A Formalization of Global Simulation Models for Continuous/Discrete Systems

L. Gheorghe, F. Bouchhima, G. Nicolescu, H. Boucheneb
Ecole Polytechnique Montréal
luiza.gheorghe@polymtl.ca

Keywords: Co-Simulation, Heterogeneous Systems, Continuous/Discrete Simulation, Formal Representation

Abstract

Many of the modern systems integrate components specific to different application domains. Frequently these systems combine continuous and discrete sub-systems and therefore their design involves solving specific global modeling and simulation problems. This paper addresses the formal representation of a continuous/discrete global synchronization model and the corresponding simulation interfaces. This representation enables the definition of generic language independent co-simulation tools that can be used to provide global simulation models for continuous/discrete heterogeneous systems. The model was validated through simulation, using UPPAAL toolbox and its verification was realized by defining and checking the main properties.

1. INTRODUCTION

Today, systems-on-chip are growing in complexity as a result of not only a higher density of hardware components on the same chip but also because of the integration of different modules that are particular to different application domains [12]. Many domains benefit from these system's advantages, among them the defense, medical, electronic, communication, and automotive. Given the diversity of concepts manipulated, the global design specification and validation are extremely challenging. A large number of these systems combine continuous and discrete sub-systems. The heterogeneity of these systems makes more difficult the elaboration of an executable model for the system's simulation (the simulation model). Generally these models are very complex; they include the execution of different components, the interconnects' interpretation as well as the adaptation between the continuous and the discrete components.

One of the methods generally used for the validation of a continuous/discrete (C/D) system is the co-simulation. The co-simulation allows joint simulation of heterogeneous components with different execution models. This technique presents many benefits. One of them is the reusability of the models already developed in a well known language and using already existing powerful tools (i.e. Simulink for the continuous domain and VHDL, Verilog or SystemC for the discrete domain). Thus, the development time, the time-to-market and the costs are reduced. This methodology requires the elaboration of a global co-simulation model (Figure 1). In this model, the co-simulation bus is in charge of transferring data between the different models [15].

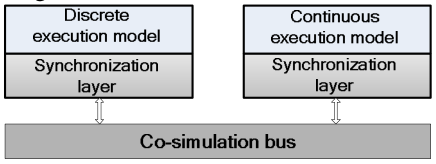


Figure 1: Generic continuous/discrete co-simulation model

For C/D systems co-simulation, the simulation interfaces have to provide efficient synchronization models for the adaptation of the domain specific models. In a C/D heterogeneous system, we find two distinct models:
- a continuous model where the computation is realized in the continuous domain by solving differential or algebraic equations;
- a discrete model where the computation is realized in cycles and every cycle represents the computation of a selected sub set of variables.

Thus, in the case of a global validation tool several execution semantics have to be taken into consideration in order to perform global simulation. A global model has to define clearly the computation and the communication (and implicitly the synchronization) and verify the behavior of the co-simulation interfaces given certain restrictions.

The automatic generation of the co-simulation interfaces is very suitable, since their design is time consuming and an significant source of errors. The simulation interfaces play an important role in accuracy and performance of global simulation. The strategy currently used for the automatic generation of the co-simulation interfaces is based on the configuration of the components and their assembly. These components are selected from a co-simulation library.

An efficient tool for the automatic generation of the co-simulation interfaces must rely on a formal definition of the simulation interfaces. Some characteristics of the formalism for interface generation are [19] the *executability* for automatic generation and *formal properties* for analyses and verifications. The formalism allows the verification of correctness, liveness and safety properties such as the deadlock, the correct answer to a change in the behavior of one of the simulators as well as the synchronization given by the continuous interface and the discrete interface.

The main contribution of this paper is the convergence of formal and co-simulation based approaches in the context of global validation of C/D systems. We focus on the co-simulation interfaces required for C/D global simulation. A formal representation of the behavior of these interfaces using timed automata is proposed. This representation was realized with respect to the generic canonical synchronization model [10]. This allows for the definition and the verification of some of the system's properties and constraints. In order to validate and check our model we used UPPAAL [5].

This formal representation (also called here model) constitutes the foundation for the definition of a generic co-simulation tool that can provides global simulation models for C/D systems validation using the automatic generation of the co-simulation interfaces. This work generalizes our previous works presented in [3] and [15]. In these works, the co-simulation interfaces were not verified formally.

