Semantics for Model-Based Validation of Continuous/Discrete Systems

L. Gheorghe, F. Bouchhima, G. Nicolescu, H. Boucheneb
Ecole Polytechnique de Montreal, Montreal, Canada
luiza.gheorghe, gabriela.nicolescu@polymtl.ca

Abstract

Continuous and discrete components can be integrated in diverse systems including defense, medical, electronic, communication, and automotive applications. Given the heterogeneity of concepts that have to be taken into consideration, their design involves overcoming specific global modeling and validation challenges. This paper presents semantics for model-based validation of continuous/discrete systems. It focuses on the simulation interfaces semantics, representation and verification. The proposed approach is applied for the validation of a continuous/discrete medical system, an automatic glycemia level regulator.

1. Introduction

Today, systems-on-chip are growing in complexity as a result of not only a higher density of components on the same chip but also because of the heterogeneity of different integrated modules that are particular to different application domains. Many fields benefit from the continuous/discrete (C/D) system's advantages, among them the defense, medical, electronic, and communication. Given the diversity of concepts manipulated, the global design specification and the validation are extremely challenging [7].

In a C/D system, we find two types of models:
- continuous models where the computation is realized in the continuous domain by solving differential or algebraic equations;
- discrete models where the computation is realized in cycles and every cycle represents the computation of a selected sub set of variables.

Currently, one of the methods used for the heterogeneous systems validation is the co-simulation. The co-simulation allows joint simulation of heterogeneous components with different execution models. The main advantages of this technique are to allow the designer to use the best execution model and tool for each domain and to provide the capabilities to validate the overall model. The co-simulation based validation of C/D systems has to take into consideration several execution semantics. This implies a complex behavior for the simulation interfaces as their design is time consuming and a significant source of error. One of the key issues for the design of these interfaces is the rigorous definition of their semantics, their modeling, and verification.

This paper proposes the semantics for the global validation of C/D systems using Discrete Event Systems Specification (DEVS) formalism and timed automata. We focus on the simulation interfaces semantics and their formal representation and verification. We illustrate the proposed approach using a C/D medical system, the glycemia regulator.

The article is structured as follows. Section 2 presents the main existing approaches for the C/D systems simulation and formalization. Section 3 introduces the basic concepts used in this paper. Section 4 proposes the operational semantics and the formalization of the interfaces. Section 5 presents the application and the experimental results. Finally, section 6 gives our conclusions.

2. Related work

The existing work in the C/D systems validation falls into one of the following categories: simulation-based and formal representation-based approaches.

In the simulation-based group the utilization of a single language for the specification of the C/D system is proposed. These tools may be obtained by extension of existing HDLs [4], [6], [11], [13]. This requires the abandonment of well established efficient tools for the continuous domain (ex. Simulink). There are tools such as Ptolemy in which the systems are designed by assembling together different components [14]. Ptolemy provides formal representation; however, the formal verification of the simulation models is not considered.

The formal representation-based approaches propose different definitions for heterogeneous systems modeling. In [9], a formal framework for comparing computation used in heterogeneous models is presented. The role of the model of computation in abstracting functionalities of complex heterogeneous systems was presented in [8] where the formalization of the heterogeneous systems is realized separating the communication and the computation aspects. In [15] the author introduced a formalism defined for the modeling and simulation of discrete event systems (Discrete EVent System

978-3-9810801-3-1/DATE08 © 2008 EDAA

Specifications - DEVS) where the time advances on a continuous time base. Based on this formalism [3] proposes a tool for the modeling and simulation of hybrid systems using Modelica and DEVS. None of these works focused on the representation and verification of simulation interfaces.

The main contributions of this paper, compared with the presented works are:

- The definition of the semantics for the continuous and the discrete simulation interfaces using DEVS formalism and their representation using DEVS models and timed automata.
- The formal verification of the simulation interfaces.
- The application of the presented approach for the model-based validation of a C/D medical system, an automatic glycemia level regulator.

3. Basic concepts

This section introduces the basic concepts that were used in this work.

3.1 Discrete Event Systems Specification

Discrete Event Systems Specifications (DEVS) is a formalism supporting a full range of dynamic systems representation, hierarchical and modular model development. It enables the symbolic specification of systems semantics.

A DEVS is defined as a structure [15]:

$DEVS = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where

$X = \{(p_d, v_d) | p_d \in InPorts, v_d \in X\, p_d\}$ set of *input* ports and their values in the discrete event domain,

S - set of *sequential states*

$Y = \{(p_d, v_d) | p_d, \in OutPorts, v_d \in Y\, p_d\}$ set of *output* ports and their values in the discrete event domain.

