Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, eds.

AUTOMATIC GENERATION OF OBJECT-ORIENTED CODE FROM DEVS GRAPHICAL SPECIFICATIONS

M. Hamri

LSIS UMR 7296
Aix-Marseille University
Avenue Escadrille Normandie Niemen
13397 Marseille cedex 20, FRANCE

G. Zacharewicz

IMS-LAPS CNRS
Bordeaux University
351 Cours de la Libration
33405 Talence cedex, FRANCE

ABSTRACT

The paper presents an approach to automatically generate object-oriented code from DEVS graphical model specification. Afterward the generated DEVS code is given to the LSIS_DME DEVS simulator to execute the corresponding behavior. This research is driven by the idea that the user of M&S, even not computer scientist and/or beginner in formal modeling like DEVS can increase his trust in the models he creates and the simulation results he is able to obtain due to the fact he is directly involved at the modeling stage and not anymore interfaced with an intermediate actor (modeler or programmer expert) that would interpret user requirements in the modeling and simulation activities. Using the proposed tool with its user friendly framework and appropriate graphical items, the user is capable to skip learning installation set up and user manual to rapidly develop his own DEVS models, to carry out simulations and to analyze them.

1 INTRODUCTION

Discrete event Modeling and Simulation (M&S) is one of the most 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 misunderstandings 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.

On 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. So the developed code is pattern-driven and model-driven specifications. This style of programming was popularized by the gang of four in the book: element of reusable object: 23 design patterns (Gamma et al. 1995) to promote the well-practices of an object programming. Some of these design patterns are useful for coding DEVS elements like: the composite pattern to code DEVS coupled models and the structure of the simulator or the observer one to exchange messages between DEVS process: root, coordinators and simulators (Hamri and Baati 2010).

Besides programming, discrete event simulation formalisms support a graphical modeling in addition to the symbolic one. For example Statecharts (Harel 1987) are purely graphics. States are modeled with opaque rectangles and transitions with directed arcs. Using such 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 items, the user becomes a modeler capable of specifying his system using a sound formalism.

However skills on programming or programmers still indispensable once model functionalities could not be supported by the graphical representation. In this case, may be programming the model from scratch could be a solution. More better, completing the graphical model afterwards generating code with programmed functionalities represents a flexible and scalable modeling approach likewise: COSMOS, CD++Builder, etc.

In this work we propose an approach that allows generating code automatically from DEVS graphical models. This will constitute a good and user friendly framework for modeling and simulation of DEVS specifications, in which the user does not need knowledge and skills 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 and enhances his skills in modeling.

The paper is organized as follows: the section 2 gives a recall on DEVS formalism and existing software tools in the field. The section 3 illustrates the LSIS_DME approach and highlights the code automatic generation technique from DEVS atomic models specified from the LSIS_DME editor. The section 4 discusses a case study in which we emphasize the limitations of the graphical modeling using the editor and how to overcome them. Finally, we conclude on the LSIS_DME approach and our future works.

2 STATE OF THE ART ON DEVS

2.1 DEVS Formalism

According to the literature on DEVS (Zeigler et al. 2000) the specification of a discrete event model is a structure, M, given by: $M = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D)$, 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 $D : S \rightarrow R^{+} \cup \infty$ represents the maximum length or the lifetime 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.

In Praehofer and Pree (1993), the authors introduced the concept of phase to partition the domains of state variables. The benefit of this approach is to get a graphical description of the corresponding atomic model. The phases are linked with internal and external transitions. For each internal transition, the user specifies the output ports and events to send out, eventually a condition and the next value of each state variable. The same definition is applied to each external transition with the difference that the user specifies the input ports and events.

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: $MC = (X_{MC}, Y_{MC}, D_{MC}, M_{d|d \in D}, EIC, EOC, IC, Select)$, where

- X_{MC}: set of external events.
- Y_{MC}: set of output events.
- D_{MC}: set of components names.
- M_d: DEVS model named d.
- 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 Existing DEVS Software Tools

In the literature of discrete event M&S, there is about a hundred of tools in the field. This 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 known by the DEVS community. In the following we give a brief description of the most popular tools:

ADEVS (ADEVS 2012) is the oldest DEVS tool developed in C++ by the Arizona university. It is an ad hoc simulator in which the user requires skills in programming to code his models. DEVS abstract classes should be extended to define atomic and coupled models; then, the user lunches the simulation. The tool automatically designs the kernel simulator according to the DEVS structure to simulate.

DEVSJAVA (DEVSJAVA 2012) is a Java framework in which the kernel simulator is ADEVS. It supports also modeling and simulation of DEVS with variable structures. The COSMOS environment integrates this framework and provides a pretty interface to define coupled and atomic models. The user defines in graphical way coupled DEVS. He reuses basic components by the drag and drop technique, then he connects them with links to define the port connections over reused models. However, at atomic level, the user should implement the corresponding DEVS behavior in Java (in our opinion the user has not enough skills to program his atomic models). He should declare the state variable then implement the DEVS interface of an atomic model by implementing the methods: `delint()`, `delext(double e, message x)`, `out()`. Through the use of the method `holdIn(String phase, double duration)`, the user defines for each state its lifetime. To enhance the user interface of this tool and presents the output simulation results in attractive and readable way, DEVS-Suite allows tracking and monitoring simulation results (running step by step or speed up the simulation, plotting output simulation, saving trace simulation, etc.).

