Defining DEVS Profiles for domain-specific modeling purpose

Stéphane GARREDU, Evelyne VITTORI, Jean-François SANTUCCI, Dominique URBANI
University of Corsica, UMR CNRS 6134, Quartier Grossetti, BP 52 20250 Corte, France
{garredu ;vittori ;santucci}@univ-corse.fr durbani@laposte.net

Keywords: Discrete event simulation, MDA, DEVS domain specific profiles, methodology

Abstract

Discrete EVent System Specification (DEVS) is a popular formalism which allows to specify and simulate models. Its main drawback is that its implementation is simulator-specific, i.e. models have to be programmed using an Object-Oriented Language (OOL).
In this paper, we introduce DEVS profiles, which are specializations/restrictions of DEVS meta-model, and we explain how to create a DEVS Profile for non-computer scientists. We distinguish two kinds of users: the meta-modelers and the modelers.
Models designed with DEVS profiles can be mapped onto platform specific models and to object code using a Model Driven Architecture (MDA) approach.

1. INTRODUCTION

DEVS formalism, introduced by Pr. Zeigler [Zeigler 1976] [Zeigler et al. 2000], has been used for years now in the community of modelling and simulation: it lies on a strong mathematical basis, and it is a generic formalism, based on discrete events.
It has great abilities to be extended, to fit several domains. DEVS also provides a simulator attached to each model. But the main drawback of such a formalism is its implementation: DEVS models must be implemented using an Object Oriented Language (OOL).
Hence, a scientist who would want to create and simulate his models has to know a domain and an OOL, in order to use a DEVS framework.
Our research team has been working for more than fifteen years on DEVS formalism and its applications, and a part of this team is currently working on DEVS adaptability, trying to enable non-computer scientists to use it without writing code. The main idea is to use DEVS abilities with software engineering to create models, in order to facilitate the modelling step, and improve portability and re-usability.
First of all, we compared, using several criteria, some languages and formalisms for which mappings towards DEVS had been performed, and we highlighted the need to create a new formalism [Garredu et al. 2006] and described the approach for the creation of such a formalism [Garredu et al. 2007].
Then we defined two major constraints:

a) This formalism must be easy to use
b) It must be able to be mapped onto DEVS formalism (i.e. it must be close to DEVS though)

The second constraint lead us to reason in those terms: if such a formalism is close to DEVS, we could see it as a particular specialization or restriction of DEVS formalism.
We can imagine that without modifying the formal definition of DEVS atomic and coupled models (i.e. without defining a new DEVS extension), it would even be possible to specialize them for different domains.
Such a specialization would be more interesting if it is platform independent and not tied to a particular DEVS framework.
It implies that mappings will have to be performed: to do so, we use knowledge inherited from software engineering to help scientists during the whole modelling process, and we chose to follow a MDA approach.
Model Driven Architecture (MDA) is an open approach that was created by the OMG: it is a set of standards and ideas in which every element is seen as a model. Even transformations between models are models themselves. The benefits of using such an approach are that a part of the models used are platform-independent, and thus reusable.

The main purpose of this paper is to introduce the DEVS Profile concept, replacing it in a larger context: a modelling environment.
This paper is organized as follows: the first section is dedicated to background; we will focus on the concepts used further in this paper (DEVS, UML, MDA, meta-models…).
The following section is about DEVS Profiles: we introduce DEVS meta-model then we present the concepts with which such a meta-model can be specialized: we give an example of such a specialization. We also explain how to provide a step-by-step help during all the modelling steps, considering two levels: the “modeller level” and the “user level” and we give a graphical representation of our profile.
In the last section, we conclude this paper by discussing about our methodology and our work in progress.

2. BACKGROUND

In this section we present several concepts which will be used in the description of our approach. The first part is dedicated to DEVS, its models and their implementation, the second one outlines UML properties and presents UML profiles, while the third one highlights the main features of a model-driven approach like MDA.

