A DEVS-Oriented Intuitive Modelling Language

S. Garredu E. Vittori J.-F. Santucci
University of Corsica
{garredu;vittori;santucci}@univ-corse.fr

Keywords: DEVS, methodology, MDA, Specification Languages, Code Generation

Abstract
The purpose of this paper is to introduce a new graphical and intuitive programming language for DEVS models. It is the result of our global approach whose general aim is to enable a scientist to create and simulate DEVS models without the constraint of asking a DEVS expert to create the formal models and a computer scientist to program them. We explain here how our language is built, and we give an example of a model designed using it. Models from this language can be mapped onto DEVS models, using a model-driven approach (based on MDA) which is really helpful to perform an automated code generation towards an Object Oriented Language.

1. INTRODUCTION

DEVS [ref.] is the most general formalism used to describe discrete event systems.
Any system can be modelled using DEVS, the only conditions which must be fulfilled are that the states of the system are finite, and the changes between states are also finite in a finite interval.
DEVS allows considering a system both in a modular or hierarchical point of view, and once the model defined, the simulator is provided.
That is why it is interesting to study complex systems using such a formalism, or one of its many extensions [1-7]. There are several modelling and simulation tools based on DEVS, for instance PowerDEVS, JavaDEVS, JDEVS, PythonDEVS and so on [Kauffman and al., 2001].

However, even if some of them provide graphical user interfaces (G.U.I.), it is not possible to use them without having a good knowledge of both DEVS and object oriented programming.
How can we make the modelling step easier ? Two constraints have to be taken into account to solve this problem:

- To make the modelling step be easier, a "simple language" (according to the criteria we defined [Garredu et al. 2007]) has to be used to reduce the complexity of this task
- The destination formalism must be Classic DEVS and the corresponding object classes automatically generated

This paper makes some proposals who take account those two constraints.
Our approach tries to make the modelling step easier by helping the scientist to create his own models, using a partially graphical and intuitive specification language, and assuming the fact that a scientist is able to recognize and make a description of a given system. This language has been given a graphical representation, implemented through a GUI. We made the choice to give the possibility to create only single models (of course with many states), i.e. our language does not support, at this time, the interconnections between several models (the components cannot be coupled).
In order to fulfil the second constraint, we follow a MDA process, chosen because of its many advantages, the most important of which is of course the ability of MDA to perform mappings from some models onto others, provided their meta-models are defined.
Thanks to the MDA PIM (Platform Independent Model) and PSM (Platform Specific Model) models, a model designed with our language can be mapped onto a basic DEVS PIM model. Finally, the DEVS PIM model can be mapped onto several DEVS frameworks, by generating the corresponding source code.
The next part is dedicated to the DEVS formalism and to the MDA approach. The third part presents the proposed specification language and its main features. Then, in the fourth part, we give an example of a model designed with the specification language. Finally, after a discussion where we explain how mappings can be performed onto DEVS, we give a conclusion.

2. BACKGROUND

In the first part of this section part we give brief overview of the system theory on which lies the DEVS formalism, and though which is used by many scientists.
In the second we present the DEVS formalism, which lies on 2 elements : Atomic Models and Coupled Models. The former describe the behaviour of a system while the latter

offer a view of the global structure of the system. We also briefly explain how a DEVS model is linked to the corresponding simulator.
In the last part, we present the Model Driven Architecture, and we show that even if it seems to be useful mainly in computer engineering, such a process can also be followed in modelling and simulation.

The *General System Theory,* introduced by L. Von Bertalanffy during the middle of the XXth century, gave a new way to consider systems. Nowadays, this theory is well-know by the scientists who use it when they need to study complex natural systems.
L. Von Bertanlanffy, fed this new way of considering systems : "a system is a mass of elements which interact". The ultimate purpose of this theory, as an interdisciplinary approach, is to find the rules which the systems studied by the sciences share.
Of course, this fantastic purpose will not be reached soon, yet we are able to find common points to several systems.
General System Theory uses two distinct concepts to describe a system, which can be named views : the behavioural view and the structural view.
The structure of a system is this inner constitution, and its behaviour is its outer manifestation.