CD++Builder (Wainer 2002)(Bonaventura et al. 2010) is a DEVS modeling and simulation environment that integrates interesting features and facilities for the user. It allows modeling and simulation of other DEVS formalisms (cell-DEVS, Quantized-DEVS, etc). It provides a DEVS graphical editor to model coupled and atomic models, and to encapsulate them through components for further reuse. It adopts a code automatic generation technique to generate a textual formatted file. The CD++simulator loads the file then produces the specified behavior by executing the different functions $\delta_{int}, \delta_{ext}, \lambda$ and D from the formatted file.

Other DEVS tools are dedicated for specific areas like: VLE (Quesnel et al. 2009) is a C++ modeling and simulation framework that integrates heterogeneous models from different scientific fields. This integration is based on the agent paradigm. JDEVS is the Java implementation of a DEVS formal framework proposed in Filippi and Bisgambiglia (2003). It supports multi-modeling paradigms based on DEVS. It ensures the interoperability among the reused components. This tool may be connected to a Geographical Information System to visualize simulation results.

By looking to these DEVS tools, we deduce that all of them automate the making of the kernel simulator and provide a user interface to define DEVS coupled model in graphical way. DEVSJAVA lets the user to program the DEVS behavior by implementing the DEVS atomic interface. With this style of modeling the user has a great freedom to implement complex behavior. However, it presents the lack that the user should have skills in programming and debugging. The CD++Builder approach is more convenient for users. It consists on providing a full user interface to specify coupled and atomic models. The user defines atomic in graphical way without skills in programming. A useful approach that avoids spending time in programming. In addition this approach is based on the generation of a textual formatted file from a DEVS graphical model. Nevertheless, the CD++ toolkit does not integrate a process (technique) to generate automatically a compilable code from the target DEVS atomic model specified with graphics. A limitation that the authors note in Bonaventura et al. (2010). Secondly the template that the CD++Builder provides, is an interface that the programmer should complete to implement the expected DEVS behavior, following comments introduced in the template. It is a way to speed up the programming process and to avoid errors. However the produced code is a less object-oriented one. In fact, there is no guidelines to objectify the items of the target atomic model (states, transitions, etc.) because often, the CD++ programmers and others use the switch-case statement in cascade to implement a such behavior. A constraint due to template design that forces the programmers to use this statement.

In our M&S approach, we take back the several points discussed above and we enhance the code automatic generation process by adopting the DEVS-phase design pattern, specific for implementing DEVS models specified with phases. In this design pattern, we objectify phase and transition into classes that extends abstract ones (the ports are only object instances) to get a modular and structured code. This allows to get a code more abstract and structured around classes. Each generated DEVS class behaves independently from others according to user model. Consequently, all phase and transition classes have their own methods (action, condition, etc.). With this design, we ensure the encapsulation of DEVS elements and the user may easily update the DEVS code if necessary. Due to the fact that each DEVS element is implemented with a class, the user may update a specific phase or transition to add new functionalities in object paradigm independently from other classes. This is a main advantage for the maintainability of code that the DEVS-phase design pattern guaranties.

In addition, our approach is based on a direct mapping from the DEVS atomic model into a Java code: each active or passive phase, internal or external transition is automatically transformed into `ActivePhase`, `PassivePhase`, `InternalTransition` and `ExternalTransition` classes respectively. To note this generated DEVS code is compilable and the Java Virtual Machine interprets it to generate byte code without amendment.

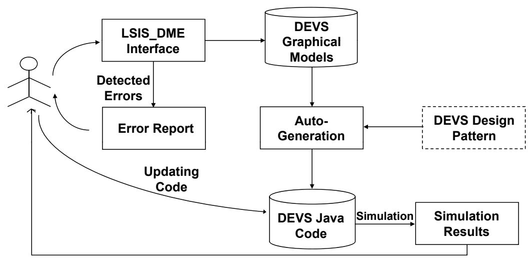
3 LSIS_DME: AN ENVIRONMENT FOR M&S OF COMPLEX SYSTEMS

The LSIS_DME (Lab of Sciences of Information and Systems_DEVS Models Editor) tool is developed by the LSIS team since 2005 to enhance the process of M&S of discrete event systems and make easier the description of models than existing tools (Hamri and Zacharewicz 2007). The environment was designed according to the following user requirements: allow a graphical modeling since the design of atomic models until coupled ones by drag & drop, to avoid the programming step and focus on DEVS modeling step. Consequently, the user, with no difficulty, is able to describe DEVS computerized models, to analyze them by simulation and avoids consuming time in coding and handling code due to modifications that often occur to enhance the specifications.