2.1. DEVS Formalism

DEVS formalism was introduced in the seventies by Pr. Zeigler, it is based on discrete events, and it provides a framework with mathematical concepts based on the sets theory to describe the structure and the behaviour of a system.
Almost any system can be modelled with DEVS, if it has finites states and finite transitions between its states, in a finite time interval.
The tiniest element in DEVS formalism is the atomic model (figure 1).
An atomic model is able to evolve by itself but it also can react to external events: it describes the behaviour (or a part of it) of the studied system and is defined as follows:

$AM = < X, Y, S, ta, \delta int, \delta ext, \lambda >$

- X is the input ports set, through which external events are received;
- Y is the output ports set, through which external events are sent;
- S is the states set of the system;
- ta is the time advance function (or lifespan of a state);
- δint is the internal transition function;
- δext is the external transition function;
- λ is the output function;

At the beginning of the algorithm, the model remains in a given state $s \in S$ for a duration $d = ta(s)$ if no external event occurs on any input port.
When d expires, the system performs $y=\lambda(s)$ (i.e. it sends a message on one of the output ports), then it triggers an internal transition from the current state s to another state: $\delta int(s)$.
If an external event $x \in X$ occurs before d expires, the system carries out an external transition from s to $\delta ext(s,d,x)$.
In both cases, the system is now in a new state s' during
$d' = ta(s')$ and the algorithm restarts.

A DEVS coupled model shows how atomic models are connected to each other, it gives a hierarchical view of the system and it describes the links between its sub-models. A coupled model can be composed of atomic and/or other coupled models. It can receive external events on its ports, and even link them to its sub-models ports. It also can link an output of one of its sub-models to one of its own outputs.

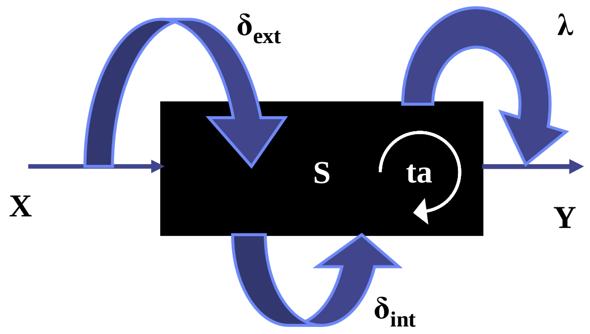

Figure 1. DEVS atomic model

.DEVS atomic and coupled models can be textually described, but, in order to be simulated, the system must be modelled within a DEVS-oriented framework: the models must be programmed using an object-oriented language.
Once an atomic model is programmed, its associated simulator is almost automatically provided, and once a coupled model is defined, its coordinator is also almost automatically provided (i.e. it coordinates all the simulators and/or the coordinators of the models it is composed of).
The simulation is initiated by the root coordinator which is the root of the hierarchical simulator.

There are several DEVS simulation environments, such as CD++ [Wainer, 2002], a framework which uses C++, or JDEVS [Filippi et al. 2004] which uses JAVA.

2.2. UML and MDA

Unified Modeling Language is a graphical set of modeling formalisms: it provides a toolkit which enables 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 2.3 [OMG 2010].
Its main advantage is that it is considered as a standard formalism by a large worldwide community of users.

An UML model (Class diagram, Use Case diagram…) 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 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 it is defined on itself, i.e. it is described in MOF terms. Hence, there is no level higher than M3 (figure 2).

