硕士学位论文

Dissertation for Master's Degree

(工程硕士)

(Master of Engineering)

恶意代码行为提取及分类系统

Implementation of a neural network simulator

Implementation of a coupling software for third parties utility tools

王维

哈爾濱工業大學

Université Blaise Pascal

2014 年 9 月

国内图书分类号：TP311 学校代码：10213
国际图书分类号：681 密级：公开

工程硕士学位论文

Dissertation for the Master's Degree in Engineering

(工程硕士)

(Master of Engineering)

恶意代码行为提取及分类系统

Implementation of a neural network simulator

Implementation of a coupling software for third parties utility tools

硕 士 研 究 生：你的姓名

导 师：HIT 导师姓名、职称

副 导 师：UBP 导师姓名、职称

实 习 单 位 导 师：实习单位导师姓名、职称

申请学位	:	工程硕士
学科	:	软件工程
所在单位	:	软件学院
答辩日期	:	2014 年 9 月
授予学位单位	:	哈尔滨工业大学

Classified Index: TP311
U.D.C: 681

Dissertation for the Master's Degree in Engineering

Coupling between a neural network simulator and a software of statistical analysis of synchronization

Candidate： Cyrille Mascart
Supervisor： Tonghua Su
Associate Supervisor: David R.C. Hill
Industrial Supervisor: Alexandre Muzy
Academic Degree Applied for： Master of Engineering
Speciality： Software Engineering
Affiliation： School of Software
Date of Defence： September, 2016
Degree-Conferring-Institution： Harbin Institute of Technology

摘　要

在神经元网络的同步样式的发现上导致根据单一的事件方法和 Hawkes 模型学习此样式，主要的不同的方法的发展。但是这些方法被限制了到对的研究神经元，因而另一个方法被开发了，命名 LASSO。模拟器在 R 被实施为了测试方法，但是局限固有对语言限制了模仿到小网络。使用平行的 DEVS 形式主义的实施，另一个队同时发展了 Integrate 和火神经元网络模拟器。使用这平行的 DEVS 形式主义实施，项目然后被创造实施一台更加快速的模拟器。题目的范围然后增加为了包括结合分析的工具的软件的创作实施套索方法和模拟器。想法然后被推断了，以便模拟器可能结合任何软件，当跑使用实时时，因此它可以连接与物理工具。在 RT-DEVS 形式主义之后，实时机械被实施。

关键词：Discrete event system specification; JAVA; Hawkes; Real time; Neural networks; Stochastic simulation; Multithreading

The discovery of synchronization patterns in neuron networks led to the development of different method to study this patterns, mainly based on the Unitary Events method and Hawkes models. These method were limited to the study of pairs of neurons nonetheless, thus another method was developed, named LASSO. A simulator was implemented in R in order to test the method, but the limitations inherent to the language limited the simulations to small networks. Concurrently another team has developed a simulator of Integrate and Fire neuron networks using an implementation of the Parallel DEVS formalism. A project was then created to implement a much faster simulator using this Parallel DEVS formalism implementation. The scope of the topic was then increased in order to include the creation of a software coupling the analysing tool implementing the LASSO method and the simulator. The idea was then generalized so that the simulator can couple any kind of software while running using real time, so that it can be interfaced with physical tools. The real-time machinery is implemented following the RT-DEVS formalism.

Keywords: Discrete event system specification; JAVA; Hawkes; Real time; Neural networks; Stochastic simulation; Multithreading

目　录

摘　要 I

CHAPTER 1 INTRODUCTION 1

1.1 BACKGROUND 1
1.2 PROJECT PURPOSE 2
1.3 THE STATUS OF RELATED RESEARCH 3
1.3.1 Parallel DEVS 3
1.3.2 Neural networks models 3
1.4 MAIN CONTENT AND ORGANIZATION OF THE THESIS 4

CHAPTER 2 SYSTEM REQUIREMENT ANALYSIS 5

2.1 THE GOAL OF THE SYSTEM 5
2.2 THE FUNCTIONAL REQUIREMENTS 6
2.2.1 THE HAWKES SIMULATOR 6
2.2.2 THE COUPLING SOFTWARE 7
2.3 THE UNFUNCTIONAL REQUIREMENTS 8
2.4 BRIEF SUMMARY 9

CHAPTER 3 SYSTEM DESIGN 11

3.1 TRANSCRIPTION AND INTERFACING: FROM R TO JAVA 11
3.2 STORING PIECEWISE CONSTANT FUNCTIONS 12
3.3 HAWKES PROCESSES 14
3.3.1 ORIGINAL METHOD 14
3.3.2 NEW ALGORITHM 15
3.4 DISCRETE EVENT SYSTEM SPECIFICATION 17
3.4.1 ATOMIC AND COUPLED MODELS 17
3.4.2 COORDINATOR AND ATOMMICSIMULATORACT 20
3.4.3 SCHEDULER 21
3.5 MULTITHREADING AND PARALLEL DEVS 22
3.6 REPRODUCIBILITY 23

3.7 THE COUPLING SOFTWARE 25
3.7.2 DESIGN 25
3.7.2 REAL TIME DEVS 26
3.7.3 PROCESSBUILDER AND PROCESSES 26
3.7.3.1 PROCESS OBJECT 27
3.7.3.2 PROCESSBUILDER 27
3.7.3.3 LIMITATIONS 28
3.8 CALCULATION ACCURACY 28
3.9 VIRTUAL FILES 29
3.10 PARAMETERS AND OUTPUT 31
3.10.1 THE GRAPH FILE 31
3.10.2 TO FILE OR NOT TO FILE 32
3.11 SCRIPTING 33
3.12 BRIEF SUMMARY 34
CHAPTER 4 SYSTEM IMPLEMENTATION AND TESTING 36
4.1 THE ENVIRONMENT OF SYSTEM IMPLEMENTATION 36
4.2 KEY PROGRAM FLOW CHARTS 36
4.2.1 SIMULATOR FLOW CHARTS 36
4.2.2 COUPLING SOFTWARE FLOW CHARTS 38
4.3 KEY INTERFACES OF THE SOFTWARE SYSTEM 39
4.4 SYSTEM TESTING 41
4.5 BRIEF SUMMARY 45
CONCLUSION 46
REFERENCES 47
STATEMENT OF ORIGINALITY AND LETTER OF AUTHORIZATION 49
ACKNOWLEDGEMENT 50
RESUME 51

Chapter 1 Introduction

1.1 Background

Back in the 80's, biologists have discovered a synchronization phenomenon among certain neurons submitted to specific stimulus. This experimental discovery led to development and usage of synchronization detection methods, first the Unitary Events (UE) method [1], which was later optimized, given birth to MTGAUE (Multiple Tests based on a Gaussian Approximation of the Unitary Events Method) and Permutation UE [2]. If this methods were real breakthrough at their time, they had been built to detect synchronizations of pairs of neurons. A more precise method for detecting neuron assembly activity was necessary for understanding the synchronization phenomenon.

A new method was proposed at Jean-Alexandre Dieudonné Laboratory (LJAD), based on the LASSO method [3] and applied to Hawkes models. It has good performances, even on short time data recordings, and allows to draw easily interpretable functional connectivity graphs. The method has been tested on both biological and simulated data, in collaboration with RNRP team from Paris 6. It has resulted in the development of an assembly detection software, written in C++, parallelized and interfaced with a Hawkes model of neural networks simulator written in R.

In parallel, a collaboration between LJAD and Computer science, Signals and Systems of Sophia-Antipolis (I3S) Laboratory has led to the elaboration of a neural activity simulation, based on an integrate-and-fire model, and developed with a local version of the DEVS-Java implementation of the Discrete Event System Specification (DEVS) formalism [4]. Simulation data are important for model validation, yet the project is involved in determining and validating hypotheses on the generation of neuronal structures linked with neural activity. A parallel stochastic simulator was proposed [5]. Currently coupling the simulator and the assembly detection method remains to be done.

1.2 Project purpose

The model developed at LJAD has been tested and validated within a large parameter range, but when strong interactions are involved, it creates fictive interactions between neurons. In order to study further the phenomenon, and to know whenever the reconstructed interaction graphs match biological data, more complex networks are needed to be simulated, which the currently used simulator struggles with. A first part of the project is so to implement the simulator in a much efficient way so that the strength of the model can be validated further.

Such a software will help positing new mathematical hypotheses based on the exploration, the construction and the study of the dynamics of different neuron models. These models will also be needed to be tested, their results analysed and compared with the results of previous models. With this idea in mind a second part of the project will consist in writing a coupling software that could concurrently launch the execution of several simulations while processing the results with analysing tools.

Scientific tools are written using a wide range of programming languages, from R to C, including Python, PHP and Java. The software described above aims at coupling all the possible software together. The coupling part will be based on the ability to launch third parties software, whatever the language used to implement them, instead of implementing legacy as it is often the case for libraries.

However, a particular model will be implemented, which may not be suitable for other applications. This is why the reusability of the utility tool created so will be a major requirement.

Finally, most of the simulations that are used by scientists are created or parameterized using experimental data. These data are gathered using sensors or other kind of tools that can be interfaced with software. The final part of the project aims at making the coupling software able to interface sensors and physical tools with any software, being a simulator, an analysing tool, etc.

To conclude, the goal of this topic is to translate a simulator source code from a language to another in order to improve efficiency while maintaining reproducibility and accuracy. A new coupling software will then be implemented to automate the simulation and validation processes, while constituting an interface for physical sensors with simulators and analysing tools.

1.3 The status of related research

1.3.1 Parallel DEVS

DEVS is a modelling and simulation formalism that has been implemented in different ways by different teams around the word. This is therefore a local version of DEVS, although it has been developed in collaboration with Bernard P. Ziegler, the inventor of DEVS.

A particular concept that has been used and developed so far is named Parallel DEVS. It is an extension of classical DEVS that is aimed at dealing with concurrent events and at being executed on machines with high parallelization capacities. A neural network simulator based on the Integrate and Fire model has been implemented using this version.

The model Integrate and Fire is based on the idea that neurons are units that can store energy from other neurons, energy that is emitted upon spikes; but when a neuron's stored intensity meets a certain threshold, then it releases it all to all the neighbourhood – that is to say to all the neurons it is connected with, included itself. This model is close to the reality of a neuron, which stores energy and then releases it during what is called a spike (it is also said that the neuron fires) after a certain threshold is met.

The simulator developed at I3S Laboratory implements this model using the DEVS formalism. A neuron is represented in this formalism as an entity named Atomic model. The graph is generated using the Erdős–Rényi binomial model: each pair of vertices is connected with identical probability p. The graph generated so is then connected to two optional layers: an input layer and an output layer.

1.3.2 Neural networks models

Models of neuronal networks are most of the time an over simplified form compared to what they are in reality. Precise model are too complex to be ran on current computers, even though in 2015, a team of researchers working on the Blue Brain project succeeded in simulating a tiny part of a rat brain, named the cortical column.[6]

1.4 Main content and organization of the thesis

The topic is divided in two parts, first implementing a much faster solution for the Hawkes model simulator written in R than the one proposed by the creator. Indeed, the slowness of the simulator, inherent to the R script machine running it, makes it difficult to simulate, and thus testing model validity with big neuronal assemblies – more than ten neurons. Ten neurons is the number of neurons that biologists working with LJAD can usually experiment on, that is to say measure the activity quite precisely.

The new simulator can be written in any language, but a Java version of the DEVS paradigm is being developed at I3S Laboratory, and is used to implement neural network simulators, including the integrate-and-fire model network simulator. Thus, feasibility study has to be conducted in order to know whether the new simulator can be written using this library or not. A special care will be given to computing accuracy, reproducibility while taking into account the code reusability offered by using the already written integrate-and-fire simulator.

After the new Hawkes simulator has been implemented, the coupling part has to be implemented. For now, as the assembly detection software has been implemented in C++, it has been interfaced with ease with the Hawkes simulator, which is written using R scripting language. The DEVS-Java library has been proposed for the Hawkes simulator, and may be suitable for the coupling software too, all the more so as a real-time part of the library is wanted to be developed. The concept has been expanded on as part of the DEVS formalism, but yet remains unimplemented in the current version of the library. Implementing it would allow easy coupling between the two simulators mentioned above, but also with other simulators developed later. As the original concept is defined for general simulation purposes in DEVS formalism, it may finally be integrated in the already developed DEVS library if the result matches the specifications as defined by the formalism.