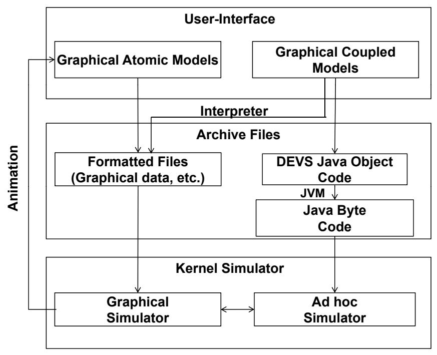
Secondly, LSIS_DME generates automatically a code from the DEVS graphical user specification to the simulation kernel. The generated code is an object-oriented code in which we map the DEVS atomic model into a class that keeps the behaviors of the specification and the DEVS coupled specification that specifies DEVS submodels and port connections into a logical structure. In fact, the generated code respects the DEVS-phase design pattern in which we objectify states and transitions. The associated process extends the class diagram according to the DEVS graphical user-specification: each state and transition is mapped into a specific class and not with an object instance; however the submodels and port connections are object instances. To note that mapping state and transition to abstract classes is not an arbitrary choice. This allows to get a code more abstract and easy to update with action and condition of transitions.

The LSIS_DME embeds the Java compiler to produce the byte code from the generated code. Afterward, it calls the kernel simulation with the byte code to make simulations and generate the output report visualized through a graphical window. The figure 1 shows the different steps and architecture of LSIS_DME. Each step calls to a process that produces the output to start the next step.

The tool can be inscribed in the MDA (OMG 2003) approach steps. The graphical modeling level is independent from any computer consideration. It can be seen as a CIM model. Then the JAVA code generated by the tool thanks to a model transformation according to the DEVS-phase design pattern can be seen as a PIM since the JAVA code is portable on main platforms. Then the code can be specialized to add different features. This step can be the PSM level. At the end, the code is compiled to be run on the machine it is destined.



(a) The LSIS_DME approach.



(b) The LSIS_DME architecture.

Figure 1: The approach and architecture of LSIS_DME.

3.1 User Specification of a DEVS Atomic Model

Some DEVS tools introduce the concept of action and condition related to transition. An action is a primary operation on a state variable. It may consist on a simple affectation of a value or a result of a mathematical operation (x = 2, x = 2*elapsed_time, etc.). An action in addition to the transition computes the next state in which all values of state variables are identified. However, a condition is a logical expression made on state variables with the logical and/or comparison operators ($\&\&, ||, <, >, ==$ and $!=$). The condition is a part of the model behavior. In fact, a transition is fired only if the corresponding event occurs and the associated condition is true. Otherwise, there is no state change to do.

At encapsulation level, the port declaration requires specifying the name and definition domain. At user interface, we restrict this domain to integer, float, interval of integers or floats, and finite set of strings. In addition to these concepts states and transitions are graphically specified. For this purpose, a sub class of DEVS models has been introduced in Zacharewicz et al. (2005). In this class of models a particular state variable entitled phase is explicitly used to represent graphical states. A phase is defined by the name and duration that expresses the lifetime. This duration may take the form of a fixed value or a combination of numerical values of state variables at a state change.

To note that a phase could be an active or a passive one based on its duration. The graphical representation is customized according to the phase type. We represent active and passive phases with continued and discontinued circles respectively. In case of a phase with a function duration, by default we represent it with a continued circle. At the end, a transition is decomposed into an internal and external transitions. We represent them with dark and dashed arrows on which we show output and input ports respectively and the associated condition and action. The figure 2 shows the mapping of a DEVS model example into classes according to the data user filled from dialog boxes.

In the left of the figure 2, we have a simple example with three states $\{S, T, U\}$, two external transitions $\{S \rightarrow T, S \rightarrow U\}$ and one internal transition $\{U \rightarrow T\}$ and its equivalent DEVS classes automatically generated from the example. The reader may believe that the example does not need an extension (with inheritance) of DEVS abstract classes, this is true. However to customize the condition and action of each transition and the lifetime function of each state, we should, according to the user specification, extends the DEVS abstract classes, then we fill them with data and functions.

3.2 User specification of a DEVS coupled model

A specification of a DEVS model consists on choosing DEVS basics models (submodels) that define the structure of the current model. Then, the user defines the different couplings over the chosen models. This modeling may be conducted in declarative or graphical way. Many DEVS tools adopt a graphical modeling

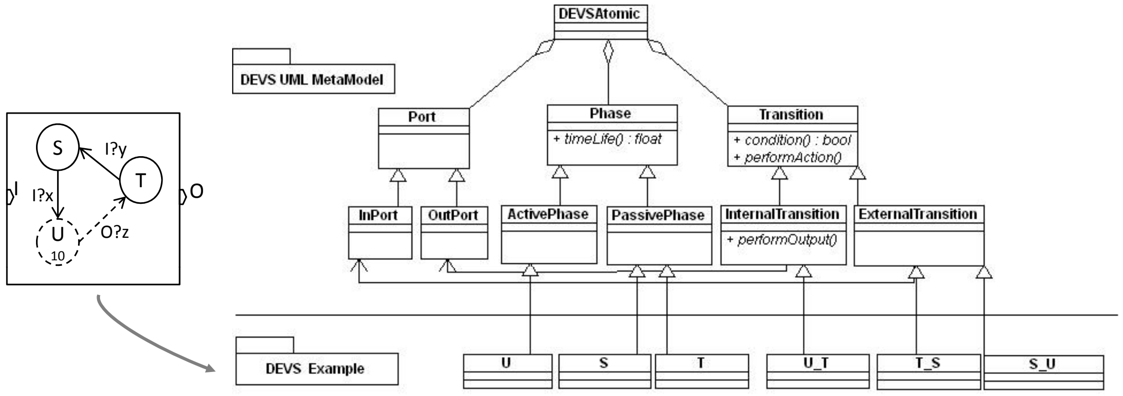


Figure 2: Class diagram of an example based on the DEVS-phase pattern.