$\delta_{int} : S \rightarrow S$ the *internal transition* function

$\delta_{ext}: QxX \rightarrow S$ the *external transition* function, where:

$Q=\{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$ set of total state,

e is the *time elapsed* since the last transition

$\lambda: S \rightarrow Y$ output function

$t_a: S \rightarrow R^{+}_{0,\infty}$ set of positive reals with 0 and ∞.

The system's state is, at any time ***s***. There are two possible situations:

- when no external events occur. In this case the system stays in this state ***s*** for the time $t_a(s)$. When the elapsed time ***e*** equals $t_a(s)$ (that is the time allocated for the system to stay in state ***s***), the system outputs the value $\lambda(s)$. The state ***s*** changes to the state ***s'*** as a result of the transition $\delta_{int}(s)$. We emphasize here that the output is possible only before the internal transitions.
- when there is an external event ***x*** before the expiration time, $t_a(s)$ (the system is in state (s,e), with $e \leq t_a(s)$), the system's state changes to state ***s'*** as a result of the transition $\delta_{ext}(s,e,x)$.

This formalism is used to define each module that composes a model.

3.2 Timed Automata

A timed automaton (TA) [1] is a formalism for modeling and verification of real time systems. It can be seen as classical finite state automata with clock variables and logical formulas on the clocks (temporal constraints) [1]. The constraints on the clock variables are used to restrict the behavior of the automaton. The logical clocks in the system are initialized to zero when the system is started and then increase at a uniform rate counting time with respect to a fixed global time frame. Each clock can be separately reset to zero. The clocks keep track of the time elapsed since the last reset [1].

4. Semantics and formal representations of simulation interfaces

This section presents the C/D synchronization model, and the simulation interfaces semantics and formal representation.

4.1 Canonical synchronization

For an accurate synchronization, each simulator involved in a C/D simulation must consider the events coming from the external world and it must accurately reach the time stamps of these events. This behavior is generally insured by the simulation interfaces. Thus, through its simulation interfaces the continuous simulator, must detect the next discrete event (timed event) scheduled by the discrete simulator when the latter has completed the processing corresponding to the current time. The discrete simulator must detect the *state events* generated by the continuous simulator. A state event is an unpredictable event, whose time stamp depends on the values of the state variables (ex: a zero-crossing event or a threshold overtaking event). The state event is not concerned with the continuous state evolution (we consider that the continuous state evolution semantics are provided by the continuous simulator). The synchronization model used in our approach respects the canonical algorithm. Fig. 1 presents the synchronization model without (Fig. 1a) and with state event (Fig. 1b).

At a given time, the discrete simulator is in the state (x_{dk}, t_{dk}) with x_{dk} the location and t_{dk} the k^{-th} discrete time. At this point, the discrete simulator has executed all the processes sensitive to the event and sends the time of the next event (t_{dk+1}) and the data to the continuous simulator. The context is then switched from the discrete to the continuous simulator (arrow 1 in Fig. 1a and Fig. 1b).

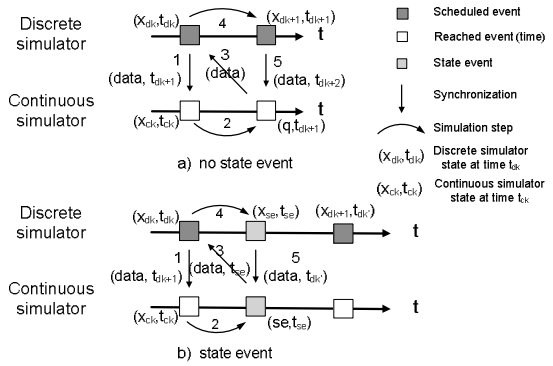

Fig. 1: The synchronization model in the C/D simulation interface without (a) and with (b) state event.

The state of the continuous simulator is (x_{ck}, t_{ck}) and the advance in time of the simulator cannot be further than t_{dk+1}, the time sent by the discrete simulator.

The behavior of the continuous simulator interface (CSI) can be described by the following transition:

$$(x_{ck}, t_{ck}) \rightarrow \begin{cases} (x_{ck+1}, t_{ck+1}) \text{ if } t_{ck+1} = t_{dk+1} & (1) \\ (se, t_{ck+1}) \text{ if } t_{ck+1} < t_{dk+1} & (2) \end{cases}$$

where:

- the state (x_{ck+1}, t_{ck+1}) is the state of the continuous simulator when no state event was generated in the time interval $[t_{ck}, , t_{ck+1}]$
- the state (se, t_{se}) represents the state of the continuous simulator when a state event *se* was generated and t_{se} represents the time when the state event occurred.

