9e Conférence Internationale de Modélisation, Optimisation et SIMulation - MOSIM'12
06 au 08 Juin 2012 - Bordeaux - France
« Performance, interopérabilité et sécurité pour le développement durable »

DEVS GRAPHICAL SPECIFICATION TO CODE GENERATION

M. HAMRI

LSIS UMR 7296 / Université Paul Cézanne
Avenue Esc. Normandie Niémen
13045 Marseille- France
amine.hamri@lsis.org

G. ZACHAREWICZ

IMS UMR CNRS 5218 / Université de Bordeaux
351, Cours de la Libération
33405 Talence Cedex - France
Gregory.zacharewicz@ims-bordeaux.fr

ABSTRACT: *The paper presents an approach to generate automatically code from DEVS graphical model specifications. The generated DEVS models code is given afterward to the LSIS_DME DEVS simulator to execute the corresponding behavior. The user, even beginner in DEVS modeling, increases his trust in simulation results due to the fact he is not interfaced with an intermediate actor (modeler or programmer) that would interpret user requirements in the modeling and simulation activities. Using appropriate graphical items, the user is capable to develop his own DEVS models, to carry out simulations and to analyze them.*

KEY WORDS: ***DEVS, Simulation, Graphical Modeling, Teaching DEVS, LSIS_DME.***

1 INTRODUCTION

Discrete event Modeling and Simulation (M&S) is the one of the popular paradigms to model dynamic systems [Banks et al., 2000]. Scientists and researchers have defined methodologies, approaches and formalisms to assist users (experts in a specific domain) to define models and simulate them.

The modeling activity is a hard task due to the fact that the modeler needs knowledge on the system to model in addition to skills on the modeling language and programming. This activity may be dispatched between three actors: the user who specifies the system in his own language (natural language, visual models, etc.), the modeler who formalizes the specification using a formal language, and the programmer who codes the formal specification using a programming language to allow simulations.

The steps from specifications conducted by the user until coding the formal specifications conducted by the programmer may lead to lack of understanding and mistakes. The modeler does not have enough knowledge on the target system to specify. However, the user with his knowledge and skills should specify the system easily but the complexity of formal languages makes this task difficult. So to bridge this gap, theorists associated graphical representations to the formalism concepts. In fact it is easy to develop formal models using graphics instead of using abstract symbols.

In the other hand, coding the specifications using an oriented-object programming language, conducts the programmer to develop patterns to design them. These patterns are generic object classes that define the computational concepts of the formalism. Then using these patterns, the programmer makes new classes that represent the elements of the specifications. So the developed code is pattern-driven and model-driven specifications. The effort to code the specification according to designed patterns is considerable. This fact evokes much development time and coding errors may occur.

In general discrete event simulation formalisms support a graphical modeling in addition to the symbolic one. For example Statecharts [Harel, 1987] are purely graphics. States are modelled with opaque rectangles and transitions with directed arcs. Using such these formalisms, the dynamic of the system may be shown while the simulation runs. So the user may validate his specifications and concludes on simulation results in case of advanced software tools. Thanks to the graphical concepts, the user becomes a modeler capable of specifying his system using a sound formalism. However, the programmer is still useful to implement the specification when no friendly user-tool exists which is the case of DEVS [Hamri and Zacharewicz, 2007]. To save the user from coding and the programmer from learning the formalism and understanding the user specifications, we propose to generate automatically code from user specifications based on graphics.

In this paper we propose an approach that allows generating code automatically from DEVS user specifications. This will constitute a good framework for modeling and simulation of DEVS specifications, in which the user does not need skills knowledge on programming or help from a programmer. The fact that there is no intermediary between modeling and simulation increases the user confidence in models and simulation results.

The paper is organized as follows: the section 2 and 3 give recalls on DEVS formalism and issues on designing finite state machines. The section 4 discusses how DEVS models should be mapped into object models. The section 5 illustrates our approach to implement DEVS user models on computer with giving an example in section 6. Finally in section 7 we conclude on this paper and we outline our future works.

2 2. RECALLS

2.1 2.1. DEVS formalism

According to the literature on DEVS [Zeigler, 2000] the specification of a discrete event model is a structure, M, given by: **M= <X, S, Y, δint, δext, λ, D>**, where **X** is the set of the external input events, **S** the set of the sequential states, **Y** the set of the output events, δ**int** is the internal transition function that defines the state changes caused by internal events, δ**ext** is the external transition function that specifies the state changes due to external events, λ is the output function, and the function **D**: S→R+U∞ represents the maximum length or the life time of a state. Thus, for a given state "s", D(s) represents the time during which the model will remain in state "s" if no external event is incurred.
Zeigler introduced the concept of total states TS of a system as: **TS = {(s, e) s ∈ S, 0 < e < D (s)}**, where **e** represents the elapsed time in state "s". The concept of total states is fundamental in that it permits one to specify a future state based on the elapsed time in the present state. Potential benefits may lie in its ability to implement event filtering, wherein a planned change of state will be realized by a model only when the time that separates two key events exceeds a predefined value, and to encapsulate otherwise the mechanical event filtering at the conceptual level. A DEVS model M is encapsulated in atomic model to make it reusable. The user defines the external interface which consists in defining input and output ports that receive on and send out events respectively.
DEVS atomic models are reusable using DEVS coupled formalism that includes the specification of DEVS components and the couplings over them. The obtained model is defined with the following structure:

