ENRICHING A DEVS META-MODEL WITH OCL CONSTRAINTS

Stéphane GARREDU[(a)], Evelyne VITTORI[(b)], Jean-François SANTUCCI[(c)], Dominique URBANI[(d)]

[(a) (b) (c) (d)] University of Corsica, UMR-CNRS 6134

[(a)]garredu@univ-corse.fr, [(b)]vittori@univ-corse.fr, [(c)]santucci@univ-corse.fr, [(d)]durbani@laposte.net

ABSTRACT

The purpose of this paper is to show how the platform-independent meta-model for DEVS formalism we have been working on can be enriched with Object Constraint Language (OCL) constraints. OCL is a declarative language (without any side effect) and allows us to control both the class attributes and the relationships between classes, in order to facilitate the modeling process and even the code generation towards a DEVS framework. To do so, we chose to follow a MDE, and in particular MDA approach, because OCL 2.0 is now aligned with UML 2.0 and MOF 2.0, which parts of MDA. The implementation of our meta-model with its OCL refinements has been done within Eclipse Modeling Framework (in which OCL 2.0 has been fully implemented) and its meta-meta-formalism Ecore.

Keywords: DEVS, modeling, MDE, MDA, EMF, Ecore, OCL, meta-modeling, PIM

1. INTRODUCTION

Since 20 years, with the great improvements in computers science, the interest for modeling and simulation has been evolving increasingly. Among the several formalisms dedicated to modeling and simulation of discrete-event systems, DEVS formalism appears to be the most famous one because of its ability to represent various systems (with its several extensions) and to simulate them. The interoperability of DEVS models is reduced because of the existing DEVS simulators.

The approach our team has been working on aims to ease the modeling process, increase the interoperability of DEVS models and improve code-writing process (using automated code generation). From our point of view, it can be done if we use some specific features of Model Driven Engineering (MDE) methodology and if we apply them to the world of modeling and simulation. The main advantage of MDE is that it is composed of a useful set of standards, and their purpose is to improve the reusability of the models and the code generation process.

An important part of our approach is to allow the description of DEVS models in a unified way: it implies that such a description has to be achieved without considering the platform in which the models will be simulated. To stick to this philosophy, each DEVS model should conform to the same pattern. This pattern should provide all the necessary DEVS concepts in order to create DEVS models in a unified way: such a pattern is called a meta-model (it is fully detailed in a paper "in press"). It describes the syntax and the semantics of a formalism.

A meta-model has to be described with a meta-formalism such as XML, or one of the concrete implementations of the well-known OMG Meta Object Facility (MOF). The one we used is the Eclipse Modeling Framework (EMF) Ecore, and can graphically be represented with UML class diagrams : we use them in this paper to present the main meta-classes of our DEVS meta-model.

The usual meta- formalisms are not refined enough to provide all the relevant aspects of a specification. They usually are cannot fully specify a modeling formalism, and they are often extended with the ability to express constraints. Even if those constraints can often be expressed with natural language, it is very hard to implement. Constraints must be expressed in a formal way. Object Constraint Language (OCL) provides such an ability. It is used to describe constraints on UML models as well as on MOF-typed meta-models, that is the reason why we said it was aligned with UML 2.0 and MOF 2.0.

We will use this language to define some important constraints on our DEVS meta-model : as a consequence, the number of meaningful models will be limited, but they will be more accurate. The purpose of this paper is to show how the needed constraints on our DEVS meta-model (and in a general way, on other meta-models) can be identified and implemented. Of course, those steps can be applied to any other kind of meta-models.

This paper is organized as follows: the first part is the background section, it focuses on the essential concepts of DEVS formalism and software engineering (MDE, and in particular UML and MDA, OCL). We conclude it by a presentation of our approach dedicated to the improvement of DEVS models interoperability and object-oriented code generation towards DEVS simulators.

The second part, which can be seen as a specific background, sums-up the most important features of the meta-model we defined for DEVS formalism : we give an overview of the meta-class hierarchy, then we focus on (using package diagrams associated to class-diagrams) the *DEVSExpression* and *DEVSRule*

concepts, and we describe how we chose to represent the couplings between models. We conclude by a discussion, which highlights some conditions, in the formal definition of DEVS formalism, which are not expressed yet in our meta-model. This will introduce the third part, dedicated to the OCL constraints we put on the meta-classes which belong to *DEVSExpression*, *DEVSRule* and *Coupling* packages. After that, we give an example of a model which could have been validated without our constraints but which is now not meaningful anymore. We finally conclude this paper by giving an outline of it, and we say a few words about our future work.

2. BACKGROUND

All caps, bold, flush left. Use Times New Roman Font and 10 points in size. Start the text on the next line. Please use the "HEADING 1**"** style.