To conclude, the goal of this topic is to translate a simulator source code from a language to another in order to improve efficiency while maintaining reproducibility and accuracy. A new coupling software will then be implemented to automate simulation and verification process. Real-time DEVS specifications for the second software, and DEVS specification for both may be used or not depending on feasibility.

Chapter 2 System Requirement Analysis

2.1 The goal of the system

This internship condenses two very different project. First, a simulator, implementing the LASSO introduced earlier has to be implemented – actually reimplemented – in order to increase execution speed. The new simulator will be inspired by a previous one, written in R while implementing a different algorithm in a different programming language.

Then a coupling tool will be developed. It is intended to automate the data transfer from the new simulator and an already existing tool for the statistical analysis of synchronizations. The goal is to facilitate both the exploration and the construction of models.

2.1.1 Simulation part

Given an interaction graph, the system has to simulate the neural network whose interactions are described by the graph and where neurons are Hawkes processes. The results are outputted in files, and are values of state variables at event occurrences during the simulation. Is considered an event the emission of spikes by neurons. The recorded variables are the time when an event occurred (relatively to the beginning time of the simulation) and the integral of the intensity over time (from the beginning of the simulation to the time of a spike).

The interaction graph may be written in a text file or generated by the software, given the number of neurons and the probability of interaction between two neurons. The last functionality was not explicitly required, although it is more convenient for big networks simulation.

2.1.2 Coupling part

The simulation results are analysed using a tool implementing the LASSO method, implemented in C++ and highly parallelized. Interfacing the new simulator and this tool is the main goal of the second part of the project.

This coupling software however has to fulfil more requirements than just the coupling. It will indeed be a part of the locally developed DEVS library, as an implementation of Real Time DEVS (RT-DEVS). Therefore it will have common real time system functionalities, that is to say checking for deadline overtaking and possible interfacing with physical tools, such as sensors or machines. Unlike many real time systems, overdue processes are not to be aborted, but will be mentioned as "due" for the user to know.

To conclude, there are two tools to be developed during the internship. The first one is a simulation tool for networks based on Hawkes models. It takes a graph or graph parameters as input and outputs state variables values from different simulation states. The second is a coupling software that handles data transfer and checks execution time between other software, mainly simulators.

2.2 The functional requirements

2.2.1 The Hawkes simulator

The functional requirements for the first tool are mainly given by the already written and yet to translate Hawkes simulator. The simulator takes as argument the complete interaction graph of the neurons – meaning interactions between all neurons have to be described, even if the interaction is null -, the intensity of the neurons at the beginning of the simulation, the simulation beginning and ending time and a warming up time delay. This "delay" models the resting time let by scientists to the mice's brains between two experiments.

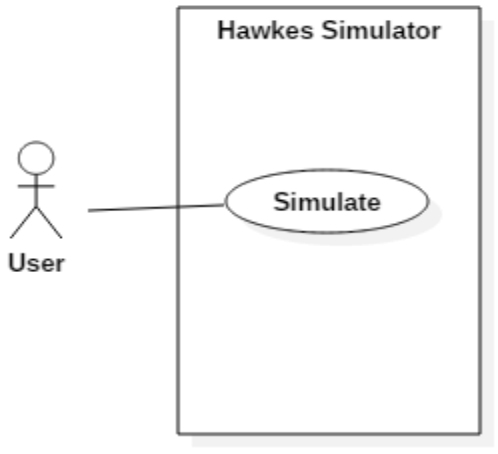

Figure 2-1 Use case of the Hawkes simulator

Simulated neurons are independent entities which are characterised by a name and an intensity. The value of the intensity determines how much spikes, on average, they may fire in a second. A spike is the data sent by a neuron to its neighbourhood, and can be exciting – modelled as a positive value – or negative – negative value. The values exchanged between neurons are defined by the interaction graph – the null values standings for no connection. The values are actually no single numbers, but piecewise constant functions. After any spiking, the next neuron to spike and the time before the event is computed, taking into account the new values of intensity of all neurons. The simulation whenever a spike time would reach the simulation ending time. The use case diagram of the simulator is shown on Figure 2-1. It is pretty simple, as the simulator only functionality is to simulate: data and configuration files are inputted, the simulation is run and results (spiking times and integrals of the intensity over time) are outputted.

2.2.2 The coupling software

The second tool, as mentioned before, is a real time coupling software, based on the requirements described by the RT-DEVS formalism. This means that the software to couple, for instance the Hawkes simulator (P1) and the analysis tool (P2), will be implemented separately from the coupling software, and called by command line. The class managing the calls is named *Atomic Model*, and these are linked within a class called *Coupled Model*. These classes can be overridden by users or their default behaviour can be used. In the second case, a file containing the interaction graph of the software (for instance, P1->P2) and the command lines to be called to execute the

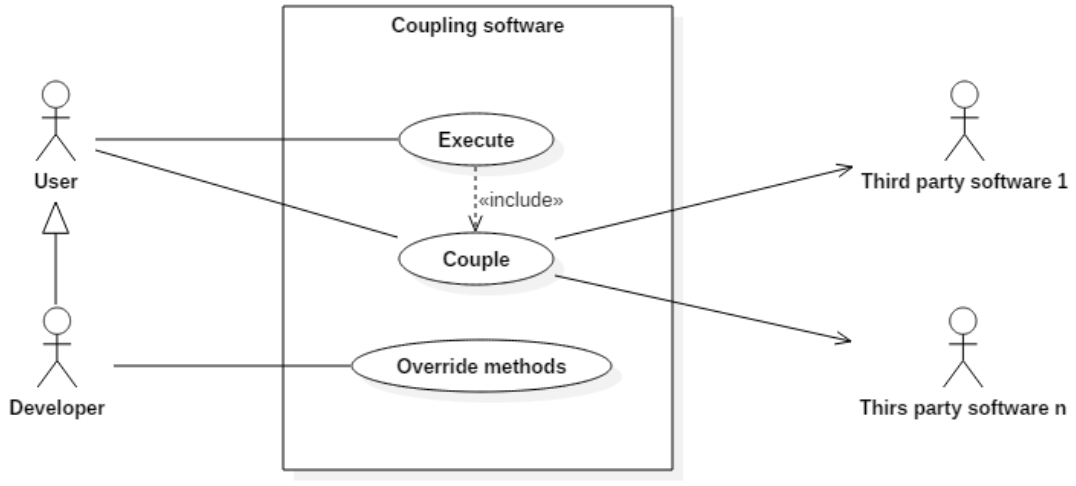

Figure 2-2 Use case of the coupling software

software. The coupled software have to take, if using the default behaviour of the coupling system, one or more file name for argument, and output one or more file name.

The second tool being a real time coupling software, minimum and maximum execution time are given so that the software are verified not to give their output before the smaller bound and are declared "due" if execution time exceeds the higher bound. The overdue software are not stopped when "due" as they may be developed from other people than the ones using the coupling software. This is particularly true if it is interfaced with drivers running sensors: drivers are indeed often proprietary software, which source code cannot be accessed to or changed, making it difficult for users to control the execution speed.

A requirement which is less a functionality than a demand is to make a comparison between the Hawkes model and the Integrate and Fire model is asked by the teams working on the project. As the Integrate and Fire simulator has been written with different requirements in mind, it will be necessary to modify the source code in order to make the comparison possible (state variables values are not stored at that moment).

The use case of the coupling software is displayed at Figure 2-2, and shows three main functionalities: executing, coupling and overriding. First, a user can execute the application, though a coupling between different third party software is needed if the user wants the software to couple something. Then as the implementation concerns as much a software implementation as a formalism implementation, a developer wanting to use the software but with a different behaviour can override certain methods of the source code to change the behaviour.

This last requirement is at the limit between a functional and an unfunctional requirement, as it concerns more the source code than what does the software offers to the user to do. But it also means the machinery has to be generic enough in what it can takes as argument so that a developer does not have to reimplement the entire system to change its behaviour.

2.3 The unfunctional requirements

The unfunctional requirements mainly focus on efficiency for one of the two parts, reusability for the second and accuracy for both. The simulation models used by the two teams have been discussed, as long as the possibility to couple them both.

As the final software, coupling simulators and analysing tool, is intended for bioinformatics engineers and biologists in general, informal requirements would focus on user-friendliness of the software.

The unfunctional requirements are all clear, complete, consistent and unambiguous. Efficiency is easily measured by total execution time and total number of neurons modelled. A single indicator, such as:

$$\frac{\text{total\_execution\_time}}{\text{total\_number\_of\_neurons}}$$

is suitable for the case.

Reusability is driven by either the reuse of the existing simulation code or by direct integration of the new Hawkes simulator – and other potential simulators, deriving or not from it - within the existing integrate-and-fire simulator. This is very interesting as the DEVS-Java implementation proposes a graphical view for users, named SimView, fitting well with the informal specification.

Worth noting that the project source code will at the end be publicly available, thus increasing even more the need to emphasize on the reusability of the software source code, as well as maintainability and the use of good coding practices.

Finally, accuracy will be achieved if the new Hawkes simulator gives similar quality results as the original one. As the simulation relies heavily on randomness, only statistical accuracy is required. This can be tested using the assembly detection software which is believed to give accurate results. Nonetheless, special care is to be given to the random number generator and computation results, in first case for reproducibility purposes (same seed, same result) and in second case checking rounding errors are avoided as much as possible – that is to say inexistent.

2.4 Brief summary

The final system will help researchers to simulate and check the validity of mathematical models of neural networks. This is achieved first by implementing a simulator software that simulates Hawkes model of neural networks, and which speed is high than the one already written in R. The simulator will meet scientific computation accuracy and will reuse code from the existing Integrate and Fire simulator while being reusable as source code for future simulators. Secondly, a tool

aimed at coupling software will be written by implementing first the RT-DEVS formalism, then using the new library part.

Chapter 3 System Design

The systems design was largely guided by the boundaries defined by the DEVS formalism, as the forst part of the project consisted in translating a program from R to Java, using a DEVS-Java library; and the second part consisted in implementing the Real-Time DEVS formalism in Java.

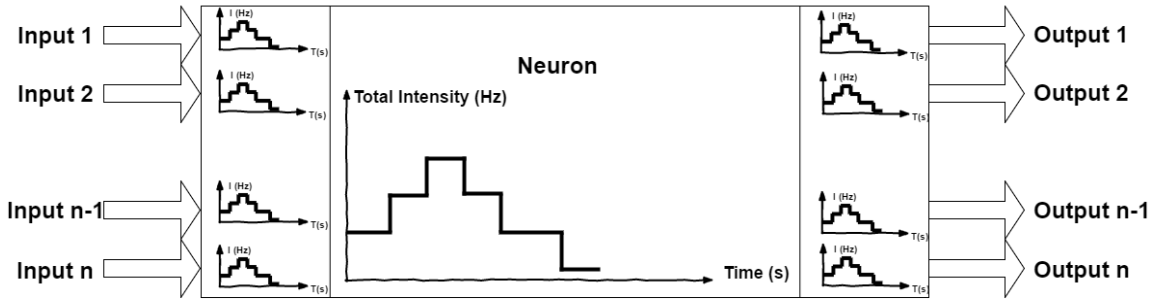

Figure 3-1 A neuron: Hawkes model

3.1 Transcription and interfacing: from R to Java

R is a scripting language, aiming at facilitate mathematical and statistical computations by providing a simple solution for matrix calculus. The machine running R scripts is built so that it can easily call C++ methods. R functions are actually often written in C++. The R language can be viewed as a high level interface for C++ vectorial calculation.

At the opposite Java is an object oriented programming language, hybrid between a compiled language and a scripting language. It is aimed at fast prototyping, but is also used as a developing language by a large community of programmers. The user friendliness of the language makes it a good choice for libraries addressed to non-computer scientists.

The two languages are very different. In addition, there are no existing library that helps interfacing an R and a Java program. This means that an algorithm written in one of the two languages has to be redesigned with both the new language possibilities and limitations in mind.

Generally, the Java language offers much more possibilities than the R language, while being naturally much faster. This allowed for neurons to be considered as independent entities from one another, defining them as distinct objects, while in the

R simulation they were abstracted as line numbers of a matrix. This allows for object/personalized definition of neurons, even multiple definition of neurons in the same simulation, thus enlarging the number of possible models.

Java is also famous for the easiness of Graphical User Interface implementations. This allows live representation of changes and communications in the system during simulation. This functionality is used in the Integrate and Fire simulation, which uses the Jung library to represent the simulated network, and available in the original DEVS library as SimView, a graphical interface for visual programming. The graphical interfacing with users is not used much in the simulation nor in the coupling software, even if a graphical representation of the simulated network is displayed by the simulation, using the Jung library just as does the Integrate and Fire simulation.