In both states, the continuous simulator will stop and send the data to the discrete simulator and then the context is switched to the time t_{dk} (arrow 3 in Fig. 1a and 1b).

In the case described by the equation (1), after the context switch, the discrete simulator will advance to the time t_{dk+1} that is the next synchronization point, where it will execute all the processes sensitive to this event. Before switching the context to the continuous simulator, the discrete simulator sends the data and the time of the next scheduled event t_{dk+2} (arrow 4 in Fig. 1a) and the cycle restarts.

Equation (2) describes the case where a state event occurred. The continuous simulator will send, through its simulation interface, not only the data but also the time when the state event occurred t_{se} (arrow 3 in Fig. 1b). The discrete simulator will advance to this time (state event detected by the discrete simulator) where it will execute all the processes sensitive to the event. Before switching the context to the continuous simulator the discrete simulator will send, through its simulation interface, the data and the recalculated time of the next scheduled event $t_{dk'}$ (arrow 4 in Fig. 1b). The time stamp can change after a state event. This time stamp can take any value bigger than t_{se} .

The canonical model requires synchronization at each simulation step of the discrete simulator and this implies some redundant synchronization steps. Despite this, the model is still efficient since it avoids any need of roll-back.

4.2 Operational semantics and formalization for the Discrete Simulator Interface (DSI)

This section presents the operational semantics of the Discrete Simulation Interfaces (DSI). The semantics was defined with respect to the synchronization model presented in section 4.1, using DEVS formalism. Table 1 presents a set of rules that show the transition between states. Its form is $\frac{Premises}{Conclusion}$.

For all the rules, the semantic of the global variable *flag* is related to the context switch between the continuous and discrete simulators. When the flag is set to '1', the discrete simulator is executed. When it is '0', the continuous simulator is executed. The global variable *synch* is used to impose the order of the different operations expressed by the rules. The first rule covers arrow 1 in Fig. 1a and 1b. The second and third rules correspond to arrows 3 (on the receiving part) and 4 in Fig. 1a respectively Fig. 1b.

In order to clarify, we detail here the first rule. The premises of this rule are: the *synch* variable has value '1', the *flag* variable has value '1', and we have an external transition function (δext) for the DSI.

This rule expresses the following actions of the discrete simulator interface:

- receiving data from the discrete model. This is an external transition (δ_{ext}) expressed by *?(DataFromDisc)*.
- sending data to the Continuous Simulator Interface (CSI) (*!DataToCSI*). The data sent to the CSI is the output function $\lambda(x_{dk}, t_{dk})$ and it is possible, according with DEVS formalism, only as a consequence of an internal transition (δ_{int}). In our case the output is represented by $!(data, t_{dk+1}(x_{dk}, t_{dk}))$. This transition corresponds to arrow 1 in Fig. 1a and 1b.
- switching the simulation context from the discrete to the continuous domain (action expressed by *flag:=0*).

All the other rules presented in this table follow the same format.

Rule
$(1)\ \dfrac{synch = 1 \wedge flag = 1 \wedge (x_{dk}, t_{dk}) = \delta_{ext}((x_{dk}, t_{dk}), 0, x)}{((x_{dk}, t_{dk}), \infty) \xrightarrow{?DataFromDisc} ((x_{dk}, t_{dk}), 0) \xrightarrow{!(data, t_{dk+1}((x_{dk}, t_{dk})));\ flag := 0} ((x_{dk}, t_{dk}), t_{dk+1})}$
$(2)\ \dfrac{synch = 0 \wedge flag = 1 \wedge \neg stateevent \wedge (x_{dk}, t_{dk}) = \delta_{ext}((x_{dk}, t_{dk}), 0, x)}{((x_{dk}, t_{dk}), e_{dk}) \xrightarrow{?Event} ((x_{dk}, t_{dk}), 0) \xrightarrow{?data; synch=1} ((x_{dk}, t_{dk}), 0) \xrightarrow{!DataToDisc} ((x_{dk+1}, t_{dk+1}), e_{dk+1})}$
$(3)\ \dfrac{synch = 0 \wedge flag = 1 \wedge stateevent \wedge (x_{dk}, t_{dk}) = \delta_{ext}((x_{dk}, t_{dk}), 0, x)}{((x_{dk}, t_{dk}), e_{dk}) \xrightarrow{?Event} ((x_{dk}, t_{dk}), 0) \xrightarrow{?(data, tse); synch=1} ((x_{dk}, t_{dk}), 0) \xrightarrow{!(DataToDisc, tse)} ((x_{se}, t_{se}), e_{se})}$

Table 1. Operational semantics for the Discrete Simulator Interface (DSI)

From these rules we can trace the state graph of the DSI as shown in Fig. 2.

