Ascertaining Important Features of the JAPROSIM Simulation Library

Brahim Belattar
Department of computer Science
University Colonel El Hadj Lakhdar
Batna 05000, Algeria
brahim.belattar@univ-batna.dz

Abdelhabib Bourouis
Department of computer Science
University Larbi Ben M'Hidi
Oum El Bouaghi 04000, Algeria
habib.bourouis@hotmail.com

Received: November 19, 2019. Revised: July 15, 2021. Accepted: November 5, 2021.

***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. In order to ascertain important features of JAPROSIM, examples are given. Further motivations are discussed and suggestions for improving our work are given.**

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

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 [1]. 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.

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 modelling and simulation.

The structure of this paper is as follows: In section 2, we present an overview of related work. In section 3 we describe the process interaction world view adopted by JAPROSIM. In section 4 major components of the simulation library and its architecture are detailed. Examples are given in order to ascertain important features of JAPROSIM. Section 6 summaries the paper and provides suggestions for future improvements of our work.

II. RELATED WORK

A large research effort has been devoted to enrich mainstream languages as C, C++, Java, Python with simulation capabilities [2]. The most common choice is to provide the additional simulation functionality through a software library. Independently of the architectural level at which they are provided (application, library, language), the simulation

capabilities embody a world view for their users. The world view is essentially the set of concepts that constitute the basic elements available to the modeler to compose and to specify the simulation. The diverse world views are functionally equivalent, but differ in expressive power and in terms of computational efficiency. 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++ [3] and YANSL [4] are based on C++, while PsimJ [5], JSIM [6] are based on Java. Discrete Event Simulation tools written in Java, like PsimJ and SSJ [7] are well designed and freeware libraries but not open source. Silk [8] 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 [9] is a set of Java packages for building discrete event process-based simulation, similar to that in Simula and C++SIM.
- JSIM [6] is a Java-based simulation and animation environment supporting Web-Based Simulation.
- Simjava [10] is a process based discrete event simulation package for Java, similar to Jade's Sim++, with animation facilities.
- jDisco [11] is a Java package for the simulation of systems that contains both continuous and discrete-event processes.
- DESMO-J [12] is a framework which supports both event and process worldviews.
- SimKit [13] is a component framework for discrete event simulation, influenced by MODSIM II and based on the event graph modeling.

SimJava and JSim are among the first implementations of the thread-based class of simulators. These early efforts pay particular attention to web-based simulation and to the Java Applet deployment model. Many simulators aim at replicating the functionality and design of Simula in Java. For example, DesmoJ supports advanced process-oriented modeling features. These include capacity-constrained resources, conditional waiting and special process relationships as producer/consumer and asymmetric master/slave. SSJ is designed for performance, flexibility and extensibility. It offers its users the possibility to choose between many alternatives for most of the internal algorithms and data structures of the simulator [2].

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

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 [14]. 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. It 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. A UML class diagram of the kernel is given below.

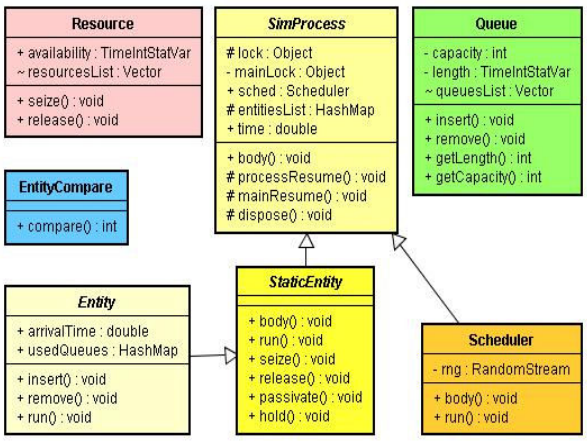

Fig. 1. The Kernel class diagram

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 [15] and [16], as Park-Miller, McLaren-Marsaglia and RandMrg in which the backbone generator is the combined multiple recursive generator (CMRG) proposed in [17]. The setSeed(long[] seed) method is used to specify seeds instead of default values.

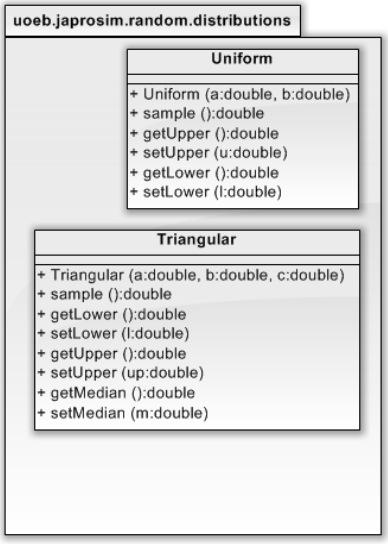

Fig. 2. The distribution sub-package

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.