by dragging and dropping the basic models and connecting port. In our tool we adopt the same modeling way to construct DEVS coupled models. We structure these models according to the class diagram shown in figure 3. In fact, we integrate in our design the composite pattern (Gamma et al. 1995) that designs an object in hierarchical and recursive way.

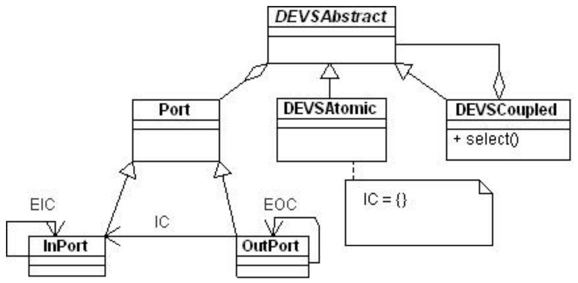


Figure 3: Class diagram of a DEVS coupled model based on the composite pattern.

However, in LSIS_DME we do not allow to the user to redefine the function select. In fact, by default, the tool chooses the first declared model in case of a conflict between DEVS basic models. This presents a limitation of our tool in case of modeling concurrent systems like logical gates, microcontrollers, etc.

3.3 Detection of errors

In the literature of DEVS, verification consists on checking the DEVS syntax correctness. In fact, the DEVS atomic and coupled are formalized with a Backus Naur Form grammar. This is the case of DEVS language definition proposed by Zeigler et al. (2000). All known DEVS tools are based explicitly or implicitly on a BNF grammar. A such checking allows to detect for:

1) Atomic models: i) check in case of a finite state machine, if all state are linked over them by transitions; ii) check for each passive state there is no internal transition that occurs and for each active state holds at least an internal transition.

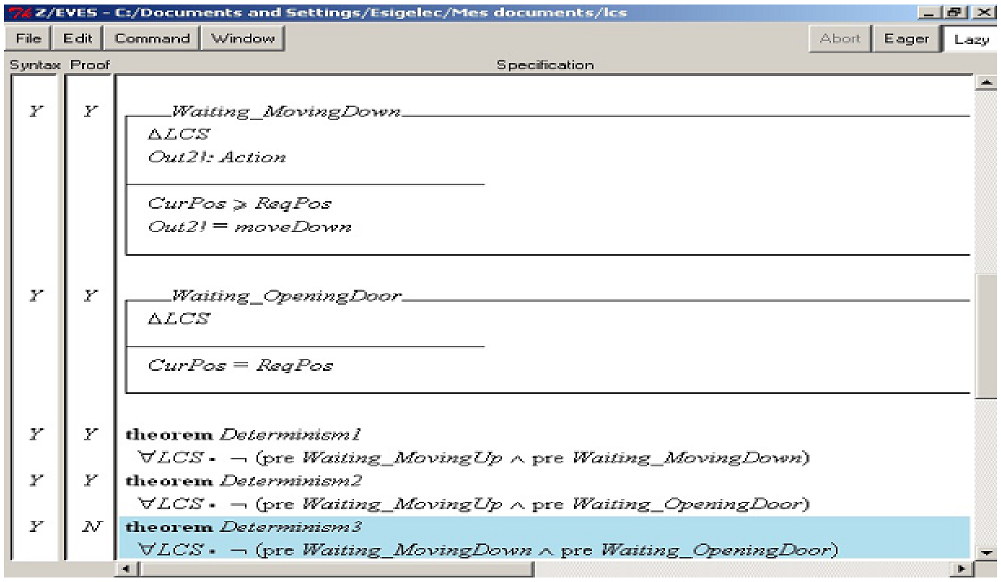
2) Coupled models: check port connections that means to verify for each port is connected to another one with avoiding that an output port of an atomic model is connect to another input port of the same model.

Nevertheless, this verification concerns only static features. In fact, logical errors cannot be detected only when the simulation runs. However, in Trojet et al. (2009) and Trojet (2010) we distinguished the logical errors into two classes:

1) At a DEVS atomic model: i) the user may define a bad initialization of the default state; ii) while conditions are introduced at transitions, the user should take care for each occurring external and internal events. In this case if the condition is false, so the model is blocked and there is no state change to do.

2)At a DEVS coupled model: i) there is no way to verify if the port connections among models are correctly defined. In other words, the user should take care if the definition domain of a port is a subset of or equal-set to that to which is connected; ii) more evident, the user should avoid a recursive definition of a DEVS model by reusing the DEVS in current specification.