MC = <X, Y, D, {Md/d∈D}, EIC, EOC, IC, Select>

X: set of external events.
Y: set of output events.
D: set of components names.
Md: DEVS models.
EIC: External Input Coupling relations.
EOC: External Output Coupling relations.
IC: Internal Coupling relations.
Select: defines a priority between simultaneous events intended for different components.

2.2 DEVS languages oriented user specification

The high level user specification language based on graphics is well suited to describe, at a conceptual level, the complex systems with discrete event models. Most of automata use the graphical specification that is easy to model behaviours. The principle is simple, states are specified with nodes on which the state name figures, and transitions are specified with oriented (directed) arcs that rely a source state to a target one. In DEVS, this concept was popularized by Seong and Kim [Seong and Kim, 1994]. For each element of DEVS, they propose a corresponding graphic:

- A state is modelled with a circular node on which the name of state and its duration are mentioned,
- An external transition is modelled with a dark directed-arc on which it is mentioned the event causing the transition (preceded by the port name if DEVS atomic),
- An internal transition is modelled with a dashed directed-arc with the output event written on (outport is added if DEVS atomic), and
- Also conditions could be specified on transitions to distinguish them when they are candidate to be triggered. When a transition could be fired, it is necessary to verify the value of the associated condition (if false the candidate transition is ignored, if true the candidate transition should be fired).

Finally this description is encapsulated in a box on which input ports are noted on the left side and output ports on the right side (e.g. Figure 1). Next these box that model DEVS atomic components are employed to define a composite (DEVS coupled) model in hierarchical way.

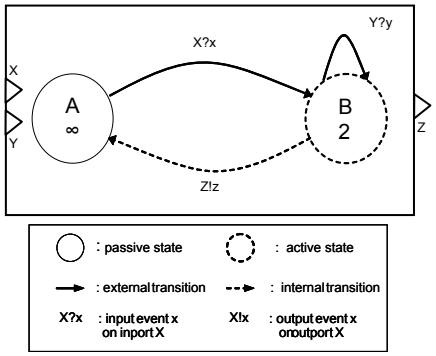


Figure 1: Example of Graphical representation of DEVS Atomic Model

2.3 Existing DEVS software tools

In the literature of discrete event M&S, there is about a hundred of tools in the field. This is shows the importance of the software tools in the cycle of M&S. The DEVS group standardization lists on his web site the most used DEVS tools that are known by the DEVS community. The oldest tool is ADEVS developed by the research team of the Arizona University using C++. The tool is an ad-hoc DEVS simulator in which the programmer who codes the DEVS user models extends the abstract class atomic to define DEVS atomic components, and the class Network to defined computerized DEVS network. Once the whole DEVS model is coded, the simulation could run. A Java version of this tool exists, on which practical facilities are offered like DEVS coupled components and defining coupling over them in graphical way.
Another interested tool is the CD++ tool developed inside the Carleton university [Wainer, 02] [Wainer et al., 01]. The tool is recognized to be beginner-friendly in

addition to its efficient kernel simulation that allows quick simulations. In fact, the tool integrates a workspace to model graphically DEVS atomic model in addition to DEVS coupled ones. This is very useful for users that have not enough skills in programming and especially in object oriented programming.
Therefore, we conclude that the graphical notation allows users to focus on modelling aspects instead of spending time on coding models.
Based on this analysis and the list of simulators available on the web site of DEVS group we identify two categories:

1) Simulators based on programming code. The programmer develops code corresponding to DEVS models. He should work strictly with the modeller to avoid of non-understanding. So, basic skills in DEVS modelling are suitable.
2) Beginner-friendly tools like CD++, DEVSJAVA, etc. in which graphical manipulations are allowed to the user, but limited to the offered facilities. Once the user defines the whole model using the chosen tool, then from this specification, the translator generates a logical structure which will be loaded by the kernel simulation to produce behaviours.

