A Java Package for Synchronous Simulation Processing

Rogério Esteves Salustiano and Carlos Alberto dos Reis Filho
School of Electrical and Computer Engineering (FEEC)
State University of Campinas (UNICAMP)
Campinas, Brazil
{rsalusti, carlos_reis}@lpm.fee.unicamp.br

Abstract - **It is common practice in the process of designing systems, protocols and circuits to implement a simulation phase aiming at to foresee performance and to identify possible flaws. In principle, the simulation of processes, which feature the occurrence of concurrent events, would require some sort of parallel processing in order to be faithful to the order of the events. However, in machines that execute one instruction at a time, this is an impossible strategy. To circumvent this problem, diverse techniques addressing pseudo-parallel processing in simulators have been proposed. This paper presents the development of a Java package with a base structure that allows the implementation of synchronous processing. Among its comprising classes there is a pair of specific ones, namely *SynchGlobal* and *SynchThread*, which provides the capability of pseudo-parallel processing. By combining object-oriented paradigm and multithreading, these two classes implement the barrier synchronization technique, which is the essential substrate for developing simulators that feature time coordination. As an illustration of possible applications, two developed simulators are discussed: a logic circuit simulator and a Wireless Sensor Network (WSN) simulator.**

Keywords- Java; parallel processing; simulation

I. Introduction

Simulation is an important tool in the development of systems, communication protocols and devices. Both academic and business worlds usually require a phase of simulation to evaluate the functionality and to estimate the performance of models. A countless number of tests, configurations and improvements can be exploited during the simulation phase, directing and quantifying more accurately the required investments in the project.

The design of specific simulators very often increases the time and cost of projects, making them impracticable. Thus, a generic simulation tool written in a programming language widely used along with an easy-to-use Application Programming Interface (API) can reduce the latency time necessary to obtain results from projects.

Java and C++ have been widely used in the design of simulation libraries. SimJava [1] and JSDESLib [2] are examples of Java libraries, while SimPack [3], simCore [4], SIM++[5] and systemC [6] are libraries implemented in C++.

Some of the advantages of Java over C++ are its direct syntax to handle exceptions [7], its portability allowed by Java Virtual Machines (JVM) and the free and standard compiler distributed by Sun Microsystems. The main disadvantage lies in the fact that programs interpreted by JVM are slower than others compiled with C++.

The event-driven control strategy presented in this paper is comparable to the ideas developed in SimJava [1] simulation package, which is based on SIM++ [5]. The synchronization package herein proposed, however, implements the synchronization among threads with priorities. This feature is important in order to control the sequence of execution of simultaneous running threads that represent entities acting independently in the real world, but keeping a pre-established communication. The sequence write-before-read in a shared memory must be respected to guarantee the perfect execution of a simulation.

The contribution of this paper is a simple Java package, which intends to assist programmers in the development of simulators. Based on event-driven approach, it has six classes that use Java multi-thread capabilities to perform a barrier synchronization technique featuring an improved priority mechanism.

II. Parallel Simulation Technique

The main problem that simulators try to solve is how to execute several processes simultaneously in the serial processing architecture of computers. By means of this approach, the execution is performed serially, however ensuring the causality of events.

Simulators models can advance the simulation clock in two different ways. In the first one, called time-driven, a fixed time interval is defined and the simulations clock advances precisely this amount, verifying if some process are waiting to execute at that specific time. The problem in time-driven approach is to find out the smallest time interval that is needed to meet the time required by all processes.

The second approach is the event-driven, whereupon the simulation next-event time is determined by the process that has the next imminent event to perform, i.e., the next process that is waiting to run in the time line. This approach has the advantage that periods of inactivity can be skipped. The causality is assured if too or more processes do not have the same next time to execute.

Barrier Synchronization [8] is a simulation technique that allows the control of processes that run simultaneously and can

This work has been supported by National Council for Scientific and Technological Development (CNPq - Brazil) and Brazilian Agency for Personnel Development (CAPES - Brazil).

be used to implement an event-driven simulator. By means of this technique, a set of processes run freely until a barrier instruction is reached and stops. Processes return to execute their instructions when all other processes have reached the same barrier. All instructions between two barriers in a process are executed simultaneously with the instructions of other processes that are in the same barrier interval.

Since processes represent devices or phenomena from the real world, they change internal parameters and/or perform actions at determinate periods of time. Thus, (1) not all processes should run at the same frequency and (2) processes that have the same or multiply periods of execution have their instructions executed concurrently and a defined rule to decide which process run first (the sequence of execution) must be established.