Despite being much faster than R, the Java programming language is bad at storing and making calculations on matrices or alike. On the other hand R offers few possibilities ofr manipulating strings of caracters. This is a key point of the software design and implementation. Another important point is how the programs can interact with the operating system and use the services an operating system provides, such as transparent memory management and networking services. Java being a lower programming language than R, it provides finer solutions for playing with this services. The difference is striking when dealing with Inputs and Outputs (IO): making a method that creates a network from a file in R is extremely difficult for there is no existing method. In such a case the best way s often to implement the method in C++ and then to use R to execute the method and use the results.

This is an exercise developers rarely face, as algorithms are most often asked to be written using one language, which is largely enough. If translating an algorithm from Java to C++ for instance is not a thoughtful task (the two languages are very similar one from another), the possibilities are much wider when changing paradigm, as it is the case from R to Java, translating from a vectorial language to an object-oriented language leads to some surprises.

3.2 Storing piecewise constant functions

On Figure 3-1, a neuron modelised as a Hawkes process is represented. There are three important parts: the input, the internal intensity and the output. The "total intensity" is the sum of the base intensity, which is constant, and the inputs. The inputs

and outputs are the signals used by neurons to interact with each other. All of these variables are piecewise constant functions of intensity over time, this is why the storage of the kind of functions is an important matter in this project.

The interaction signals are short in length and most of all fixed at the beginning of the simulation (actually they are defined in the data file). For instance, if the neuron 1 can send signals to the neuron 2, then the signal will always be the same whenever the simulation time. On the Figure 3-1, they are represented as the functions on the input and output rows. They can be either exciting – positive value - or inhibiting – negative value -, even switching over time.

The intensity is the result of the sum of these interaction signals and of a base intensity. It is longer in length than the interaction signals, and it is worth noting that at a given time T of simulation, the values of TotalIntensity(t < T) are constant, as no signals can modify the past. In addition, what is important, aside from the intensity value they represent, is the value of their integral over time. Long story short, their integral is equivalent to a certain extent to a probability, the probability of having a new event at time $t+\Delta t$ (cf. Section 3.3).

The first and easiest way to represent the intensity is to use a list whose elements are changes in the intensity. The intensity function is then recomputed whenever any new event occurs, which is time consuming for highly connected networks. The advantage of this method is that the intensity function is clearly represented, with past and future events stored in a list.

One step further in design is achieved considering past events are no longer of use for the calculations. Indeed, the two important values to be stored as event occurrences are the elapsed times since beginning of the simulation and the integral of intensity over time since the beginning. From any time t, the integral of intensity from beginning to t will remain constant, thus making it possible to be stored while deleting non useful information.

Starting from the previous idea, are there other parts of the method that are stored or computed that could not be? It is important to keep the future events as variable, as new events may occur between current time t and further time $t+\Delta t$.

Another available lever is the computation part, as it may be interesting not to build the intensity function whenever an event happens but whenever it is needed. This is achieved by storing events and source of events instead of the result of the event. The intensity and intensity integral at time t are computed on demand, and unnecessary

elements are eliminated at the same time. This approach is interesting for systems with much higher communication rate than storing rate, which is not the case here since the spiking rate (therefore storing rate) highly depends on the intensity value which depends on spikes. It is mentioned here as this solution was implemented during the development phase for a wrong version of the Hawkes process implementation.

3.3 Hawkes processes

3.3.1 Original method

Hawkes processes are mathematical abstractions of diverse point processes. [7] Here the model uses multivariate Hawkes processes, which describes the evolution of a set of points defined by their intensity function – hence the piecewise constant function representation problem described above. Points in the model interact with other points, exciting or inhibiting them, during what is called an event. The intensity value of a point at a time t determines the point "interaction rate" or "event rate" (the higher the intensity the higher the "interaction rate"). After an event occurred, the next event time is calculated by the system.

There are two ways of calculating the time to the next event, which are mathematically equivalent and only influence how the neurons are represented in the simulation paradigm. The first possibility is to pick both the time to the next event and

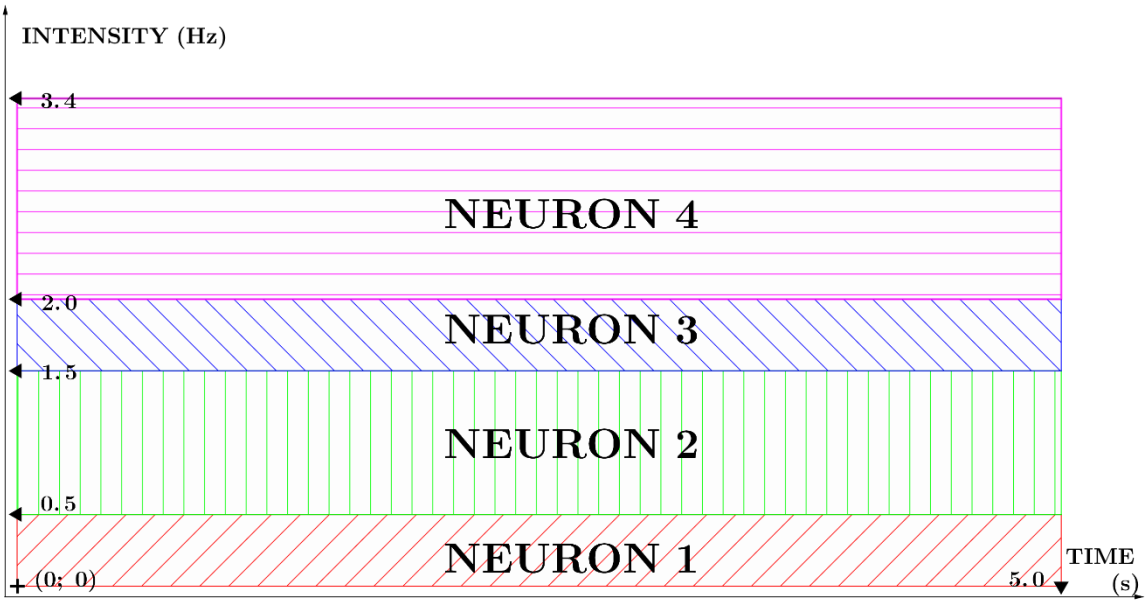

Figure 3-2 Simple intensity exemple

the neuron at the source of the event. The neurons here are considered as a set with a unique intensity – the sum of the neurons intensity – and the calculation is achieved by picking a random number used to determine the time to the next event.

The algorithm begins by picking a random floating number between 0 and 1, using a uniform random number generator. The opposite of the logarithm of this number is equivalent to an area under the total curve. For instance, if the random number is $X = 0.001113$, then the time to the next event is computed by the formula: $T_{next} = \frac{-\log_e(X)}{\sum_N \max_N(Intensity_N,0)}$, where N is a neuron. On the Figure 3-2 the result is indicated by the arrow's abscissa. The intensity of all the neurons on this figure is constant and positive for purposes of simplification. In the case where the intensity of a neuron is negative it is considered as null. The unit of time is the second and the unit of the intensity is the Hertz.

So the time to the next event is computed by picking a random number, then applying a formula. Once the time is fixed, the neuron source of the event is determined by picking another random floating number included between 0 and 1, and compare it to the normalised intensities. On Figure 3-2, the sum of the intensities is equal to 3.4 Hz. The neuron 1 is allocated the area $\frac{0.5}{3.4} \approx 0.15$, so it has 15% chance to be the source of the next event. Neuron 2 has $\frac{1}{3.4} \approx 0.30$, being 30%, etc. If the random number is 0.1, then the origin of the next event is the neuron 1. If it is 0.4, then the origin is the neuron 2, and so on.

This algorithm is the one that is used in the original R-implemented simulation, which is why the intensities are considered all together and not separately. It is indeed easier and much faster to compute using vectors of elements in R, therefore to group all the intensities in a single vector and calculate when and from which neuron the next event is going to occur.This algorithm is also used in the implemented Java simulation, but in a slightly different form as we are going to see.

3.3.2 New algorithm

So the algorithm used in the new simulator is almost the same as the one implemented for the R simulation. The main difference between the two simulations

is lies in the instanciation of the neurons: they are all comprised in the same object in the R simulation, while in the Java simulation they are considered as independent objects. Thus, for every neuron, the algorithm uses its intensity only to compute how long to wait before the next event. This is done independently and concurrently for every neuron instanciated. The neuron which will effectively provoke the next event being the one with the minimum proposed time of all.

At this point we can compare the two algorithm as for the efficiency and possibility. On one hand the first algorithm uses only two random numbers to compute two values (a time and a neuron identifying number), while the second needs as many random numbers as there are simulated neurons. This can seem as a waste of computation time which is should be avoided, but on the other hand, the problems are very different in both cases. Vectorial calculus is less efficient in Java than it is in R while being a convenient way to compute the intensity functions. Quiting the much centralised solution to embrace a more decentralised one is also important for the cohabitation of different models, for instance models that have a different view of the intensity or even no intensity value at all.

Finally, the possibility of concurrent execution is a major improvement supplied by the translation form R to DEVS-Java. It is an interesting feature when considering big networks as a single computer often doesn't have the hardware capacity to manage all the network. For instance, on a computer with a one core CPU @ 2.40 GHz and 2 GB of RAM, the simulation takes about 8 seconds for a 10 neurons network, 15 seconds for a 100 neurons networks and more than 40 minutes for 1000 neurons.

3.4 Discrete Event System Specification

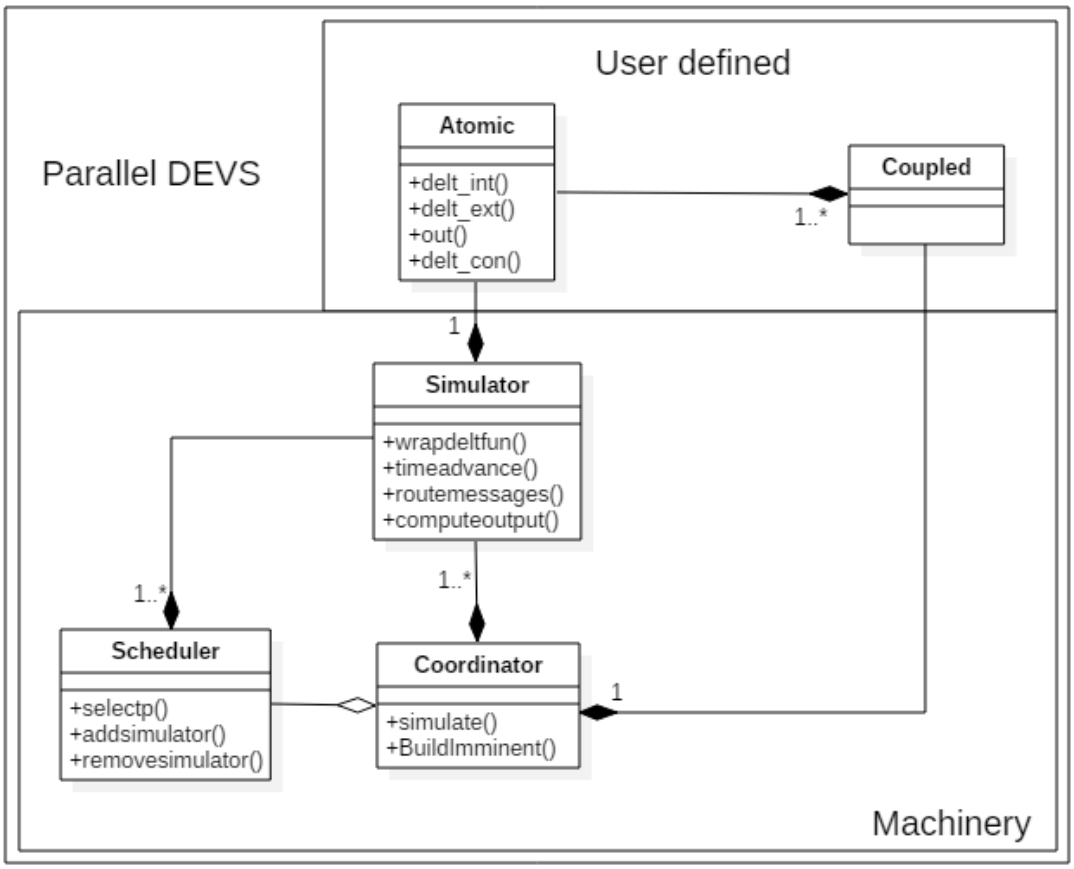

Figure 3-3 Simplified Class diagram of parallel DEVS

In this section we are going to describe the class hierarchy of a DEVS simulator. All the classes are not going to be described here, as there are many and most of them exist only for practical uses (the current code was about 30 000 lines). We are going to focus first on the two main classes a user has to struggle with (the ones implementing the expected behaviour).