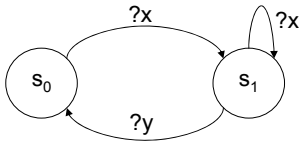
We point as an essential remark that each category offers advantages to users. The first category by implementing DEVS user specifications into byte code, the simulation process is fast. However the users are not able to conduct simulations alone, they need a programmer to code correctly their models and run efficient simulations. Otherwise, the beginner-friendly tool gives a friendly framework to users for M&S their specifications. But the simulation may be delayed due to the fact existing tools do not give attention into translators that transform graphical models into computerized ones by transforming them into a set of logical structures. Based on these statements, we will aim to improve beginner-friendly tools by focusing on translators to obtain object code readable and making simulations safe. By exploring the automatic code generation field from automata, we should be able to raise this challenge. The next section gives an overview of the most research works in this field.

3 DESIGN OF USER FINITE STATE MACHINES

Designing Finite State Machine (FSM) is an interesting field of software engineering from 90's. Researchers and practitioners developed approaches to transform discrete event specifications based on graphics to statement code. In [Dijk and van Gurp, 99], the authors discuss techniques to code finite state machines (classic automata) and classify them into three classes according to the designed code.

3.1 Code based on switch statement

Many works adopted the switch statement to code a finite state machine. The basic idea consists on coding states with an enumerate set of states and when events occur, according to the current state, the switch statement deduces the next state and updates the current state variable. The difficulty of the switch statements is that when the number of states increases. This fact leads to statements hard to read and maintain. In case of hierarchical state like Statecharts, the imbricate switch should be used to allow a waterfall test. Unfortunately, this is still ambiguous for the reader and the hierarchical structure of state is not explicit in the code.




```
switch(event){
   case x: switch(current_state){
          case s0: current_state = s1;
                   break;
          case s1: current_state = s1;
                   break;
         }
          break;
   case y: switch(current_state){
          case s0: current_state = s1;
                   break;
          case s1: current_state = s1;
                   break;
          }
          break;
}
```

Figure 2: Coding a FSM using the switch statement

3.2 Code based on structures

This approach proposes a logical and structured representation for the automata. Given a finite state machine, the suitable structure could be a table with two dimensions in which the set of states defines lines and the set of input event defines columns. Each element of the table represents the new current state to consider when a transition is fired. If Output events appear on the automata, the table should be extended (see the figure below). However, dynamic computations like storing conditions and actions on variables are not supported by this structure.

state\event	x	y	x	y
S_0	S_1	-	-	-
S_1	S_1	S_0	-	z

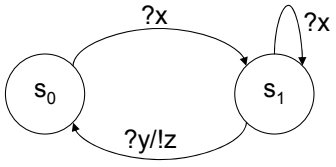


Figure 3 : Mapping a FSM into a two-dimensional matrix

3.3 Code based on object-oriented paradigm

The object oriented paradigm offers more programming concepts and possibilities to well design concepts for the target formalism. Aggregate should be used to describe states in hierarchical form, inheritance to reuse existing

models, the encapsulation to encapsulate models, etc. This approach gives more freedom to design the formalism elements. The works of Adamczyk [Adamczyk, 2003] shows how a FSM may be implemented using an object-oriented language. Also, The DEVS-Sim proposes an interesting approach to generate object code from a DEVS textual language that respects a Backus Naur Form grammar.

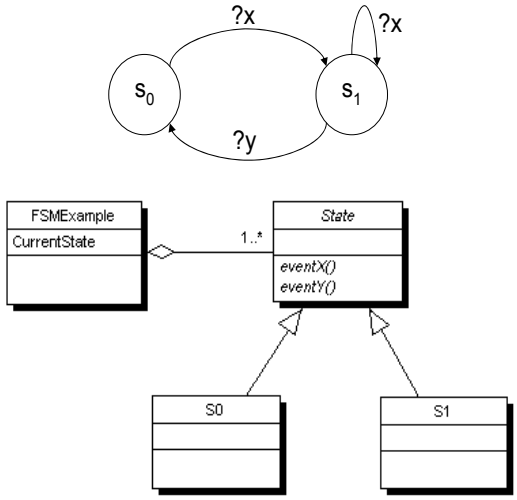


Figure 4 : A designed class diagram from a FSM

Indeed, designing DEVS with object-oriented paradigm is an interesting issue and should lead a lot of advantages to DEVS community. In the next section we propose an approach to design DEVS models (atomic and coupled) into object paradigm with showing practical results and performances.

4 DESIGN OF DEVS MODELS USING OBJECT PARADIGM

The first object oriented language was appeared in 1970 under the name SIMULA and Smalltalk language. It is proposed as an alternative to procedural languages that present a lot of lacks like: code not safe, difficult to reuse, etc. The Smalltalk society popularized this language and a new paradigm is born. The object paradigm is well recognized by the software engineering community to design complex systems and software applications. This paradigm is based on the concept class. A class is defined as an object with characteristics named attributes or variables and methods that act on attributes and provide services to other classes. The object paradigm allows encapsulating attributes and methods in classes and limits the visibility to other classes using specific keywords (public, private, etc.). New classes may be designed by reusing old ones and extending them to add new attributes and methods thanks to the inheritance. Finally the polymorphism allows writing classes with safety.
Therefore, it is more interesting to map DEVS models with object paradigm to obtain computerized models instead of using switch statements or logical structures like table.