The behaviour can be seen as the way a system interacts with its environment, the only knowledge we have is that this system has input ports where it receives events, and output ports through which it sends information to the external world.
The most known representation of the behaviour of as system is the famous black box (see Figure 1) which gives a relationship between inputs received by the system and its outputs. The structure of the system is not known, only its behaviour, that is the reason why it is represented by a black box. An input-free system is called an autonomous system (i.e. it evolves itself without receiving external events).

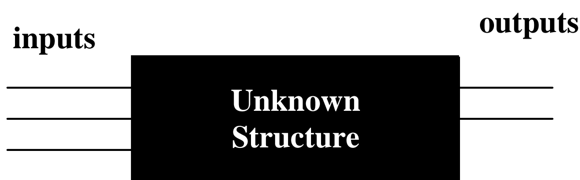

Figure 1. A system represented by a black box

The internal structure of a simple system is composed by states and rules which define how inputs make the system go from one state to another.
But the structure of a system may also give information on its coupling relationships and its hierarchical structure: if the output ports of a system are coupled to the input ports of another one, then there is a coupling relationship. The resulting system can be seen as a new system, named coupled system, which can itself be coupled to other systems. Those coupling relationships illustrate the hierarchical structure of a system, and are tightly linked to the specification level concept.
If we know the structure of a system, then its behaviour can be deducted, but the other way is not true : it is not always easy to find the structure of a system only observing its behaviour. Though, in modelling and simulation, we often have to give a valid representation of a system based on the study of its behaviour.
The basic concepts General System Theory are known and used all around the world by many scientists who belong to several domains, that is the reason why when we introduce our language we assume that some concepts can easily be handled, even by non-computer scientists.

DEVS is a formalism which is based on the general systems theory; it was introduced by Pr. Zeigler [Zeigler 2004].
It allows to describe a system in a modular and hierarchical way, and to consider a global system as a set of other more simple subsystems, in order to reduce the system's global complexity.
Modelling and simulation are distinct phases, and once the model designed the simulator is provided.
DEVS also has great genericity properties, it can be used in many study domains, always considering the fact that a system evolves with events; moreover, DEVS has a good evolutivity, and has often been extended to be able to take into account various systems : dynamic systems (i.e. which structure evolves with time) with DSDE [3] and DynDEVS [4], systems with uncertain parameters with fuzzy-DEVS [5] and min-max-DEVS [6], systems with evolutions in their interfaces with Cell-DEVS (cellular approach) [7] and Vector-DEVS (vectorial approach) [8].

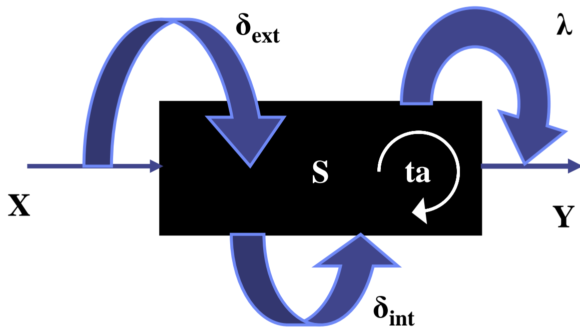

Figure 2. Atomic DEVS model behaviour

The atomic model (Figure 2) is defined by:
AM = < X, Y, S, t_a, δ_{int}, δ_{ext}, λ >
With:
- X the input ports set, through which external events are received;
- Y the output ports set, through which external events are sent;
- S the states set of the system;
- t_a the time advance function (or lifespan of a state);
- δ_{int} the internal transition function;
- δ_{ext} the external transition function;
- λ the output function;

The simulation is carried out by associating to each component a simulator making it possible to implement the corresponding simulation algorithms.