The first two classes are the one the user can override, being the atomic and the coupled model. Then we are going to describe three other important classes that manage the first two, the simulator and the coordinator. Finally, the scheduler role is developed in its own part.

3.4.1 Atomic and Coupled Models

The DEVS paradigm establishes a new, hybrid simulation surrounding, in addition to the traditional discrete time and event based simulation kinds. It is used to model

and simulate systems either described by differential equations depending on continuous time or equivalent variable (continuous state systems), or systems described by state transition tables (discrete event systems). The formalism also accepts hybrid systems, described partly by differential equations and state transition tables.

The DEVS formalism sees any model as the sum of its elementary subprocesses. They can be neurons in a brain model, hardware components in a computer model, etc. Thus a DEVS-based model is highly hierarchical, which allows for concurrent execution of the subprocesses. These subprocesses are characterised by a state and a lifespan, whose values are changed upon event arrival. An event can either be internal, which means the state of an elementary process has reached his lifespan, or external, meaning another process's state has reached the allowed lifespan, and communicates it with the neighbourhood.

The general structure of the system is described in what is called a *Coupled Model*, while the inner behaviour is captured by what are called *Atomic Models*. They are both modelled as tuples, and what is inside the tuples depends on what flavour of DEVS is used. Nevertheless, all DEVS variations are based on a basic one, called *classical DEVS*. The classical *Atomic DEVS model* is defined as a 7-tuple: AM=<X, Y, S, ta, δext, δint, λ>

X, Y and S are sets of respectively input event values, output event values and sequential states (for instance "passive", "active", etc.). The three next elements describes the functionalities of an *atomic model*:

- ta is the *time advance function*. It takes no argument – actually it should take a state as argument according to the formalism – and returns the lifespan of the current state.
- δext is the *external transition function*. It takes an elapsed time from the last event occurrence and an input value as arguments, and returns nothing. This is where is defined the changes in the state of the atomic model upon external events.

```
@Override
public void deltext(double e, message x) {
Continue(e);
for (Iterator iterator = x.iterator(); iterator.hasNext();) {
    content c = (content)iterator.next();
    switch (c.getPortName()) {
        case "portName 1":
            break;
        case "PortName N":
            break;
        default:
            Logging.log("Unknown message port in deltext of
        neuron " + this.name);
            break;
    }
}
    holdIn( "phase", time("phase") );
}
```

Code 3-1 Example of external transition function

- δint is the internal transition function. It takes no argument and returns nothing. This is where is defined the changes in the state of the atomic model upon internal events.

```
@Override
public void deltint() {
    holdIn("active", time("active));
}
```

Code 3-2 Example of internal transition function

- λ, also called out, is the output function. It takes no input and outputs a set of messages, which content depends on the current state of the atomic model and is viewed as input for the external transition functions of other atomic models.

```
    @Override
    public message out() {
        message m = new message();
        m.add(new content("portName", new Entity(content)));
        return m;
    }
```

Code 3-3 Example of output function

```
@Override
public void timeAdvance(double t) {
    double oldtN = tN;
    tL = t;
    tN = tL + myModel.ta();
    if(tN < DevsInterface.INFINITY) {

        if(scheduler!=null) {
            ((Scheduler)scheduler).addSimulator(tN, this,
myModel.act());
            if(oldTN != tN)
                ((Scheduler)scheduler).removeSimulator(oldtN, this);
        }
        else
            Logging.log("atomicSimulator scheduler not set!!",
Logging.full);
    }
}
```

Code 3-4 Example of time advance function

3.4.2 Coordinator and atommicSimulatorAct

These two classes serves as the real running machine of the DEVS formalism. If the atomic models and the coupling models are what is going to be executed, the coordinator and the atomicSimulatorAct are how it is going to be executed. Each coordinator – there can be several, we are going to see that in a minute – is instantiated

by giving a coupled model to simulate. As coupled models can be viewed as atomic models (for formalism closure), coordinators are classified following a hierarchy with on top the Root coordinator and below all the other coordinators.

A coordinator is a multi-objective classes, but it only contains one much needed method: the simulate one. This function can take as argument whether a double, for a simulation time, or an integer, for a number of loops. The two functions are very much the same, except that the first one encapsulates the second one by looping over time over it.

The time progresses by fits and starts, from an event time to another event time. An event time is over when the computing of all the internal, the external and the output methods that must be computed at current event time tN is finished. It is worth noting that the method stops if all of the atomic models have their state associated with an infinite lifespan. This is obviously done to avoid simulating when all atomic models are in passive state.

With each Atomic Model is associated an atomicSimulatorAct, whose main role is to execute the state transition functions, time advance and output functions. This class also provides functions that reload the Scheduler so that previous events are erased while the new events are registered.

3.4.3 Scheduler

The scheduler is an important part of the DEVS library as it is the system managing the time for future events to come for all the atomic models. The provided methods varies from selecting the events occurring at a given time *t*, adding and removing the events. When speaking about events, one means that even if the Scheduler actually stores simulators references, it is all about the events the atomic models linked to the said simulators are going to generate or undergo.

The scheduler plays a great role as the synchronisation point of the system. It is indeed unique for all the system, which is an important point to know when dealing with parallelization, as we are going to see in a moment.

```
@Override
public synchronized void removeSimulator(double t,
coreSimulatorInterface sim) {
    for (int i = 0 ; i < 2 ; i++)
    {
        TreeMap<Double, Set<coreSimulatorInterface>> tr =
scheds.get(i);
        if(tr.containsKey(t)) {
            Set<coreSimulatorInterface> list = tr.get(t);
            list.remove(sim);
        if(tr.get(t).isEmpty()) {
            tr.remove(t);
        }
    }
}
```

Code 3-5 Example of scheduler removal function

On Code example 3-5 we can see the average structure of a Scheduler method. The TreeMap data structure contains the future events. What is stored are simulators (the mother class being coreSimulatorInterface), the back end part of atomic models.

3.5 Multithreading and Parallel DEVS

Multithreading is an advantage in event-based simulations since processes routines (and corresponding events) are independent at current time *t*, thus allowing developers to parallelize the computation of process actions. This is particularly the case with any model following the DEVS formalism, as the independent processes are naturally described by atomic models. Nonetheless a new function is needed for the formalism to manage all the cases of an event parallel simulation: the *confluent function*, noted δcon, which describes which method between the internal transition method and the external transition method is executed first when a process faces an

external and an internal event at the exact same time. A classic example is presented on Code 3-6.

The idea that was developed was to keep the scheduler as the centre of the system, which role is to keep track of the future event and their date. Meanwhile, the coordinators will continue to manage locally their part of the system, by calling runnable classes from a pool of threads for computing the independent tasks.

A consequence of such a highly parallelized system is that testing can become a highly difficult task, for the task execution order become unpredictable with a high number of threads involved. Indeed, during the internship, an error was hidden in the task managing process: whenever an atomic model had an event forecasted, the event was not destroyed if external events happened modifying the forecasted event time. This was hardly detectable in the Integrate and fire simulation, as the results are not predictable. As the Hawkes simulator had an already working version, a result comparison led to the discovery of the issue.

The time advance function, called by atomic simulators after an event occurred for an atomic model, keeps the time of the said atomic model up-to-date, while informing the Scheduler that a new event has been scheduled for this atomic model. The issue was that first, the removeSimulator simulator was not called, as the addSimulator function was expected to do the removal part. This could be expected as future events are stored using a TreeMap object, which replaces old values of an object if the given key already exists in the tree view. This error is now corrected.

```
@Override
public void deltcon(double e, message x) {
    deltint();
    deltext( 0.0, x );
}
```

Code 3-6 Example of confluent transition function

3.6 Reproducibility

Reproducibility is an essential element of simulation software, as scientific researchers seek reproducibility of their simulation experiments. Stochastic simulations achieve reproducibility by fixing the seed (or status) of their random number generators, but in parallel applications further techniques have to be

implemented to ensure reproducibility. The thread execution order is unpredictable in Java, so a single RNG for all the thread processes will give unpredictable results after each execution for the order the threads processes are picking the next number from the generator is unknown and impossible to know. Therefore, random number generators have to be instantiated in a way the randomness of thread execution order does not interfere with which activity picks up a new random number at time *t* [8]. In order to do this, each stochastic variable in an atomic model was assigned its own random number generator, independent from the others [9].

Another solution is to have only one generator but the stream is divided between the different atomic models. If there are 10 atomic models, then the generator stream will be divided in 10 non-overlapping parts. Having only one generator for all threads or one generator for each threads will give statistically equivalent results at the end as long as the period is long enough to never be reached. This may be an issue for very large simulations, although the latest random number generators have hugely long periods, and that it may be much more easier to manage reproducibility with only one or a few generators, as this means only few different seeds are needed – the output of a generator being fully determined by the seed used at initialisation, as many different seeds as there are generators involved are needed to guaranty different output from all the generators.

The generator that is being used in the system is the *MRG32k3a*, which is part of the *umontreal.iro.lecuyer.rng* package for the management of random number generator in Java. This generator has a period length of $\rho = 2^{191}$, but is divided in adjacent streams of size $Z = 2^{127}$, that are also partitioned into V substreams of size W ($V = 2^{51}$ *and* $W = 2^{76}$). This generator acts more like a virtual generator that is used to instanciate objects that will use the substreams to generate random numbers. As the period is very large, and so are the size of the different partitions, many generators can be instanciated without risking any overlapping between two generators. We can also note that the solution that have been implemented here is the one with one very large stream partitioned in many smaller streams.

The state of the generators can be described with six 32-bits long doubles, allowing the execution of several simulations with different streams for each. This is important as for statistical verifications several different runs are needed. The state of the generators can be obtained via command line arguments or serialization of the classes, depending on what does need the application.

3.7 The coupling software

3.7.1 Design

Before writing such a software the most important part was dedicated to the design. Several articles were dissected in order to understand what could be the best solution for what we had in mind. The primal idea was to implement a software that could manage several applications concurrently, mostly simulations, while being still anchored enough in the reality so that it could be interfaced with physical tools such as sensors or machines.

The main ideas were that the system would be real time and would be able to execute third parties software, meaning that some part of the system would not be modifiable by the user. As the DEVS formalism offers a good solution for easily implementing simulations as long as the formalism is followed, the idea emerged to develop a part of the DEVS formalism that was defined but not yet implemented: RT-DEVS for real time discrete event specification system.

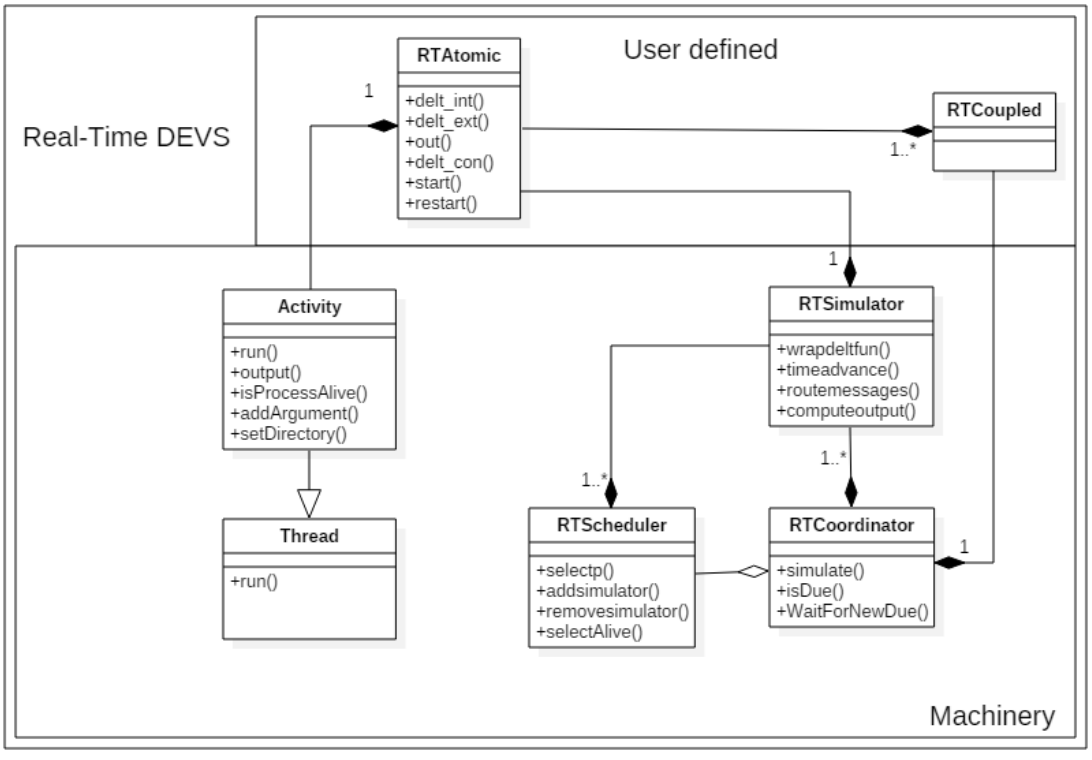

Figure 3-4 RTDEVS simple class diagram

3.7.2 Real Time DEVS

So far we have defined two different versions of the DEVS formalism. These are the versions that are the mostly used, and that were used during the development process until the last part of the project was reached. At this point it becomes so necessary to define this other formalism.