4.1 Designing DEVS atomic models

According to past works in software engineering the rule to design FSM is to map states with classes and transitions with methods (see figure 3). When events occur, they provoke calling the corresponding methods. That is to say, events are assimilated to the call of methods. This may be useful to obtain fast simulation. However the developed code is less readable and less abstract due to the fact transitions are mapped into methods. In our proposal, we decide to map transitions to classes, in which the target state is stored in the form of an attribute. This technique was adopted by [Dijk and van Gurp, 1999] to code FSMs. They show the advantages of this kind of mapping that consists on obtaining a readable code easy to maintain and modify.
In our approach, we extend this rule to distinguish internal transitions from external ones. This distinction allows associating the method *output()* that implements the output function λ of DEVS to classes that code internal transitions only. Thanks to the inheritance of classes a such specialization is possible which guarantees a clean and safe code. A code without this distinction, we must implement an output method with an empty body in case of external transition classes and that pollutes the code. Based on these statements of mapping, we design a class diagram to implement a DEVS atomic model shown in figure 4. This diagram proposes a clear separation between states and transitions by designing them with independent classes. Then, relations exist between these classes to obtain a coherent structure. A given class of state should reference all external transition classes that go from the considered state. However the internal transition classes are only referenced by active states with respect to the DEVS operational semantics. This fact leads us to specialize the class *State* and extend it into two classes *PassiveState* and *ActiveState*. We note that cardinalities are expressed on this diagram, to control and respect also the DEVS semantics.

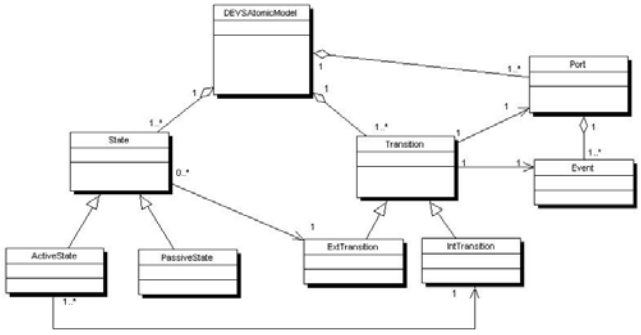


Figure 5: Class diagram of a DEVS atomic definition

4.2 Designing DEVS coupled models

Making a DEVS coupled model consist on reusing defined DEVS model that the user saved in top up way. Next he defines connections (oriented ones) among reused DEVS models by specifying external input, ex-

ternal output and internal couplings (EIC, EOC and IC respectively).
All the elements of DEVS coupled appear on the class diagram. The ports are mapped into class *Port* within references to other ports. In fact, these references define the possible couplings EIC, EOC and IC. The class DEVS coupled consists of other classes that implement DEVS coupled and atomic models. The class *Abstract-Model* plays the role of intermediary allowing saving them in a unique object (vector, list, etc.) by the cast technique. Still the last element, the function select which is encapsulated like a method in the class DEVS coupled and should define the priority between DEVS components at the same level with the same parent.

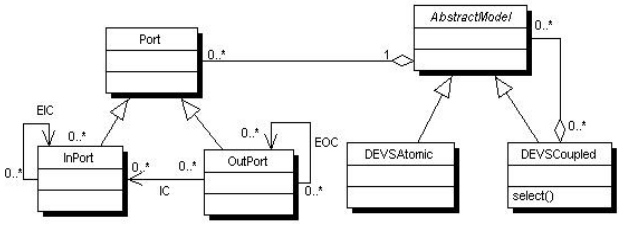


Figure 6 : Class diagram of DEVS coupled definition

Now, we will discuss our approach to model and simulate DEVS user specifications using a software tool that we developed at LSIS lab.

4. LSIS_DME: An environment for M&S of complex systems

The LSIS_DME (Lab of Sciences of Information and Systems_DEVS Models Editor) tool is developed by the LSIS team to enhance the process of M&S of discrete event systems in scholar courses and make easier the description of models than existing tools. The environment was designed according to user requirements: allow a graphical modeling since design atomic models until coupled ones by "drag & drop", to avoid the programming step and focus on learning DEVS concepts and principles. Consequently user with basic knoweldge on DEVS should be able to describe DEVS computerized models and analyse them by simulation. Step by step he improves his skills on M&S and especially DEVS; instead of consuming time in coding models and handling code due to modifications that often occur on his models.
In addition we propose an original approach that consists on generating automatically code from DEVS user specifications. This process is illustrated on the following figure, when a user adopts LSIS_DME to model and simulate discrete event systems.