The headings for the Abstract, Acknowledgements, Appendix, References and Authors Biographies sections are not numbered. Please use "HEADING" style. Insert one blank line before each heading.

Do not include any kind of page numbers, headers or footers. Final page numbers will be inserted by the publisher.

2.1. DEVS

DEVS formalism, introduced in the 70's by Pr. B.P. Zeigler (Zeigler 1989) (Zeigler et al. 2000) is based on discrete events, and it provides a framework with mathematical concepts based on the sets theory and systems theory concepts to describe the structure and the behavior of a system.

Almost any system can be modeled with DEVS formalism, if it has finites states and finite transitions between its states, in a finite time interval, and interacts with its environment through events sent and received on its communication ports. A DEVS model represents, as other kinds of models do, a simplified version of reality. DEVS formalism is modular and hierarchical; it allows the definition of 2 kinds of models: atomic models and coupled models. DEVS makes an explicit separation between a model and its simulator: the latter is "automatically" built from the former. It has been formally proved that the entity "simulator" is able to execute correctly the behaviour described by the entity "model". It has also been formally proved that DEVS is closed under composition, which means that a coupled model (composed of several models) can be seen as a unique atomic model.

As there exist many DEVS-oriented simulators and frameworks (DEVS Standardization Group 2012) the interoperability of DEVS models is reduced.

2.1.1. DEVS Atomic Models

The tiniest element in DEVS formalism is the atomic model. It is specified as follows.

AM = $< X, Y, S, ta, \delta_{int}, \delta_{ext}, \lambda >$, where :

- $X = \{(p,v) | p \in InputPorts, v \in X_p\}$ is the input events set, through which external events are received; *InputPorts* is the set of input ports and X_p is the set of possible values for those input ports
- $Y = \{(p,v) | p \in OutputPorts, v \in Y_p\}$ is the output events set, through which external events are sent; *OutputPorts* is the set of output ports and X_p is the set of possible values for those output ports
- S is the states set of the system;
- $ta: S \rightarrow R_0^+ \cup +\infty$ is the time advance function (or lifespan of a state);
- $\delta_{int}: S \rightarrow S$ is the internal transition function;
- $\delta_{ext}: Q \times X \rightarrow S$ with $Q = \{(s,e)/s \in S, e \in [0, ta(s)]\}$ is the external transition function;
- $\lambda: S \rightarrow Y$, with $Y = \{(p,v) | p \in OutputPorts, v \in Y_p\}$ is the output function;

The simplest transition is called the internal transition : at a given moment, a system is in a state $s \in S$. Unless an external event occurs on an input port, the system remains in the *s* state for a duration defined by *ta(s)*. When *ta(s)* expires, the model sends the value defined by $\lambda(s)$ on an output port $y \in Y$, and then it

changes to a new state defined by $\delta_{int}(s)$. Such a transition, which occurs because of the expiration of $ta(s)$, is an internal transition.

The other transition type is called the external transition, because it is triggered by an external event. In this case, it is the $\delta_{ext}(s,e,x)$ function which defines which state is the next one (s is the current state, e is the elapsed time since the last transition, and $x \in X$ is the event received).

In both cases, the system is now in a new state s' for a new duration $d' = ta(s')$ and the algorithm restarts.

2.1.2. DEVS Coupled Models

A coupled model is composed of at least one submodel (atomic or coupled). A coupled model can be seen as a parent-model which describes a hierarchy (i.e. a list of sub-models, the links between itself and the submodels it is composed of, and the links between the submodels themselves). A coupled model is formally defined by

MC = <X, Y, D, {Md|d∈D}, EIC, EOC, IC, select>, where :

- X = {(p,v)|p∈InputPorts, v∈Xp} is the input events set, through which external events are received; InputPorts is the set of input ports and Xp is the set of possible values for those input ports
- Y = {(p,v)|p∈OutputPorts, v∈Yp} is the output events set, through which external events are sent; OutputPorts is the set of output ports and Xp is the set of possible values for those output ports
- D is the set of component names, d∈D
- Md is a DEVS model (either atomic or coupled)
- EIC is the set of external input couplings; an external input coupling is a link between the input port of the current coupled model and the input port of any of its submodels
- EOC is the set of external output couplings; an external output coupling is a link between the output port of the current coupled model and the output port of any of its submodels
- IC is the set of internal couplings; an internal coupling is a link which involves the output port of a submodel and the input port of another submodel
- select is the tiebreaker function

Figure 1 is an example of a coupled model containing two submodels: an atomic model and a coupled model, which contains itself two atomic models. The coupling functions (EOC, EIC, IC) are indicated and the ports are represented by black diamonds. For instance, Coupled 2 has two input ports, and one output port. Atomic 3, contained by coupled 2, has one input port, and one output port.