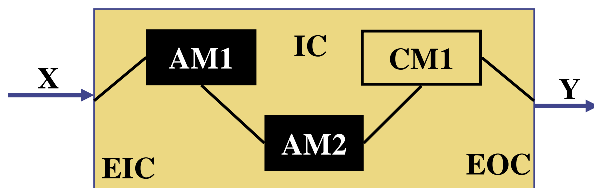

Figure 2. DEVS coupled model

Thus, DEVS enables the specialist to be completely abstracted from the simulators implementation. Once the model is built and whatever its form is, B.P. Zeigler defined a simulator able to take into account any model described according to DEVS formalism, and developed the concept of abstract simulator [1] [2] [10]. The architecture of such a simulator represents an algorithmic description making it possible to implement the implicit instructions of the models resulting from formalism DEVS, in order to generate their behaviour. The major advantage of such a simulator is that its building is independent from the model.

As the main drawback of DEVS is its difficulty to be used by the non-computer scientists, our approach takes place in the improvements already made to help the modeller to create his models. Some DEVS-oriented specification languages have been created to reach this goal.
Among those improvements, we can quote the DEVS Definition Language which enables to give a textual representation (written in pseudo-code) of DEVS atomic and coupled models [Zeigler et al., 2000]. This language is close to DEVSpecL [ref.], but they both do not have a corresponding graphical formalism.
Among the graphical improvements, a graphical representation of atomic models, based on states (vertices) and transitions (edges), was given in 1994 by H.S. Song. GDEVS is an environment which enables to specify DEVS models using this graphical formalism. [ref univ marseille]. (citer GGAD) However, they only provide a low-level representation and even a graphical formalism to express DEVS models has been introduced.

DEVS is a very good tool for the modeling and simulation of complex systems but it does not allow a lambda user, who is not a computer scientist, to specify his models.
A both graphical and textual specification language, establishing the link between knowledge of an expert and a DEVS environment, would make it possible to extend the advantages of DEVS.

MDA (Model Driven Architecture) [11] is a software design approach initiated by the OMG (Object Management Group) in 2001 to introduce a new way of development based upon models rather than code. It defines a set of guidelines for defining models at different abstraction levels, from platform independent models (PIMs) to platform specific models (PSMs) tied to a particular implementation technology. The translation between a PIM and one ore more PSMs is to be performed automatically by using transformation tools.
OMG provides a set of standards dedicated to this approach. Although the Unified Modeling Language (UML [12]) was at the beginning the basis of the OMG works on MDA, it is now MOF (MetaObject Facility) which appears to be the most basic standard.
It is a metaformalism (e.g. formalism used to define formalisms). According to this standard, every formalism involved in a MDA process at any level (PIM, PSM) is to be specified by a metamodel expressed in terms of MOF elements. The QVT (Query Views Transformation) standard provides a standard formalism to define transformation between models expressed in MOF compliant formalisms.
The UML [12] still provides a common and useful visual notation for the description of software artefacts at several levels and from several points of view. But it now appears

as a formalism (among the others) that may be used to specify models but theoretically a MDA process can be followed without using it. However, OMG still advocates its use as a favourite formalism arguing that it becomes a real used standard in modelling area and its metamodel is fully defined [11].

MDA^{TM} is a trade mark hold by the OMG. However, nowadays a more global approach called MDE Model Driven Engineering is used by the international research community to refer to the same principles even if all the OMG standards are not imposed. MDE focuses on the use of metamodeling and model transformation but is not tied to the MOF.

In our approach, we believe that a MDE process will be useful to specify our intuitive language and also its automatic transformation process into DEVS formalism.

Some links between the computer engineering world and modelling and simulation, have been introduced by A. Tolk [Tolk and Muguira 2004], he establishes a link between MDA and DEVS formalism.

3. PROPOSED SPECIFICATION LANGUAGE

In this section, we first present our specification language, and we show how it can be graphically represented and we give its metamodel.

In the background section, we saw that most of the scientists are able to watch a system and identify boundaries between this system and the rest of the world. Every model is linked to its environment by receiving and sending values, through its input and output ports.