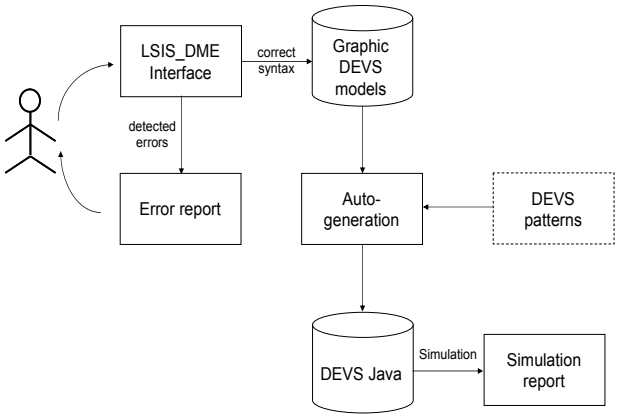


Figure 7 : LSIS_DME approach in DEVS M&S

4.1 Graphical representation of DEVS models

The LSIS_DME provides a worksheet to model DEVS atomic and coupled models. The elements of DEVS atomic state, transition and port are in grouped into a visual palette. The declarations are realized only with a click on the palette and a drop inside the worksheet. Once the user has modeled his system, he selects the corresponding library in which the current model will be stored. In this way DEVS basic models are saved and may be reused in top up approach to define new models in modular and hierarchical fashion.
To get more freedom with the graphical definition of atomic DEVS, we extend the state variable S (the third element of DEVS atomic definition) to the concept phase in which other state variables could be added. Consequently we can define in a graphical mode actions and conditions on state variables. Actions are basic arithmetic operations (addition +, subtraction -, multiplication *, division /) that compute the new values of state variables. These actions are defined on the transitions (external and internal ones) and executed when the associated transition is fired. Conditions also carry on state variables and used to distinguish candidate transitions (that mean once an event occurs, if there are more than one transition may be fired it should be only one in which the associated condition is true at this time).
We note that also the extension of the state variable is done in graphical manner by declaring the other state variables in the "init" phase and set them with the corresponding initial values.

4.2 Formal verification of the syntax and semantics of DEVS models

To save correctly the DEVS models defined by users, a formal verification is conducted. We define a BNF (Backus Naur Form) grammar to determine the precise syntax of both DEVS atomic and coupled models. The BNF grammar is a formal mathematical way to describe specification languages. It consists of rules named production rules that are applied to obtain a sentence (in our case a DEVS model). The grammar that we use resembles slightly to DEVS definition language [Zeigler et al., 2000]. However the DEVS BNF grammar that we define

is fuller than DEVS definition language and its objective is different. The DEVS definition language was introduced to define untimed DEVS models in which the time advance function D is not specified.
With this grammar, syntax (static) errors are detected before simulation and the corresponding error messages are displayed on a shell. For example let us consider an DEVS atomic model, if the user forgets to associate an internal transition to an active state "s", the tool displays the corresponding message "phase s without an internal transition". Errors are also detected in DEVS coupled models, the grammar can verify if all couplings are done to have correct DEVS coupled models.

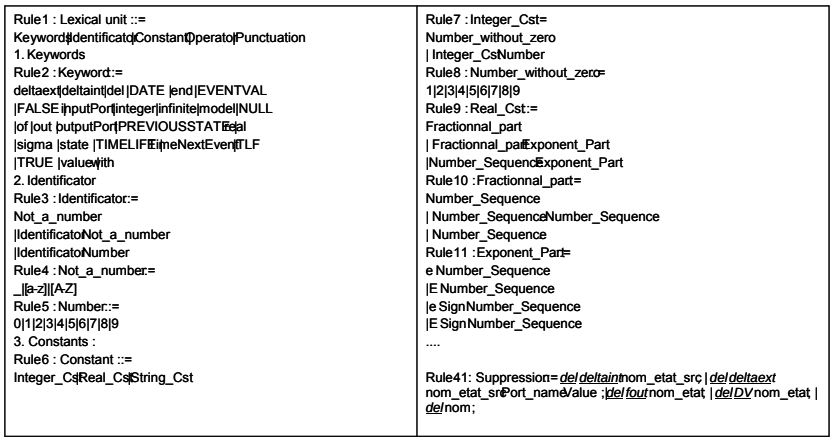


Figure 8 : DEVS BNF grammar associated to LSIS_DME models

An important feature to take into account too is to verify the determinism property for DEVS atomic models. To enhance the simulation, semantic errors must be avoided. We define a set of prevent messages that encourage the user to verify the determinism of models. These errors are such as defining more than one external transition for an input or two internal transitions that are not exclusive. This problem leads to a new issue on verification and validation of DEVS models that we should explore in the near future.
We note that the BNF grammar defined above concerns models edited with LSIS_DME that consists of 41 rules of production. We note also that the type of state variables is basic (integer, float and string). This fact limits the use of other types like arrays, lists, stacks.