The article is structured as follows. Section 2 will present several of the related works and previous approaches to the simulation of the C/D systems. Section 3 introduces some of the basic concepts such as the synchronization model, timed automata and UPPAAL, a tool that is used to verify systems that can be modeled as timed automata. Section 4 presents the canonical synchronization model in the context of the C/D co-simulation, the co-simulation interfaces, their behavior and formal models. Section 5 shows the experimental results; more precisely the model simulation and validation as well as the properties verification are detailed. Finally, section 6 gives our conclusions.

2. RELATED WORK

Some of the existing works on C/D systems validation propose the utilization of a single language for the specification of the C/D system. These tools may be obtained by extension of existing HDLs [9], [11], [16], [17]. 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 [18]. These works do not consider the accurate synchronization for the continuous/discrete integration. Moreover, the different systems and components need to be developed in the same environment in order to be compatible thus they do not solve the problem of real components-based approach to system design.

Several formal representation approaches propose definitions for heterogeneous systems modeling. In [14], a formal framework for comparing computation used in heterogeneous models is presented. The authors propose a formal classification framework that makes it possible to compare and express differences between models of computation. The intent is "to be able to compare and contrast its notions of concurrency, communication, and time with those of other models of computation" [14]. The role of the computation in abstracting functionalities of complex heterogeneous systems was presented in [13]. The author proposes the formalization of the heterogeneous systems by separating the communication and the computation aspects; however verification on the interfaces between domains were not taken into consideration.

In [20] the author introduced an interesting formalism defined for the modeling and simulation of discrete event systems (Discrete EVent System Specifications - DEVS) where the time advances on a continuous time base. DEVS is a formal approach to build models, using hierarchy and modularity and more recently it integrates object-oriented programming techniques. Based on this formalism, [7] proposes a tool for the modeling and simulation of hybrid systems using Modelica and DEVS. The models are "created using Modelica standard notation and a translator converts them into DEVS models" [7].

Compared to these works, we concentrate on the formal models of the co-simulation interfaces that enable the interfaces verification.

3. BASIC CONCEPTS

This section introduces some of the basic concepts that are used in this work. The first part of the section presents the canonical synchronization model. The next two parts will introduce the timed automata formalism and a brief presentation of the UPPAAL modeling and validation tool.

3.1. Canonical Synchronization Model in Discrete/Continuous Co-simulation

The synchronization, defined as coordination with respect to time, represents an important aspect in co-simulation models. The synchronization between the continuous domain and the discrete event domain is realized using a canonical algorithm as it was presented in [10]. For a rigorous synchronization the discrete kernel has to detect the events generated by the analog (continuous) solver and the continuous solver must detect the scheduled events from the discrete kernel.

For the discrete event processes, the time does not advance during the execution. The next execution time is the next time in the event queue. The execution of the analog solver advances the simulation time. Let be t_k the synchronization time for the discrete kernel and the analog solver. The analog solver advances to the next synchronization time t_{k+1}, known in advance from the digital kernel. At this point the analog solver suspends while the digital kernel resumes and the events in t_{k+1} are executed. If a state event occurs in the time interval $[t_k, t_{k+1}]$, the analog solver suspends to allow the digital kernel to take this event into consideration. This way the analog solver and the digital kernel are synchronized again.

3.2. Timed Automata

Proposed by Alur and Dill in 1990 [1], timed automata is a formalism for modeling and verification of real time systems.

A timed automaton can be seen as a classical finite state automata with clock variables and logical formulas on the clock (temporal constraints). 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 [1]. The clock constraints are the guards on the transitions. A transition can be taken when the clocks' values satisfy the guard labeled on it. Figure 2 illustrates an example of a timed automaton.

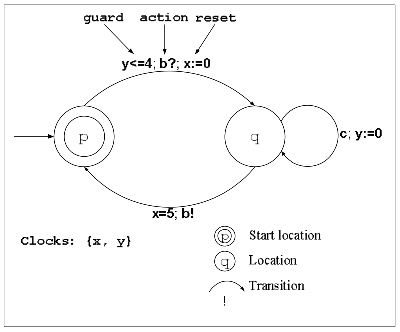


Figure 2: Example of a timed automaton

The process shown in Figure 2 starts at the location p with all its clocks (x and y) initialized to 0. The values of the clocks increase synchronously with time at the location q. At any time, the process can change the location following a transition $p \xrightarrow{g;a;r} q$ if the current values of the clocks satisfy the enabling condition g (guard). A guard is a Boolean combination of integer bounds on clocks and clock-differences. With this transition, the variables are updated by r (reset) which is an action performed on the clocks. The actions are used for synchronization and are expressed by a (action) [2]. A synchronization label is of the form *Expression?* or *Expression!* where ! represents the operator send and ? represents the operator receive.