Scientists also can identify how the system evolves with time, and the ways (events) to go from one state to another.

To put it in a nutshell, we are convinced that every scientist can handle the concepts of the general systems theory, even if modelling with DEVS is not trivial.

The basic elements of our language are: models, states, events, transitions, ports.

Even if those elements seem very close to DEVS formalism, and to every other formalism based on states and transitions, we are convinced that they will be easier to handle, because the G.U.I. provides a step-by-step help and makes some concepts more transparent. Moreover, they have a great portability, because they are platform-independent.

The purpose of our language is to provide an intuitive G.U.I. which will help the scientists during the different steps of the modelling. This G.U.I. will provide a step-by-step help, as an interactive tutorial. The main steps we identified are:

- **Init step**: the user gives here the time unit which will be used later by the simulator (millisecond, second, hour, day, year…), depending on the context.
- **Model Identification step**: the user first identifies the model, gives it a name, and defines its ports.
- **State Identification step**: here the user is asked to identify (and name) the different states of the system. In his point of view, there are two types of states: a set of finite states, defined only by their names, and a set of states defined by their name and their state variables.
- **Transition identification step**: in this step the G.U.I. expects a specification of the links between the states. Several parameters are checked, such as the respect of determinism, and if each state is linked to at least one another.
  Once the transitions are identified, the user can specify whether the transition is driven by time (i.e. is fired after time duration expires) or by an event from the external world. Graphically, those transitions are also different. In this case, the port where the event is expected and the type/value of the event (message, real number…) have to be specified. A list of events may be written.
- **Action identification step**: to each transition, an action can be associated. This action can modify the states, their variables, send an output (if at least one output port has been defined)…and so on.

A simple pseudo language is used to define events and actions, as we will see in the following examples.

Graphically, they can be represented easily. We give here the representations we chose for each element, and the corresponding BNF notation. We also explain how the G.U.I. helps during the modelling step.

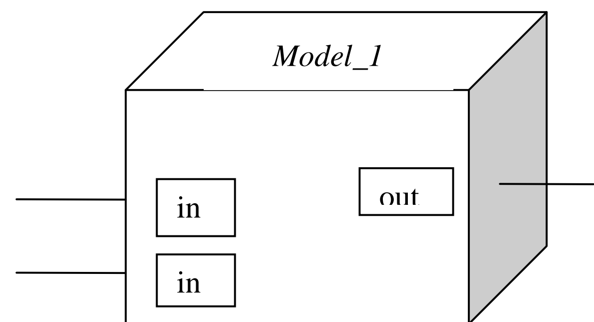

Figure 3. A simple model with ports

Figure 3 is a 3-D rectangle which shows a simple model, named Model_1, with 2 input ports (in1 & in2) and one output port (out1). A model must have a name, and may have input and/or output ports. In this case, the ports must be named. An input port takes place on the left side of the state, while an output port takes place on the right side. A

model must have, at least, one state. According to the BNF form, a model is written like that :
<model>::=<state>{<state>}{<inport>}{<outport>}

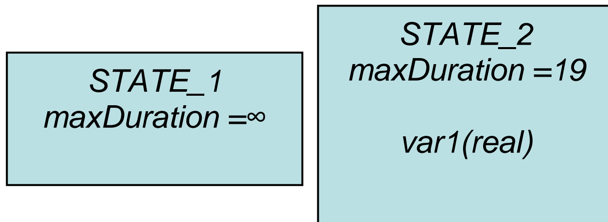

Figure 4. Two states with different durations and variables

Figure 4 shows two states, the first one “STATE_1” is a default state with an infinite duration, while the second one has a modified duration and a state variable which is a real number (and which can evolve during the simulation).
A state, once created, must be given a name.
When the user specifies a state, sometimes he adds at least one state variable which can take several numerical values, and he considers there is one single state with a state variable.
However, from the modeller’s point of view there are as many states as the state variable can have.
But we chose to represent the state as a simple one. Usually, it is easier to simulate states with a few state variables.
Each state has a time duration value (considerered as a particular state variable), by default this duration is infinite, but must be changed by the user, as we see on the previous picture. When the duration expires, it generates a particular event, because it comes from the model itself. The typical form of a state is
<state>::=“(“<max_duration>{“,”<variable>[<value>]}“)”