Therefore, an adaptation of the Barrier Synchronization must be performed to solve the two above stated problems. To solve (1), multiple barriers can be created and not all processes need to be blocked at the same barriers. The problem (2) can be solved adding priorities to processes, so if two or more processes have been blocked at the same barrier, their priorities will determine the sequence of execution.

Fig. 1 shows the proposed adaptation in a diagram, where four processes (A, B, C and D) are defined with their periods and priorities. Process C, for instance, has a period of 1.5s and a priority of 1. This means that every 1.5s of simulation (00:00:0.000, 00:00:1.500, 00:00:3.000 and so on) the process is unblocked from the barriers. As process C has a priority of 1, if other processes are unblocked at the same barrier (Simulation Time), it will execute after processes D and A, respectively (if they are unblocked) and before process B (if it is also unblocked).

The execution sequence of processes A, B, C and D, respecting the time sequence (barriers determined by processes periods) and the priority sequence, from 0 to 4s will be: D→A→C→B→B→A→B→C→B→D→A→B→B→A→C→B→B→D→A→B.

This improvement of Barrier Synchronization increases its potentiality toward implementing more accurate simulators, although it may result in slower simulations due to the increased number of operations needed to release a process from each barrier. Java threads facilitate its implementation and allow an optimized memory management.

Process	*Period*	*Priority*
A	1.0s	2
B	0.5s	0 (lower)
C	1.5s	1
D	2.0s	3 (higher)

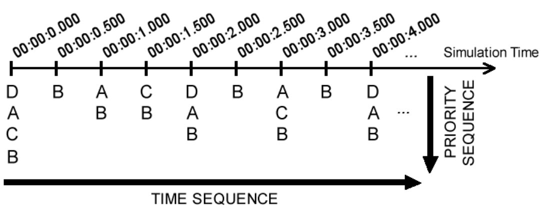

Figure 1. Execution sequence of processes.

III. Java Implementation and Synchronization Package Details

The parallel synchronization technique above described can be easily implemented in a programming language that supports multi-thread. Java is an excellent choice since it is a native threaded language, i.e., it includes thread primitives as part of the language itself. With the advent of multi-core processors, Java threads will gain power using this benefit to run parallel instructions in a real way [9]. Furthermore, Java is multiplatform, which is an important feature to guarantee the portability for simulators developed.

Some features of Java allow the control of threads running at the same time or just prevent the concurrent access of specific objects or methods. *Synchronized* methods resolve the *producer/consumer* problem, guaranteeing that one thread do not execute a method while other thread is executing it. The package *java.util.concurrent* has a lot of facilities to control the execution of threads. The class *CyclicBarrier*, e.g., implements the basic barrier synchronization, i.e., it allows that a set of threads run until they reach a common barrier; from this barrier, all threads are released until another barrier is reach and this occur in a cyclic way. This class, with some adjustments, can be easily used to implement a simple time-driven simulator.

Despite the Java's features mentioned above already available in its current version (1.6.0_16), threads that are running are difficult to control. Standard *stop*, *suspend* and *resume* methods could be used, despite being considered as unsafe operations [10] by Java developers. The *sleep* method is unsafe to perform precise delays in threads execution because the sleep time required is not guaranteed by the Java Virtual Machine [11].

Moreover, the sequence that threads execute in the shared processing cannot be established by the default priority (range 1 to 10) of *Thread* class. Threads with higher priorities run until they complete, sharing the "priority space" controlled by a *round-robin* scheduler. Lower-priorities threads will run only when higher-priorities threads complete their executions. So, threads with lower-priorities could never run if higher-priorities threads never complete.

Therefore, the classes from the standard Java API cannot be directly used to implement the stated parallel synchronization. Thus, a java package compound of six classes, namely *br.eng.rsalustiano.synchronization*, was developed to allow a rapid, efficient and precise design of simulators. Fig. 2 shows the class diagram of the package. A brief description of all classes in the package is made below.

- ***SynchPeriod*** is used to define the period of execution of a *SynchThread*. This period can be defined as a period in seconds or as a frequency in hertz.
- ***SynchThread*** is the synchronized thread. Developers must implement synchronous threads extending this abstract class, passing a *SynchPeriod* and a priority (integer value) to the super-class constructor. Methods *synchRun*, *synchInit* and *synchFinalize* must be implemented. The method *synchInit* will be called at the start of simulation and *synchFinalize* at the end.