4.3 Automatic code generation of Java code from DEVS user models

The automatic code generation is a part of software engineering. The DEVS scientists and developers proposed particular tools that insure partially the step of code generation like DEVSJAVA tool of ACIM and VLE of LIL labs. These tools provide for users to generate skeleton and formatted files from designed DEVS models. However the users do not have dedicated "components" to model DEVS behavior which is defined by the variables and functions of DEVS atomic. We believe with this lack such tools can not be used in scholar courses to learn DEVS because the modeling consists on designing models and not only reusing them, the facility (the reuse of DEVS coupled models) given by many DEVS tools. In other terms the users should not code first atomic models and reuse them in coupled ones - even if skeleton files may be automatically generated - they should focus on specifying models and maintain them when errors or modifications occur.
The LSIS_DME provides this option. So based on DEVS class diagrams, it generates Java code from DEVS user models. In other terms, the tool instantiates the DEVS class diagram according to the user model to produce Java code that will be given next to the simulator. In first, given a DEVS atomic model we map it into a DEVS java class. Next internal classes are added into the main class and which correspond to the different states and transitions of user model. Each internal class extends an abstract class: given a state, it is mapped into a class that extends the abstract class *Phase* and given a transition (internal or external one), the corresponding class extends an abstract one too. To conduct this automatically, we develop a module that execute the following steps:
Let us consider a DEVS atomic model *Model*, we:

1) create the class *Model*,
2) add two arrays that code the set of input and output ports.
3) if the state variable is extended, adds the corresponding variables mapped into attributes of the class Model.
4) for each defined state (phase), extend the abstract class *Phase* to create the corresponding class. Each extended class should return the duration of the state that maps it and the out transitions by implementing the corresponding abstract method.
5) for each defined transition, extend the abstract class transition and implement the abstract method *output()* that returns the output event. If the transition depends on a condition, the method

condition() should return the logical value of condition. Otherwise, it returns true.

We note that these steps are conducted automatically and no effort to spend for coding a DEVS atomic model specified using graphics.

Concerning DEVS coupled models done also with graphics, the user who is using the tool defines only a structural model in which ports are connected. However for simplicity, we ignore the function select. So in the computational model, we map only the port couplings EIC, EOC and IC into arrays in which the index represent the sources of couplings and the elements of array represent the targets. These arrays are stored in a serializable class, and then the simulator uploads this structure to route messages when simulation starts.

4.4 Limits of graphical DEVS representations

With LSIS_DME some modelling difficulties may occur when DEVS atomic models require complex objects to define state variables (except for integer, float and symbolic variables), or the phases of the system cannot be specified. To declare state variables as complex objects, the user can access to the formatted file that represents the Java code of the system modelled after had generated it with the editor, adds the state variables ignored in the graphical model (not allow to be defined with the dialog box), and finally saves the modifications and recompiles the program to generate the DEVS application.

The second difficulty occurs when the model to simulate is a non-phase-based model. This category of models is called implicit state-based model in which the user tries to describe the event-driven behavior through the DEVS functions without phases. In this case, the user implements the corresponding specification using Java code. However to simplify the coding step, we advise the user to define the external view of the DEVS atomic model i.e. the input and output ports with the editor. Then complete its internal view (the description of DEVS functions) in the formatted file.

4.5 Simulation kernel

The interpretation of the behaviour of a DEVS model is given with the DEVS conceptual simulator. It consists of processors that represent a root coordinator, coordinators that are associated to DEVS coupled models and basic simulators that capture the behaviour of DEVS atomic models. A set of messages are exchanged between processors grouped into rising and falling messages. These messages are:

- i-message that activates the init state and its actions,
- x-message that informs the simulator about an external event arriving and allows the fire of an external transition,
- *-message is a scheduled event according to the time life of the current state that causes the execution of the output function and the corresponding internal transition change,
- y-message that specifies the output function result, and
- d-message that expresses the fact that the x-message or*-message was treated by the simulator.

To enhance the simulation performances, Kim et al. and Zacharewicz et al. proposed to flatten the simulation processors by transforming DEVS hierarchical models into non-hierarchical ones [Kim et al., 2000][Zacharewicz et al., 2005]. This transformation is realized progressively along the modelling step in LSIS_DME. Since the user defines a new coupled model, we save both the hierarchical and non-hierarchical models. The first one is used for the modelling requirements, in order to keep the user hierarchy specification. The second (non-hierarchical) model defines the layout of the DEVS simulator. In addition, we replace the root and sub coordinators with a unique processor named Local Coordinator (see figure 8) in which the coupling arrays (EIC, EOC and IC) are associated. However we keep the basic simulator algorithm as defined by Zeigler in [Zeigler et al., 2000]. The atomic simulator invokes the functions of a DEVS atomic model through the object class *Model* that defines both the external and internal transition and output functions implemented via internal classes.