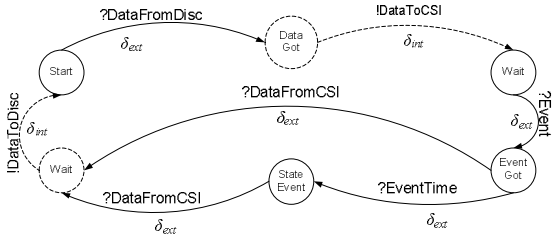

Fig. 2: State graph of the DSI represented using DEVS

The dashed lines represent internal transitions and the corresponding states and the plain lines represent external transitions and the corresponding states.

The equivalent timed automaton of the DSI is given in Fig. 3. The timed-automata model completes the DEVS graph with the addition of the timing evolution notions [5]. Thus, the transitions label may include operation for the time variable update. For instance, during the transition from the *WaitDataFromCont* state to the *Start* state, the value of the *NextTime* variable is assigned to the *td* variable expressing the time of the discrete simulator. The *NextTime* variable represents the next synchronization instance and its value is calculated respecting the canonical synchronization model.

Since several formal verification tools (ex UPPAAL) are based on timed-automata models, the transition from the DEVS graph to the timed-automata offers a model for verifiability.

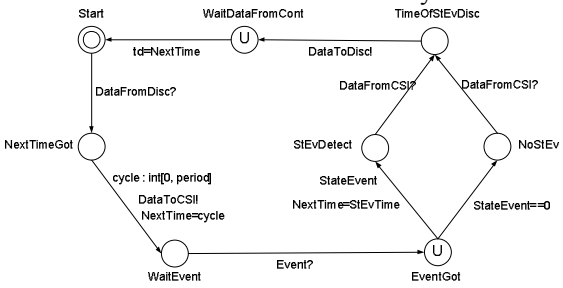

Fig. 3: The DSI represented as a TA

4.3 Operational semantics and formalization for the Continuous Simulator Interface (CSI)

The operational semantics for the CSI is given by the set of rules presented in Table 2. In these rules, the *Data* notation refers to the data exchanged between the DSI and the discrete simulator.

From these rules we can trace the state graph of the CSI as shown in Fig. 4.

Rule
$(4)\ \dfrac{synch = 1 \wedge flag = 1 \wedge q_k = \delta_{ext}((x_{ck}, t_{ck}), 0, x)}{(x_{ck}, \infty) \xrightarrow{?(data, t_{dk+1});\ synch := 0} (x_{ck}, t_{ck}) \xrightarrow{!(DataToCont, t_a(x_{dk}, t_{dk}))} (x_{ck}, t_{ck})}$
$(5)\ \dfrac{synch = 0 \wedge flag = 0 \wedge \neg stateevent \wedge q_{k+1} = \delta_{\text{int}}((x_{ck}, t_{ck}))}{(x_{ck}, t_{ck}) \xrightarrow{?(DataFromCont)} (x_{ck}, t_{ck}) \xrightarrow{!(data);\ flag := 1} (q, t_{d,k+1})}$
$(6)\ \dfrac{synch = 0 \wedge flag = 0 \wedge stateevent \wedge q_{k+1} = \delta_{\text{int}}(q_k)}{(x_{ck}, t_{ck}) \xrightarrow{?(DataFromCont)} (x_{ck}, t_{ck}) \xrightarrow{!(data, t_{se});\ flag := 1} (se, t_{se})}$

Table 2. Operational semantics for the Continuous Simulator Interface (CSI)

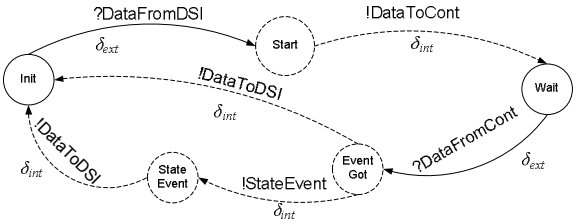

Fig. 4: State graph of the CSI represented using DEVS

The equivalent timed automaton of the CSI is given in Fig. 5.