The semantics for a time automaton is defined as "a transition system where a state or configuration consists of the current location and the current values of clocks" [2]. Thus, the state is represented by the tuple: (*location*, x) where x is the clock. Given the system, we can have two types of transitions between locations: a delay transition when the automaton may delay for some time or an action transition when the transition follows an enabled transition. Timed automata can be extended with parallel composition that is the product of the automata.

3.3. A Verification Tool for Timed Systems – UPPAAL

UPPAAL [5] is an integrated tool environment for modeling, simulation and verification of timed automata developed jointly by Aalborg University in Denmark and the Uppsala University in Sweden. It consists of three parts: a model descriptor, a simulator and a model-checker. The descriptor models systems that can be represented as a collection of non-deterministic processes with finite control structure and real-valued clocks (i.e. timed automata), communicating through channels and (or) shared data structures. A model consists of one or more concurrent processes (also named simulators), local and global variables, and channels. There are three types of locations in UPPAAL: *normal locations* with or without invariants, *urgent locations* and *committed locations.* No delay is allowed in urgent or committed locations. The transitions out from an urgent location have higher priority than that of time progress.

The expressions cover clocks and integer variables and are used with the labels: guards, synchronization, assignments or invariant. The models synchronize with each other via channels. In UPPAAL the assignments are evaluated sequentially (not concurrently). On synchronizing transitions, the assignments on the !-side (the emitting side) are evaluated before the ?-side (the receiving side).

The model checker engine in UPPAAL is based on the theory of timed automata and its query language is a subset of computational tree logic, the timed computational tree logic (TCTL). The query language [5] consists in path formulae and state formulae. The states formulae describe individual states while the path quantifies over traces of the model.

The main advantage of UPPAAL is that the product automaton is computed on-the-fly during verification. This reduces the computation time and the required memory space. It also allows interleaving of actions as well as hand-shake synchronization.

In our approach UPPAAL was used for the formal representation and verification of the simulation interfaces. Our formal representation needs to support concurrency between the two models, the continuous and the discrete therefore they were represented as a parallel composition of several timed automata.

4. METHODOLOGY

To enable the design of flexible, modular, scalable and accurate co-simulation tools, a methodology independent of the simulation tools used for the continuous and discrete components of the system should consider the following main steps:

1) Definition of the operational semantics for the synchronization in C/D global execution models
2) Distribution of the synchronization functionality to the simulation interfaces
3) Formalization and verification of the simulation interfaces behavior
4) Definition of the library elements and the internal architecture of the simulation interfaces
5) The analysis of the simulation tools for the integration in the co-simulation framework.
6) The implementation of the library elements specific to different simulation tools.

The work presented in this article covers steps 2 and 3 defined above: the distribution of the synchronization functionality to the simulation interfaces and the formalization and the verification of the simulation interfaces.

5. FORMAL DEFINITION OF THE CONTINUOUS /DISCRETE SIMULATION INTERFACES

This section details the global simulation model for a C/D system. In the global simulation model already presented in section 1 (see Figure 1) the co-simulation interfaces are in charge with the synchronization and the communication between the simulators.

5.1. Application of the Canonical Synchronization Model in Co-simulation

The continuous time system is described by the state space equations:

$$\dot{x}_c(t)=A_c x_c(t)+B_c u(t)$$
$$y(t)=C_c x_c(t)+D_c u(t) \qquad (1)$$

where x_c is the state vector, u the input signal vector, y the output signal vector and A_c, B_c, C_c and D_c are constant matrixes that describe the dynamic of the system.

The discrete system can be described by the state-space equations [6]:

$$x_d(t_{k+1})=f(x_d(t_k), u(t_k), t_k) \qquad x(t_0)=x_0$$
$$y(t_k)= g(x_d(t_k), u(t_k), t_k) \qquad (2)$$

where x_d is the state vector, u the input signal vector, y the output signal vector.

For the linear discrete systems, the equations 2 become:

$$x_d(t_{k+1})= A_d x_d(t_k)+B_d u(t_k)$$
$$y(t_k)=C_d x_d(t_k)+D_d u(t_k) \qquad (3)$$

where A_d, B_d, C_d and D_d are matrixes that can be time-varying and describe the dynamic of the system.

For the model presented here, the time interval for the continuous system (equation 1) is included in $[t_k,t_{k+1}]$. The input signal vector for the continuous domain is the output signal vector from the discrete domain and vice versa.