Simulations often use an abstract simulation time for internal cooking. But simulation time is different from processor time, which is different from the real world time. The last two times are very close, as processors' time is close to the nanosecond, meaning processors can manage time with a precision organic beings are unable of. Still, it is discrete, thus making it possible to create real time applications, using processor time instead of simulation-defined time.

As we can see on the Figure 3-4, Real time DEVS adds to the other classic versions of DEVS the notion of *Activity* and *Activity Mapping Function* [10]. An *Activity* is a sub process, the kind an *Atomic* Model would usually execute, which is mapped to a particular state of the *Atomic Model* thanks to an *Activity Mapping Function*. The difference between a RT-DEVS and a classic DEVS application is that the subprocesses represented by the atomic model take time to be executed, so that it is much more interesting to run the system alongside to the subprocesses. In addition, some of these subprocesses may need to be executed at a specific time in the real world, or need results of another subprocess at a very specific time.

The solution is then to launch the time consuming processes at event occurrence instead than computing them at event occurrence. Such a system will then wait for real elapse time instead than just jumping from a state to another.

The *Activity Mapping Function* is not present on the Figure 3-4 as it has not been implemented. Indeed, the Atomic Model can execute only one *Activity* at a given time, and most of the time it will always execute the same *Activity*.

3.7.3 ProcessBuilder and processes

Usually, applications are language consistent, that is to say applications are written using the same language at every part of the development process. Some languages, often scripting languages, have been interfaced with compiled languages in

order to increase the execution speed. This is the case of R for instance, which is interfaced with C++ so that C++ tools can be natively called from an R script.

In our case, what was wanted was a software that could launch any other program whatever the source language. This is a very difficult task in Java, mainly because of the Java Virtual Machine (JVM) that creates a special execution environment for any launched java program. That means there is no access to the rest of the machine from a Java point of view, except to the operating system command line system.

As Java is such a particular language from other languages point of view, there are few solutions for interfacing it with other languages as it can be seen in R or ADA with C++ and C respectively. Thus, using calls to the operating system shell is the only solution. Thankfully the base Java library provides two handy classes to manage this kind of problematic: Process and ProcessBuilder.

3.7.3.1 Process object

The Process class was introduced early in the Java library (actually it exists since SDK1.0). This class " provides methods for performing input from the process, performing output to the process, waiting for the process to complete, checking the exit status of the process, and destroying (killing) the process" (from the Oracle documentation). The new processes created so does not own their own terminal or console: the subprocess is executed in the same environment and working directory as the Java process owning the Process object associated with the subprocess, and the standard I/O are redirected to different streams which the user can look in at any time. [11]

3.7.3.2 ProcessBuilder

The ProcessBuilder was later created in order to facilitate the management of subprocesses represented by Process objects. The idea is that a subprocess will still be represented by a Process object, but instead of calling the Runtime.exec() function to create the Process instance, here the Process instance is created using ProcessBuilder.start(). The main difference relies in the possibility to define, in addition to the classical command line, an environment and a working directory for the subprocess to be executed in.

This means that the range of subprocesses that can be launched by an application using a ProcessBuilder object is very much larger, as some applications may need to access certain files that are not situated in the same place where the mother application was launched. It also provides the possibility to work with different versions of the same language or software by changing the environment variables. In addition, the standard input and outputs can also be redirected, and for more convenience the class methods throws more exceptions upon errors than the previous Process class methods.

3.7.3.3 Limitations

In addition to the limitations already mentioned for the Process class, there are a few other limitations. First it is not very secure as the subprocess will continue to execute asynchronously from the mother application if there is no reference on the Process instance. In general, there is no control over if the process represented by a Process object will execute concurrently or asynchronously with respect of the Java process that owns the Process object. Finally, a subprocess represented by a Process object may deadlock if an input is needed if the buffer is not empty. This is a major limitation as some systems natively limit the size of input and output buffers.

The class also suffers some limitations, mainly from the fact it is unsynchronized, meaning that if multiple threads want to access or modify the attributes of a given ProcessBuilder object, the access and modifying synchronization has to be set manually. However for our application this is not an issue, a subprocess is equivalent in the DEVS formalism to an activity, which means it can be accessed only by one atomic model at all.

3.8 Calculation accuracy

Perfect accuracy in usual applications is not a very important issue, and the minimal precision provided by the language is largely suitable. This is obviously not the case for scientific applications, as the higher precision as possible is needed. But what is the precision of Java on computational tasks?

Different functions were involved with calculations, but the most important concerns the integral calculation. At that time, a library was sought that would help to make exact calculations. The Java standard library provides a class object which is

called BigDecimals [12][13]. It allows making perfectly accurate calculus thanks to not using binary representation of numbers. This kind of solution is quite slow though, so even if it was implemented on, another version of the classes is used which uses doubles instead of BigDecimals. The double precision is very high is Java, and for the moment no errors in the calculations have been found, so the double format and precision is used as long as no error is found.

3.9 Virtual Files

As we have seen before, the system relies highly on input and output facilities to carry on communication between the sub processes launched by the system. Yet, most onerous of all data manipulations are the one involving the hard drives. This is one of the reason database management systems are so complicated, as they must manage input and output data so that the transfer from the Random Access Memory (RAM) to the Hard drives is optimized as much as possible. Is such a system a necessary feature in our case?

The writing process is handled by the third party software, as well as the reading process. Hence a database management system would not be a good solution in this case, as all the management part is carried out by the third party software. An idea used by database management systems and that applies here is to use in memory files. They are files managed by the operating system and that exist only in memory, for the application lifetime. Being in memory, they are of faster access time than any file in hard drive, and as they are managed by the operating system any application whatever the language it has been written can use it. The average flow for a hard drive I/O operation is proportional to 1 octet/microsecond while the value decreases to 1 octet/nanosecond for an in-RAM I/O operation! In addition, if the execution of a third party application fails, the files will still be created, thus making the global application quite safe.

The Java programming language does not natively provide such facilities, but different solutions exist. The first solution would be to implement it, which would be difficult as Java does not let the user manage the memory. Another solution consists of creating an in memory file system. Instead of a single file, this is a whole file system which is created in the RAM. This is practical as it allows a good management of the files such created, but is not the solution implemented for now. Indeed there exist a

memory mapped file solution natively implemented. Memory mapped files are files that exist on the hard drive but are mapped by the operating system to the RAM. The consistency of the files is insured by the operating system, so that the application only deals with the in memory part. In addition, if the application crashes the operating system will insure the last modifications of a file will be written on the hard drive before garbaging the memory content. This solution is faster than the Stream IO, making it a great solution for any IO operation. Finally the file will be stored outside of the Java heap memory, making it possible to be used as a solution for interprocess communiation. This is the solution that is being used for the software.

```
n1:20.5
n2:20
n3:20
n4:20
n5:-20
n6:20
n7:20
n8:20
####################################
Comp1
0;100;1
####################################
####################################
n1;n2[[0;25][0.02;0]]
n2;n2[[0;-25][0.02;0]]
n1;n3[[0;25][0.02;0]]
n3;n1[[0;25][0.02;0]]
n3;n2[[0;25][0.02;0]]
n5;n8[[0;25.2][0.02;0]]
n6;n5[[0;25][0.02;0]]
n7;n6[[0;25][0.02;0]]
n8;n7[[0;25][0.02;0]]
```

File Format 3-1 Example of entry file

The inconvenient of such a system is the extended need for memory. The files outputted by the applications can be quite big, as this is often the results of a simulation or of an analysis tool. This is not the case in the presented system, as the results for a typical interaction graph with the simulator weights around 300kB for one file.

3.10 Parameters and output

3.10.1 The graph file

The first developed application, the Hawkes simulator, has different parameters depending on what is wanted to be simulated. The basic possibility is to give the simulator a file name, which associated file contains the graph of the network to simulate.

On File Format 3-1 is an example of the file format the simulator takes as input. There are three parts: a header, a configuration part and the interaction graph, the last two parts constituting what is called a comportment. The header describes the neurons involved with their label and their base intensity. The intensity is a real number, so it can be positive, negative or floating. The format is neuron_name: intensity_value.

In the configuration part are described the caracteristics of a comportment, that is to say a label and the simulation beginning, ending and warming up times, followed by an interaction graph. Here there is only one comportment, named Comp1. The simulation will "warm-up" for 1 second – meaning that the simulation will run from simulation time -1 to time 0 but there will be no records – and will begin at simulation time 0 to finish one hundred seconds later. The warming-up period is equivalent to the time between experiments when scientists do not trigger the whiskers of the rats so that there is no interference between the experiments.

The interaction graph describes the edges (the vertices being the neurons which labels are known) with the convention: destination;source. Then the interaction function is described: [[time_1; intensity_1]…[time_n; intensity_n]]. The time_n value is the time to be elapsed after the source neuron has spiked for the intensity_n modifier to be applied to the base intensity of the destination neuron.

Several behaviors can be strung together, so that it is possible to recreate the communication changes in neuron flow. This reflects the experimental data that

measures a flow in a direction during a period, then a flow in another direction, sometimes opposite, during a following period of time.

Another possibility is to give the simulator parameters for it to algorithmically build a graph. The solution is interesting to simulate large graphs without having to write down all the interactions. The graph is then generated by randomly connecting nodes, each with the same probability p. The method is close to the Erdős–Rényi method. All the neurons have the same interaction function, for the objective is to allow an easy simulation of networks for which writing all the connections and the interactions by hand would take a long time.

3.10.2 To file or not to file

The question can be raised of whether or not a system, even a basic one, should interact with the user through lines of text in files, and if yes, what file format should be used.

On the one hand, communicating with files is a risky thing to do: I/O methods are often very complex and prone to errors, while a lot of errors also come from the users that can easily make mistakes while writing manually in files, even more when the files are big.

On the other hand, using fancy user interfaces, even more graphical interfaces, often does not help the user in their task. For instance, defining the interactions in a graph is a boring task, may it be done by hand writing or dragging and dropping. Plus, files are needed in order to keep track of what was simulated for instance, or for launching the same simulation on different machines and/or at different times. Also, developing graphical interfaces is time consuming reducing research time.

Another plus for the all-file interactions is that no information is needed for the user to be seen aside from the final results of the analyser. If the simulator displays the graph of the neurons being simulated, this is more for the user to check if the simulation actually simulates what was asked or if there is an error in the parameter files.

The final results, obtained by processing the data output from the existing R simulation in the analysing tool, are displayed for now in the form of a matrix of images, informing the user whether an interaction was detected or not. This is the place were a graphical user interface would have been perfect, as the result involves images. But this is actually a temporary output, as the analysing tool is wanted to be modified

so that is printed not a matrix of images but a matrix of probability of interactions. This is much more readable for users when a hundred neurons are involved in the simulation process, than a very big image composed of tens of thousands of little graphs on which some of them may show false positives (detected interactions that does not exist in reality) or true negatives (undetected interactions).

The other question was what is the file format that should be used? Indeed, the Java language proposes natively two possibilities for writing a file: writing a text file or writing a binary file. The second option was first chosen for its robustness, and easiness of use: parsing a binary file is a much easier task in Java than parsing a text file. Plus, binary files are often faster too read from and write to while weighting less than a text file.