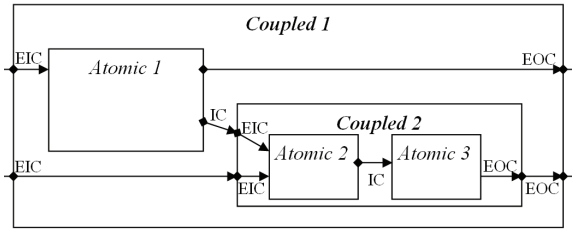

Figure 1: A DEVS coupled model

2.2. DEVS Software engineering and meta-modeling

2.2.1. UML

Unified Modeling Language (UML) is a graphical set of modeling formalisms: it provides a toolkit which enables one to model the structural aspects of a system as well as its behavior (Booch et al. 1998).

UML is owned by the Object Management Group, and its current version is UML 2.4.1 (OMG 2011). Its main advantage is that it is considered as a standard formalism by a large worldwide community of users. We use in this paper UML class and package diagrams to represent the meta-classes of our DEVS meta-model.

2.2.2. UML and meta-levels

A UML model, for instance a UML class diagram (that we will use later to describe our meta-model) is an abstraction of a system from the real world located at the lowest abstraction level: M0. Such an abstraction takes place at a higher level: M1. It is defined by its meta-model at, once more, a higher level: M2. This meta-model describes, using a language or formalism, the elements that can be used to design the model and their relationships with each other. Such a description is defined at a higher level by Meta Object Facility (MOF), a language used to describe other languages. This level is M3. MOF is defined on itself, i.e. it is described in MOF terms. Hence, there is no level higher than M3 (Figure 2).

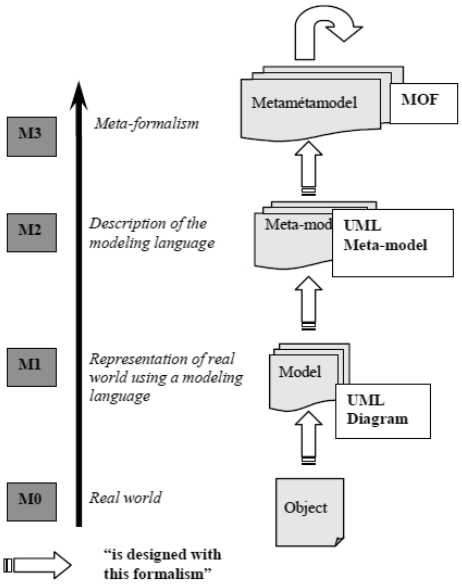

Figure 2: UML and the "meta" levels

2.2.3. MDE, MDA and EMF

Model Driven Engineering is a software development methodology which focuses on creating and exploiting domain models. MDE is a generic approach, and its most famous implementation is Model Driven Architecture, owned by the OMG.

We can also quote the Eclipse ecosystem of programming and modelling tools (Eclipse 2012). EMF (EMF 2012) is a particular part of Eclipse which contains and implements a set of MDA standards: in fact, MDA and EMF are very close one to each other, the ladder uses and implements the concepts inherited from the former, and we use both of them in our approach. MDA (Model Driven Architecture) (OMG, 2001) is a software design approach initiated by the OMG in 2001 to introduce a new way of development based upon models rather than code. With MDA approach, everything is a model, even the transformations between models are considered as models.

MDA defines a set of guidelines for defining models at different abstraction levels, starting from Computational Independent Models (CIMs) to platform independent models (PIMs), then from PIMs to platform specific models (PSMs) which are tied to a particular technology (i.e. platform). The translation from one PIM to one or several PSMs is to be performed automatically by using transformation tools. MDA also enables transforming a PSM into source code. The advantage of such an approach is the great reusability of models. OMG provides a set of standards dedicated to this approach. Although UML was at the beginning the basis of the OMG works on MDA, it is now Meta-Object Facility which appears to be the most basic standard.

According to this standard, each formalism involved in a MDA process at any level (PIM, PSM) is to be specified by a meta-model expressed in terms of MOF elements. As MOF was not given a concrete syntax, our meta-model was implemented using EMF. The MOF equivalent in EMF is Ecore. Ecore can be seen as an implementation of MOF, in the same way that EMF uses the MDA concepts.

2.2.4. OCL

Object Constraint Language (OMG 2006) is a formal and strongly typed modeling language proposed by the OMG in order to specify unambiguous constraints on UML 2.0 models, and even MOF 2.0 meta-models (i.e. fully aligned with MOF and MDA).

As it is a pure specification language OCL enables us to write expressions without side effects. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model, nor in the meta-model. Moreover, it is not possible to write program logic or flow control in OCL. It is not possible to invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, OCL expressions are not by definition directly executable.

In this paper, we will use OCL to specify *invariants* on our meta-classes and their relationships (i.e. conditions that have to be fulfilled by any instances of the meta-classes) so that the instances of the meta-model must conform to those conditions.