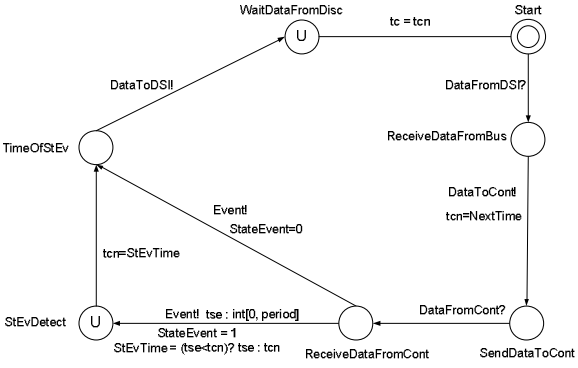

Fig. 5: The CSI represented as a TA

4.4 Formal verification of the simulation interfaces

The representation of the interfaces as timed automata allows for their formal verification. In our work we used UPPAAL [2] to check three types of properties:

1. *Safety properties* - the system does not get into an undesirable configuration. The safety properties verified for the simulation interfaces are:

- P1: Absence of deadlock.
- P2: No state event is detected by the discrete simulator interface if no state event was sent by the continuous simulator.

2. *Liveness properties* - some desired configuration will be visited eventually or infinitely. The liveness properties verified in our approach are:

- P3: The respect of the causality principle
- P4: A state event sent by the continuous domain leads to a state event detected in the DSI.

3. *Reachability properties* – the system always has the chance of reaching a given situation.

- P5: Invariantly both DSI and CDI in the *Start* location (initial state) and each of them executed one cycle `imply` the time in the continuous domain t_{ck} is equal with the time in the discrete domain t_{dk}**:**

```
A[](      (IDiscrete.Start      and
IContinu.Start   )      imply      (
IContinu.tc - IDiscrete.td == 0))
```

5. Implementation and results

The presented verified models were used as basis for the implementation of simulation interfaces for the SystemC [12] discrete simulator and Simulink [10] continuous simulator. The implemented DSI is a SystemC model respecting the functionality presented in section 4.2 while the implemented CSI is a Simulink model respecting the functionality presented in section 4.3.

The implemented interfaces were applied for the validation of a medical C/D system, a glycemia level regulator. This system allows the utilization of a new technique for diabetes therapy: the insulin or glucose infusion based on real time values of patient's glycemia level. This technique is a more beneficial alternative to the conventional therapy consisting in daily insulin injections.

As shown in Fig. 6, the glycemia system includes two sub-systems, a continuous sub-system modeling in Simulink the patient, the insulin and glucose pumps, and the insulin and glucose injection and a SystemC discrete sub-system for the injection control.

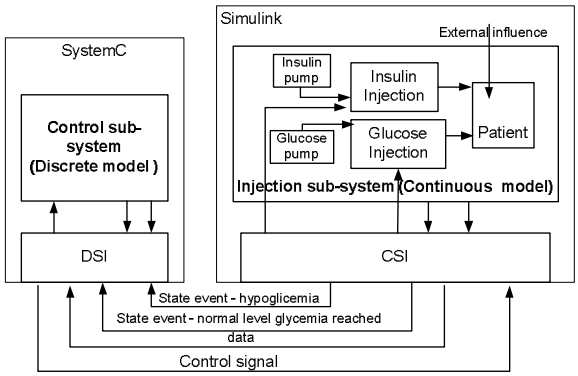

Fig. 6: The glycemia regulator system

The patient glycemia level (that is the level of glucose in the blood) is read and compared with the normal level in the "Injection sub-system" and the result is sent to the "Control sub-system". Depending on the value, the "Control sub-system" activates either the insulin or the glucose pump. If the level of the glycemia drops under 60mg/dl, this corresponds to the state of hypoglycemia, and the glucose pump will be activated immediately. In the case of this application, two types of state events are generated:

- the state events generated when a normal level of glycemia is reached (120 mg/dl)
- the state events generated when the glycemia drops under a reference value (60 mg/dl) - hypoglycemia.

Fig. 7a and 7b illustrate the evolution of the patient's insulinemia (units of insulin/dl) during 24 hours monitoring respectively the generation of a

state event. The state event is generated at the time 22.2481 when the patient' glycemia reaches the normal level (120mg/dl) (see Fig. 7b). We observe from Fig 7a that the insulin injection stops at the same time 22.2481,as a consequence of the state event detection. Fig. 8 shows the messages displayed by the SystemC simulator signalizing the state event detection and the insulin injection.