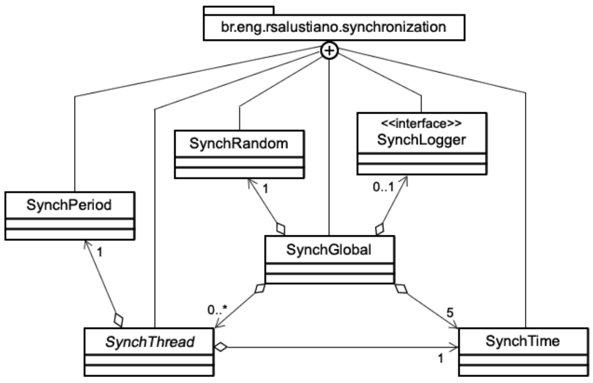

Figure 2. Package br.eng.rsalustiano.synchonization class diagram

The instructions to be executed synchronously with other *SynchThreads* must be defined in the method *synchRun*. Special methods *waitTime*, *waitNextPeriod*, *waitNextNPeriods* and *waitNextNRandomPeriods* can be used to block the *synchRun* execution until *SynchGlobal* unblocks the *SynchThread*. A set of instructions defined in the *synchRun* method and between two waits execute when the nearest precedent wait instruction is released by *SynchGlobal*.

- ***SynchTime*** defines a time, which is used to determine the end of a simulation or a specific simulation time to suspend the execution of a *SynchThread*.
- ***SynchGlobal*** controls the synchronization itself. It has a global clock and it is responsible to unblock *SynchThreads* when it is time to run next instructions (release *SynchThreads* from their time barriers), always verifying the priorities of *SynchThreads* to guarantee casual precedence.
- ***SynchLogger*** is an interface that allows programmers to develop a global logger. All *SynchThreads* controlled by the same *SynchGlobal* has access to this global logger if it is passed as a parameter in the *SynchGlobal* constructor. Methods *logInit* (called in the start of simulation), *logFinalize* (called in the end of simulation) and *log* (to receive log events) must be implemented. It is not mandatory that a *SynchThread* has a logger.
- ***SynchRandom*** is a class used as a global parameter that can be passed in the *SynchGlobal* constructor. It creates random numbers by demand and, as *SynchLogger*, it is shared globally by *SynchThreads*. A seed can be used to allow repeated simulations.

The major design rule that must be followed is not to create loops in the *synchRun* method without a wait instruction inside it. If this case occurs, a *deadlock* state will be created and all *SynchThreads* will stop to execute.

To illustrate a simple usage of the classes from the package, Fig. 3 shows a definition of a class namely *SynchWrite*, which only prints in the standard output its name and a counter (that increments every period).

```
public class SynchWrite extends SynchThread {
     private int counter;
     public SynchWrite( String name, SynchPeriod synchPeriod,
                        int synchPriority ) {
          super( name, synchPeriod, synchPriority );
     }
     public void synchInit() {
       this.counter = 0;
     }
     public void synchRun() {
          while ( true ) {
               this.counter++;
               System.out.println( super.getName() +
                                   " [counter=" +
                                   this.counter + "]" );
               super.waitNextPeriod();
          }
     }
    public void synchFinalize() {
    }
}
```

Figure 3. *SynchWrite*: an example of class that extends *SynchThread*.

```
1 public static void main( String args[] ) {
2    SynchGlobal synchGlobal = new SynchGlobal();
3    synchGlobal.add( new SynchWrite( "A",
            new SynchPeriod( "1.0", SynchPeriod.PERIOD_S ), 2 ) );
4    synchGlobal.add( new SynchWrite( "B",
            new SynchPeriod( "0.5", SynchPeriod.PERIOD_S ), 0 ) );
5    synchGlobal.add( new SynchWrite( "C",
            new SynchPeriod( "1.5", SynchPeriod.PERIOD_S ), 1 ) );
6    synchGlobal.add( new SynchWrite( "D",
            new SynchPeriod( "2.0", SynchPeriod.PERIOD_S ), 3 ) );
7    synchGlobal.setSimulationTerminateTime(
            new SynchTime( "0:0:4.000" ) );
8    synchGlobal.start();
9 }
```

Figure 4. Configuring and starting the simulation. *SynchWrite* objects are created with periods and priorities defined in Fig. 1.