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. JAPROSIM BY EXAMPLES

This section discusses two simulation models which have purposely been simplified in order to promote understanding JAPROSIM capabilities. They give a good impression of the wide applicability of simulation in production and logistics. However, it is important to underline that with JAPROSIM, a simulation model is simply a Java source program which merges process interaction modeling features provided by the library simulation packages with powerful features of the Java programming language.

A. a Simple Queueing Network Scenario

Queueing network models have been used extensively as a modeling paradigm for deriving analytical as well as simulation based performance measures. They are commonly used to model a wide range of discrete event systems. Kendall's notation is a mean for describing queueing networks especially in case of simple systems. For complex ones, a graphical notation with textual annotations is used instead. To analyze the model either by simulation or by mathematical analytic tools, the model is commonly coded and saved directly in a proprietary file format.

In order to show JAPROSIM capabilities, we consider a simple queuing network as depicted in Fig. 3. 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.

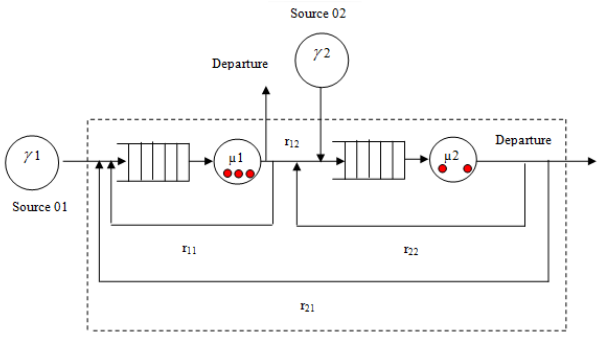

Fig. 3. A simple queuing network

The corresponding parameters of the simulation model are:

- γ_1=3.57 and γ_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).
- μ1=4.15, μ2=5.96, where μi is the parameter of the exponential distribution of a server of the service station number i.
- c1 = 3, c2 = 2, where ci is the number of identical parallel servers at the ith service station.
- r11 = 0.17, r12 = 0.33, r21 = 0.23, r22 = 0.18, where rij is the probability that a transaction moves from station i to the station j.

As we can see, this is a single-class open network with FCFS multi-server nodes, unlimited waiting rooms, reliable servers and probabilistic routing. Thus the analytical solution is given by the following equation system:

- $$\begin{cases} \lambda_1 = \gamma_1 + r_{11}.\lambda_1 + r_{21}.\lambda_2 \\ \lambda_2 = \gamma_2 + r_{12}.\lambda_1 + r_{22}.\lambda_2 \end{cases}$$

Where λ_i is the effective rate at node i. The analytical solution for this equation system is:

- λ_1=6.2041 and λ_2=6.8669

The stations utilization is:

- ρ_1=0.5360, ρ_2=0.7184

To compute the steady state network performances we used the RAQS (Rapid Analysis of Queueing Systems) software developed at the Center for Computer Integrated Manufacturing (CCIM) Oklahoma State University. The results obtained are:

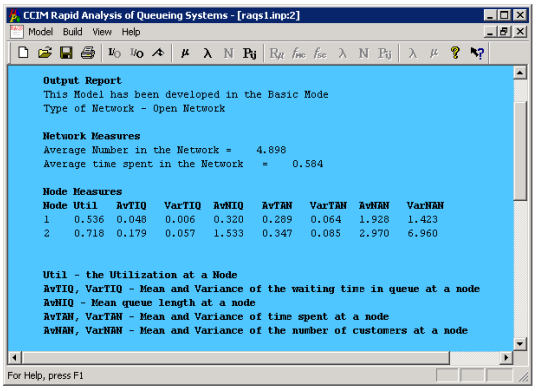

Fig. 4. RAQS output window

To build a simulation model for this example using JAPROSIM, we can first identify two resources which represent the two stations of the network. The first resource has a capacity of three units and the second has a capacity of two units. The two input arrivals in the network lead us to distinguish between two active entities with distinct life cycles. Since all transactions have the same priority, we can use one class (named Transaction) for which the source code is:

```
1.  import uoeb.japrosim.random.distributions.*;
2.  import uoeb.japrosim.kernel.*;
3.  public class Transaction extends Entity{
4.  static Exponential arrival1 = new Exponential(3.57),
5.                     arrival2 = new Exponential(4.82),
6.                     serv1    = new Exponential(4.15),
7.                     serv2    = new Exponential(5.96);
8.  static Queue queue1 = new Queue("Queue 01"),
9.                queue2 = new Queue("Queue 02");
10. static Resource  server1 = new Resource("Station 1",3),
11.                  server2 = new Resource("Station 2",2);
12. static Uniform selection = new Uniform(0.0, 1.0);
13. int trID;
14. double choice;
15.    public Transaction(int id) {
16.       trID = id;
17.    }
18.    public void body(){
19.       if(trID == 1){
20.             new Transaction(1).beginAfter(arrival1.sample());
21.             intoStation1();  }
22.       else{
23.             new Transaction(2).beginAfter(arrival2.sample());
24.             intoStation2();  }
25.    }
26.    public void intoStation1(){
27.          queue1.insert(this);
28.          while(server1.getAvailability()<1){
29.                passivate();    }
30.          seize(server1, 1);
31.          queue1.remove(this);
32.          hold(serv1.sample());
33.          release(server1, 1);
34.          choice = selection.sample();
35.          if(choice<=0.17){ intoStation1();  }
36.          else{ if (choice<=0.5){ intoStation2(); }
37.    }
38.    public void intoStation2(){
39.          queue2.insert(this);
40.          while(server2.getAvailability()<1){
41.          passivate(); }
42.          seize(server2, 1);
43.          queue2.remove(this);
44.          hold(serv2.sample());
45.          release(server2, 1);
46.          choice = selection.sample();
47.          if(choice<=0.18){  intoStation2(); }
48.          else{ if (choice<=0.41){ intoStation1(); }
49.    }
50. }
```

Fig. 5. The transaction class.

From Fig. 5, it appears that the class structure consists of data declarations (lines 4-14) which define the characteristics of the simulation entities created from this class. The body() method (line 18-25) is used to modify 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 entities waiting queues
- server1 and server2 are service stations (resources).

In this example, each Transaction creates (lines 20/23) 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 32/44). 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 28, 29, 40 and 41). The selection represents a CDF function values where 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 between transactions is made according to their trIDs. The intoStation1() and intoStation2() methods specify 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.

To run a JAPROSIM simulation, we need another class which constitutes a starting point for any Java program. This class contains a main() method for standalone programs or the init() method for browser-based applets. In the example, this class is called OpenNetwork and its java source code is:

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

Fig. 6. The OpenNetwork class.

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.

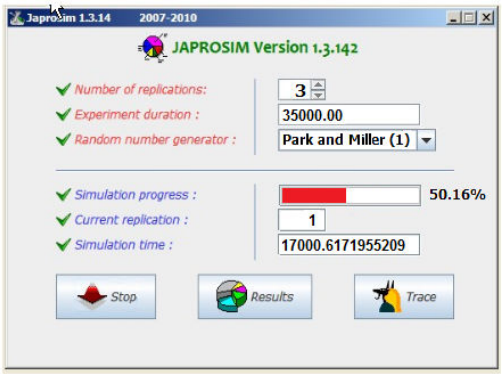

Fig. 7. JAPROSIM experimentation frame.

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

The JAPROSIM model of this example was executed for 35000 time units. The statistical results for the first replication are given below:

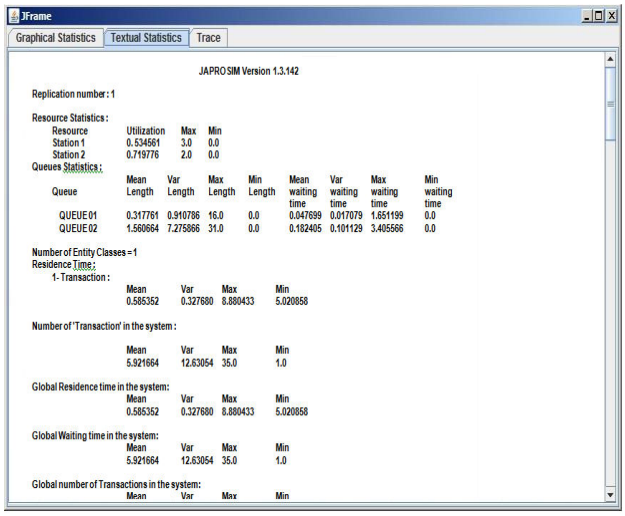

Fig. 8. Statistical results.

The statistical results obtained have been compared with those given by the analytical method (RAQS). Fig. 9 summarizes important performance measures as reported by JAPROSIM at the end of simulation and the same measures as calculated by RAQS.