An event is defined as follows (using the BNF notation) :
<event>::= “(“ <inputport> | <outputport> “,” <value> “)”
For instance, if the scientist expects an input with the real value 3.14 on the third input port, the event will be:
E_1= (in3, 3.14).
An event specified using an output port will automatically be sent on the given port just before the autotransition of the current state is fired.

A transition is graphically represented by an oriented arrow between two states. There are two different arrows, depending on the transition type: if it is triggered by an external event, the arrow will be full, if it is triggered by a clock event, when maxDuration expires (we name it autotransition) it will have a thin white line inside.
Formally, a transition is written as follows (still using the BNF notation):
<transition> ::=<state>”,”<state>[<event>] [<action>],
where the event is not specified if it is a time transition, and where the corresponding action is optional.
An autotransition between two states (STATE _1 and STATE _2) can be written:
T_1::= STATE_1, STATE _2

As we see, an action can be associated to any type of transition: at this time, we write it using a simple pseudo-language, that is the reason why an action is only composed of a string. An action is often used to update the states (their lifespan and/or their state variables).
Using the BNF notation, an action can be written as follows:
<action> ::= “(“{<state>[<variable>] <value> “;”}”)”
The target-state of the action and a value must always be specified, if only a numerical value follows the state name, the action will change the maxDuration. If there is also a variable name after the state name, then the system updates the variable with the given value.
An action can be composed of several updates.
An action which changes the maxDuration of STATE_1 and gives the value 3 to a variable named var1 in STATE_2 will be written :
<A1>::=(STATE_1 maxDuration 150; STATE_2 var1 1;)
or <A1>::=(STATE_1 150; STATE_2 var1 3;)

Based on what we said, we give the metamodel of this specification language, using a UML 2.0 class diagram (see figure 5)

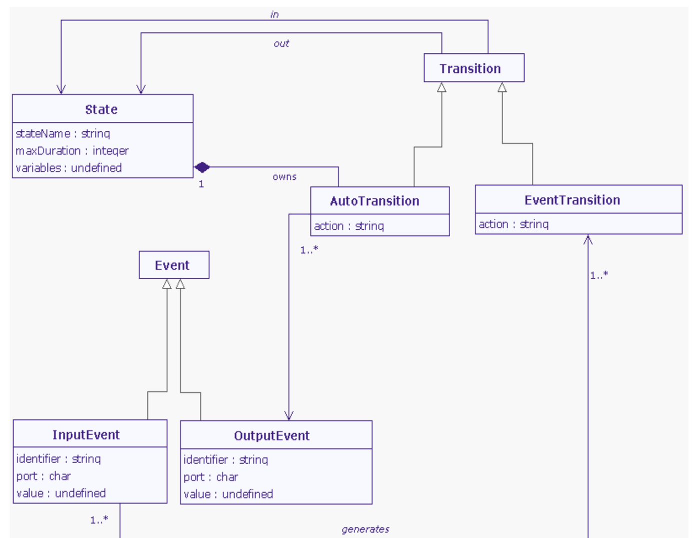

Figure 5. The language’s metamodel

4. AN EXAMPLE OF USE: A STUDENT

To illustrate our modelling method, let us take a very simple system: a student. We first study his behaviour, then we use our method to model this system.
From an expert's point of view, a student may be sleeping, working, eating or idle.
The default time a student needs to sleep is 12 hours, but this time can be reduced if an external event from his alarm clock radio disturbs him. In every case, once he wakes up, he is in an idle state, for 12 hours. As long as he remains in this state, he can go to work if he receives a message from one of his teachers, or he can start eating for 2 hours if someone calls him to eat ("come with us, let's eat !"). As a lambda student, he can not receive messages while eating. After his meal, he will say to the world that he feels good, and then he returns in the idle state. While working, the student sends regular messages to his teacher (every hour), telling him the number of exercises he did (a random number between 1 and 5). He is able to work for an hour, the he gets back in the idle state unless he receives again a message, or he falls asleep. While working, he cannot be disturbed by an incoming message.
Every operation which needs energy (working and eating) must be taken into account: the time before he goes back to sleep must decrease.