2.3. Our Approach

The DEVS meta-model we designed is located at the M2 level and uses the concepts given by the highest level (M3). The meta-meta-formalism we used is EMF Ecore.

Some theoretical aspects of our approach are presented in a more detailed way in (Garredu et al. 2011). Figure 3 shows a larger view of our philosophy : the DEVS meta-model enables to specify platform-independent models (DEVS PIMs) which are in fact instances of this meta-model. Thanks to transformation rules, those models can be used for an automated object-oriented code generation towards a DEVS simulator. But the other huge advantage of following a MDE approach is that the DEVS meta-model can be seen as a single entry-point for other DEVS-like (i.e. based on states, transitions and events) formalisms: also using transformation rules at the M3 level, it is possible to specify mappings from any DEVS-Like formalism towards our DEVS meta-model. In other words, any model written in a DEVS-like formalism can be transformed in a DEVS model. Then, as a normal DEVS model, it can be used, as we said before, to generate object-oriented code.

3. MAIN ELEMENTS OF OUR DEVS METAMODEL

3.1.1. Overview of the hierarchy

The most important package of our meta-model is the *DEVSModel* package (Figure 4). It shows that a DEVS

model can be either atomic or coupled. If it is coupled, it must contain at least a submodel.

Types are defined differently following the object-oriented languages. So, our meta-model must be able to handle types, in a generic way. We chose to represent only 4 basic types, but this can be easily extended. The package representing types is given by Figure 5.

Every DEVS model has at least an output port and can have an input port. Every port must be given a name and a type, and can be either an input port or an output port (Figure 6).

We introduced with this meta-model an important basic concept which is common to many elements of this meta-model: the *DEVSExpression*.

3.1.2. DEVSExpression package

In a formal point of view, a DEVS atomic model is composed of a finite set of its possible states *S* linked by deterministic transitions. Those states are distinct values; it implies that the fact of changing a state may lead to the creation of another state.

This is not a problem for the systems of which the states are known (and can be enumerated) but it becomes one when we have to deal with states which take their values in infinite sets, for instance $[0;1] \in R$.

To solve this problem, we chose to represent a state by instance, the simplest one is given by the *ta* function; it describes the fact of returning a value after evaluating a state.

what we call a *state variable*. It takes a new value when the state changes (i.e. each new state change will lead to a change of the value of the state variable). Therefore, only a state variable is used to represent a collection of states which belong to the "same kind". A state variable must be named, and must be typed. It can also be affected a literal value.

We chose to describe state variables and types, and they can be included in a larger set which is called *DEVSExpression*.

It is one of the basis of our meta-model. As a *StateVar* is a *DEVSExpression*, a *LitteralBasicValue* (see Figure 9) is also *DEVSExpression* but a simpler one: in fact the simplest one because it is composed of a unique typed value. We built this package (see Figure 7) in a modular way, in order to facilitate its modification by enriching it with other sub-classes. *DEVSComplex* can be a starting point to do so.

3.2. *Rule* package

In spite of the differences between the DEVS functions, we can notice that every function can describe a test, an action on a variable, a message. Those descriptions follow a sort of pattern which is often the same: a set of enumerations. We call those enumerations *DEVS Rules*.

The purpose of a rule is to represent a set of operations on specific elements. To be more accurate, these are not exactly operations but descriptions. For