6. Conclusions

Given the diversity of concepts manipulated in the continuous/discrete systems, the global validation stage is very challenging. It requires a complex behavior for the simulation interfaces. One of the key issues for the design of these interfaces is the rigorous definition of their semantics, their modeling, and verification.

This paper presented the simulation interfaces semantics and their formal representation using DEVS formalism as well as timed automata that allowed the formal verification. The validation of an automatic glycemia level regulator illustrated the proposed approach.

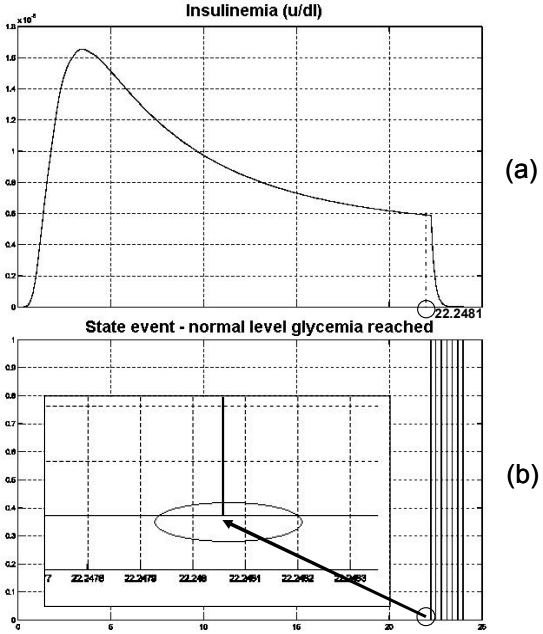

Fig. 7: Patient's insulinemia and state event generation by CSI

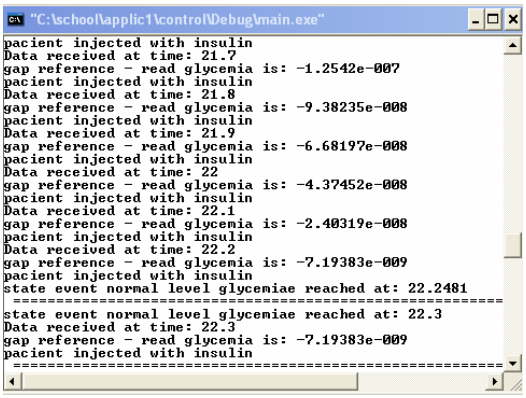

Fig. 8: State event detection by DSI

.

References

[1] Alur, R., Dill, D.,: "Automata for modeling real-time systems", in *Proc. 17-th Int. Colloquium on Automata, Languages and Programming*, vol. 443, 1990.

[2] Behrmann, G. et al.: "A Tutorial on UPPAAL", *Real-Time Systems Symposium, Miami*, 2005

[3] D'Abreu, M.; Wainer G.: "M/CD++ : modeling continuous systems using Modelica and DEVS", *IEEE Int. Symposium of MASCOTS'05*, 2005

[4] Frey, P. et al.: "Verilog-AMS: Mixed-signal simulation and cross domain connect modules", *Behavioral Modeling and Simulation Workshop*, 2000

[5] Giambiasi, N., Paillet J-L., Chane F., :"From timed automata to DEVS", *in Proc. of the 2003 Winter Simulation Conference*., 2003

[6] IEEE Standard *VHDL Analog and Mixed-Signal Extensions* (1999), IEEE Std 1076.1-1999

[7] International Technology Roadmap for Semiconductor Design, at http://public.itrs.net/.

[8] Jantsch, J. "Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation". Morgan Kaufmann Publishers, June 2003.

[9] Lee, E.A. and Sangiovanni-Vincentelli A.L.: "Comparing Models of Computation" in: *Int. Conf. on Computer-Aided Design (ICCAD)*, 1996.

[10] Matlab-Simulink at www.mathworks.com

[11] Patel, D. H, and Shukla, S. K.: "SystemC kernel – Extensions for heterogeneous System modeling" Kluwer Academic Publishers, 2004

[12] SystemC LRM, available at www.systemc.org

[13] Vachoux, A. et al.; "Analog and mixed signal modeling with SystemC", *ISCAS'03.*

[14] Ptolemy project at http://ptolemy.eecs.berkeley.edu/

[15] Zeigler, B.P.; Praehofer H. and Kim, T.G.: "Modeling and Simulation - Integrating Discrete Event and continuous complex dynamic systems". Academic Press, San Diego, 2000