CUSTOMIZING CODE OF DEVS MODELS ACCORDING TO USER REQUIREMENTS USING LSIS_DME

M. HAMRI and R. MESSOUCI
LSIS UMR 7296- Aix-Marseille Université
Domaine universitaire de Saint Jérôme
Avenue Escadrille Normandie Niemen
13397 MARSEILLE Cedex 20
amine.hamri@lsis.org, rabah.messouci@lsis.org

Keywords: DEVS, design of atomic models, DEVS queuing system

Abstract

In this paper, we discuss two ways to code DEVS atomic models the first one is based on the switch-case statement in which no phase is described for the atomic model to simulate. The second one is based DEVS-phase design pattern in which we objectify phases and transitions. These ways of coding are integrated into the LSIS_DME tool to generate a complete compilable code of the DEVS-phase model and a partial code (template) that the user should complete in case of a DEVS atomic model without phases.

1. INTRODUCTION

In last decade, many DEVS tools have been developed to provide frameworks to model and simulate DEVS models. ADEVS (ADEVS, 2006), the oldest one, is a toolkit that suggests to the user to design DEVS models in object paradigm. DEVSJAVA (DEVSJAVA, 2012) provides a friendly-user framework to the user to make graphical DEVS coupled models and at his charge to design atomic models by coding them. Other frameworks avoid the user a coding step and propose to make both atomic and coupled models in graphical way. A process insures the transformation of graphical models into interpretable ones for the simulation process. By analyzing DEVS frameworks of the literature, we identify two categories:

1) DEVS frameworks in which the design of atomic models is let to the user. Often he should extend an abstract class or implement an interface to define the corresponding DEVS atomic model, and

2) DEVS frameworks in which a standard design of atomic models is defined to allow graphical or XML description.

However, we remark that there is no framework allowing different designs of atomic models in unique one. In our approach, we propose to catch different designs of atomic models inside a unified framework and provide to the user the more adequate design to the current atomic model to code.

The paper is organized as follows: Section 2 gives a recall on DEVS concepts and discusses the current LSIS_DME approach. Section 3 proposes two ways of software designs to help the user in coding atomic models. Section 4 integrates the discussed designs in LSIS_DME to make variable the process of code automatic generation. An example of a queuing system is shown and discussed in Section 5. Finally, we conclude on the proposed and future works.

2. RECALLS

2.1. DEVS Formalisms

A DEVS model (Zeigler et al., 2000) consists of DEVS coupled and atomic models. An atomic model is structured as follows:

$Atomic = (X, Y, S, \delta_{int}, \delta_{ext}, D)$ where

X, Y: sets of input and output ports respectively.

S: set of state variables

$\delta_{int} : S \to S$, the internal transition function. It defines the set of autonomous transitions.

$\delta_{ext} : QxX \to S$ such $Q = \{(s,e) | s \in S, 0 \leq e \leqslant D(s)\}$, the external transition function. It defines according to the occurring event x and the elapsed time e on the current state s which transition to fire.

$D : S \to \mathbb{R}^+$, for each state the function D defines the max duration in which the model still to fire an autonomous transition.

However, based on the encapsulation of atomic models, coupled ones are made. This construction is in respect with the following structure:

$Coupled = (X, Y, D, M_{d \in D}, IC, EIC, EOC, select)$

X, Y: are sets of input and output ports respectively

D: names of component and M_d: model of D component

IC, EIC and EOC: sets of internal, external input and external output couplings respectively.

$Select$: function priority between components.

2.2. LSIS_DME tool

LSIS_DME tool (Hamri and Zacharewich, 2007) is a full environment allowing DEVS and GDEVS modeling and simulation. The user defines DEVS and GDEVS in graphical way using a bottom-up approach. Firstly, he starts with modeling

atomic models, and then he stores them in his library who may share it with other users for further reuse. Secondly, he reuses these basic models (both atomic and coupled models) by drag and drop to build new coupled models. Also, these new models are stored in a library for further reuse again. The storage format of any DEVS, LSIS_DME distinguishes atomic models from coupled ones. Once, the user stores an atomic model, the tool transforms this graphical description into a Java code to express the pure DEVS (or GDEVS) behavior and a formatted file that contains structural and graphical data of the concerned model. Still the coupled model is transformed into a formatted structure according to definition of DEVS coupled formalism.
To summarize, we recall the LSIS_DME approach shown on figure below. Firstly, the user describes an atomic or coupled model by drag and drop necessary elements (state, transition or basic model). Then, he checks the current model using DEVS-Compiler to avoid logical errors (determinism, ambiguity, completeness and port coupling) before lunching simulation. Once, the user saves its model he can start simulation. Then an output simulation report is given to the user to analyze the corresponding behavior.