	JAPROSIM	RAQS
Resources (nodes) utilization	Station 1: **0.534561**	node 1: **0.536**
	Station 2: **0.719776**	node 2: **0.718**
Mean queue length	Queue 01: **0.317761**	node 1: **0.320**
	Queue 02: **1.560664**	node 2: **1.533**
Mean waiting time in queue	Queue 01: **0.047899**	node 1: **0.048**
	Queue 02: **0.182406**	node 2: **0.179**

Fig. 9 Simulation versus analytical solution.

We observe that the simulation results are closer to the analytical solution. This justifies our claims that the JAPROSIM library is well designed and results produced are of high accuracy. However, it is known that simulation can never give the exact solution as analytic methods do. This is why we must run the model more than one time until we reach the best solution. A statistical analysis of simulation results will help in deciding to do other replications or not.

B. The Tvs inspection and adjustment Example

The second example illustrates a simplified simulation model of a TVs inspection and adjustment process as described in [18]. 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.

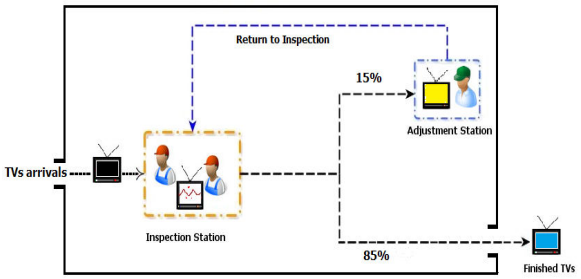

Fig. 10 The TVs Inspection example.

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. A class diagram of the JAPROSIM simulation model for this example is shown below:

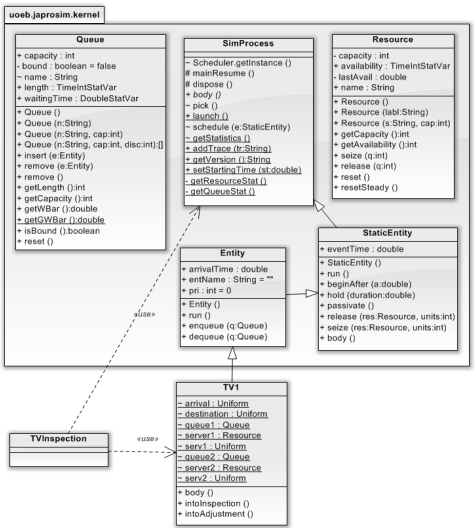

Fig. 11 A class diagram of the simulation model.

The JAPROSIM simulation model of the example uses two classes named respectively TV1 and TVInspection

```
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. 12 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.

Like the first example, to run this simulation model, we need another class which contains the main() method. It is where simulation model would be initialized, and the scheduler started. This class is called TVInspection and its java source code is:

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

Fig. 13 Source code of the TVInspection class.

Beside classical statistical results like those presented with the first example, JAPROSIM allows graphical presentation of selected performance measures. Example of such presentation is given in Fig. 14. It shows the utilization of the two resources used in the simulation model during each replication.

Fig. 14 Graphical Simulation results.

C. *Important features of JAPROSIM*

The examples presented reveal 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.

Furthermore, we can observe in the source code of the classes used in the JAPROSIM simulation models, 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. From the examples presented, many advantages of the object-orientation of JAPROSIM and the process interaction worldview have been exhibited. Today, JAPROSIM is a fully functional library which has been tested thoroughly. JAPROSIM is distributed since several years as an Open Source project. 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 and interoperability with other tools.

REFERENCES

[1] Korichi Ahmed, Belattar Brahim, Towards a Web Based Simulation Groupware: Experiment with BSCW, WSEAS transactions on Business and Economics, Issue 1, Volume 5, pp. 9-15, January 2008.

[2] Antonio Cuomo, Massimiliano Rak, Umberto Villano: Process-oriented Discrete-event Simulation in Java with Continuations - Quantitative Performance Evaluation. In Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies, Technologies and Applications, pp. 87-96, 2012, Rome, Italy, 28 - 31 July, 2012.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] A. Buss, "Component Based Simulation Modeling with SimKit", 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. 243-249, December 2002.

[14] Iulia Dumitru, Ioana Fagarasan, S. St. Iliescu, Grigore Stamatescu, Nicoleta Arghira, Veronica Barbulea, A Modular Process Simulator with PLC, In Proceedings of the 9th WSEAS International Conference on SIMULATION, MODELLING AND OPTIMIZATION (SMO '09), pp. 391-395, Budapest Tech, Hungary, September 3-5, 2009.

[15] 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.

[16] 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.

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

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

Creative Commons Attribution License 4.0 (Attribution 4.0 International, CC BY 4.0)

This article is published under the terms of the Creative Commons Attribution License 4.0
https://creativecommons.org/licenses/by/4.0/deed.en_US