The LSIS_DME calls to an original approach that consists on transforming a DEVS model into a Z specification to check such errors (see figure 4). Then, the Z/EVES prover (Saaltink 1997) checks the following theorems: determinism, ambiguity, completeness and port connection. The approach is computerized through the DEVS-compiler (Trojet 2010). In fact the user, may start a formal checking on its DEVS model before gives it to simulation. So the trustiness in the checked model increases and the user could lunch the simulation with a trusted model.


Z/EVES - C:/Documents and Settings/Esigelec/Mes documents/lcs
File
Edit
Command
Window
Abort
Eager
Lazy
Syntax
Proof
Specification

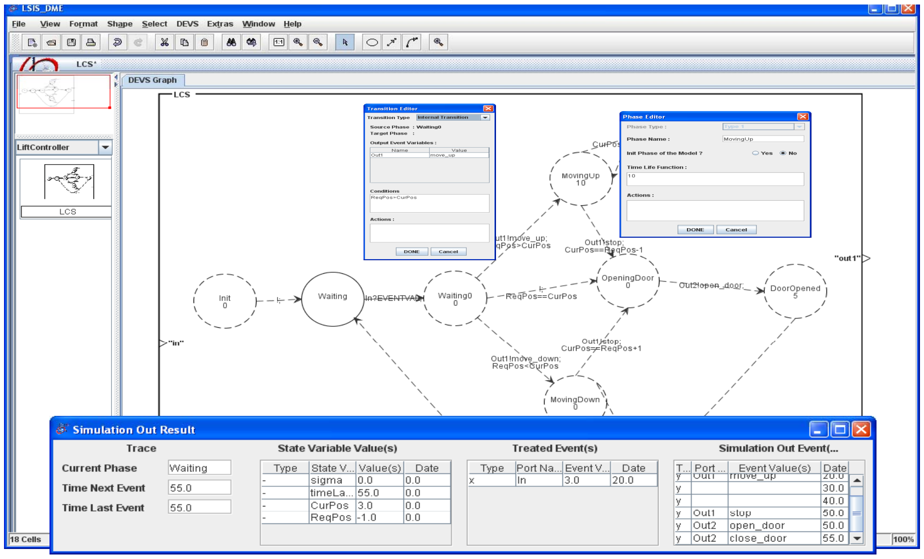
```
Y  Y   Waiting_MovingDown
       ΔLCS
       Out2!: Action
       CurPos ≥ ReqPos
       Out2! = moveDown

Y  Y   Waiting_OpeningDoor
       ΔLCS
       CurPos = ReqPos

Y  Y   theorem Determinism1
        ∀LCS • ¬ (pre Waiting_MovingUp ∧ pre Waiting_MovingDown)
Y  Y   theorem Determinism2
        ∀LCS • ¬ (pre Waiting_MovingUp ∧ pre Waiting_OpeningDoor)
Y  N   theorem Determinism3
        ∀LCS • ¬ (pre Waiting_MovingDown ∧ pre Waiting_OpeningDoor)
```

Figure 4: The Z/EVES window showing that the model of the below example does not respect determinism property. In fact the model defines two possible internal transitions from the phase Waiting. An error that the simulator detects if and only if the corresponding scenario is simulated.

4 A CASE STUDY: THE LIFT CONTROLLER

4.1 A Graphical Modeling of the Lift Control System

A lift control system (LCS) is required to control a single lift in a multi-floor building. Requests for the lift can either be made by someone waiting on a floor of the building, or by another travelling in the lift. The task of the LCS is to satisfy these requests by moving the lift to the required floor. If the lift is empty, it serves always the first request. Otherwise, it serves the request of the customer inside. There are no waiting requests, only one request is satisfied. The lift still in the final position reached until it moves to satisfy a new request. The DEVS model of the LCS is described in the figure 5 using LSIS_DME.



Transition Editor
Transition Type: Internal Transition
Source Phase : Waiting0
Target Phase :
Output Event Variables :

Name	Value
Out1	move_up

Conditions
ReqPos>CurPos
Actions :
DONE Cancel

Phase Editor
Phase Type : Type 1
Phase Name : MovingUp
Init Phase of the Model ? Yes [✓ No]
Time Life Function : 10
Actions :
DONE Cancel

Simulation Out Result

Trace

Current Phase	Waiting
Time Next Event	55.0
Time Last Event	55.0

State Variable Value(s)

Type	State V...	Value(s)	Date
-	sigma	0.0	0.0
-	timeLa...	55.0	0.0
-	CurPos	3.0	0.0
-	ReqPos	-1.0	0.0

Treated Event(s)

