Specification languages as front-ends towards the DEVS formalism

Stéphane Garredu, Evelyne Vittori, Jean-François Santucci, Alexandre Muzy

SPE UMR CNRS 6134 University of Corsica20250 CORTE
{garredu, vittori, santucci, a.muzy}@univ-corse.fr

***Abstract* - This paper constitutes a state-of-the art of specification languages relevant to be used as front-ends towards the DEVS (Discrete EVent System Specification) formalism. Comparison criteria are defined to evaluate specification languages for the description of DEVS structures. Finally, the need for building an original front-end, accounting for the whole criteria, is discussed.**

Index Terms - Modeling, Discrete event simulation, Specification languages.

I. INTRODUCTION

In simulation, once a model is simulated, usually two people at least worked on it: the domain specialist and the computer science one. On the other hand, in spite of the increasing number of specialists creating and simulating specific models, few are able to create and translate automatically a given model into a simulation one.

To achieve this goal, DEVS can be used. This powerful framework proved to be relevant for helping people in the definition of many kinds of simulation systems.However, this framework can be tricky to manipulate for non-computer science specialists. It takes time to the domain-specialist to learn both DEVS and Object-Oriented Programming.

The idea here is to enable a domain specialist to design a model using specifications at a higher level, with its own notations. After, a modelling tool can be used for the automatic mapping to DEVS for simulation.

In this paper a study is achieved of relevant well-known formalisms for which DEVS mappings have already been defined. Marks are given according to a criteria list. Using this simple method helps to find the best language to do so.

The rest of the paper is organised as follows. Section two presents the background attached to the DEVS formalism. A set of comparison and analysis criteria is introduced in section three. Section four presents the four specification languages selected here. A comparison of these languages is presented in section five according to the criteria defined in section three. Finally the conclusions and perspectives are given in the last section.

II. BACKGROUND ON THE DEVS FORMALISM

The Discrete EVent System Specification (DEVS) is a mathematical formalism for describing discrete event systems. DEVS is grounded in general systems theory. Within DEVS, discrete event systems are described by two kinds of structures. Atomic models describe the behaviour of non-decomposable units via event-driven state transition functions. More complex models can be constructed hierarchically by coupling atomic models into coupled models.

An Atomic Model (AM) DEVS is a structure:

$$AM =< X, Y, S, \delta_{ext}, \delta_{\text{int}}, \delta_{conf}, \lambda, t_a >$$

Where,

X is the set of input values, Y is the set of output values, S is the set of sequential states, $\delta_{ext} : Q \times X^b \to S$ is the external transition function (where $Q = \{(s,e) / s \in S, 0 \le e \le t_a(s)\}$ is the total set of states, e is the elapsed time since the last state change, and $X^b \in X$ is a bag of input values), $\delta_{\text{int}} : S \to S$ is the internal transition function, $\delta_{conf} : S \times X^b \to S$ is the confluent transition function, $\lambda : S \to Y^b$ is the output function, $Y^b \in Y$ is a bag of output values, and $t_a : S \to \Re^+$ is the time advance function.

The external transition function describes how the system changes state in response to an input. When an input is applied to the system, it is said that an external event $x \in X$ has occurred. The next state *s'* is then calculated according to the current state *s*, the elapsed time *e* and the value of the external event x : $s' = \delta_{ext}(s,e,x)$.

The internal transition function describes the autonomous (or internal) behavior of the system. When the system changes state autonomously, an internal event is said to have occurred. The next state *s'* is calculated only according to the current state *s*: $s' = \delta_{\text{int}}(s)$.

The confluent transition function determines the next state of the system when an external event and an internal event coincide. The default definition of the confluent transition function is: $\delta_{conf}(s,x) = \delta_{iext}(\delta_{\text{int}}(s),0,x)$. This is the same approach as the classic DEVS. But here the internal transition is allowed to occur first and to be followed by the effect of the external transition function on the resulting state: $\delta_{conf}(s,x) = \delta_{\text{int}}(\delta_{ext}(s,ta(s),x))$. Finally, another behavior can be defined.

The output function generates the outputs of the system when an internal transition occurs.

The time advance function $t_a(s)$ determines the amount of time that must elapse before the next internal event will occur, assuming that no input arrives in the interim.

An atomic model is in a *passive* state when $t_a(s) = \infty$. An atomic model is in a *active* state when $0 \le t_a(s) < \infty$.