Fig. 4 shows a *main* method that constructs four *SynchWrite* objects with names, periods and priorities equals to the example of Fig. 1.

It is important to notice that the period (or frequency) is defined in the *SynchPeriod* constructor as a *String* and not as a *float* or *double*. This is a special feature of all classes that manipulates time in the synchronization package: all use IEEE 754R standard to store time. The implementation, based on the *BigDecimal* class from *java.math* package, guarantees 34 decimal digits to numbers that represent time in seconds.

The *start* method (used at line 8 of Fig. 4) from *SynchGlobal* class initiates the simulation. Once initiated, there are three ways to finalize a simulation: by simulation time and/or by execution time, or by force. The simulation time corresponds to the current time of the simulation, while the execution time corresponds to the time that simulation is running (processing time). The method *forceTerminate* from the *SynchGlobal* class can be used in a Graphic User Interface (GUI), e.g., to allow users terminate a simulation when required.

IV. Examples of Simulators Implemented

The proposed synchronization mechanism can be used in a sort of non real-time applications, but especially as the core of simulators. This section presents two simulators implemented, demonstrating how the java package can be used in the design of different simulators.

A logic circuit simulator that simulates a serial-to-parallel converter and a Wireless Sensor Network Simulator (a more complex example) are explained in the next sub-items.

A. *Logic Circuit Simulator*

Logic circuits have blocks that do not respond immediately after an input signal change. In order to simulate logic circuits, these intrinsic delays must be included in the model.

Thus, a D flyp-flop, e.g., has its proper delay between it receives a signal in the D port and put it out in the Q port. Clocks and input signals did not change their values instantly either. These logic elements can be described in classes that extend *SynchThread*.

A simple Logic Circuit Simulator was designed with some basic logic elements (flip-flops, AND, OR and NOT gates, clocks and buses) to test the feasibility of this kind of simulator in the synchronization package developed. Fig. 5 shows a graphic representation of a serial-to-parallel converter that was described in Java language, while Fig. 6 shows the waveform created by a specific logger that implements the *SynchLogger* interface.

This Logic Circuit Simulator can be improved and used as a didactical instrument to teach students how to describe logical elements in details. A graphical interface can be designed to accomplish a full Logic Circuit Simulator.

B. *Wireless Sensor Network Simulator*

A Wireless Sensor Network is a collection of thousands (even millions) sensor nodes that self-organize to perform a collaborative sense of some environment. These nodes have restricted energy supply to accomplish the task of deliver their sensing data to one or more base stations. Commonly, such delivery is done using a multi-hop strategy, using intermediary sensor nodes to create a path to a base station.

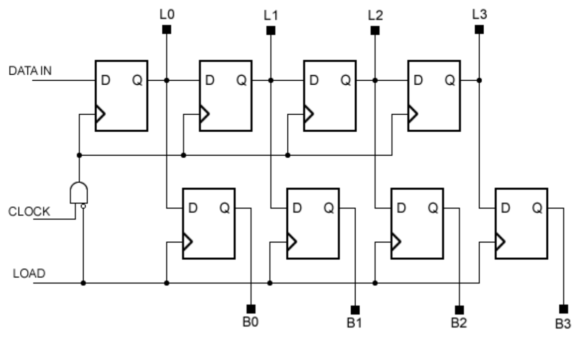

Figure 5. The serial-to-parallel circuit used to test the Logic Circuit Simulator

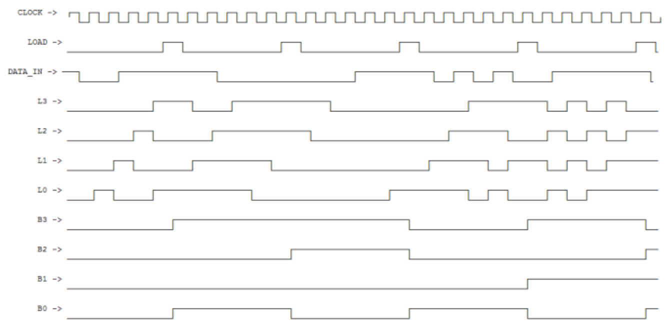

Figure 6. The waveform generated by the Logic Circuit Simulator when simulating the serial-to-parallel circuit of Fig. 5.