The events exchanged between the discrete and the continuous simulators are:
- *occurred /scheduled events* that are timed events scheduled by the discrete simulator.
- *state events* that are unpredictable events generated by the continuous simulator. Their time stamp depends on the values of state variables (e.g. a zero-passing or a threshold crossing).

Figure 3 presents the synchronization model in the C/D co-simulation interface with and without state event.

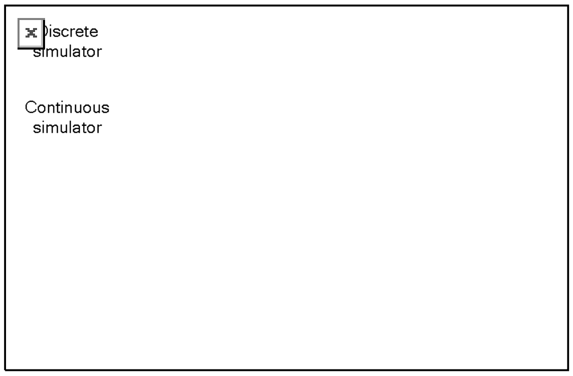


Figure 3: The synchronization model in the continuous/discrete simulation interface without (a) or with state event (b)

In the case of the co-simulation, the global model that integrates two different simulators has to respect the synchronization model previously presented. The simulators have to be controlled by the co-simulation interfaces in order to provide the functionalities described below. At a given time, the discrete simulator is in the state (x_{dk},t_{dk}). At this point, the discrete simulator had executed all the processes sensitive to the event with the time stamp x_{dk} and sends the time of the next event t_{dk+1} and the data to the continuous simulator and switches the context from the discrete to the continuous simulator before advancing the time. The state of the continuous simulator is (x_{ck},t_{ck}) and the advance in time of the simulator cannot be further then t_{dk+1}, the time sent by the discrete simulator. Consequently the behavior of the continuous interface can be described by the following transition state:

$$(x_{ck},t_{ck}) \longrightarrow \begin{cases} (x_{ck+1},\ t_{ck+1})\ \ if\ t_{ck+1}=t_{dk+1} & (4) \\ (se,t_{se})\ if\ t_{ck+1}< t_{dk+1} & (5) \end{cases}$$

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 $[t_{ck}\ ,t_{ck+1}]$ while 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 situations the continuous simulator will stop and send the data to the discrete simulator and then switch the context to the time t_{dk}. This work takes into consideration the event generated within the time interval $[t_k,t_{k+1}]$ after the context switch from the discrete domain to the continuous domain at the time t_k. This event can be a state event or the detection of an event scheduled by the discrete simulator (in both cases a synchronization point)

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

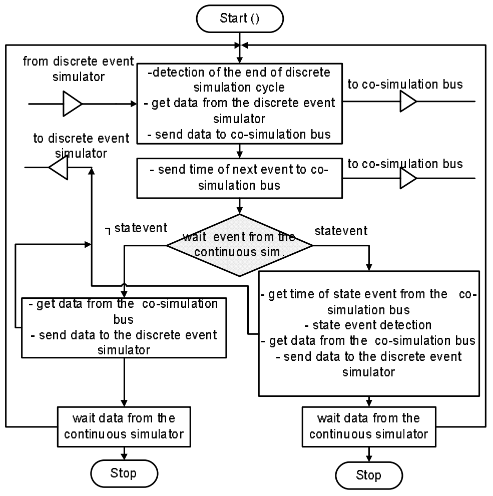
In the case of the equation (5) the continuous simulator will send not only the data but also the time when the state event occurred t_{se}. The discrete simulator will advance to this time (state event detected by the discrete simulator) where it will execute all the processes sensitive to the event. Before switching the context to the continuous simulator the discrete interface will send the data and the recalculated time of the next scheduled event $t_{dk'}$. Usually the developers assume that the sampling intervals are constant. In this work the time stamp can change after a state event. This time stamp is bigger than t_{se} and it can be smaller, equal or bigger then the time stamp that previously existed in the discrete simulator's queue – t_{dk+1}.

The advantage of this model is that it avoids any need of roll-back even if a state event was generated. This model is used if the signal update events are unpredictable.

A particular version of this model is when the events are periodic [4]. In this case the time stamps of the update events and sampling events are put in a queue. The time stamp of the next output event is known by accessing the queue to take out the smallest time stamp and send it to the continuous model.

5.2. Discrete domain co-simulation interface

Figure 4 presents the flowchart of the behavior of the discrete domain interface.