Using our modelling method, we first choose the time unit, which is one hour. The model is easy to identify, it is the student himself, he has one input port (to receive messages) and two output ports (one to send messages, the other to send values).
There are four states for this student : SLEEPING, WORKING, EATING and IDLE.
The autotransitions will be between sleeping and idle, idle and sleeping, eating and idle, working and idle. The transitions based on events will be between sleeping and idle (if the alarm-clock rings), idle and eating, idle and working.
There are 3 possible input messages : "*eat*", *bip*", "*work*".
There are 2 types of outputs, one is a message "*I feel good now !*" (after his meal) and the other is a random value between 1 and 5 (after his work).
We must remember to decrease the maxDuration of the IDLE state : this is an action to associate to transitions. Figure 6 illustrates the representation of the studied system using our specification language.

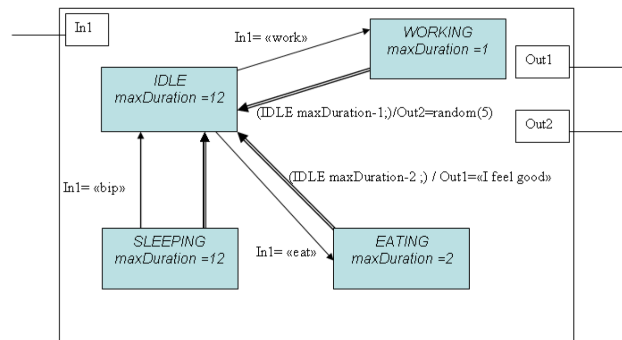

Figure 6. The student modelled

5. CONCLUSION AND FUTURE WORK

We followed step by step the methodology defined in part 3. However, the main difficulty when modeling systems is to have a valid simulator.
Every model designed with our language is platform-independent, a PIM according to the MDA terminology.
We know the DEVS metamodel, and the proposed language metamodel : hence, it is possible to transform a PIM written using our language onto a DEVS PIM using simple transformation rules (for instance, MOF QVT). MOF is a meta-metaformalism, i.e. all the metamodels can be defined using MOF.
Once the DEVS PIM created, we need to chose an Object Oriented Language in which we want to generate object code : the metamodel of this language must also be known.
An environment which supports MDA is able to generate such an object code (using templates).
Figure 7 sums up those mappings.

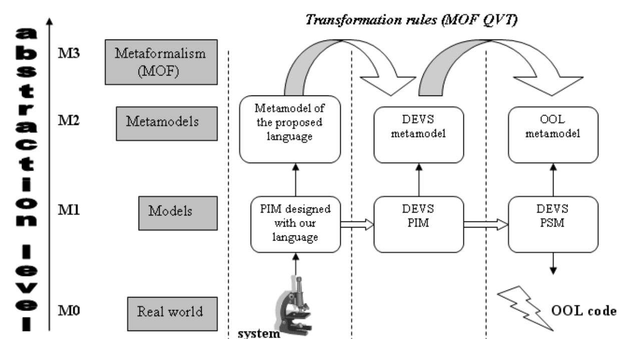

Figure 7. DEVS code generation starting from a model designed with our specification language

The aim of this paper was to present a methodology to help scientists to specify DEVS models : this methodology is applied using an intuitive specification language. Once a model created, it is interesting to perform a code generation towards a DEVS model in order to perform the simulation, that is why we chose a MDA approach : the advantages of this software engineering method could improve the reusability of the models.
We designed our model using the proposed language, the main advantages of this language is that it is graphical and intuitive.
In a near future, we plan to finish our G.U.I. and its integrated help, and to add the G.U.I. the possibility for the user to specify advanced actions (writing code for instance).
We also plan to take into account coupling relationships between models, in order to simulate DEVS coupled models. Finally, we are working on a state of art of all the MDA-Oriented Environments in order to compare them and to chose the best one to perform our mappings and our code generation towards a PythonDEVS-oriented platform.