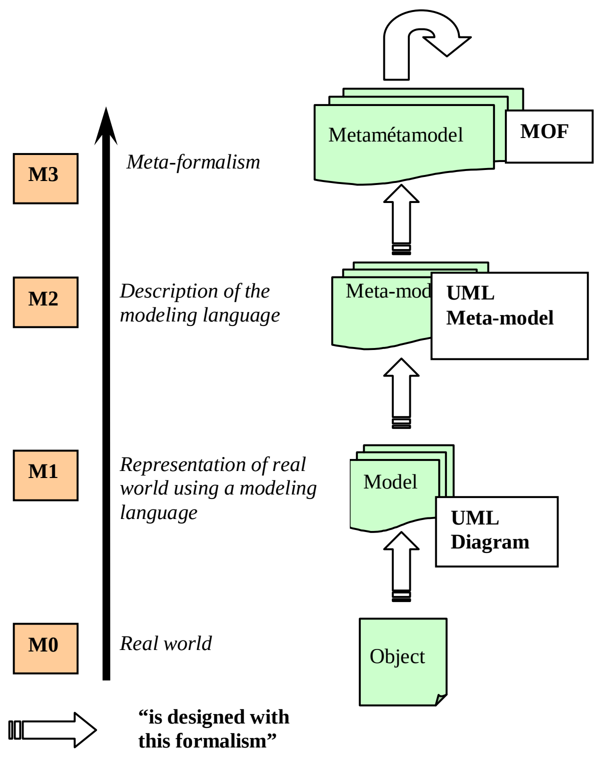
M3
Meta-formalism
Metamétamodel
MOF
M2
Description of the modeling language
Meta-mod
UML Meta-model
M1
Representation of real world using a modeling language
Model
UML Diagram
M0
Real world
Object
"is designed with this formalism"

Figure 2. Modeling levels hierarchy and relationships with UML

MDA (Model Driven Architecture) [MDA 2001] is a software design approach initiated by the OMG in 2001 to introduce a new way of development (figure 3) 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 to transform a PSM into source code.
The great 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 MOF which appears to be the most basic standard.

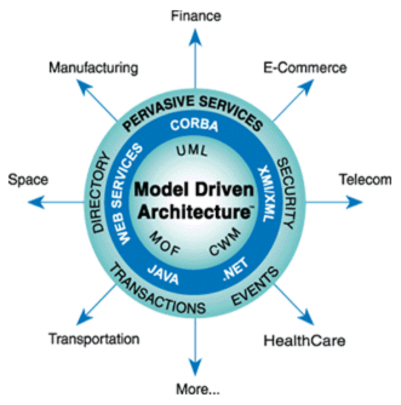

Figure 3. Overview of MDA abilities

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. UML still provides a common and useful visual notation for the description of software artifacts at several levels and from several points of view.
Theoretically, a MDA process can be followed without using UML. However, OMG considers it as a favourite formalism, arguing that it has become a real used standard and that its meta-model is fully defined

Since its 1.3 version, UML also includes a set of mechanisms which provide a help during the modelling process, in all its steps: UML Profiles [Fuentes-Fernández et al. 2004].
UML profiles allow customizing UML meta-model by specializing some of its elements, for instance putting restrictions on them, but without adding anything to it: this meta-model has to be considered, following the OMG, as a "*read-only model, that is extended without changes by profiles*".
A Profile can be defined to fit a particular platform (J2EE, .NET…) or domain (real-time…). It must be able to be applied to or retracted from a model.
Profiles are composed of three elements: *stereotypes*, *tagged values*, *constraints:* their purpose is to customize the meta-model.
Those elements allow the use of a terminology that can fit particular domains and the use of different symbols than the usual ones supplied by UML. They also enable to specify existing semantics or to add new ones to the meta-model, to give additional information when a model is transformed into another model (mapping rules), to restrict the meta-model by adding it constraints (written in Object Constraint Language for instance).

Even if our approach is not directly linked to UML profiles (even if they are a part of MDA architecture, and used in model mappings), we thought it was necessary to take a glance at them and their philosophy: indeed, as UML profiles allow the specialization of UML meta-model by adding restrictions to it, DEVS Profiles are a way to specialize DEVS meta-model: the following section explains how.

3. DEVS PROFILES

In this section, we explain our vision of DEVS extensibility, introducing DEVS Profiles. Such profiles can be built for a domain-specific modeling purpose.
As it was mentioned before, extending DEVS with profiles does not add anything to DEVS meta-model, i.e. it is not added new elements (neither new sets, nor new functions…). We only specialize DEVS meta-model by specifying it and modifying its classes.