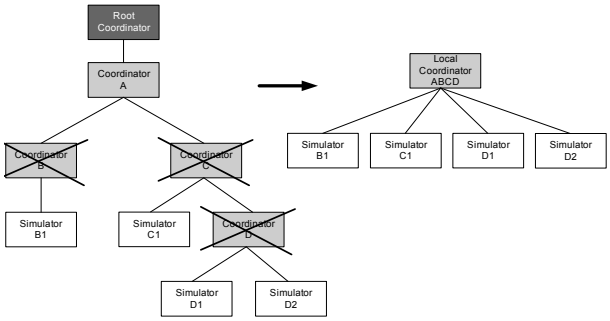


Figure 9 : Put in flat the DEVS hierarchical simulator

We believe that the simulation process may be improved by exploring technique from software engineering and parallel computation. Also a formal analysis of the simulation process based on complexity of the classical and flatten simulators should give a clear answer and confirm whether or not the experimental results realized at this subject.

5 AN ACADEMIC EXAMPLE: THE LAMP-USER INTERACTION

Let us consider a lamp controlled per a human user. He switches on the lamp since he perceives that it is switched off. We estimate the reaction delay of the user to 2 u.t (units of time). The light is maintained about 5 u.t, then the lamp switches off.

From this specification we obtain the DEVS models shown on the below figure.

HumanUser= (X, Y, S, δ_{int}, δ_{ext}, λ, D) X = {(In, OFF)} Y = {(Out, ON)} S = {perceived, reacted} δ_{int}(aperçu)=réagi δ_{ext}(reacted, e, In ? OFF) = perceived λ(perceived) = (Out !ON) D(perceived) = 2 D(reacted) = infinity	Lamp = (X, Y, S, δ_{int}, δ_{ext}, λ, D) X = {(In, ON)} Y = {(Out, OFF)} S = {switched_off, switched_on} δ_{int}(switched_on) = switched_off δ_{ext}(switched_off, e, In ?ON) = switched_on λ(switched_on) = (Out !OFF) D(switched_off) = infinity D(switched_on) = 5
HumanLampSys = (X, Y, D, {Md | d∈D}, EIC, EOC, IC) X = Y = {} D = {HumanUser, Lamp} EIC = EOC = {} IC = {((HumanUser, Out), (Lamp, In)), ((Lamp, Out), (HumanUser, In))}	

Figure 10 : The different DEVS models of the user-lamp interaction

Next, we design these models in LSIS_DME incrementally. The following figure shows explicitly the DEVS atomic model of the lamp and the simulation trace. The two other models (user and coupled ones) are saved in the library *HumanLamp* viewed at the left side of the tool. When the user attempts to save the models, the tool verifies the syntax of the described models before saving them. If there is no error, the models are saved and the corresponding object code is generated. Otherwise, this generation fails and error messages are displayed to the modeler.

The example shows us how modeling and simulation using LSIS_DME is easy to do. Animations of atomic models are possible by starting simulation in step per step mode. This allows making verification and validation by including the modeler in this process.

Note that we have integrated LSIS_DME in simulation courses dispensed at Paul Cezanne University. We identified important impact and advantages on courses and students too:

1) the DEVS courses give a training part that offering to students to apply their academic knowledge without coding effort,
2) enhance the student skills on DEVS M&S by stating that describing a model is an art and simulation is a mechanical process,
3) learn to how making decision on simulation results and allow to students to guide the verification and validation process, and
4) by forming students in discrete event simulation, we make this discipline popular in industry.