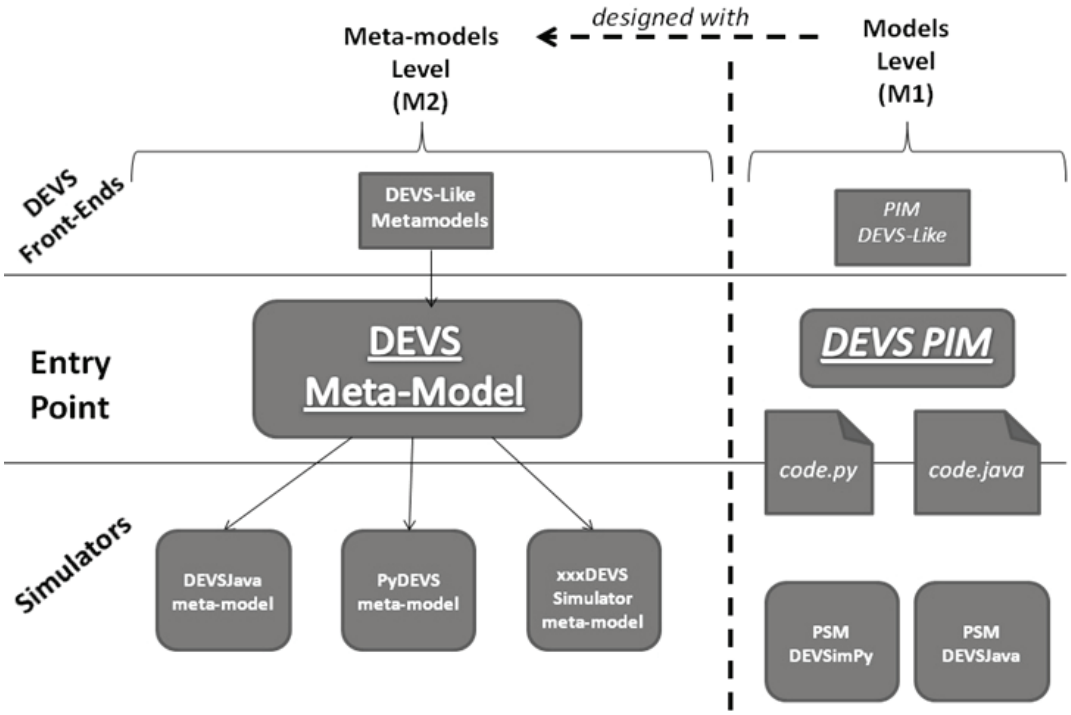

Figure 3: The DEVS meta-model used as a single entry-point towards DEVS simulators

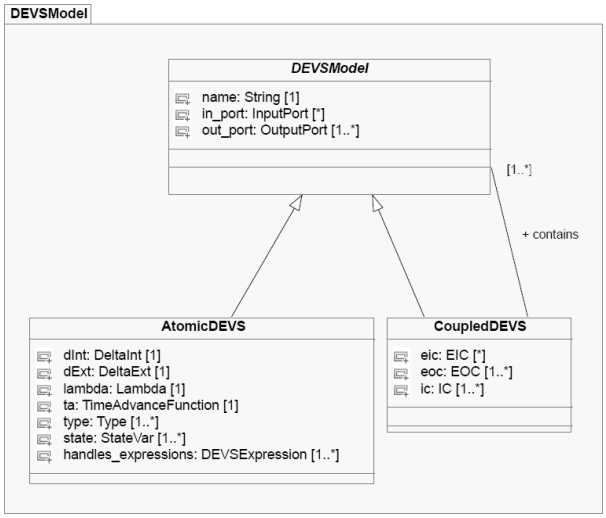

Figure 4: The *DEVSModel* package

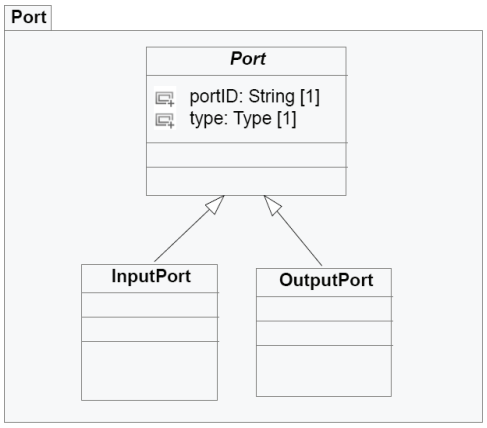

Figure 5: The *Port* package

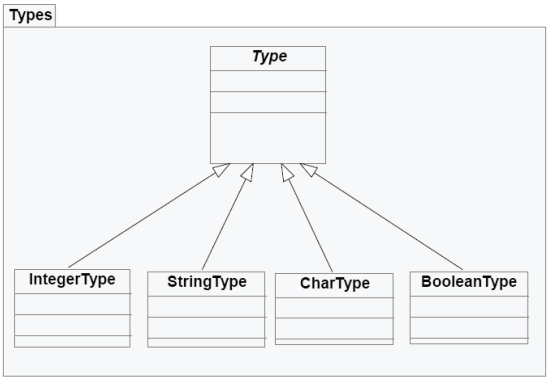

Figure 6: The *Types* package

3.3. Rule package

In spite of the differences between the DEVS functions, we can notice that every function can describe a test, an action on a variable, a message. Those descriptions follow a sort of pattern which is often the same: a set of enumerations. We call those enumerations *DEVS Rules*.

The purpose of a rule is to represent a set of operations on specific elements. To be more accurate, these are not exactly operations but descriptions. For instance, the simplest one is given by the *ta* function; it describes the fact of returning a value after evaluating a state. A rule is always composed of a *condition* and an *action*. Figure 8 shows the organization of the *Rule* package, it uses the two next packages : see Figures 9 and 10. Those figures represent the *Conditions* and *Actions* used in the Rules. A *condition* (see Figure 9) is described by a test: a left member, a comparator, and a right member. It can be a test on an input port (in the case of an external transition function) or on a state variable (in every DEVS atomic function, there is a test on a state variable). An *action* is the description of an action: an output action (on a port), or a state change action (in the case of a transition function). Figure 10 shows the *Action* package.