3.1. The DEVS meta-model

The DEVS meta-model can be designed using UML class-diagrams. To do so, the first step is to identify the classes. A coupled model is composed of two models at least, whether they are atomic or coupled.
Every atomic model has at least one state. A state must have at least one variable defined: the duration of lifespan of the state. Hence, it must have at least one transition defined: the one accessed with $\delta int(s)$. A transition is a path from one state to another one, and it can be internal or external.
Every kind of model has ports. A port can be waiting for external messages (input events), which will trigger δext function, or it can send messages (output events) when λ function is triggered. λ can be added associated code. A port is able to belong to a coupled or an atomic model and it can be linked to another port. Figure 4 shows a basic DEVS meta-model.

3.2. DEVS profiles philosophy

The DEVS meta-model must be seen as a starting point for defining DEVS profiles.
In our philosophy, DEVS meta-model can not be added new classes, because it would mean that DEVS models definition have changed. The purpose of DEVS profiles is not to create a new DEVS extension but to simplify the modeling process, and of course simulation.
The user who wants to create a DEVS profile is seen as a "meta-modeler". He defines the concepts which will be handled by the final user, or "modeler".
Creating a DEVS profile allows the meta-modeler to modify classes in order to adapt the DEVS meta-model to a particular domain. Classes variables which remain undefined can be given a particular type, relationships between classes can be redefined, and the attributes of the classes can be changed if those changes are compatible with the initial DEVS meta-model.
The concepts handled by the meta-modeller can be given a graphical notation. The meta-modeler uses a meta-modeling framework.
The modeler, who can be the meta-modeler himself or another person, will use this meta-model within a modeling framework. He will design his models using the concepts defined at the higher level by the meta-modeler.
The metamodeler must define the rules which will link the modified metamodel to the basic one.
Such a user designs UML models without being aware of the code used in the UML modeling framework he is using, the modeler designs his DEVS models without having any knowledge of the code behind. Once his models defined, they can be mapped onto DEVS PIMs, those DEVS PIMs onto a DEVS PSM, and this PSM onto object code in order to be simulated: the main idea is the transparency of those operations.
A meta-modeler has to know DEVS formalism, whereas a modeler has not.

3.3. A formalism which can be defined as a profile

In [Garredu et al. 2009], we defined a new formalism to help non computer scientists to specify DEVS models.
As this formalism is not too different from DEVS formalism, we show here how to create a profile for this language.
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. 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.
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 modeler'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 (considered as a particular state variable), by default this duration is infinite, but must be changed by the user. When duration expires, it generates a particular event, because it comes from the model itself. An event is defined with an input/output port, and a value.
An event specified using an output port will automatically be sent on the given port just before the autotransition of the