Type	Port Na...	Event V...	Date
x	In	3.0	20.0

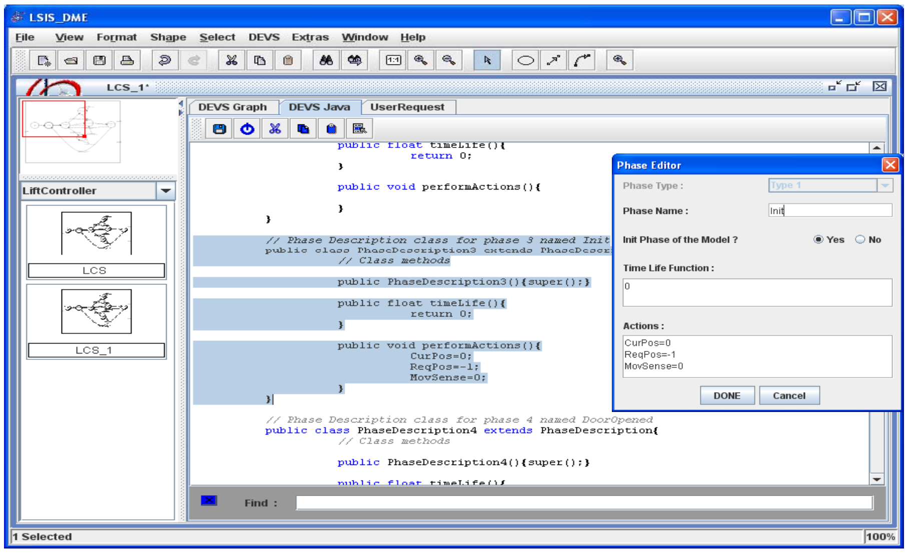
Simulation Out Event(...

T...	Port ...	Event Value(s)	Date
y	Out1	move_up	20.0
y			30.0
y			40.0
y	Out1	stop	50.0
y	Out2	open_door	50.0
y	Out2	close_door	55.0

Figure 5: The correct DEVS atomic model of the LCS with LSIS_DME.

First, the user defines the name of DEVS atomic model and input and output port numbers. The tool creates a box with the different ports that the user names. Then he drags and drops inside this box from the tool palette all required states and transitions for the model and fills data of each item.

Once the model is checked, the user saves it and the tool generates the corresponding object code (see figure 6). Then, the JVM produces the corresponding byte code. The simulation kernel lunches the byte code, when the user decides to run the simulation process.



```
            public float timeLife(){
                    return 0;
            }

            public void performActions(){

            }
    }

    // Phase Description class for phase 3 named Init
    public class PhaseDescription3 extends PhaseDescr
            // Class methods

            public PhaseDescription3(){super();}

            public float timeLife(){
                    return 0;
            }

            public void performActions(){
                    CurPos=0;
                    ReqPos=-1;
                    MovSense=0;
            }
    }

    // Phase Description class for phase 4 named DoorOpened
    public class PhaseDescription4 extends PhaseDescription{
            // Class methods

            public PhaseDescription4(){super();}

            public float timeLife(){
```

Phase Editor
Phase Type : Type 1
Phase Name : Init
Init Phase of the Model ? [✓ Yes] No
Time Life Function : 0
Actions :
CurPos=0
ReqPos=-1
MovSense=0
DONE Cancel

Figure 6: The object code of the LCS model generated automatically with LSIS_DME.

4.2 A Scalable Modeling Approach with LSIS_DME

The graphical DEVS modeling could be an obstacle for some models in which a complex function should be implemented and then used by the DEVS behavior (ex: create a connection to a database to store output events at run time). In fact, a such function cannot be expressed inside an action of a transition, it should be designed with a programming language. Fortunately, LSIS_DME approach is based on an automatic generation of Java code from the graphical DEVS model. The user may access to the corresponding code and complete it with the new features that the tool could not keep them through the graphics. Still to guaranty that the modification is correct and does not change the specified behavior.

Let us consider the LCS specified above with a memory variable that stores all user requests. However, the strategy of request satisfaction changes, it consists on reducing the change of moving. According to the moving sense and the current floor, the LCS chooses the request to satisfy. Consequently, the LCS should store each user request and then decide which one to satisfy among those stored.

The request satisfaction algorithm could not be implemented inside an action of the LCS model. In fact, to avoid the overloading of the concerned actions with a dozen of code lines, we propose to encapsulate this algorithm into an external class `UserRequest`. Then, the external transition, that the LCS model firing, passes to this class, the following parameters: current floor, moving sense and stored user requests. The sequence diagram shown in the figure 7, illustrates the interaction between the external transitions that store the current request and ask the new `ReqPos` with the `UserRequest` class that implements the user request satisfaction algorithm.