Start ()
from discrete event simulator
-detection of the end of discrete simulation cycle
- get data from the discrete event simulator
- send data to co-simulation bus
to co-simulation bus
to discrete event simulator
- send time of next event to co-simulation bus
to co-simulation bus
¬ statevent
wait event from the continuous sim.
statevent
- get data from the co-simulation bus
- send data to the discrete event simulator
- get time of state event from the co-simulation bus
- state event detection
- get data from the co-simulation bus
- send data to the discrete event simulator
wait data from the continuous simulator
wait data from the continuous simulator
Stop
Stop

Figure 4: Flowchart for discrete domain co-simulation interface

The behavior of the discrete domain interface can be described by its processing steps detailed subsequently.

During the first step, the interface:

- detects the end of the simulation cycle in the discrete;
- receives data and the time of the next event in the discrete domain;
- sends the time and the data to the continuous simulator via the co-simulation bus and
- switches the context from the discrete to the continuous domain.

After the first step is finalized, the discrete interface waits for the signal from the continuous simulator, the behavior of the interface depending on the data received from the continuous interface and if a state event is generated. Two cases are possible (see Figure 4).

Based on this flowchart we formalized the discrete co-simulation interface. Figure 5 shows the formal model (using timed automata) for the discrete domain interface. The model has only one initial location (a double circle in Figure 5) *Start*.

The discrete interface will change location from *Start* to *NextTimeGot* following the transition $\boldsymbol{Start} \xrightarrow{DataFromDisc?} \boldsymbol{NextTimeGot}$. This is an external transition realized with zero time and it is triggered by the receiving of the data (that is also synchronization between the discrete simulator and the interface) from the discrete simulator (`DataFromDisc?`). Here the interface receives the data from discrete simulator and the time of the next event in the discrete domain.

The location changes to *WaitEvent* following the transition: $\boldsymbol{NextTimeGot} \xrightarrow{DataFromBus!,\ NextTime=cycle,\ cycle:int[0,\ period]} \boldsymbol{WaitEvent}$. In order to change the location, the continuous interface sends to the discrete interface the time of the next event (occurred/scheduled event) in discrete (the synchronization `DataFromBus!`). The variable `NextTime` is the time of the next event in the discrete domain. This variable takes, in this mode, the value `cycle`. The theory normally assumes equidistant sampling intervals. This assumption is not usually achieved in practice. For an accurate simulation we assume that cycle takes random values in an interval defined here as *[0, period]*.

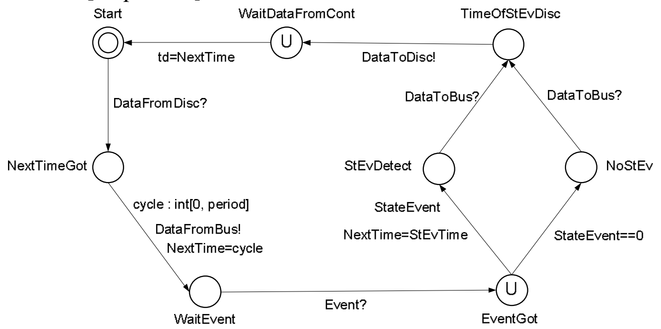


Figure 5: The discrete domain interface model (IDiscrete)

In *WaitEvent* location, the context is switched from the discrete to the continuous simulator.

When the context is switched back to the discrete simulator, the location is changed to *EventGot* following the synchronization transition: $WaitEvent \xrightarrow{Event?} EventGot$. During this transition the discrete interface receives from the continuous interface the synchronization `Event?`. In this location the occurrence of a state event in the continuous domain is considered. *EventGot* is an urgent location (as defined in section 3.4). This will not allow the discrete model to miss a state event generated by the continuous model. Two cases are possible:

1) When no state event was generated by the continuous domain, the location changes from *EventGot* to *NoStEv*. The transition $\boldsymbol{EventGot} \xrightarrow{StateEvent==0} \boldsymbol{NoStEv}$ is annotated in this case only with the guard `StateEvent==0`.

2) When a state event was generated by the continuous domain the location changes from *EventGot* to *StEvDetect* following the transition:

$$\boldsymbol{EventGot} \xrightarrow{StateEvent,NextTime=StEvTime} \boldsymbol{StEvDetect}.$$

This transition is annotated with a guard (`StateEvent`) and the update of the `NextTime` in the discrete domain as the time when the state event occurred in the continuous domain `StEvTime` (for a rigorous synchronization, the discrete domain has to consume this event and stop at the time when it was generated by the continuous domain interface) This is the time of the next event that is going to be sent to the continuous simulator.