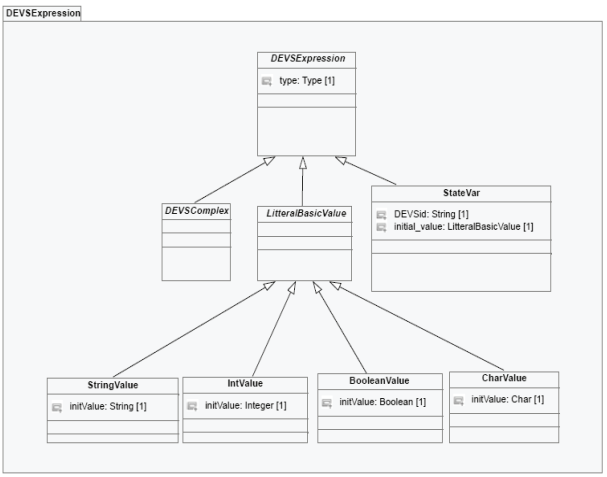

Figure 7: The *DEVSExpression* package

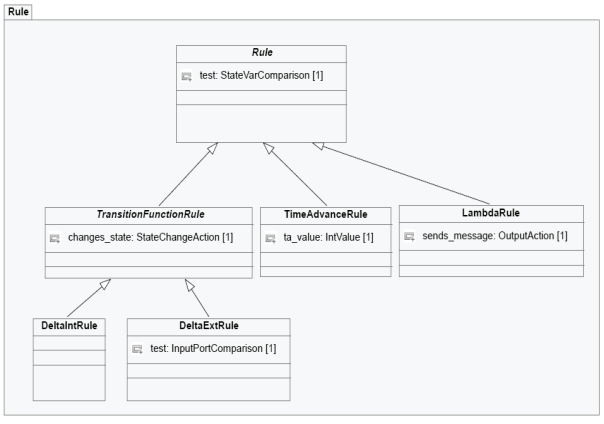

Figure 8: The *Rule* package

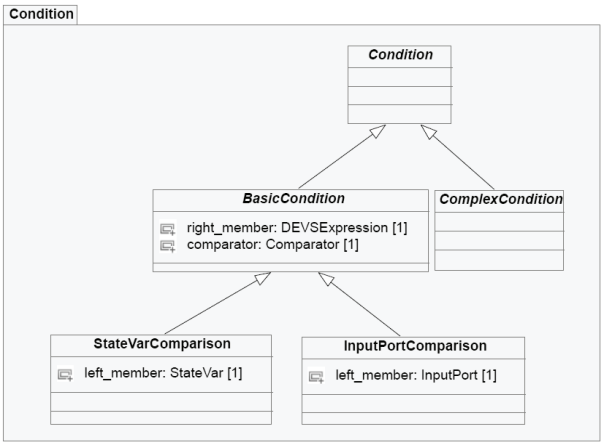

Figure 9: The *Condition* package

3.4. Coupling Package

The meta-class which describes the coupling functions (in the coupled models) is given by Figure 11. As usual, there is an abstract class (*Coupling*) from which inherit 3 sub-classes: EIC, EOC, IC.

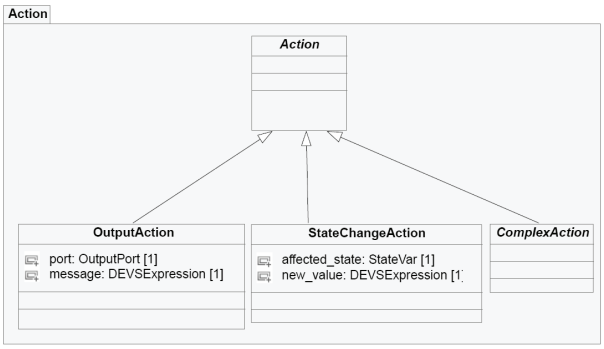

Figure 10: The *Action* package

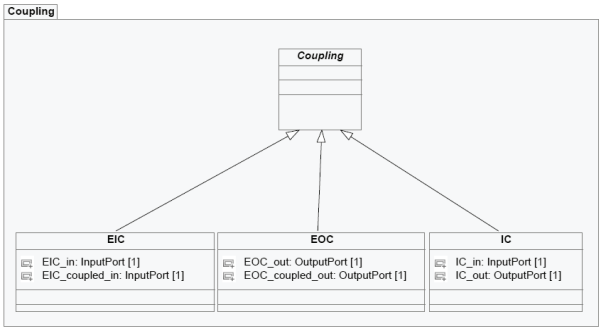

Figure 11: The *Coupling* package

3.5. Discussion

The DEVS meta-model we show here is accurate enough to specify several DEVS models. Though, the modeler must take care of several parameters while he designs a system. If we look at the DEVS formal definition, we can see that : an IC function involves an input port and an output port, an EIC function involves two input ports, and an EOC function involves two output ports. That is transcribed in the that a previous figure representing the coupling functions. But, how can we specify that, for instance, an IC function must use the ports of two sub-models of the current coupled model?