An atomic model allows to specify the behavior of a system. Connections between different atomic models can be performed by a coupled model (CM):

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$$

Where,

X is the set of input values, Y is the set of output values, D is the set of model references, For each $i \in D$, M_i is an atomic model, I_i is the influencer set, $Z_{i,j}$ is a function: the *i* to *j* output translation with (*c.f.* Figure 1): $Z_{MC,j} : X_{MC} \to X_j$ is the input coupling function, $Z_{i,MC} : Y_i \to Y_{MC}$ is the output coupling function, and $Z_{i,j} : Y_i \to X_j$ is the internal coupling function.

III. Criteria for Specification Language Analysis

This section defines the basic goals of a DEVS front-end. Several specification languages and formalisms are compared. Most have already been related to the DEVS formalism.

Criteria are splitted into two parts: the mandatory ones, related to the basic requirements for the language to be mapped to DEVS from a high level of specification, and the extension ones, related to ability of the language to deal with complexity.

A. Mandatory Criteria

1) Temporal aspects

We mean by "temporal aspects" the way time goes by, concerning the relationships between state and time. If state durations can be described, the language should be able to account for temporal aspects. Of course this is to be related to the *ta* (Time Advance) function of DEVS atomic models. In discrete event simulation, one of the most important principles is the strong link between an event and its occurrence date.

2) Transition and output Functions

This criterion considers the capability of the language to map specifications to transition and output functions of DEVS atomic models. These functions allow managing between external and internal events. Indeed, when an event occurs, it is necessary to specify if this event comes from outside the model (in case of an external event, activating an external transition function), or if this event was internally triggered (in case of an internal event, linked to an internal transition function).

On the other hand, we need to know when output external events are sent on output ports, through the output function.

3) Composition hierarchy

In DEVS, coupled models are a description of the hierarchy used to describe a model. Atomic models have a clearly defined set of inputs and outputs, and

its state space is completely encapsulated.

For example, let's take the following coupled model : CM1 = [AM1 + CM2 [AM2 + AM3]].

A language, which could interpret this kind of description, would have the ability to work using a composition hierarchy.

4) Understandability

This criterion means that a language should be easy to understand and to use for non-computer scientists.

Some modeling languages are easier to understand than others. This depends on the ability of a modeling language to abstract, merge and organize operations and information managements. For instance, the closer to the "natural language" a programming language is, (*i.e.,* close to the way we represent ideas with words), the easier it is to understand. On the other hand, formal languages appear to be not easy to understand, as they require expertness in mathematics. Finally, modifiable visual and domain-specific notations enhance understandability. Hence, for easy understanding, a language must provide high level visual (or textual) notations. This language can include graphical notations (arrows, lines, circles and basic pictures) as well as textual structures to specify details or constraints.

B. Extension Criteria

In computer science, a complicate system contains much information, exchanged and computed by many components. Then, the complexity of a real system corresponds to the degree of interactions of the sub-systems. Having tools for adaptive-based and distributive-based descriptions of particular sub-problems, allows tackling this complexity. Then, if structure changes can be depicted, a complex system can be even more easily described.

1) Adaptability

As the specification language aims to be dedicated to generic DEVS models, it does not have to be attached to a specific domain. However, it must be adaptable to fit different kinds of application areas. Adaptability is closely related to extensibility but it does not only means ability to be extended but also ability to be specialized or restricted.

This feature may be evaluated in two directions. On the first hand, experimentally, if some extensions have already been defined for a language, this can show a good level of extensibility. We have to appreciate the complexity of the existing extension processes and their potentiality to be reused and adapted to define new ones. On the other hand, the structure of the language it self may allow it to get specialized or extended more or less easily. For instance, if its abstract syntax is specified through a meta-model, this will increase its power of adaptability especially if the design of its basic structures is based on object oriented patterns. New features may then be introduced without corrupting its basic elements. Mechanisms allowing such incremental extensions may also be provided.

If adaptability is good, the language will have many abilities to fit several different domains.

2) Dynamic structures

This criterion denotes the ability of a language to describe dynamic changes in structure during a simulation. Using dynamic structures allows to more faithfully and truly describe a system. Complexity is more easily depicted. Changes in structure can be the addition and deletion of components, as well as couplings between them and changes in their internal structure. An extension of DEVS, called Dynamic Structure DEVS (DSDEVS) [3] has been designed.

3) Cellular models

In the real world, complex systems necessitate the spatial description of interacting sub-systems. In simulation, the description of sub-systems through interacting components can be achieved trough cellular models. A cellular model contains many sub-components (called cells), whose states timely depend on other influencing components.

IV. Specification Languages for DEVS Models

In this section, we present a brief overview of four formalisms for which DEVS mappings have already been defined. We chose to focus on general-purpose formalisms rather than on domain-specific languages.

