[3] J. A. Joines, S. D. Roberts: "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, pp. 65-72, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, December 1996.

[4] J. M. Garrido, "Object-oriented Discrete Event Simulation with Java". Kluwer/Plenum, NY, September 2001.

[5] J. A. Miller, Y. Ge, and J. Tao, "Component Based Simulation Environments: JSIM as a Case Study Using Java Beans", *In Proceedings of the 1998 Winter Simulation Conference*, ed. D. J. Medeiros, E. F. Watson, J. S. Carson and M. S. Manivannan, pp. 373-381, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, December 1998.

[6] P. L'Ecuyer, L. Melian, and J. Vaucher, "SSJ: A framework for stochastic simulation in Java", *In Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, pp. 234–242, December 2002.

[7] R. A. Kilgore, "Silk, Java and Object-Oriented simulation", *Proceedings of the 2000 Winter Simulation Conference*, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, pp. 246-252, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, December 2000.

[8] M. C. Little, "The JavaSim User's Manual", Department of Computing Science, University of Newcastle upon Tyne, 1999.

[9] F. Howell and R. McNab, "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, January 1998.

[10] K. Helsgaun, "Discrete Event Simulation in Java", DATALOGISK SKRIFTER (writings on computer science), Roskilde University, 2000.

[11] B. Page, T. Lechler and S. Claassen, "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.

[12] A. Buss, "Component Based Simulation Modeling with SimKit", *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, pp. 243-249, December 2002.

[13] P. L'ecuyer, "Uniform Random Number Generator", *In Proceedings of the 1998 Winter Simulation Conference*, ed. D. J. Medeiros, E. F. Watson, J. S. Carson, and M. S. Manivannan, pp. 97-104, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, December 1998.

[14] P. L'ecuyer, F. Panneton, "Fast Random Number Generators Based on Linear Recurrences Modulo 2: Overview and Comparison", *In Proceedings of the 2005 Winter Simulation Conference*, ed. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, pp. 110-119, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, December 2005.

[15] P. L'ecuyer, "Good parameters and implementations for combined multiple recursive random number generators". Operations Research, vol. 47(1), pp 159–164, 1999.

[16] C. D. Pegden, R. E. Shannon, and R. P. Sadowski, *Introduction to Simulation Using SIMAN*. New York McGraw-Hill Inc., 1990.

B. Belattar 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.

A. Bourouis 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.