Reasoning in the same way, how can be sure that the initial value of a *StateVar* and its type belong to the same type? How can we verify that a *StringValue* has the correct corresponding type (*StringType*)?

All those questions have the same process: a comparison between the formal definition of DEVS and our meta-model, or a reasoning on what makes a model be meaningful or not (a non-meaningful model always leads to "modeling mistakes").

The next part is dedicated of all the refinements we can apply to our meta-model in order to make it be far more accurate than before.

4. ADDING OCL CONSTRAINTS TO OUR METAMODEL

In this part we use the EMF OCL 3.1.0 language to describe the constraints on our meta-model. The constraints were directly implemented in the EMF framework, and (added to the corresponding meta-classes).

4.1. DEVSExpression Constraints

4.1.1. *StateVar* Constraints

As we have seen before, we must verify that a *StateVar* which is given a String type cannot contain a reference to a value from another type. This can easily be done :

```
invariant StateVarTypeConstraint:
self.type = self.initial_value.type;
```

A *StateVar* must also be identified: its attribute DEVSid must not be empty, so we write :

```
invariant idNotEmpty:
self.DEVSid.size()>0;
```

4.1.2. LitteralBasicValue Constraints

As a child of *DEVSExpression*, a *LitteralBasicValue* can be a type which is different from the value it carries. We must prevent that. Every new *LitteralBasicValue* must have the same type than the type of the value it carries. Moreover, a *StringValue* must not be empty :

- *IntValue* must be typed with *IntegerType*

```
invariant intIsInt:
self.type.oclType().name =
'IntegerType';
```

- *CharValue* must be typed with *CharType*

```
invariant charIsChar:
self.is.type.oclType().name =
'CharType';
```

- BooleanValue must be typed with BooleanType

```
invariant boolIsBool:
self.type.oclType().name =
'BooleanType';
```

- *StringValue* must be typed with *StringType* and not empty

```
invariant stringIsString:
self.type.oclType().name =
'StringType';
invariant nameNotEmpty:
self.str_val.size()>0
```

4.2. *DEVSRules* Constraints

In every comparison, we must verify that the type of the left member (a *Port*, a *StateVar*…) belongs to the same type as the element it is compared to:

- In the *StateVarComparison* meta-class :

```
invariant svcTypeConstraint:
self.left_member.type=self.right_memb
er.type;
```

- In the *InputPortComparison* meta-class :

```
invariant ipcTypeConstraint:
self.left_member.type=self.right_memb
er.type;
```

In the same way, we must verify that the types match in an *Action*. In the case of an *OutputAction*, we must verify that the output port given as a parameter belongs to the currrent atomic model :

- For a *StateChangeAction* :

```
invariant stateChangeTypeConstraint:
self.affected_state.type=self.new_val
ue.type;
```

- For an *OutputAction* :

```
invariant outputActionTypeConstraint:
self.port.type=self.message.type;

invariant portBelongsToCurrentAtomic:
self.port.oclContainer()=self.oclCont
ainer();
```

4.3. Coupling Constraints

Here we present 3 groups containing 3 constraints each. Those constraints are very important because they prevent hasardous couplings between ports. The last constraint of each group verifies that two ports have the same type.

- EIC must have a reference to an input port from the current coupled model and a reference to the input port of a submodel

  $(Md|d \in D)$

```
invariant EICcurrentModelInputPort :
self.oclContainer()=self.EIC_coupled_
in.oclContainer();

invariant EICsubmodelInputPort :
self.oclContainer()=self.EIC_in.oclCo
ntainer().oclContainer();

invariant EICtypes:
self.EIC_coupled_in.type =
self.EIC_in.type;
```

- It is the same thing for EOC, but with output ports :

```
invariant EOCcurrentModelOutputPort :
self.oclContainer()=self.EOC_coupled_
out.oclContainer();

invariant EOCsubmodelOutputPort :
self.oclContainer()=self.EOC_out.oclC
ontainer().oclContainer();

invariant EOCtypes:
self.EOC_coupled_out.type=self.EOC_ou
t.type;
```

- IC must link the input and output ports of two submodels :

```
invariant ICsubmodelInputPort :
self.oclContainer()=self.IC_in.oclCon
tainer().oclContainer();

invariant ICsubmodelOutputPort :
```