But then the issue of the writing action came soon in reality, as using binary files necessitates the use of a piece of software for managing both writing and reading processes. This solution did not appear to be a necessary feature of the simulator while raising the same issues than a graphical user interface would have raised as discussed above, thus disappearing in favour of the more common text file solution.

3.11 Scripting

The coupling softwares launches subprocesses by executing commands. The input expected by a subprocess can differ from what is outputted by a previous subprocess, and vice versa. For instance, the analysing tool outputs state information during execution, so that the user can know which parts of the analysing process have been carried on and which parts remain to be done. This cannot be controlled as the source code of the analysing tool cannot be changed, and only a tiny part of this information is useful for the system. Actually the systems launches another simulation once the analysis is finished, so no part of the output really necessary.

These scripts are often longer than just a command line as the output of an application often does not feat what is asked by the coupling software, that is to say only string of caracters. For instance the simulation outputs different state variables

values during the execution process, which are filtered by the script presented in Code 3-7, so that only a configuration file is kept and given to the analysing tool.

```
#!/bin/bash
OUTFILE="/tmp/SimuOut.out"
mvn exec:java -Dexec.args="/path/to/file/simu.conf" > $OUTFILE
cat $OUTFILE | while read line
do
        SUBSTRING=$(echo $line| cut -c1-15)
        if [[ "The data file: " = $SUBSTRING ]]
        then
                ARG=`echo $line| cut -d' ' -f  4`
                echo ${ARG##*/} | tr -d '\n'
        fi
done
rm $OUTFILE
```

Code 3-7 Example of a script interface

Other scripts can be necessary when the data are stored in uncompatible formats for different functions. For instance in R the *read.csv* function, that creates a list of lists of data from a Comma-Separated Values (CSV) file, considers that a column is equivalent to a variable (for instance, Intensity_neuron1 ⇔ Column_1). But the *BoxLassoV* function, that computes coefficients for the LASSO method, requires the data in a matrix format, with rows corresponding to variables. There the easiest way was to write a R script managing the different formats before the execution of the functions.

Finally the testing part of the developing process required a statistical analysis of the results, which were first conducting using the Kolmogorov-Smirnov test before using the LASSO method. This tool constitutes the biggest part of the scripting made for the project. This is further developed in the Section 4.4.

3.12 Brief summary

The two programs that have been written are very different in terms of functionalities but also concerning the techniques and tools that had been necessary in order to achieve the implementation.

The first program is a simulation whose design relies upon the differences between a Java algorithm and an R algorithm, the understanding of what are Hawkes processes and how works a DEVS simulation. There was no real difficult part except for the comprehension of what is executed by the original R simulator.

The second program is a coupling software, which was a more difficult task to achieve. If it can be said that the real-time machine was partly written because it is based on the Parallel DEVS classes and methods, with some changes on how the time is managed by these classes and methods, some other commodities, like the virtual files and the ProcessBuilders were much difficult to come to.

Chapter 4 System Implementation and Testing

4.1 The environment of system implementation

The system was implemented on a personal computer running a linux operating system on a virtual machine. The operating system disposed of 2GB of RAM and of a single processor core cadenced at 2.40GHz. All the results were obtained with this configuration, unless otherwise stated.

Different programing languages were used in order to develop the different tools. The simulator and and the coupling software were implemented using the latest Java release, being the version 1.8.0_91. R and Bash, in their 3.2.5 and 4.3.46 version respectively, were used for the implementation of the analysing tools and script interfaces. Two Integrated Development Environment (IDE) were being used, first Netbeans, version 8.1, for Java, secondly RStudio for the R scripts.

The Maven build manager is used in order to get distant libraries source code at building time. Hence it is also used for the execution of the Java softwares, whether it be the simulator or the coupling software.

4.2 Key program flow charts

As there are two programs described here, there will be two main parts, one focusing on the simulator and the other focusing on the coupling software.

4.2.1 Simulator flow charts

This flow chart is an over simplified view of the running simulator process. The programs begins by initialising its state variables, which values are determined by configuration and data files. The simulated graph can also be randomly generated, which is suitable for large scale simulations. Then the program enters the main loop, which basically checks if the simulation can finally be stopped due to the absolute time

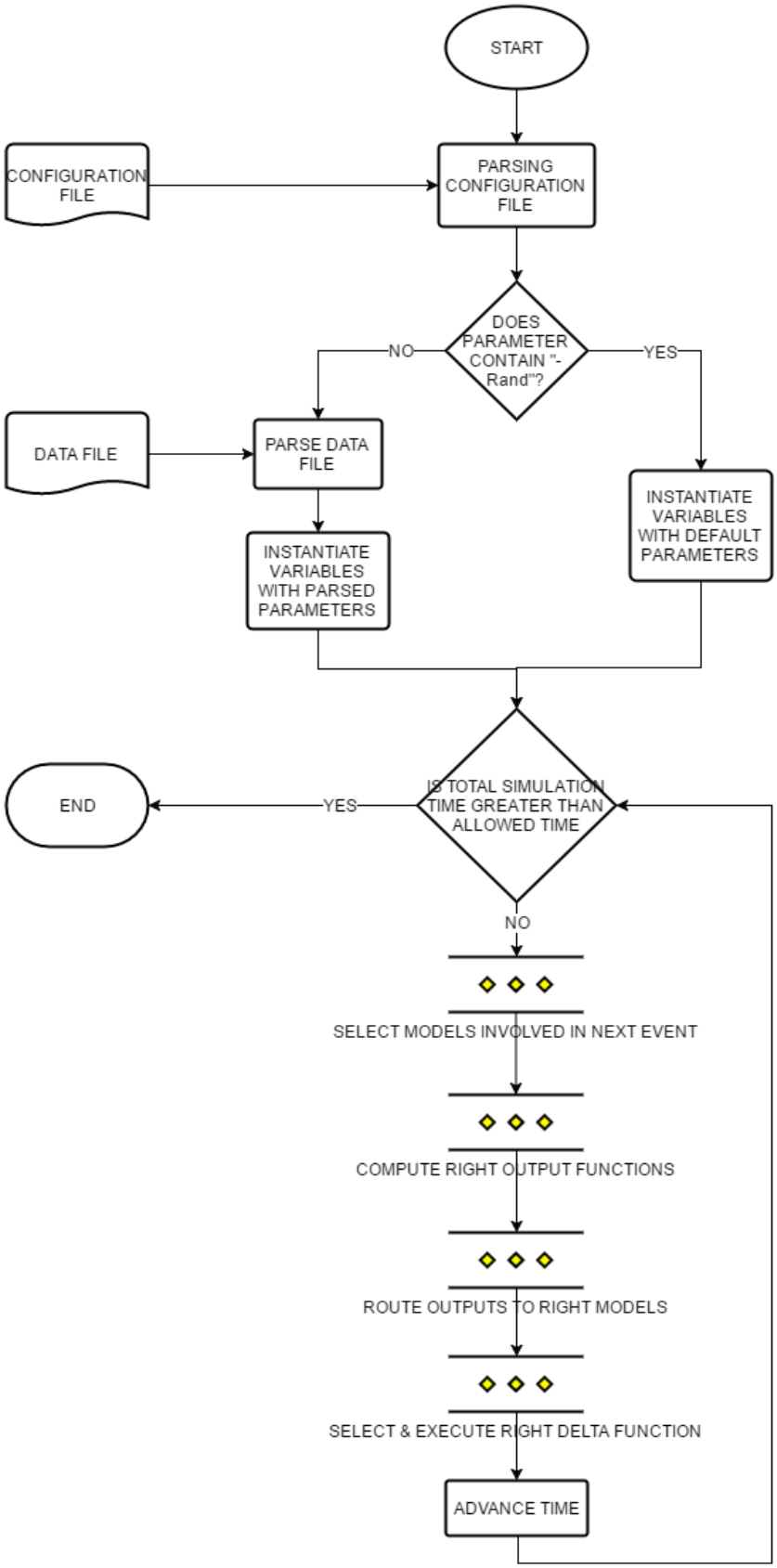

Figure 4-1 Simulator's flow chart

of occurrence of future events being much larger than the simulation allowed time. If

there is still time left for executing new events then the coordinator concurrently selects the models externaly or internaly involved in the next event. The models involved internaly have their output functions computed, and the resulting messages are routed to the right simulators (still concurrently for both). Once all the messages are ready the delta functions – deltint, deltext and deltcon – are concurrently executed, and the time is updated for all involved simulators and atomic models. Finaly the coordinator is updated its iternal clock too and the algorithm returns at the loop condition phase.

4.2.2 Coupling software flow charts

One could expect the coupling software to have a very more complex flow chart than the simulator it encapsulates, but both the simulator and the coupling software were written using the DEVS formalism. That is to say that the overall process will be the same in both cases. The two major changes are the absence of initialisation phase as there was been for the simulation. The software is indeed considered to be initialised at running time, the configuration relying on the connection between atomic models in the coupled model and on the class legacy of the atomic models. The other difference can be found lower in the diagram, where the algorithm selects the next event to jump to and wait for it. The waiting here corresponds to the waiting method of the main thread. It is necessary so that the *Activities* can be processed. If an activity should have finished but has not, the next event is still the one that the termination of the activity should have trigerred. Then the waiting times will be shorter for catching the termination as closely as possible.