6. REFERENCES

B.P. Zeigler, Multifaceted modelling and discrete event simulation. Academic Press, 1984.

B.P. Zeigler, H. Praehofer, and T. Kim, Theory of Modeling and Simulation, Second Edition. Academic Press, 2000.

F. Barros. “Dynamic structure discrete event system specification: a new formalism for dynamic structure modelling and simulation”. In Proceedings of *Winter Simulation Conference* 1995, 1995.

A. Uhrmarcher. “Dynamic Structures in Modeling and Simulation: A Reflective Approach,” ACM *Transactions on Modeling and Computer Simulation*, Vol. 11, No. 2, April 2001, Pages 206–232, 2001.

Y. Kwon, H. Park, S. Jung, and T. Kim. “Fuzzy-devs formalism: Concepts, realization and application”. *Proceedings AIS* 1996, pages 227–234, 1996.

N. Giambiasi and S. Ghosh, “Min-Max-DEVS: A new formalism for the specification of discrete event models with min-max delays,” *13th European Simulation Symposium*, Marseille, France, pp 616-621, 2001.

G. Wainer, C. Frydman and N. Giambiasi "An environment to simulate cellular DEVS models". *Proceedings of the SCS European Multiconference on Simulation*. Istanbul, Turkey. 1997.

J. B. Filippi, F. Bernardi, and M. Delhom. “The jdevs environmental modeling and simulation environment”. *IEMSS, Integrated Assessment and Decision Support*, Lugano Suisse, pages 283–288, 2002.

L. Zadeh, Fuzzy Sets. Inform Control, 1965.

H. Vangheluwe, “The discrete event system specification DEVS Formalism,” 2001.

http://www.omg.org/mda/

G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1998.

S. Garredu, E. Vittori, J.-F. Santucci and A. Muzy. “Specification languages as front-end towards DEVS formalism”. ISEIM: The First International Symposium on Environment Identities and Mediterranean Area, Corte, 2006.

Thomas R. Gruber (1993). Toward principles for the design of ontologies used for knowledge sharing, Originally in N. Guarino and R. Poli, (Eds.), *International Workshop on Formal Ontology,* Padova, Italy. Revised August 1993. Published in *International Journal of Human-Computer Studies*, Volume 43 , Issue 5-6 Nov./Dec. 1995, Pages: 907-928

B. Bouchon Meunier, La logique floue et ses applications. Broché. 1985.

A. Kaufmann, *Introduction à la théorie des sous-ensembles flous*. Number 1. Masson edition. 1973.

P.-A. Bisgambiglia, E. De Gentili, J.-F. Santucci and P.A. Bisgambiglia. "DEVS-Flou: a Discrete Events and Fuzzy Sets Theory-Based Modeling Environment", ISSCAA: 1st International Symposium on Systems and Control in Aerospace and Astronautics. Harbin, CHINA, 2006.

“Unified Modeling Language: Superstructure”, *V 2.0 OMG document formal/05-07-04,* april 2005

J. Bézivin, “On the Unification Power of Models”, Software and System Modeling. – 2005, Vol. 4, No. 2, p. 171-188

J. Miller and J. Mukerji, “Model Driven Architecture guide version 1.0.1”, OMG document number /03-06-01, january 2003.

[Hong et al., 2005] –**Hong, K.J.**, **Kim**, **T.G**. – *DEVSpecL: DEVS specification language for modelling, simulation and analysis of discrete event systems,* 2005

A. Tolk, J.A. Muguira, M&S within the Model Driven Architecture, *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC) 2004*