From both locations *StEvDetect* and *NoStEv*, the system can reach the next location: *TimeOfStEvDisc*. In both cases the model performs synchronization (`DataToBus?`). At this point the discrete interface will synchronize and send data to the discrete simulator (`DataToDisc`) and changes the location to *WaitDataFromCont*. The next location is *Start*, the discrete time variable is initialized on this channel (`td=NextTime`) and the cycle restarts.

5.3. Continuous domain co-simulation interface

Figure 6 presents the flowchart of the behavior of the continuous domain interface. The following paragraph gives the description of the processing steps in this domain.

During the first processing steps the continuous simulator:
- receives the data from the co-simulation bus;
- sends the data to the continuous simulator;
- sends the time of the next event in the discrete domain to the continuous simulator.

The second step consists of receiving the data from the discrete simulator. The behavior of the interface depends on the data received from the continuous simulator and if a state event was generated. Regarding the occurrence of a state event, two cases are possible (as shown in Figure 6). Figure 7 shows the continuous domain interface formalization that was developed based on the flowchart representing the behavior of this interface.

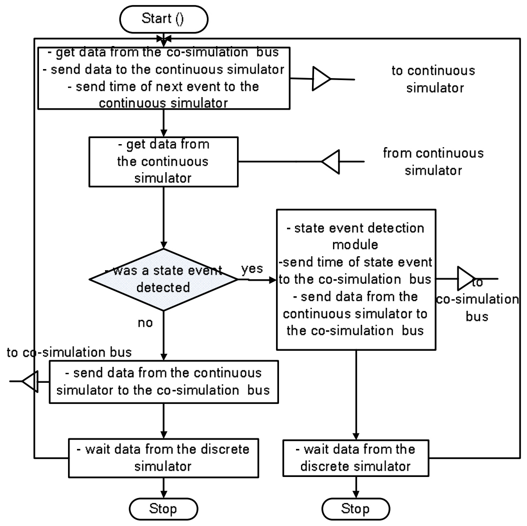

Start ()
- get data from the co-simulation bus
- send data to the continuous simulator
- send time of next event to the continuous simulator
to continuous simulator
- get data from the continuous simulator
from continuous simulator
- was a state event detected
yes
no
- state event detection module
-send time of state event to the co-simulation bus
- send data from the continuous simulator to the co-simulation bus
to co-simulation bus
to co-simulation bus
- send data from the continuous simulator to the co-simulation bus
- wait data from the discrete simulator
- wait data from the discrete simulator
Stop
Stop

Figure 6: Flowchart for the continuous domain co-simulation interface

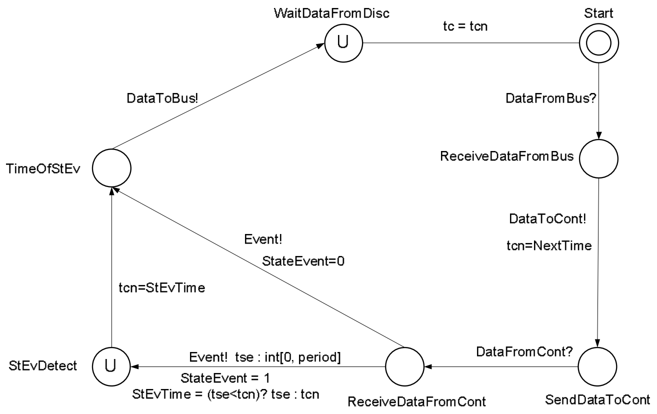


Figure 7: The continuous domain interface model (IContinu)

The continuous interface will leave the initial location *Start* following the transition:

$\boldsymbol{Start} \xrightarrow{DataFromBus?} \boldsymbol{ReceiveDataFromBus}$. This is also an external transition realized with zero time and it is triggered by the receiving of the data from the discrete interface (`DataFromBus?`) that is also the first synchronization point between the discrete interface and the continuous interface. The interface receives the data from discrete and the time of the next event in discrete. From *ReceiveDataFromBus* location the process will move to the next location *SendDataToCont* following the transition

$$ReceiveDataFromBus \xrightarrow{DataToCont!,tcn=NextTime} \boldsymbol{SendDataTo\ Cont}$$