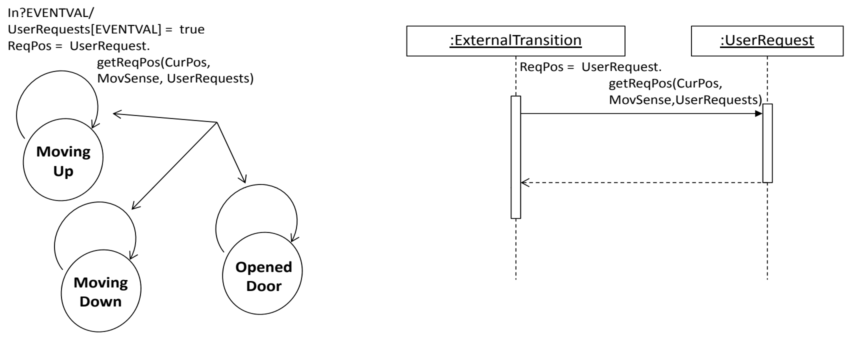
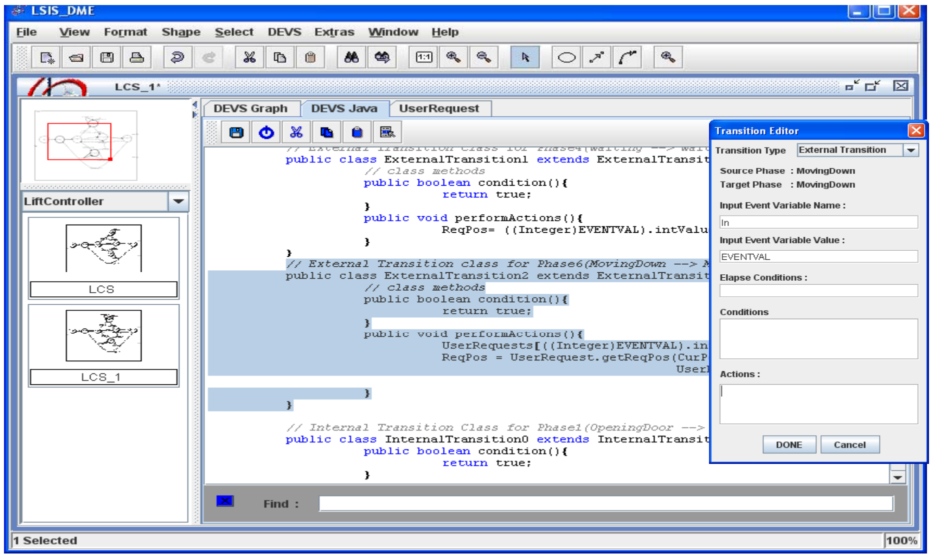

In?EVENTVAL/
UserRequests[EVENTVAL] = true
ReqPos = UserRequest.getReqPos(CurPos, MovSense, UserRequests)
Moving Up
Moving Down
Opened Door
:ExternalTransition
:UserRequest
ReqPos = UserRequest.getReqPos(CurPos, MovSense,UserRequests)

Figure 7: The modification introduced in the LCS model through the external transitions.

Based on these modifications, we develop a new controller model under the name LCS_1 (figure 8).



```
public class ExternalTransition1 extends ExternalTransit
            // class methods
            public boolean condition(){
                    return true;
            }
            public void performActions(){
                    ReqPos= ((Integer)EVENTVAL).intValu
            }
}
// External Transition class for Phase6(MovingDown --> M
public class ExternalTransition2 extends ExternalTransit
            // class methods
            public boolean condition(){
                    return true;
            }
            public void performActions(){
                    UserRequests[((Integer)EVENTVAL).in
                    ReqPos = UserRequest.getReqPos(CurP
                                                    User

            }
}

// Internal Transition Class for Phase1(OpeningDoor -->
public class InternalTransition0 extends InternalTransit
            public boolean condition(){
                    return true;
            }
```

Transition Type: External Transition
Source Phase : MovingDown
Target Phase : MovingDown
Input Event Variable Name : In
Input Event Variable Value : EVENTVAL
Elapse Conditions :
Conditions
Actions :

Figure 8: A hand-update of the external transition.

The same approach is supported by internal transition, state duration and output function. We can call methods from external classes to compute a state variable or an output event of the DEVS model. In fact, the user gets more freedom with object code than graphics to define the DEVS model. However, the graphics modeling step still an important one that allows the user to define the template of the DEVS model, to avoid starting from scratch and to spend less time in coding. Thanks to the DEVS code generation process that generates from the DEVS model the corresponding Java class, the user can update the code by adding the classes that interact with the class of DEVS atomic model when the simulation runs.

5 CONCLUSION

In this paper, we pointed the automatic code generation process that LSIS_DME approach integrated. This process consists in mapping a DEVS atomic model specified graphically into a Java class. It encapsulates the different phases and transitions classes and expresses the DEVS behavior specified through data user. The user may describe complex DEVS behaviors by defining additional actions and conditions based on state variables and received events, and specifying lifetime of phases with equations based on state variables. A DEVS coupled model is specified in a graphical way by reusing stored atomic models and coupling them using graphical connections, then the tool saves the coupled model (into a formatted file) to be also reused in a modular and hierarchical approach. In addition, our approach calls to a Z verification to check a DEVS specification and to avoid logical errors that the static verification based on BNF could not detect.

The DEVS kernel simulator simulates the DEVS model based on the byte code of the generated Java classes and the formatted file that describes graphical and structural data. The simulation results are shown within a window that user can analyze.

Other features of LSIS_DME were not detailed in this paper like XML storage of DEVS models and distributed simulations using High Level Architecture (Zacharewicz et al. 2010). In fact, a bridge between this work and DEVS standardization works (Sarjoughian and Chen 2011) could be established to standardize the storage formats of DEVS models specified from LSIS_DME and the simulated code.

Currently, we are extending the tool to model and simulate DEVS models in which no phase was specified. In this case, the user should declare the input and output ports, then he codes in Java the specified model by implementing the specific interface that defines the interaction between the simulator and model. This issue will give to the user more freedom to design DEVS models according to fixed requirements like: speed up time execution, clarity of code, etc.