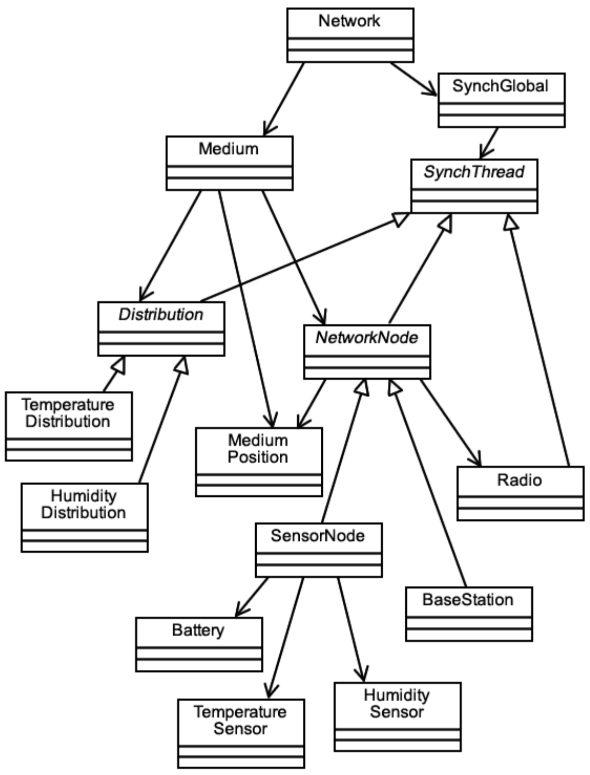

Figure 7. Wireless Sensor Network Simulator class diagram.

The protocol that coordinates the communication among sensor nodes and base stations (usually in a multi-hop way) must be efficient to save energy from nodes and guarantee the longevity of the entire network. The development of energy-efficient protocols is the major goal of recent researches in WSN.

The problems in the development and validation of WSN protocols *in situ* are related to the cost of production of a large number of sensor nodes and their deployment in some environment, which usually are places of difficult access. A simulator capable to describe in details both the network itself and the environment variables allows the developers to evaluate the energy performance of different communication protocols.

A set of Java classes was designed based on the synchronization package to construct a WSN Simulator. Fig. 7 illustrates a simplified class diagram of the WSN Simulator. The class structure represents a network with sensor nodes that measure temperature and humidity.

Distribution, *NetworkNode* and *Radio* (and their subclasses) are classes that extends *SynchThread* and have their periods established in accordance with their operation frequencies in real components: instances of *NetworkNode* (*SensorNode* and *NetworkNode*) represents devices with microcontrollers that operates in a determined frequency; instances of *Radio* also have specifics operation frequencies; and *TemperatureDistribution* and *HumidityDistribution* are responsible to simulate temperature and humidity (the environment variables), respectively, having their periods much longer than other *SynchThreads* objects.

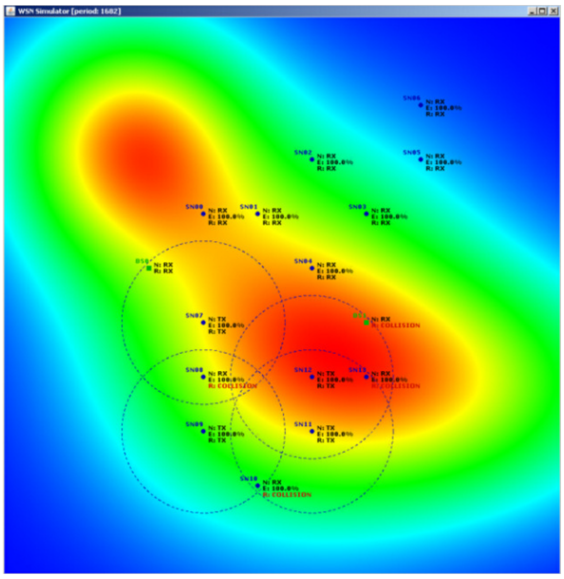
WSN Simulator [period: 1682]
SN06 N: RX E: 100.0% R: RX
SN02 N: RX E: 100.0% R: RX
SN05 N: RX E: 100.0% R: RX
SN00 N: RX E: 100.0% R: RX
SN01 N: RX E: 100.0% R: RX
SN03 N: RX E: 100.0% R: RX
BS0 N: RX R: RX
SN04 N: RX E: 100.0% R: RX
SN07 N: TX E: 100.0% R: TX
BS1 N: RX R: COLLISION
SN08 N: RX E: 100.0% R: COLLISION
SN12 N: TX E: 100.0% R: TX
SN13 N: RX E: 100.0% R: COLLISION
SN09 N: TX E: 100.0% R: TX
SN11 N: TX E: 100.0% R: TX
SN10 N: RX E: 100.0% R: COLLISION

Figure 8. A simulation in progress with 14 sensor nodes and 2 base stations (sink nodes) in the Wireless Sensor Network Simulator.