The value `NextTime`, the time of the next event (occurred/scheduled event) in the discrete simulator is assigned to `tcn`, the next time in the continuous simulator. In our model, the synchronization on this transition is between IContinu and SimCont (where SimCont is the continuous domain simulator), the interface sends to the simulator data received from IDiscrete and the time of the next event in the discrete domain. The system changes the location from *SendDataToCont* to *ReceiveDataFromCont* following the synchronization transition:

$$\textbf{\textit{SendDataToCont}} \xrightarrow{DataFromCont?} \textbf{\textit{ReceiveDataFromCont}}$$

During this transition the continuous interface receives data from the continuous simulator and eventually, if a state event occurs, the time of the state event. In the *ReceiveDataFromCont* location, the continuous interface evaluates if a state event was generated. Two cases are possible:

1) When no state event is generated, the location changes from *ReceiveDataFromCont* to *TimeOfStEv* following the transition

$\textbf{\textit{ReceiveDataFromCont}} \xrightarrow{Event!StateEvent=0} \textbf{\textit{TimeOfStEv}}$. The transition is annotated in this case by the synchronization `Event!` and with the update `StateEvent=0`.

2) When a state event is generated, the location changes from *ReceiveDataFromCont* to *StEvDetect* following the transition : *ReceiveDataFromCont*

$$\xrightarrow{Event!\ StateEvent=1, StEvTime=(tse<tcn)?tse:tcn,\ tse:int[0,\ period]} \textbf{\textit{StEvDetect}}$$

This transition is annotated with a synchronization (`Event!`) and three variable updates: `StateEvent=1` (for the detection of a state event), `StEvTime=(tse<tcn)?tse:tcn,tse:int[0,period]` (for the time of the state event that occurs during the time interval *[0,period]*; this time will be sent to the discrete simulator). *StEvDetect* is an urgent location. The location *StEvDetect* changes to *TimeOfStEv* following the transition:

$$\textbf{\textit{StEvDetect}} \xrightarrow{tcn=StEvTime} \textbf{\textit{TimeOfStEv}}$$

At this point there is no synchronization, just an update of the time in the continuous domain having assigned the time of the state event `StEvTime`: `tcn=StEvTime`.

TimeOfStEv location is common for both cases, `StateEvent=0` or `StateEvent=1`. This location changes to *WaitDataFromDisc*. The system performs synchronization (`DataToBus!`) between the continuous interface and the continuous simulator. The next location is *Start*, the continuous time variable is initialized on this channel (`tc=tcn`) and the cycle restarts.

The model descriptor from UPPAAL allows the description of the interfaces behavior as a network of automata. The global formal model proposed in this paper is formed by four sub-models (processes): the continuous domain simulator (SimCont), the continuous domain interface (IContinu), the discrete domain simulator (IDiscrete) and the discrete domain interface (SimDisc).

Figure 8 shows the global formal model including the continuous domain and the discrete domain simulators and their interaction. The transitions show the synchronizations between the simulators and interfaces as well as the synchronization between the interfaces. The initial location for the global formal model is the discrete simulator; however, the continuous simulator is the first that advances in time.

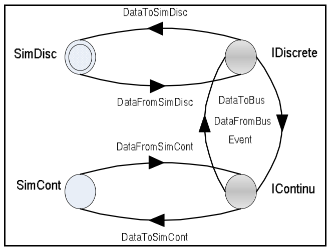


Figure 8: The global formal simulation model

6. MODEL VALIDATION

UPPAAL allows the validation of the global formal model by simulation. We simulated all the possible dynamic executions of our synchronization model and we formally verified our model.

The formal verification consists of checking properties of the system for a broad class of inputs [8]. In our work we checked properties that fall into two classes:

- Safety properties - the system does not get into an undesirable configuration, (i.e. deadlock etc) [8].
- Liveness properties - some desired configuration will be visited eventually or infinitely (i.e. expected response to an input, etc.) [8].

The properties verified in order to validate the synchronization model are described below.

***P0* Deadlock (safety property)**

Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set. In UPPAAL deadlock is expressed by a formula using the keyword `deadlock.` A state is a deadlock state if there are no outgoing action transitions either from the state itself or any of its delay successors [4].

The absence of the deadlock property was verified (and the property was satisfied).

The property is expressed by the following query:

`A[] not deadlock`

***P1* State event detected by the discrete domain (liveness property)**

The indication of a state event by the continuous interface and its detection by the discrete interface is very important for C/D heterogeneous systems. We defined a liveness property in order to check this behavior:

Definition: A state event detected in the continuous domain `leads` to a state event detected in the discrete.

The property is expressed by the following query:

`IContinu.StEvDetect→IDiscrete.StEvDetect`

(where *IContinu* and *IDiscrete* are the processes and *StEvDetect* the locations).