REFERENCES

ADEVS 2012. Accessed March. 1, 2012. http://www.sourceforge.net/projects/adevs.

Banks, J., J. S. Carson, B. L. Nelson, and D. M. Nicol. 2000. *Discrete-Event System Simulation*. 3rd ed. Prentice Hall, Inc.

Bonaventura, M., G. A. Wainer, and R. Castro. 2010. "Advanced IDE for Modeling and Simulation of Discrete Event Systems". In *Proceedings of the Spring Simulation Multiconference*. San Diego, CA, USA: Society for Computer Simulation International.

DEVSJAVA 2012. Accessed March. 1, 2012. http://www.acims.arizona.edu/software/software.html.

Filippi, J. B., and P. Bisgambiglia. 2003. "JDEVS: an implementation of a DEVS based formal framework for environmental modeling". *Environmental Modelling & Software* 19:261–274.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

Hamri, M., and L. Baati. 2010. "On Using Design Patterns for DEVS Modeling and Simulation Tools". In *Proceedings of the Spring Simulation Multiconference-Symposium Theory of Modeling & Simulation-DEVS Integrative M&S symposium*. San Diego, CA, USA: Society for Computer Simulation International.

Hamri, M., and G. Zacharewicz. 2007. "LSIS_DME: An Environment for Modeling and Simulation of DEVS Specifications". In *Proceedings of the International Modeling and Simulation Multiconference*, edited by N. G. F. Barros, C. Frydman and B. Zeigler, 55–60. Buenos Aires - Argentina: LSIS, Aix-Marseille III University.

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

OMG 2003. "OMG: MDA Guide Version 1.0.1". Technical Report 2003-06-01 edn, Object Management Group.

Praehofer, H., and D. Pree. 1993. "Visual Modeling Modeling of DEVS-based Multi-Formalism Systems based on Higraphs". In *Proceedings of the 1993 Winter Simulation Conference*, edited by G. W. Evans and M. Mollaghasemi, 595–603.

Quesnel, G., R. Duboz, and E. Ramat. 2009. "The Virtual Laboratory Environment - An Operational Framework for Multi-Modelling, Simulation and Analysis of Complex Dynamical Systems". *Simulation Modelling Practice and Theory* 17:641–653.

Saaltink, M. 1997. "The Z/EVES System". In *International Conference of Z Users on The Z Formal Specification Notation*, edited by Springer-Verlag, 75–82. London, UK.

Sarjoughian, H., and Y. Chen. 2011. "Standardizing DEVS Models: An Endogenous Standpoint". In *Proceedings of the Spring Simulation Multiconference*, 266–273. San Diego, CA, USA: Society for Computer Simulation International.

Trojet, M. W. 2010. *Méthodes de vérification et de validation basées sur la spécification Z et le formalisme DEVS*. Ph.D. thesis, Aix-Marseille III University, Marseille, France.

Trojet, M. W., C. S. Frydman, and M. Hamri. 2009. "Practical Application of "lightweight" Z in DEVS Framework". In *Proceedings of the Spring Simulation Multiconference-Symposium Theory of Modeling & Simulation-DEVS Integrative M&S symposium*. San Diego, CA, USA: Society for Computer Simulation International.

Wainer, G. 2002. "A Toolkit to Develop DEVS Models". *Software–Practice and Experience* 32 (3).

Zacharewicz, G., N. Giambiasi, and C. Frydman. 2005. "Improving the Lookahead Computation in GDEVS/HLA Environment". In *Proceedings of the IEEE International Symposium on Distributed Simulation and Real-Time Applications DS-RT*, 273–282. Montreal, Canada.

Zacharewicz, G., M. Hamri, C. Frydman, and N. Giambiasi. 2010. "A Generalized Discrete Event System (GDEVS) Flattened Simulation Structure: Application to High-Level Architecture (HLA) Compliant simulation of workflow". *Simulation* 86 (3).

Zeigler, B., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. Academic Press.

AUTHOR BIOGRAPHIES

MAAMR HAMRI is Associate Professor at Aix-Marseille University and member of LSIS UMR 7296 lab. He obtained the PhD. in 2005. His research are conducted in discrete event simulation and focused on DEVS formalisms and its extensions GDEVS and MinMax DEVS. Currently he draws near to formal specification community to bring on light the possible contribution of research works on discrete event systems to this community. He has published more than 30 articles in international conferences and journals. His email address is amine.hamri@lsis.org.

GREGORY ZACHAREWICZ is Associate Professor in Bordeaux 1 University (IUT MP). His research interests include Discrete Event Modelling (e.g., DEVS, G-DEVS), Distributed Simulation, Distributed Synchronization Algorithms, HLA, FEDEP, MDA, Short lived Ontologies, ERP, BPMN, and Workflow. He recently focused on Enterprise Modeling and Interoperability. He has published more than 40 papers in Conferences and International Journals. He is Reviewer in Conferences (Spring, Summer SCS, WinterSim,...) and SCS Simulation journals. He is involved in several French, European and Transatlantic projects. His email address is gregory.zacharewicz@u-bordeaux1.fr.