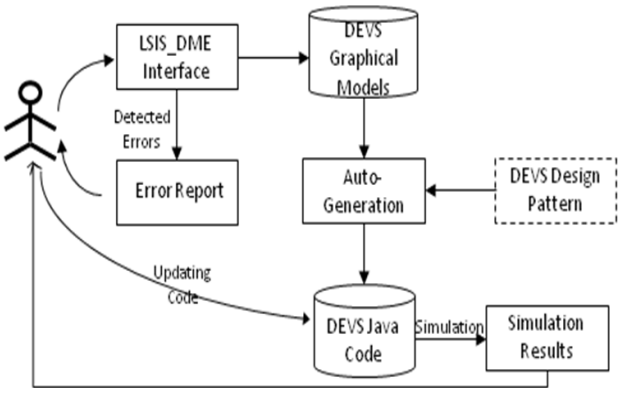

Figure 1. LSIS_DME approach

LSIS_DME allows modeling only DEVS-phase atomic models i.e. DEVS models in which the behavior is specified using phases (Praehofer and Pree, 1993). Sometimes, the user may use an infinite state machine to describe a DEVS atomic model, due to the fact he employs state variables with infinite domain. However (Honig and Seck, 2012) propose an approach based on DEVS-phase modeling called PhiDEVS. So we can imagine that any DEVS or GDEVS atomic model could be reduced to a DEVS-phase model. In addition, almost DEVS models seen in the literature are phase-based models and at least they are based on two phases: active and passive (Zeigler et al., 2000). Consequently, the DEVS-phase design stills valid to implement any DEVS models, by adopting specific design for LSIS_DME simulator. However to get an architecture to take into account different designs of atomic models, we update the communication between the simulation process and model to simulate.
In the next section, we discuss how LSIS_DME user may choose between a DEVS-phase design and an ad-hoc design based on the switchâcase statement.

3. A SOFTWARE DESIGN APPROACH TO MODEL AND SIMULATE DEVS ATOMIC COMPONENT

The abstract simulator of DEVS consists of a process that manages messages to send out or to receive from an atomic model. At design level, (Zeigler et al., 2000) define an interface to make interaction between the simulator and the atomic model to simulate. In fact, this is a contract between the user who will design the atomic model and should implement this interface. Next, the simulator acts on the model to simulate through the implemented methods to create DEVS behaviors. This strategy is employed in DEVSJAVA.

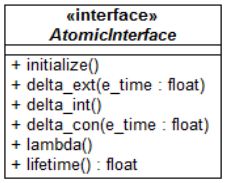

Figure 2. DEVS interface of atomic models

In LSIS_DME, we reuse this advanced design to make a structured communication between each atomic model to simulate and the simulator. In fact using such an interface, we give to the user a great freedom to design its atomic models and we guaranty interoperability between them. However two constraints are necessary to correctly build atomic models: i) the atomic model should hold a reference on the current state of the model; and ii) the DEVS atomic model should implement all methods of DEVS interface. In the following, we discuss two possible designs of DEVS atomic models.

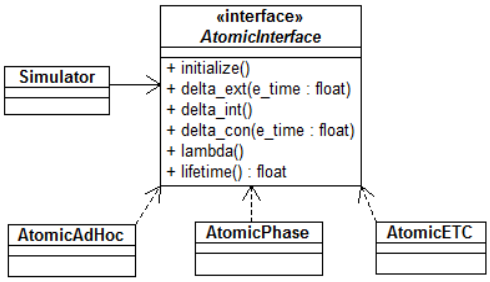

Figure 3. Communication between LSIS_DME simulator and different designs of atomic models

This architecture that distinguishes the interface of the model

to simulate from the model itself provides the possibility to integrate atomic models with new designs.