A. Statecharts

Statecharts are a high-level graphical-oriented formalism used to describe complex reactive systems. They have been developed by David Harel [4] and are an extension of state-transition diagrams (the diagrams representing a Finite State Machine or Automaton) [5]. They were added three concepts: orthogonality (the way parallel activities are achieved), composition hierarchy (depth nesting of states), and broadcast communication (events sent from one to many elements).

Statecharts are composed of eight basic elements: Labels, Transitions, States, Actions, Conditions, Events, Expressions and Variables.

The statechart formalism has been evolving for years now. The most popular derived formalisms are the Classical UML, and those implemented by Rhapsody [6], [7]. Moreover, several other semantics provide them a pretty good evolutivity.

Recently, an extension able to include probabilities has been proposed [8]. With this extension, it would be possible to specify some very simple fuzzy functions if the future.

A mapping from Statecharts to DEVS has been proposed [9].

B. Petri nets

Petri Nets are a low-level graphical modeling formalism, developed by Carl Adam Petri in the sixties [10]. The basic elements are Places, Transitions, Arcs, Arc Weights and Tokens.A set of rules defines the semantics for model dynamics.

A variant of Petri Nets named Grafcet is often used in the industry [11]. Even a non-computer scientist can use Petri Nets for modeling simple systems.

In a basic Petri net, conditional transitions are fulfilled without timing constraints.

Extensions have enabled them to be hierarchical (Hierarchical Petri Nets), and to take into account temporal aspects with synchronised Petri Nets for instance [12]. There are many other extensions, such Time Petri Nets, Timed Petri Nets, Colored Petri Nets, etc.

An implementation where Petri Nets are mapped into DEVS formalism has been proposed [13].

C. Timed Input/Output Automata

Timed Input Output Automata provide a mathematical framework to describe and analyse mathematical systems [14].

Timed I/O Automata are eventually non-deterministic machines that can represent systems and components.

A Timed I/O Automata is composed of states described by state variables (each state variable has both a static type and a dynamic type). A state can change when actions occur (by instantaneous discrete transitions). It can also change following a trajectory (a function of time which describes how the state changes between discrete transitions)

Actions are classified as external or internal.

The TIOA (Timed Input/Output Automata) is a programming language used to specify Timed I/O Automata [15]. It is a variant of the IOA language used with untimed input/output automata. A tranformation method from Timed I/O Automata to DEVS models has been proposed [16].

D. DEVSpecL

DEVS SPECification Language is a modeling language for DEVS models which supports development of design tools [17]. Its built-in types include float, integer, string, void and time.

This modular language is able to specify the structure of atomic models and coupled models.

A DEVSpecL function can support random number generation functions and also links to other object-oriented programming languages.

DEVSpecL syntax is similar to those of classical object-oriented programming languages.

V. Analysis and Comparison

Each criterion, applied to a specification language is given a mark. There are 3 different marks: BAD = **X**, MEDIUM = ~, GOOD = √.

Moreover, each criterion has a weight. This

weight can be '***' for the basic criteria, '**' for the intermediate criteria, and '*' for the optional one. Of course, this weight always remains the same for each criterion applied to each language.

The marks were given taking into account the possible extensions of a language or a formalism, for example the classical Petri Nets do not have the ability to take into account temporal aspects, whereas the synchronized ones do.

Figure 1 is an array giving us an overview of the main advantages and drawbacks of each language studied according to our criteria.

When a mark is given, a language and all its extensions have been considered. That is why Petri Nets are said to be able to take into account temporal aspects even if time was not supported by the basic formalism.

All the studied languages and formalisms can deal with time. All got the best mark.

Due to its closure to DEVS, DEVSpecL is the only one able to perfectly differentiate functions, while the other studied modelling languages offer some similar properties, more difficult to use.

Two languages, among those studied, have a bad understandability: DEVSpecL and TIOA. Due to its closure to DEVS, DEVSpecL is hard to understand for a non-computer scientist, and the TIOA because of the strong mathematical basis required to use them got a bad mark too.

On the contrary, Petri Nets (and their variant Grafcet) and Statecharts, due to their clear graphical formalism, are easier to use and to understand.

Statecharts and DEVSpecL are able to make models with complex composition hierarchies. This ability is one of the most important properties of Statecharts, and its closure to DEVS explains once again the good mark given to DEVSpecL for this criterion.