***P2* No state event in discrete if no state event in continuous domain (safety property)**

In order to avoid false responses from the discrete simulators, we defined a safety property to verify if the system will "detect" a state event in the discrete when it was not generated (and indicated) by the continuous domain:

Definition: `Invariantly` a state event detected in the discrete domain `imply` state event in the continuous.

The property is expressed by the following query:

```
A[](IDiscrete.StEvDetect imply StateEvent)
```

P3 **Synchronization between the interfaces (liveness property)**

One of the most important properties characterizing the interaction between the continuous and the discrete domains is the communication and implicitly the synchronization. This property verifies that after a cycle executed by each model, both of them are at the same time stamp (and by consequence they synchronize)

Definition: `Invariantly` both processes in the *Start* location (initial state) and each of them executed one cycle `imply` the time in the continuous *tc* is equal with the time in the discrete *td*.

The property is expressed by the following query:

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

P4 **Causality principle <u>(liveness property)</u>**

The causality can be defined as a cause and effect relationship. The causality of two events describes to what extent one event is caused by the other. This property verifies that when a state event was generated by the continuous domain, the discrete domain will detect this event at the same precise time (the cause precedes or equals the effect) and not some other possible event existing at a different time in the continuous domain.

Definition: `Invariantly` both processes in the *StEvDetect* location (detection of state event) `imply` the time in the continuous *tc* is equal with the time in the discrete *td*.

The property is expressed in UPPAAL by the following query:

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

7. CONCLUSION

An efficient tool for the automatic generation of the co-simulation interfaces must relay on the formal definition of the simulation interfaces. This paper proposes solutions for two of the steps of a generic methodology for C/D design: the distribution of the synchronization functionality to the simulation interfaces and formal representation and verification of the behavior of the C/D co-simulation interfaces. The formalization was realized with respect to a generic canonical synchronization model using timed automata. The model was validated through simulation. In order to verify the formal representation, some properties were defined and checked using the model checker from UPPAAL.

References

[1] Alur, R., Dill, D., 1990: "Automata for modeling real-time systems". In Proceedings, 17-th International Colloquium on Automata, Languages and Programming, vol. 443, 322-335.

[2] Bengtsson, J., Yi, W, 1996: "Timed automata: Semantics, Algorithms and Tools", Department of Computer Science Uppsala University. Denmark.

[3] Bouchhima, F. et al. 2005: "Discrete–Continuous Simulation Model for Accurate Validation in Component-Based Heterogeneous SoC Design", Rapid Systems prototyping, pp 181-187.

[4] Bouchhima, F. et al. 2006: "A SystemC/Simulink Cosimulation Framework for Continuous/Discrete-Events Simulation", Behavioral Modeling and Simulation Conference, San Jose, California.

[5] Behrmann, G., David, A. and Larsen, K. 2005: "A Tutorial on UPPAAL", RTS Symposium, Miami.

[6] Cassandras, C. G. 1993: "Discrete event systems: Modeling and performance analysis", Richard Irwin, New York.

[7] D'Abreu, M.; Wainer G. 2005: "M/CD++: modeling continuous systems using Modelica and DEVS", in Proceedings of the IEEE International Symposium of MASCOTS'05.

[8] Edwards, S., Lavagno, L., Lee, E., and Sangiovanni-Vincentelli, A.L. 1997: "Design of Embedded Systems: Formal Models, Validation, and Synthesis" In Proceedings of the IEEE, vol. 85, pp. 366-390.

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

[10] Ghasemi, H.R. 2005: "An effective VHDL-AMS simulation algorithm with event", International Conference on VLSI Design.

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

[12] International Technology Roadmap for Semiconductor Design, available on line at http://www.itrs.net/links/2006

[13] Jantsch, J. 2003: Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation. Systems on Silicon. Morgan Kaufmann.

[14] Lee, E.A. and Sangiovanni-Vincentelli A.L. 1996: "Comparing Models of Computation", In: Proceedings of the International Conference on Computer-Aided Design (ICCAD), IEEE Computer Society. pp.234-241.

[15] Nicolescu, G. et al. 2002: "Validation in a component-based design flow for Multicore SoCs", in Proc. of ISSS.

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

[17] Vachoux, A. et al. 2003: "Analog and mixed signal modeling with SystemC", In: Circuits and Systems, (ISCAS'03).

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

[19] Romitti, S., Santoni, C., François. P. 1997: "A design methodology and a prototyping tool dedicated to adaptive interface generation", In Proceedings of the 3rd ERCIM Workshop, Obernai, France.

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