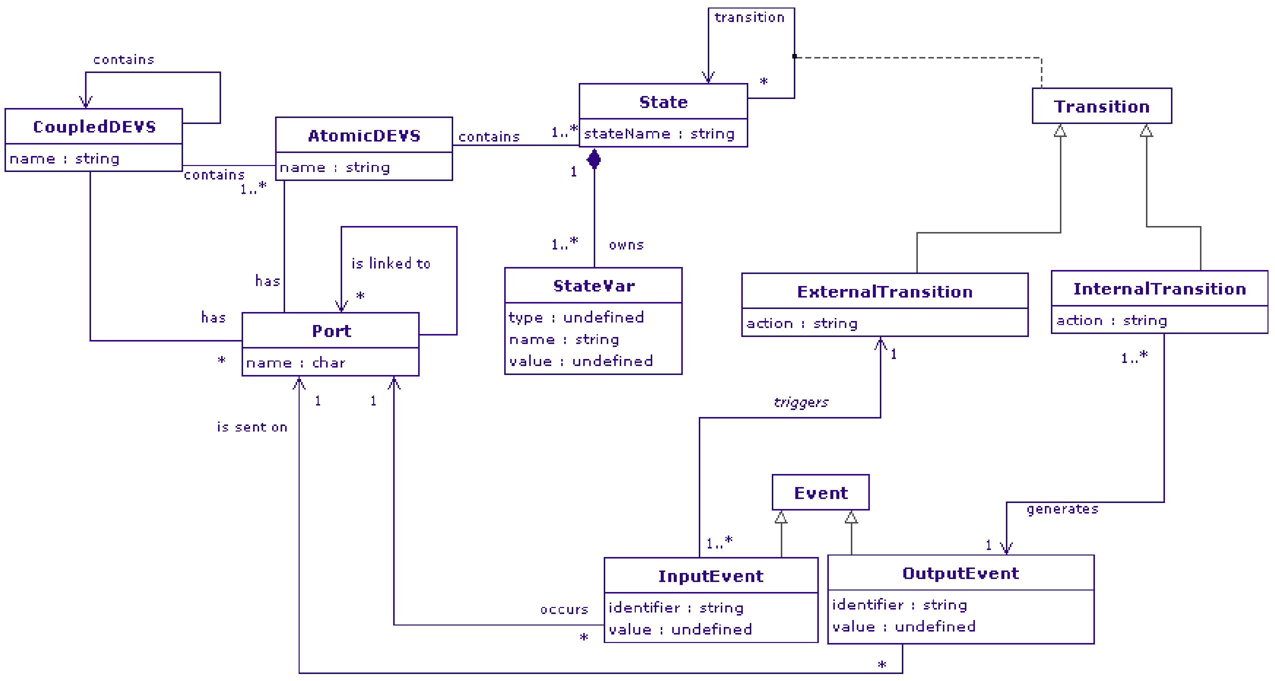

Figure 4. A basic DEVS metamodel

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.

3.4. Defining the profile

Starting from the DEVS meta-model, we can define the profile of this formalism.
As its purpose is to enable non-computer scientists to define simple DEVS models, we can only work on DEVS atomic models. We do not use the CoupledDEVS class anymore. The ExternalTransition and InternalTransition classes become EventTransition and AutoTransition.
As links between ports no longer exist, we can suppress the Port class and use ports as simple attributes of EventTransition and AutoTransition
The AtomicDEVS class can be removed, it is not used by our formalism, as long as there are no connexion between two AtomicDEVS.
We consider maxDuration as a mandatory attribute of the State class, and we consider that a state may have variables seen as attributes. We do not use the StateVar class anymore. We give in figure 5 the meta-model of the DEVS Profile for non-computer scientists which could have been designed by the meta-modeller.

3.5. Using the profile

Now, the modeler can use this meta-model in a modeling framework in order to create a model.

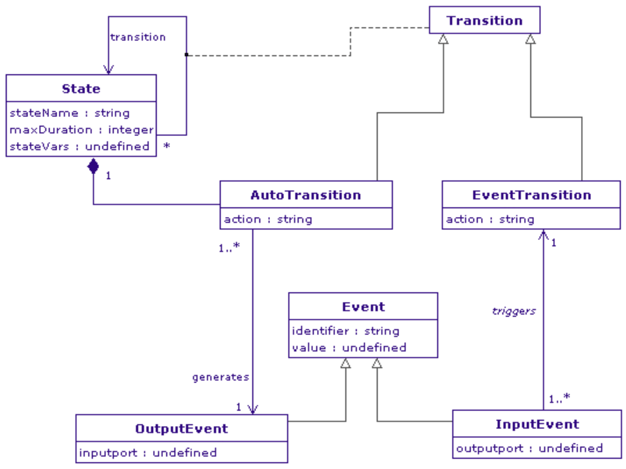

Figure 5. The tailored DEVS metamodel which describes our profile

Such a model can be under the form on the example shown on figure 6.
It is helped by a contextual step by step defined by the meta-modeler himself. At a higher abstraction level, the tailored meta-model can be linked to DEVS basic meta-model using rules. This can be done under a MDA approach. The purpose of this approach is to define links between those

two meta-models, in order to transform a DEVS Profile PIM to a basic DEVS PIM.