3.1. Ad-hoc design of DEVS atomic models

The ad-hoc design of DEVS atomic models consists on implementing the DEVS interface to define a DEVS component. The user may define infinite or enumerated state variables. Then based on these variables, the user defines the evolution of the model from a state to another through the methods delta_ext() and delta_int(). This design is useful to implement models like DEVS SISO models in which only an input and an output ports, both expressed with real (Zeigler and Sarjoughian, 2005).

```
public void initialize(){
phase = "passive";
sigma = INFINITY;
store = 0;
response_time = 10;
super.initialize();
}
public void Deltext(double e,double input){
Continue(e);
if (phaseIs("passive")){
if (input != 0) // 0 is query
store = input;
else holdIn("respond", response_time);
}}
public void deltint( ){
passivate();
}
public double Out( ){
if (phaseIs("respond")) return store;
else return 0;
}
}
```

Figure 4. DEVS behavior of a ramp

However, designing complex behavior of atomic models with different phases and ports, still difficult due to the fact the user will employ the switch-case or if-else statement to implement such a behavior. In this case, the code is less structured, less readable and hard to maintain. In fact, once the user makes changes on the model, he should reflect them on the code. Unfortunately, this will take much time due to inflexibility of such a code.

3.2. DEVS-Phase pattern to design atomic models

To remind to the ad-hoc design of atomic models, we propose the DEVS-phase design pattern. In this pattern we objectify phases and transitions to make the code of atomic models object-oriented. We recall briefly this design pattern according to the following structure:

Name: DEVS-Phase design pattern

Context: this pattern is useful to design DEVS behaviors expressed with phases to obtain a corresponding object-oriented code.

Solution: the AtomicPhase class holds a reference on the current phase of the model. It also holds the input and output ports vectors and concrete phase and transition classes inside vectors. This class has the responsibility to make state changes according to the received event via the method delta_ext() or delta_int(). It identifies the concrete ExtTransition or IntTransition object to fire. These transition classes hold a reference on their future phase and the simulator will act the state change by updating the current phase reference.
In addition, this pattern encapsulates through transition classes the following methods:
i) $action()$: this abstract method computes the new values of state variables outside the phase variable, and
ii) $performOutput()$: this abstract method identify the outport and computes the output event to send out of the internal transition to fire.
iii) $guard()$: a Boolean method that decides whether or not the transition may be fired.

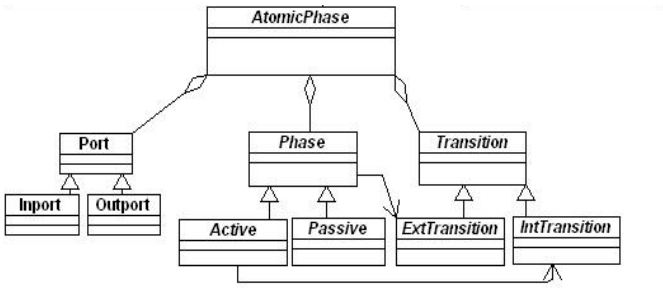

Figure 5. Class diagram of the DEVS-phase pattern

Advantages: +) well-structured code
+) easy code to maintain
+) flexible structure to add new DEVS behaviors at run-time (adding new phases and transitions, or modifying the target of a transition, etc.).

Lacks: -) skills on object programming are necessary to manipulate directly the code of this design pattern.
-) no mechanism to check if the updated model at run-time stills correct.

4. CODE AUTOMATIC GENERATION FROM DEVS ATOMIC MODELS

The field of code automatic generation is promoted by Model Driven Architecture. It consists on transforming a model expressed with some language into a different language or a programming language. The benefits of such an approach are: i) avoiding coding errors and speeding-up the implementation process; and ii) providing a direct mapping from the atomic models to designed ones.
The DEVS standardization group recognizes that automatic code generation is in progress for DEVS field and should be more developed to get significant advances. In fact the current works are focused on transforming DEVS atomic and coupled models written in Java and C++ into DEVS XML files (and vice-versa) through the DEVSML language. The main advantage of this language is to allow interoperability

between various models.
Obviously, we integrate code automatic generation in LSIS_DME approach by proposing to the user to choose which kind of generated code is useful for him. According to the atomic model type (SISO or phase-based model) the user may customize the generated code through the tool.