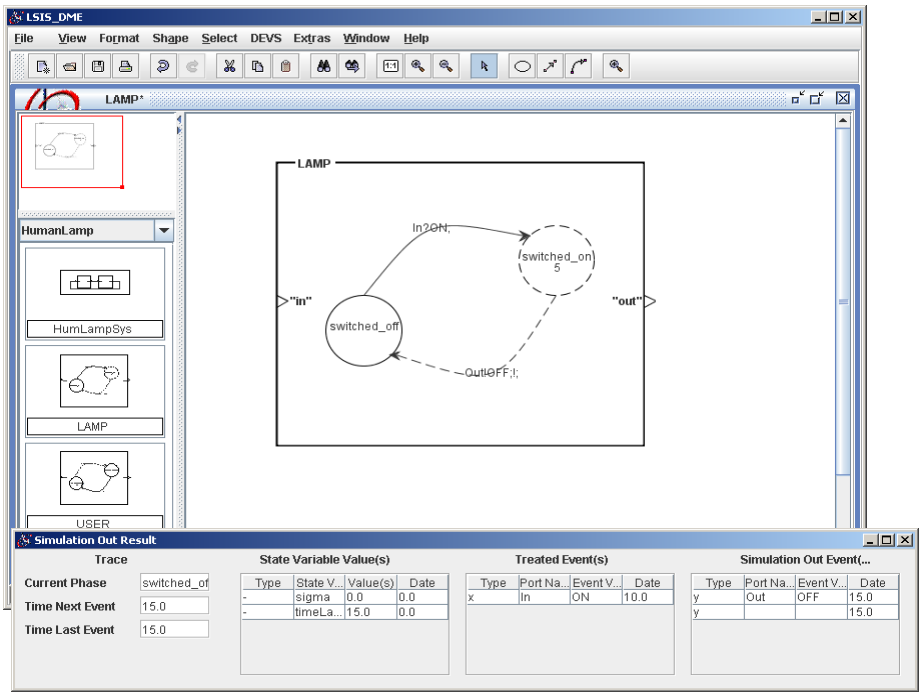


Figure 11 : LSIS_DME GUI for DEVS Atomic models

6 CONCLUSION

In this paper we give different forms of coding DEVS models according to the literature of FSM design. Therefore, we identify three ways to implement a classic DEVS behavior: 1) using the switch case statement, 2) using a tabular (matrix) form and 3) using the object paradigm. Based on this paradigm, we propose different models (class diagrams) to design DEVS coupled and atomic ones. Next, we define an approach to generate automatically code from DEVS user models. These user models are done using the graphical user representations of DEVS (arrows to model transitions, box to encapsulate models, etc.). From these specifications, we generate an object code based on the proposed DEVS class diagrams. Then, the simulation engine loads the generated object code to interpret the target model.

Knowing the advantages of our approach, we developed an environment based on Java language. This environment is totally graphic and all DEVS elements are manipulated with "drag & drop" technique to define DEVS

user models. This environment is useful and helpful to student users who like to enhance their skills on modeling. In fact we avoid them, thanks to this environment, a programming step of models that requires skills on programming language (computer science). From a technical view, the simulation process is faster than existing tools based on XML that provide to users to model DEVS specifications. Because in the case of these tools, the simulation process generates trajectories using the XML descriptions and not a byte code that speeds up the simulation. Still to show it by giving performance comparison of each simulation conducted on different tools and approaches.

In the near future, we aim to conduct a research work on computing the complexity of different forms of simulation (hierarchical and flatten ones). This work will allow discussing formally and will mark another step further the experimental results discussed in [Wainer, 2002], [Zacharewicz et al., 2005] and more recently [Jafer, 2011]. In the other hand, we will re-design the environment to be an IDE (Integrated Development Environment) like Eclipse © and AnyLogic ©. In perspective to give to the user a unified framework in which he describes graphic models and could analyze the generated code that correspond to the described models and to overtake the limits of DEVS graphical models.

REFERENCES

Adamczyk, P. 2003. The Anthology of the Finite State. In Proceedings *Pattern Languages of Programming conference*, PLoP.

Banks, J., Carson, J.S., Nelson, B.L. and Nicol, D. M. 2000. *Discrete event system simulation*. Third edition Prentice Hall.

Giambiasi, N., Escudé, B., and Ghosh, S. 2001. Generalized Discrete Event Simulation of Dynamic Systems", in: SCS Transactions: 18(4), p. 216-229

Hamri, M., Zacharewicz G. LSIS-DME: An environment for modeling and simulation of DEVS specifications. in: AIS-CMS International modeling and simulation multiconference, Buenos Aires - Argentina, February 8-10 2007, pp. 55-60, ISBN 978-2-9520712-6-0.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *International journal science of computer programming.* 8(3), pp. 231-274.

Jafer, S., Wainer, G. Global Lookahead Management (GLM) Protocol for Conservative DEVS Simulation. *Proceedings of DS-RT 2010*, Virginia, USA. 2010

Kim, K., Kang, W., Sagong, B., and Seo, H. 2000. Efficient distributed simulation of hierarchical DEVS models: Transforming model structure into a non-hierarchical one. *IEEE XXX*, p. 227-233.

Niaz, I., and Tanaka. J. XXXX. An object-oriented approach to generate java code from UML Statcharts. *International journal of Computer & Information Science*, vol. 6(2), p. 83-98.

Song, H. S., and Kim, T. G. 1994. The DEVS framework for discrete event systems control. *Proceedings of AI, Simulation and Planning in High Autonomy Systems conference*, Gainesville, FL, USA, 1994.

van Gurp, J. and Bosh, J. 1999. On the implementation of finite state machines. *Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications*, IASTED/Acta Press, Anaheim, CA, pp. 172-178.

Wainer, G. 2002. CD++: a toolkit to develop DEVS models. *Software, Practice and Experience*, Wiley, 32(3), p.1261-1307.

Zacharewicz, G., Giambiasi N. and Frydman C.: "Improving lookhead computation in GDEVS/HLA compliant", *Proceedings IEEE DS-RT conference*, Ottawa Canada October 10-12 2005, pp. 272-282.

Zeigler, B., Praehofer, H. and Kim T. G. 2000. *Theory of modeling and simulation.* second edition Academic Press.