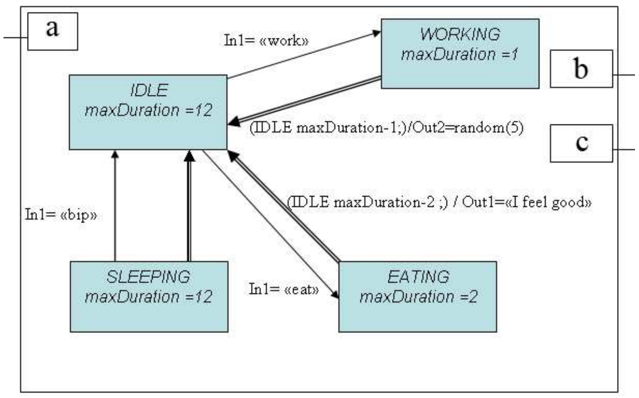

Figure 6. An example of a model designed using our profile

Those links can be expressed with MOF QVT. Then, this basic DEVS PIM could be mapped onto a DEVS PSM and this PSM can be used to generate the corresponding classes in a DEVS-oriented framework. MDA-compatible software are able to do so.
This approach is summed-up in figure 7.

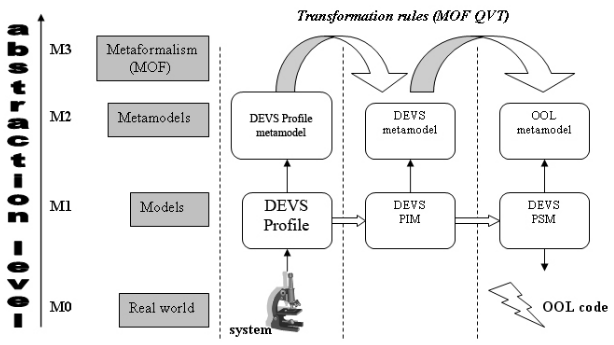

Figure 7. A MDA approach to transform DEVS Profiles onto code

CONCLUSION

We shown in this paper how to define DEVS Profiles. Such profiles could be used to specialize DEVS meta-model for domain-specific purpose. We chose the example of a profile for non-computer-scientist. We are currently programming an environment which enables the meta-modeler to create DEVS profiles, which could be used by the modeler to design models according to this profile. This modeler does not need to know a programming language to create his models.
Following a MDA approach, those models will be mapped onto object oriented code. We are using MDA-oriented software, such the Eclipse plugin Acceleo, in order to complete those mappings.

Reference List

[Booch et al. 1998] – G. Booch, J. Rumbaugh, and I. Jacobson. “The Unified Modeling Language User Guide”. Addison-Wesley, 1998.

[Filippi et al. 2004] – Jean-Baptiste Filippi and Paul Bisgambiglia JDEVS: “an implementation of a DEVS based formal framework for environmental modelling” Original Research Article Environmental Modelling & Software, Volume 19, Issue 3, March 2004, Pages 261-274

[Fuentes-Fernández et al. 2004] – Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. “An Introduction to UML Profiles” in UPGRADE : European Journal for the Informatics Professional, Vol. V, No. 2, April 2004

[Garredu et al. 2006] – S. Garredu, E. Vittori, J.-F. Santucci and A. Muzy. “Specification Languages As Front-end Towards DEVS Formalism”, ISEIM 2006: The First International Symposium on Environment Identities and Mediterranean Area, Corte

[Garredu et al. 2007] – Garredu, S., P.A. Bisgambiglia, E. Vittori and J.F. Santucci. 2007. “Towards The Definition Of An Intuitive Specification Language”. In Proceedings of the Simulation and Planning in High Autonomy Systems (AIS) & Conceptual Modeling and Simulation (CMS).

[Garredu et al. 2009] – Garredu, S., E. Vittori and J.F. Santucci. 2009. “A DEVS-oriented intuitive modeling language”. In SpringSim Proceedings of the 2009 Spring Simulation Multiconference, article n°155

[MDA 2001] http://www.omg.org/mda/

[OMG 2010] OMG Unified Modeling Language: Superstructure, version 2.3
http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

[Wainer, 2002] – Wainer, G. “CD++: a toolkit to define discrete event models”. Software, Practice and Experience. Vol.32, No.3. pp. 1261-1306. November 2002

[Zeigler 1976] – Zeigler, B. *Theory of Modeling and Simulation* – Academic Press, 1976

[Zeigler et al. 2000] – Zeigler, B. P., H. Praehofer, and T. Kim. 2000. *Theory of Modeling and simulation*. 2nd ed. Academic Press