4.1. Partial code automatic generation from Ad-hoc models

Such models are not phase-based models. Consequently, the DEVS-phase pattern is not applied. However we make only automatic the generation of the model interface; that means the port sets and domain of each port in addition to the DEVS methods are generated automatically. Then the user completes the body of each DEVS method to implement the expected behavior and to define the set of state variables.

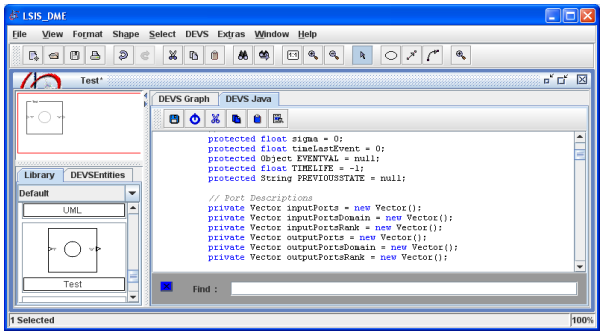

Figure 6. DEVS template of an atomic model generated from LSIS_DME tool

Unfortunately, the consistence of the code is not insured and the user should take care to avoid undesired scenarios.

4.2. Code automatic generation from DEVS-phase based models

Once a DEVS-phase based model description is established by the user, the tool generates a Java code from the graphical. The code is object-oriented and its architecture is designed according to the DEVS-phase pattern.

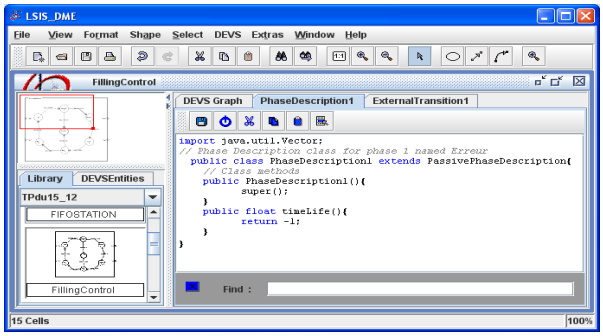

Figure 7. A phase class generated with LSIS_DME

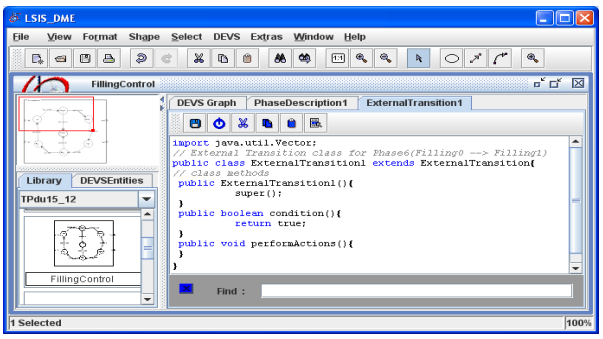

Figure 8. An external transition class generated with LSIS_DME

The generated code is open to hand-modification. The user may introduce pieces of code that are not supported by the graphical description due to grammar syntax limitation of LSIS_DME.

5. QUEUING SYSTEM

5.1. Example

Let us consider a queuing system consisting of one queue and one resource. Jobs arrive at any time, they are served according to their arrival date i.e. first in queue first out. To note that jobs arrive with a Poisson distribution $(\lambda = 1)$. In addition, jobs occupy the resource for some units of time (u.t). The inter-departures from the resource are exponentially distributed $(\mu = 2)$. With this analytical model, we compute mathematically some indicators like average waiting time using the following equation:

Average waiting time = $(\mu - \lambda)/\mu(\mu - \lambda) = 0.5$ u.t

The simulation that we propose to simulate such a system is well-known in the literature of DEVS. Three DEVS components are necessary: Generator, Queue and Resource.