Criteria	**W**	St.	P.N.	TIOA	DSpec
Understandability	***	~	~	**X**	**X**
Temporal aspects	***	√	√	√	√
Differentiate functions	***	~	~	~	√
Composition hierarchy	***	√	~	~	√
Adaptability	**	√	~	~	~
Dynamic structures	**	**X**	**X**	**X**	√
Cellular models	*	**X**	**X**	**X**	**X**

Fig. 1. : Criteria applied to the studied languages

VI. Conclusions

A set of criteria, identifying the basic requirements of a front-end specification language, for DEVS models, and dedicated to non-computer scientists, has been proposed.

As shown in Figure 1, as far as we know there are no any existing formalisms fitting correctly our needs. The main limit consists in their understandability. Nevertheless, Statecharts would be the one, which seems to have the best advantages. Many of these advantages are due to its great adaptability.

A perspective of this work is to investigate in more details the features of the standard modelling language UML 2.0 [18], [19]. It includes a special version of Statecharts called "state machines". Furthermore, UML defines other new interesting formalisms as timing diagrams, dedicated to the support of temporal aspects. Extension mechanisms are also provided to customize these formalisms and even extend them to specific domains.

A choice will have to made now between: (i) designing a completely new language, or (ii) defining an extension/restriction of an existing adaptable formalism such as UML.

Anyway, this will leads us to a Model Driven Engineering approach (MDE) [20], [21]. Meta-modelling will be the key for specifying the language and then its automatic transformation process into DEVS formalism.

VII. REFERENCES

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

[2] B.P. Zeigler, *Theory of Modeling and Simulation* , John Wiley, New York, 1976.

[3] Barros F.J. *The Dynamic Structure Event System Specification formalism*, Transactions of the Society for Computer Simulation International, Vol. 13, No 1, 1996.

[4] D. Harel, "Statecharts : A visual formalism for complex systems", *Science of Computer Programming*, 8(3):231-274, 1987.

[5] J.E. Hopcroft, R.Motwani, J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation,* 2nd edition. Addison Wesley. 2001

[6] M.L. Crane and J. Dingel, "UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal", *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML'05). Montego Bay*, Jamaica. October 2005.

[7] D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts", *Preliminary Version., SoftSpez Final Report,* 2004 , 325-354

[8] N.L. Vijaykumara, S.V. Carvalhoa, V.M.B. Andradeb, V. Abdurahimanc, "Introducing probabilities in Statecharts to specify reactive systems for performance analysis", *Computers & Operations Research 33*, (2006) 2369–2386

[9] S. Borland and H. Vangheluwe, "Transforming Statecharts to DEVS", *In Proceedings of the 2003 SCS Summer Computer Simulation Conference,* Montreal, QC. Canada. 2003.

[10] C.A. Petri, "Introduction to general net theory", in: W. Brauer (Ed.), Net Theory and Applications, *Proc. Advanced Course in General Net Theory, Processes and Systems,* Springer-Verlag, New York, 1980.

[11] R. David and H. Alla, *Petri nets and Grafcet: Tools for modeling discrete event systems*, 1992, Prentice Hall.

[12] M. Moalla and J. Pulou and J. Sifakis – "Synchronized Petri nets: A Model for the Description of Non-autonomous Systems", *7th MFCS, LNCS 64, Springer Verlag,* 1978, pp. 374-383

[13] C. Jacques, G. Wainer, "Using the CD++ DEVS Toolkit to Develop Petri Nets", *In Proceedings of the 2002 Summer Computer Simulation Conference,* San Diego, U.S.A. 2002.

[14] R. Alur and D.L. Dill, "A theory of time automata", *Theorical computer science*, Vol. 126, No 2, pp 183-235

[15] D. Kaynar, N. Lynch, S. Mitra, and S. Garland. "The TIOA language", *MIT CSAIL, Cambridge, MA,* USA, February 2005.

[16] N. Giambiasi, J.L Paillet, F. Châne, « From Timed Automata to DEVS Models », *in proceedings of the 2003 Winter Simulation Conference,* New Orleans, LA 2003.

[17] K.J. Hong and T.G. Kim, "DEVSpecL : DEVS specification language for modeling, simulation and analysis of discrete event systems", *Information and Software Technology*, Vol. 48, No. 4. (April 2006), pp. 221-234.

[18] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, (2nd ed.), Addison-Wesley, Boston (2004)

[19] "Unified Modeling Language: Superstructure", *V 2.0 OMG document formal/05-07-04,* april 2005.

[20] J. Bézivin, "On the Unification Power of Models", *Software and System Modeling. – 2005*, Vol. 4, No. 2, p. 171-188

[21] J. Miller and J. Mukerji, "Model Driven Architecture guide version 1.0.1", *OMG document number /03-06-01*, january 2003.