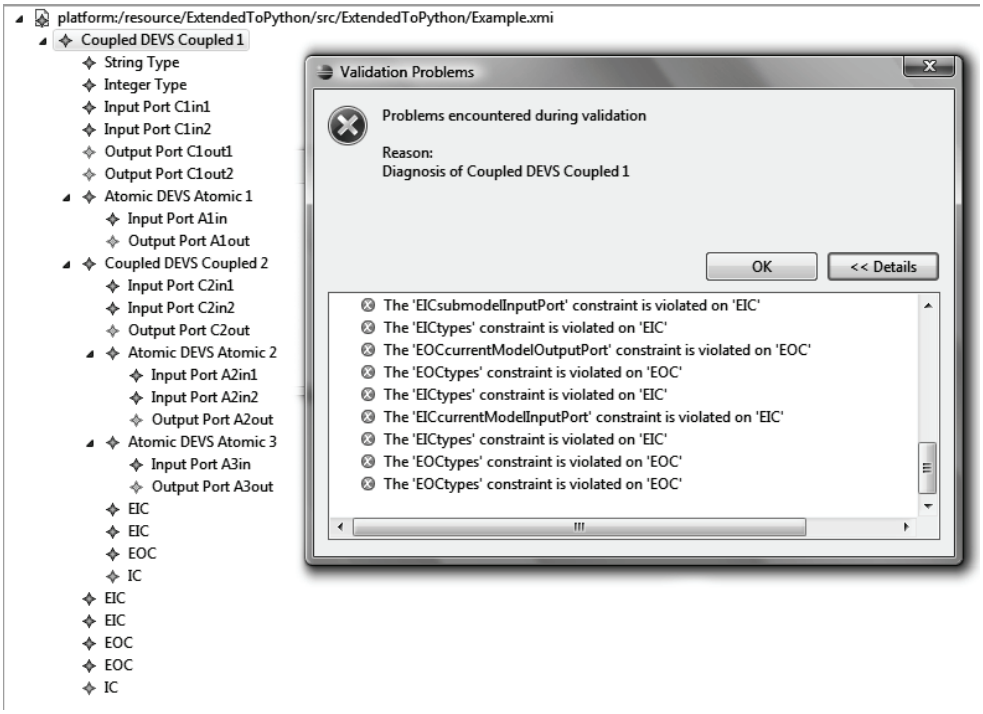

Figure 12: Testing OCL constraints with Eclipse Modeling Framework and our DEVS meta-model

```
self.oclContainer()=
self.IC_out.oclContainer().oclContain
er();

invariant ICtypes:
self.IC_out.is_typed=self.IC_out.is_t
yped;
```

5. EXAMPLE

We chose to re-create the hierarchy of Figure 1 in this example in order to test the class invariant constraints we put on the meta-classes of *Coupling* package. We tried to make voluntary mistakes in the definition of the coupling functions and tried to validate our model instance. The errors raised by EMF are shown in the following screenshot (Figure 12). The other kind of constraints we put on other classes are fully functional too.

6. CONCLUSION

In this paper, we have presented the main meta-classes of our meta-model for DEVS then we have shown how they could be enriched and refined with OCL constraints. Without those constraints, even if the instantiated models would have been "correct" in terms of our meta-model, from a DEVS point of view they would not. OCL can be seen as a super-layer which cannot work the fundamental layer created by the meta-model. This super-layer depicts more precisely the meta-attributes and the meta-relationships between the meta-classes. Using those two layers was possible because of the tight links which exist between OCL and MOF : those links exist in EMF too between Ecore and EMF OCL.

In a near future, we plan to describe how OCL can be used as a super-layer for Model To Model transformations. This is another side of OCL usage: instead of being used to describe class invariants, it is used this time to create queries and navigate in the links between the instances of a meta-model.

REFERENCES

G. Booch, J. Rumbaugh, and I. Jacobson. 1998. "The unified Modeling Language User Guide". Addison-esley.

DEVS Standardization Group, Carleton University Website, 2012 http://cell-devs.sce.carleton.ca/devsgroup/?q=node/8

Eclipse 2012 http://www.eclipse.org/

EMF 2012 http://www.eclipse.org/modeling/emf/

S. Garredu, E. Vittori, J.-F. Santucci, D. Urbani, "A methodology to specify DEVS domain specific profiles and create profile-based models", IEEE-IRI 2011, 3-5 Aug. 2011, Las Vegas, NV, USA, pp. 353 - 359

OMG 2001. Model Driven Architecture homepage http://www.omg.org/mda/

OMG 2006. Object Management Group website, OCL section, http://www.omg.org/spec/OCL/2.0/

OMG 2011. Unified Modeling Language: Superstructure and infrastructure, version 2.4.1, August 2011 http://www.omg.org/spec/UML/2.4.1/

Zeigler B.P., 1989. "*DEVS Representation of Dynamical System", in* Proceedings of the IEEE*,* Vol.77, pp. 72-80

Zeigler, B. P., H. Praehofer, and T.Kim. 2000. Theory of Modeling and simulation. 2nd ed. Academic Press