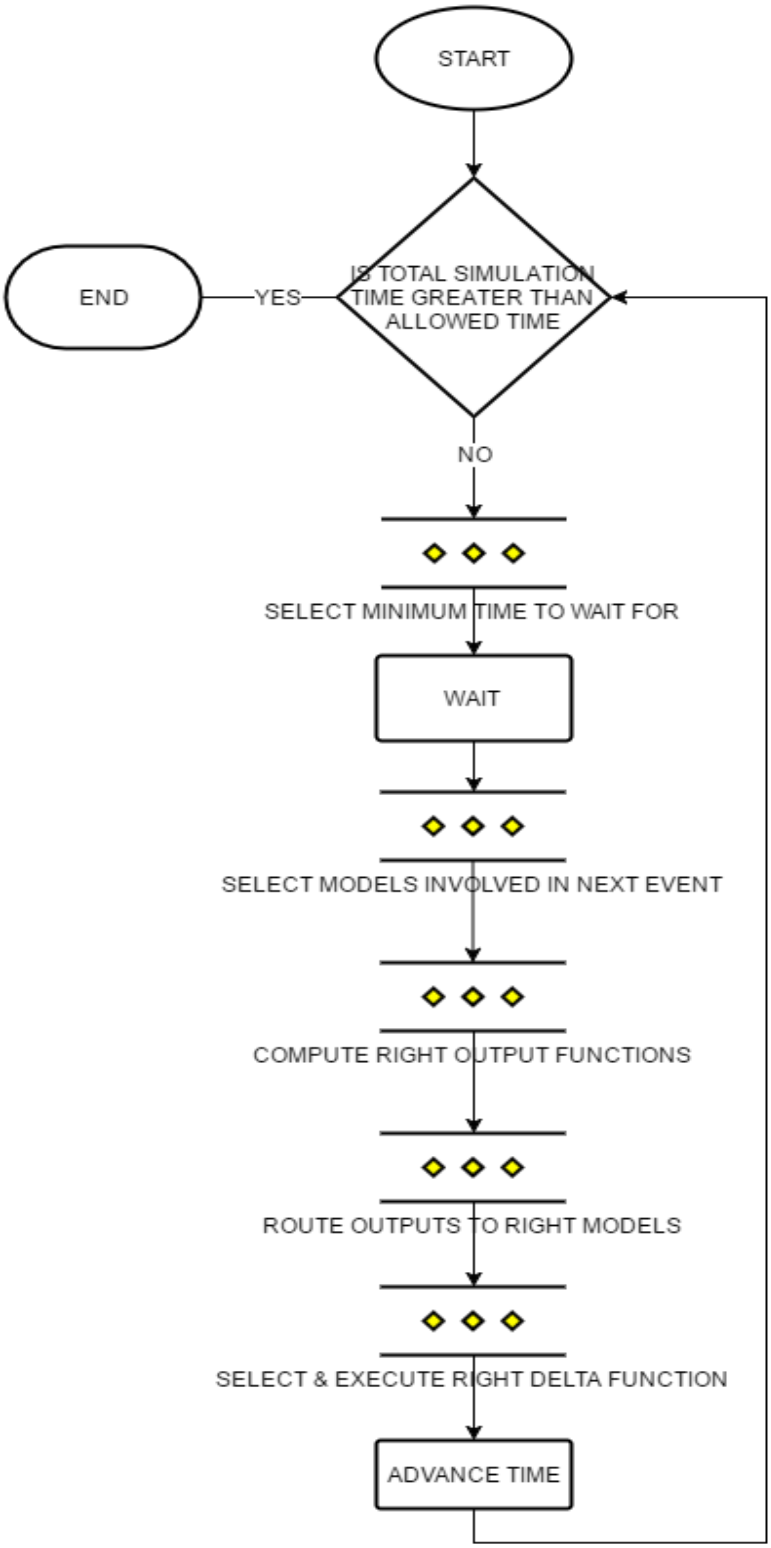

Figure 4-2 Coupling software's flow chart

4.3 Key Interfaces of the software system

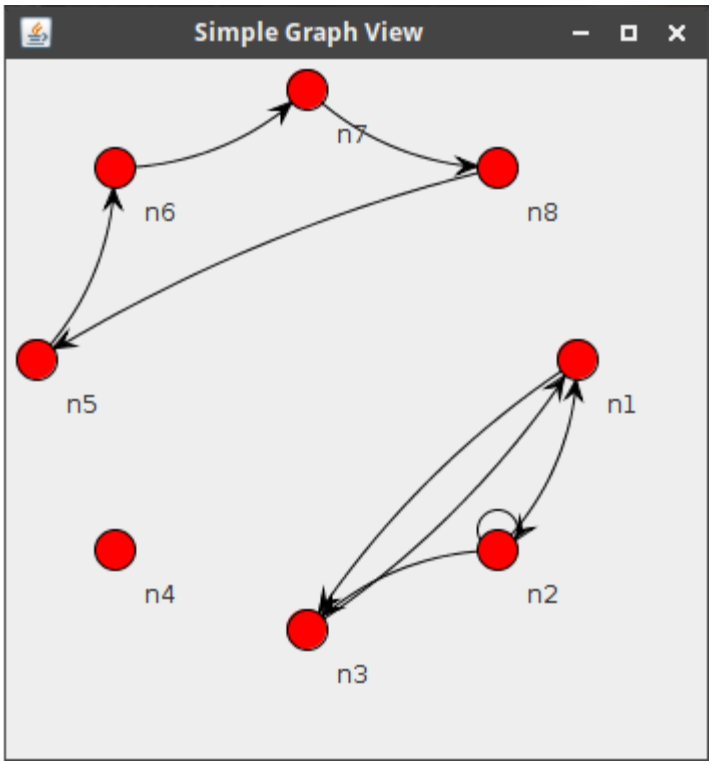

Figure 4-3 Simple graph view from the simulator

There are very few interfaces as one of the main goal of the project was to automatize as much as possible the processes (simulation and analyse). Actually, there are two interfaces: the input arguments and the output files.

Inputs arguments of the simulator is where necessary data are given to the software using text files. There is no need indeed to look for any form of user input during the simulation, as the objective of the simulator is to simulate the flow in a given graph as fast as possible. Hence, the possibility to introduce later an improved graphical interface for users to visualize better what is going on at certain times of the simulation is not dismissed.

A second interface takes form in the output of the system, which is mainly data written in files – whatever the file format – but also graphical user interfaces launched by third parties systems. Finally, the third kind of interface is the one between the system and other software, which is managed by the ProcessBuilder class, and the shell scripts that it runs. These shell scripts are often much more than the software command line way to be executed. On Code 3-7 is an example of such a shell script. It executes the software using maven, and stores the outputs in a temporary file. As the outputs is filled with different values of the system state variables, it must be filtered so that only the important information (the name of the files where the results are stored) is

recorded. Then the temporary file is erased, as not needed anymore. This is a simple example of why this treatments may be necessary and how to make it, but any much more complex treatments could also be considered.

4.4 System Testing

Testing has been an important part of the development process. This is truer because the results outputted by the simulator have been shown to and tested with the people who need it.

Thus for the first part of the development process no testing tool was necessary, the analysing tool being the only true black box testing tool. Indeed, aside from bad inputs from the users, upon which an error log is launched, the important part of the simulation process is the validity of the results. There are several ways to test the validity of the results outputted from the simulator, which have been assembled in a single analysing tool, implemented in R and C++. Given a file containing the results of a run of the Java simulator, it executes a simple test based on the Kolmogorov-Smirnov method. The idea of the test is to compare the values with a reference, here an exponential of parameter one. If the tested function is "close" to the reference, the indicators will be high for the p-value and low for the statistic. The indicators values are written in a text file and formatted to match the requirements of a LaTeX table argument. This means that the file can be included in a LaTeX table without any change.

On Figures 4-4 and Table 4-1 we can see a part of the results from this analysis, only for the neuron 1 on Figure 4-4. The test is made on the values of the integrals of the intensity at the internal event times. The graph on the left is a histogram of the differences between two consecutive values of these integral values. As the red curve shows, the values should be shared out as an exponential. The result can greatly differ

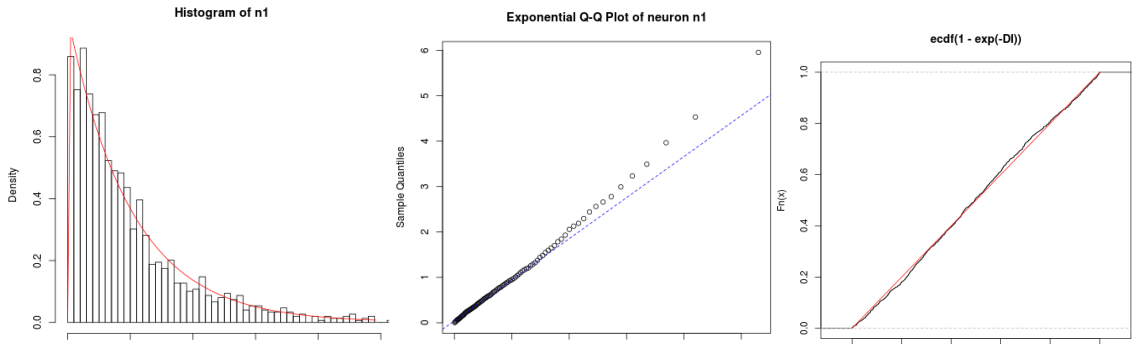

Figure 4-4 Different statistical tests

from an actual exponential as it is valid for an infinite simulation time. The two other graphs display the same information, but using a Quantile-Quantile plot. The graph should display something close to a straight line, which is the case. This test is more precise than the histogram.

The next Figure is a table of the p-value and the statistics of the 8 neurons that have been simulated. The statistic reflects the distance between the result and the theoretical curve (for an infinite time). This is why the numbers are relatively low. The p-value is also an indication of the closeness between the data and the theoretical curve, but it is more a "the closer to one the merrier" indicator.

Neuron name	Statistic	p-value
N1	0.0240728560656834	0.353670791492095
N2	0.0324561292358713	0.405808582013856
N3	0.0183797367433974	0.692234645885691
N4	0.028942118575323	0.902909700015841
N5	0.020060476498267	0.915596353040034
N6	0.0250295516444078	0.73377986016329
N7	0.0337899086069106	0.353872174137624
N8	0.0248171768983048	0.728566417104132

Table 4-1 Table of the statistical results of the first analysing tool

The second test is the one developed at LJAD, based on the LASSO method. Given a graph and the results of a simulation, it tries to reconstruct the graph from the results of the simulation. The test is not as objective as the Kolmogorov-Smirnov test can be. Indeed, the graph being reconstructed will be wrong most of the times, but on multiples

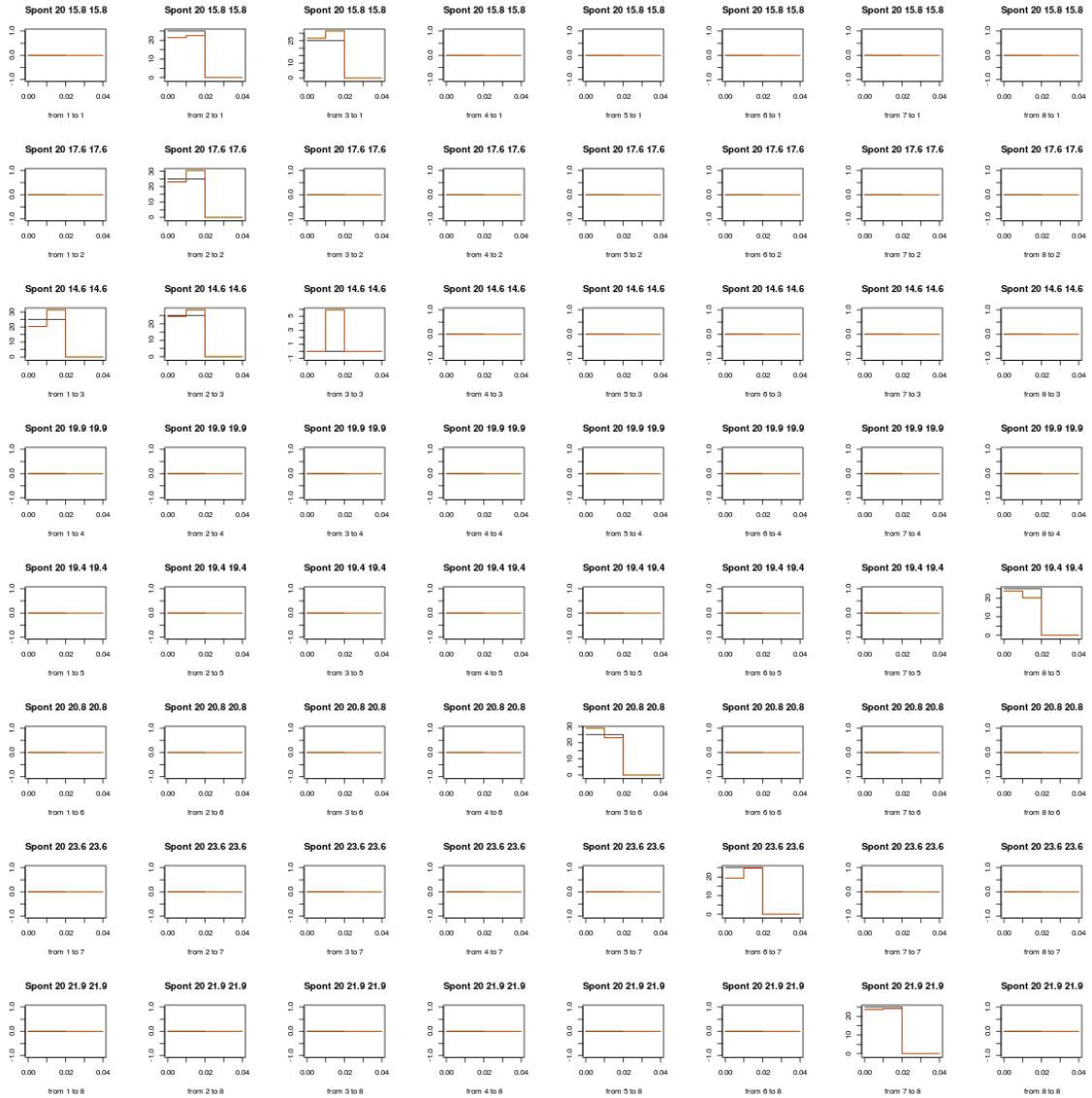

Figure 4-5 Example of the LASSO result of the R simulation

runs with multiples values, what is being simulated will appear. On the Figures 4-5 and 4-6 we can see the result of the LASSO method on the same graph, simulated with the R program for the first one and the Java program for the second. The graph being simulated is the one displayed on the Figure 4-3. The red curves is what have been reconstructed by the analysing tool from the results, in black what should have been reconstructed ideally. There are small differences in the two results, which can be

explained by the difference in the random generator seed. We can also see that in the two cases the tool reconstructs an interaction which does not exist in reality, from neuron one to itself. This kind of variation appears and disappears from a simulation to another, which is why stochastic simulations must be run numerous times to smooth this kind of edge effect.

This second test is not even enough for verifying the statistical validity of the program. The last step is to run the first two tests multiple times, and look for the uniformity of the results. For instance the p-values of the Kolmogorov-Smirnov test have to be uniformly distributes between 0 and 1.