The implementation of the *Radio* class is the one that requires more attention because two cases must be treated to represent real situations: (1) at the same time that a bit is being sent by a *Radio*, the receivers must read this bits (as they work at the same frequency) and (2) more than one *Radio* can transmit at the same time, creating collision situations in some areas. The priority scheme adopted in the *SynchThread* abstract class can be used to solve both cases.

A virtual buffer was created to emulate the receiving of a bit. Thereby, at the same frequency the transmitter puts a bit in the receiver buffer, the receiver reads its buffer and clears it. When a *Radio* is transmitting, its priority must be higher than *Radios* that are receiving. Thus, the *Medium* puts a bit in the destination buffers before receivers read the value (solution for (1)) and as all *Radios* that are transmitting are set with higher priorities, if a receiver buffer already has a bit when some *Radio* is propagating a bit, the collision can be detected, solving (2).

Fig. 8 shows a screen capture of the WSN Simulator interface. The user of the WSN Simulator can view the simulation progress with the coverage area of radios (dotted circles), state of nodes (off, transmitting, receiving, processing or sleeping) and state of radio (off, transmitting, receiving, *sleeping*, *idle* and *collision* detected). The amount of energy available (remaining) in nodes also is shown near each sensor node.

The WSN Simulator developed uses real parameters of commercial radios and microcontrollers. This painstaking allows an accurate simulation to evaluate the energy consume in sensor nodes and analyze the performance of communication protocols.

V. Conclusions and Package Distribution

The package developed for synchronous simulation processing accomplished its purpose, allowing not only the time coordination of threads, but also the guarantee of a specific sequence of execution in instructions that represents parallel actions in real world.

The portability of Java makes it an attractive language to be used in the development of simulators. The only six classes of the package encourage its learning and usage, even for simple simulations or to use it for educational purposes.

The simulation speed did not exceed the performance of any similar Java simulator, but a more accurate analysis, which is out of the scope of this paper, must be done to compare the performance of packages and their features.

Both the Logic Circuit Simulator and the more complex Wireless Sensor Network Simulator have demonstrated flexibility in hardware description and plausible speed to perform simulations with a large number of parallel executing elements.

The *br.eng.rsalustiano.synchronization* package described in this paper can be downloaded for free at the site: http://www.lpm.fee.unicamp.br/~rsalusti/synchronization. An Application Programming Interface (API) created by JAVADOC can be accessed at the same link, as well as more details on the package usage.

References

[1] R. McNab and F. W. Howel, "Using Java for Discrete Event Simulation" in Proceedings of 25th UK Computer and Telecommunications Performance Engineering Workshop (UKPEW), University of Edinburgh, pp. 219-228, 1996.

[2] L. F. W. Goes et al., "JSDESLib: a library for the development of discrete-event simulation tools of parallel systems" in Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, 6pp., April 2005.

[3] SimPack web page. < http://www.cise.ufl.edu/~fishwick/simpack >.

[4] Y. Jung et al., "simCore: an event-driven simulation framework for performance evaluation of computer systems" in Proceedings of the 8th International Symposium on Modeling and Simulatino of Computer and Telecommunication Systems, pp. 274-280, 2000.

[5] SIM++ web page. <http://www.simplusplus.com>.

[6] System C web page. <http://www.systemc.org>.

[7] E. Niewiadomska-Szynkiewicz et al., "Application of a Java-based framework to parallel simulation of large-scale systems," Internationa Journal of Applied Mathematics and Computer Science, 2003, Vol. 13, No. 4, pp. 537-547.

[8] R. M. Fujimoto, "Parallel and distributed simulation systems," A Wiley-Interscience Publication, New York, USA, 2000, 300p.

[9] P. Bertels and D. Stroobandt, "Java and the power of multi-core processing" in the Proceedings of the 2008 Conference on Complex Intelligent and Software Intesive Systems, pp. 627-631, 2008.

[10] Java Platform API Specification, Standard Edition 6. <http://java.sun.com/javase/6/docs/api/>.

[11] H. M. Deitel and P. J. Deitel, "Java: how to program," 3rd edition, Prentice-Hall, New Jersey, USA, 1999, 1355p.