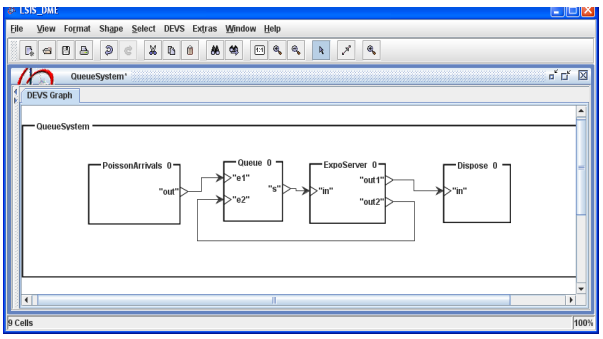

Figure 9. DEVS coupled model of queuing system

The generator, as the name indicates, generates jobs with a Poisson distribution. In addition it computes the number of generated jobs. Each job remains in the queue until the resource is released. The queue model consists of two phases "active"and "passive". In passive phase, the length of the

queue is zero (there is no job waiting) and the resource is free. Once a new job arrives, it is immediately sent out to the resource and the queue model changes phase from passive to active. To note that the generator is a DEVS ad-hoc design, the other models Queue, Resource and Dispose follow a DEVS-phase design.

5.2. Simulation results

The table 1 shows simulation results of the model of the queuing system example.

Based on the table 1, the average waiting time (AWT) is compared to that provided by the model M/M/1. This comparison shows that the waiting time of the simulation result converges to the result of mathematical one when simulation duration (the replication length) increases. So we can suppose that the model is valid.

Table 1. Simulation results

Replication	Average Waiting Time (u.t)		
	Simulation duration (u.t)		
	10^4	10^5	10^6
1	0,7210	0,7205	0,5572
2	0,7211	0,7204	0,5572
3	0,7213	0,7206	0,5572
4	0,7214	0,7206	0,5573
5	0,7215	0,7207	0,5573
6	0,7215	0,7211	0,5573
7	0,7220	0,7211	0,5573
8	0,7217	0,7211	0,5573
9	0,7216	0,7213	0,5575
10	0,7216	0,7215	0,5579
Average waiting time of replications (u.t)	0,7215	0,7209	0,5573

Complex scenarios may be defined through the simulation model of queuing system like the resource fails, customizing the service time of jobs, etc. for which it is difficult to define a mathematical model.

6. CONCLUSION

This paper proposes different forms of DEVS atomic designs. The most used one consists on using a DEVS behavior using the switch-case statement. However, this design suffers from the following lacks: the corresponding code is difficult to read and not well-structured. In addition it is difficult (in some cases impossible) to make further modifications or extensions. From that, we propose an alternative way to design atomic models based on DEVS-phase pattern. These two designs are integrated into a unique framework that supports new designs of atomic models. Consequently DEVS models implemented differently are simulated by the same simulator. In the near future, we will develop experimental studies to compare the proposed designs and to show which one may be useful according to the user requirements. In other works, we will define software metrics to evaluate indicators like time execution, heap memory size, etc. to help user to choose which design is more suitable to implement his atomic models.

7. REFERENCES

ADEVS, 2006 software site: http://sourceforge.net/projects/adevs. Last accessed March 2012.

DEVSJAVA, 2012. ACIMS software site: http://www.acims.arizona.edu/SOFTWARE/software.html Last accessed March 2012.

Hamri, M. and G. Zacharewicz. 2007 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. Honig and M. Seck, 2012. PHI-DEVS: Phase Based Discrete Event Modeling. Symposium on Theory of Modeling Simulation - DEVS Integrative MS Symposium (DEVS). SpringSimâ12, Florida-USA March 26-29 2012.

Praehofer H. and D. Pree Visual Modeling Modeling of DEVS-based Multi-Formalism Systems based on Higraphs Proceedings of the 1993 Winter G. W. Evans, M. Mollaghasemi, Simulation Conference pp 595 â 603.

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

Zeigler B and H. Sarjoughian, 2005. Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models. http://www.acims.arizona.edu/PUBLICATIONS/publications.shtml. Last accessed March 2012.