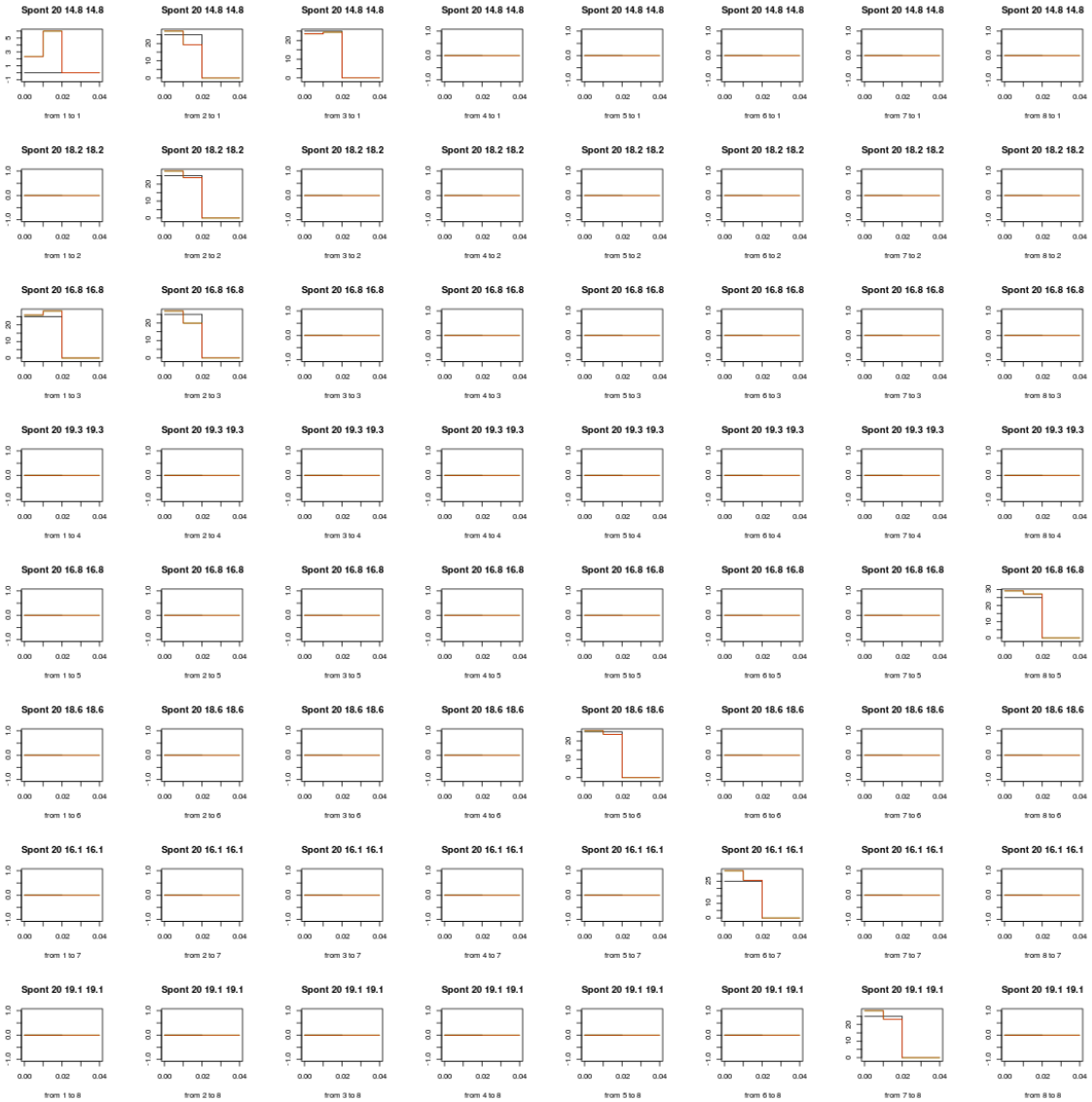

Figure 4-6 Example of the LASSO result of the Java simulation

White box testing can then be carried out during development process by testing every single method that is implemented before using it. This is true for methods aimed

at computing mathematical operations, but more difficult when dealing with the basic DEVS functions. The tests focus on the accuracy and the efficiency. For the simulation software, the tests were necessary during the implementation of the intensity storage solution. It was facilitated by the possibility to execute the simulation, so that the only methods to be tested were the ones in relation with the intensity storage system. Before implementing the intensity storage system, the intensity related methods were returning constant values, so that the rest of the system could be built more easily.

Advantage was taken of the debug facilities embedded in the Netbeans IDE. It was indeed possible to visualize the content of every variable while the simulation is paused. Finding the causes of an error, for instance a negative value when only positive numbers are involved, is then quite straightforward. This is not the case for the next testing phase.

Testing a real-time system can be a pain as the system is not exactly predictable. Plus, when trying to debug such a system, the behaviour can be distorted from the expected behaviour as the time is not flying at a normal rate but is being discretised by the pause and continue actions. But the real game begins when subprocesses are involved, that is to say when the mother application calls for the execution of other applications the IDE cannot control. Verifying which objects are being executed at a given time, are the beginning and ending time as expected, etc.

4.5 Brief summary

The systems overall functioning is quite simple at first glance. The thing is that it is not a software aimed and designed to interact with users, but a software that helps the users in their work by automatizing processes with simple solutions. The system testing phase, while being extremely important for the system is involved with scientific calculations, is based essentially on two tests, one for each program. The simulator is tested by analysing the results with a dedicated tool and the coupling software tests involves checking the coupled programs are launched and their end is detected at the right time. Though, the development process was feedbacked by testing, meaning every pieces of the software were tested before used which explains why there was so little different testing to be done once the software were built.

Conclusion

As a student I wanted to work in the domain of neural networks, and the link they share with artificial neural network. The topics of this internship is related to this domain from the simulation part to the coupling software implementation part.

The real goal of this topic lies in the will for researchers to have a software allowing them to both simulate and analyze fastly the results of their simulations. The implementation of a simulator as a first part of the project became quickly an evidence. The implementation was first interesting for learning the functioning of Hawkes models. This step was necessary in order to verify the correctness of the newly written simulator in comparison with the ancient one. Knowing how to analyze the results of the simulator with the analysing tool, beginning the coupling between the two software became possible. And finally, by improving the simple coupling between two programs to a general and real time coupling system, the goal of the project have been nearly met.

First, the internship begun with a discovery of both the Hawkes simulator and the statistical theory behind, but also the DEVS formalism and the implementation carried on by the I3S team. This was also a chance to improve my knowledge in R and Java, mostly concerning existing solutions for implementing multithreading applications.

After that, several implementations were proposed, depending on what have been understood of the functioning behavior of the Hawkes simulator. The models were tested on the number of points that were outputted, the quality of the results given by the Kolmogorov-Smirnov test and finally on the interactions detected or not by the method developed by the LJAD team.

For the final version of the Hawkes simulator, the number of points were very similar. For instance, a simulation with 8 neurons with same initial intensity (20, there are no real units, but can be considered as Hertz) and interactions function (+25 from 0 seconds to 0.02 seconds after spiking, nothing after). The graph was the same in the two situations.

After this simple test was passed, Kolmogorov-Smirnov's test was necessary in order to check out the goodness of fit of what was simulated. The vector of the differences between two consecutive measures of the integral of the intensity of a given neuron has to follow the distribution of an exponential variable of parameter 1. A little

checking tool was written, that outputs the result so that it can be directly understood by a LaTeX document as a table. This is nonetheless impractical when dealing with a high number of neurons, as the resulting table would be too large to be readable. About the indicators, there are two of them: the p-value and a statistic value. The first has to be uniformly distributed between 0 and 1 upon multiple tests, while the second has to be as low as possible. For the moment the tests focused on obtaining the good p-values and passing the last test. As the p-value were rather high while changing completely when the simulation time was changed too (for instance passing from 20 seconds of simulation to 100 seconds, the p-value of some neurons changed from around 0.8 to 0.06, while others stayed even), this was a good sign for uniformity.

So the third statistical test was the method developed by the LJAD team, named LASSO. The method takes as input argument the simulation results and the interaction graph, with the neurons initial parameters. The method then reconstructs what should be reconstructed (knowing the interaction graph) and compares it with what it can reconstruct from the simulation result. It ends up with a matrix of images, each of them informing the user whether there was an interaction detected (a function of which the integral of the absolute values is not null) or no interaction detected (a function of which the integral of the absolute values is null). Sometimes the analyzer will detect interactions that does not exist or will not detect interactions that effectively exist. This is due to statistical fluctuation, and can be eliminated by multiple simulations.

These are the results of the first part, the writing of a simulator, which results still need to be checked by multiple execution of the same models to confirm the statistical robustness of the simulator.

The second part of the project consisted in writing a coupling software for the simulator and the analysis tools, but more generally for programs, and that will manage the simulations using real time so that it can be interfaced with physical tools.

The software is at that time nearly finished and works as expected. It manages in time the different programs it has to launch and transmit the results between them. Its efficiency with a large number of subprocesses remains to be tested, though it is expected to keep good performances as there is little executed in the machinery.

Finally this project was a good experience and a success in the results as all part were implemented on time. It can be released any time but making a documentation first for helping users is necessary.

References

[1] Riehle, A., Grün, S., Diesmann, M., and Aertsen, A.M., Spike synchronization and rate modulation differentially involved in motor cortical function, Science, 1997, 1950:3

[2] Melisande A., Bouret Y., Fromont M., Reynaud-Bouret P., A distribution free unitary events method based on delayed coincidence count, 2015, 31

[3] Malot C.**,** Reynaud-Bouret P., Rivoirard V., Grammont F.**,** Test d'adéquation pour les processus de Poisson et les processus de Hawkes, 45ème Journées de Statistique, 2013, 4

[4] Zeigler B.P., Praehofer H., Kim T.G., Theory of modelling and simulation, 2000, location 1484

[5] Muzy, A., Lerasle, M., Grammont, F., Dao, V.T., Hill, David R.C., Parallel and pseudorandom discrete event system specification vs. networks of spiking neurons: Formalization and preliminary implementation results, The 2016 International Conference on High Performance Computing & Simulation (HPCS 2016), International Workshop on Parallel Computations for Neural Networks (PCNN 2016), 2016, 6

[6] Reconstruction and simulation of neocortical microcircuitry, Cell, 2015, 2

[7] **http://mathworld.wolfram.com/HawkesProcess.html**

[8] Hill, D.R.C., Mazel C., Passerat-Palmbach J., Traore M.K., Distribution of random streams for simulation practitioners, Wiley Online Library (wileyonlinelibrary.com) , 2012, 1430:3

[9] Hill, D.R.C.Parallel random numbers, simulation and reproducible research, IEEE Computer Society, 7:9

[10]Seong Myun Cho, Tag Gon Kim, Real-Time Devs Simulation: Concurrent, Time-Selective Execution Of Combined RT-Devs Model and Interactive Environment, 2:3

[11] **https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html**

[12]**http://java-performance.info/bigdecimal-vs-double-in-financial-calculations**

[13] **https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html**

Statement of Originality and Letter of Authorization

学位论文原创性声明

Statement of Originality

本人郑重声明：此处所提交的学位论文《中文题目 English Title》，是本人在导师指导下，在哈尔滨工业大学攻读学位期间独立进行研究工作所取得的成果。且学位论文中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本学位论文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名：　　日期：　　年　　月　　日

学位论文使用权限

Letter of Authorization

学位论文是研究生在哈尔滨工业大学攻读学位期间完成的成果，知识产权归属哈尔滨工业大学。学位论文的使用权限如下：

（1）学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文，并向国家图书馆报送学位论文；（2）学校可以将学位论文部分或全部内容编入有关数据库进行检索和提供相应阅览服务；（3）研究生毕业后发表与此学位论文研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。

作者签名：　　日期：　　年　　月　　日

导师签名：　　日期：　　年　　月　　日

Acknowledgement

First I want to thank Alexandre Muzy for proposing the subject, but most of all for his support and accompanying all the project long, from the discovery of the topic to the writing of the report.

A special thank to Patricia Reynaud-Bouret who is the second person at the source of the project. She was of great help for explaining the theory behind the method and the functioning of the original R simulator she implemented.

Also thank to David Hill for putting me and Alexandre in contact in the first place, and for his rewieing of the documents.

Finally, thank to Tonghua Su, Lanshun Nie and Naiqian Zhang for their advices and time during the rewiewing phase.

Resume

Confirmation of Supervisors

HIT Supervisor
Signature: Date:
UBP Supervisor
Signature: Date:
Internship Supervisor
Signature: Date: