Des Autom Embed Syst (2015) 19:243–275
DOI 10.1007/s10617-014-9156-3

A generic conceptual framework based on formal representation for the design of continuous/discrete co-simulation tools

Luiza Gheorghe Iugan · Hanifa Boucheneb · Gabriela Nicolescu

Received: 23 May 2013 / Accepted: 17 November 2014 / Published online: 28 January 2015
© Springer Science+Business Media New York 2015

Abstract Modern systems integrate components specific to different application domains. Frequently, these systems combine continuous and discrete sub-systems and therefore their design involves overcoming specific global modeling and validation challenges. In order to generate global simulation models of heterogeneous systems the designers need efficient tools for systems' validation. Therefore, a new type of designers emerged, the designers of co-simulation tools. Their main objective is to provide coherent tools for the co-simulation models' designers. Given the diversity of abstractions, languages and simulation tools, the design of co-simulation tools may be costly and time consuming. Thus, the key for the improvement of the validation process is to define a model-based generic approach before the implementation of these tools. This requires new skills on formalism and formal verification domain. This paper proposes a generic conceptual framework based on formal representation of the co-simulation interfaces for co-simulation tools design. The framework can be used to provide rigorous global formal co-simulation models for continuous/discrete heterogeneous systems. It allows the definition for implementation of the co-simulation interfaces starting with their formal definitions that are gradually refined and verified. The global formal model also provides the rules for the implementation and the generation of the interfaces. The framework is the skeleton on which the designers can build accurate tools for global execution models of continuous/discrete heterogeneous systems. The approach was used to design a co-simulation tool that is presented in this paper.

Keywords Continuous time systems · Discrete event systems · Formal verification · Modeling · Co-simulation

1 Introduction

System on chip (SoC) trends of the past decade observed the shrinking of the chips' size simultaneously with the growth in complexity. In response to the challenges of systems

L. G. Iugan (✉) · H. Boucheneb · G. Nicolescu
Ecole Polytechnique de Montréal, Montreal, Canada
e-mail: luiza.gheorghe@polymtl.ca

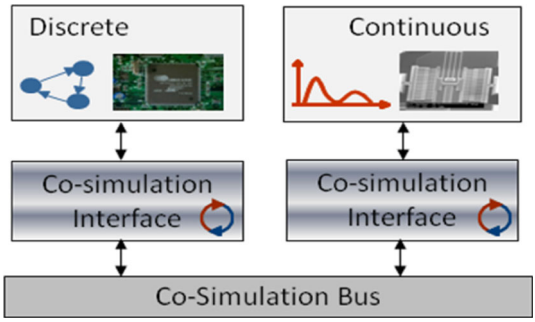

Fig. 1 Global co-simulation model

miniaturization, the International Technology Roadmap for Semiconductors (ITRS) emphasizes the More Than Moore's Law Movement that focuses on system integration rather than increasing transistor density and leads to a functional diversification in integrated systems [1]. Thus, SoCs are currently characterized by the heterogeneity of different modules that are particular to different application domains such as optical, electronical, mechanical, hydraulics and biological. These multi-domain systems are the main driver of the development of a wide range of products across a broad and diverse spectrum of applications in many industries, but not limited to Automotive, Aerospace, Health Care and Consumer Electronics. Given the diversity of concepts manipulated, the global design specification and the validation are extremely challenging. The heterogeneity of these systems makes difficult the elaboration of a global simulation model for the overall validation.

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. This technique allows the designer to use the best execution model and tool for each application domain and provides capabilities to validate the overall model. 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) has beneficial results: the development time, the time-to-market and the costs are reduced. This approach requires the elaboration of a global co-simulation model (Fig. 1).

There are three types of basic elements that compose this model [2]:

- The *execution models* of the different components that form the heterogeneous system (corresponding to the Continuous model and the Discrete model in Fig. 1). An execution model of the different components can be viewed as the interpretation of the computation model. To each computation model we can associate one or more execution models.
- The *co-simulation bus.*
- The *co-simulation interfaces.*

The *co-simulation bus* handles the interconnections between the different components of the system. It serves only as a simple communication means between the two models: the continuous and the discrete, it does not provide synchronization between them.

The *co-simulation interfaces* enable the communication of different components through the simulation bus. They are in charge of the adaptation of different simulators to the co-simulation bus in order to guarantee the transmission of information between simulators executing the different components of the heterogeneous systems. They also have to provide efficient synchronization models for the modules adaptation.

The implementation and the simulation of an execution model in a given context is called a *co-simulation instance*. It refers strictly to the execution model and its simulation for a given application. Several instances may correspond to the same execution model and these instances may use different simulators and may present different characteristics (e.g. accuracy and performances).

In this context, a new type of designers emerged, the designers of co-simulation tools. Their main objective is to provide coherent tools for the co-simulation models' designers. Given the diversity of abstractions, languages and simulation tools, the design of co-simulation tools may be costly and time consuming. Thus, the key for the improvement of the validation process is to define a model-based generic approach before the implementation of these tools. This requires new skills on formalism and formal verification domain.

The main challenge in the definition of new co-simulation tools for continuous/discrete (C/D) systems is due to the heterogeneity of concepts that are manipulated by the discrete and the continuous components.

In a C/D heterogeneous system, we find two distinct models: (1) a continuous model where the computation is carried out in the continuous domain by solving differential or algebraic equations and (2) a discrete model where the computation is carried out in cycles and every cycle represents the computation of a 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 co-simulation. This results in a complex behavior of the interfaces that leads to time consuming design and can be a significant source of errors. Therefore, their automatic generation is mandatory.

An efficient tool for the generation of the co-simulation interfaces must rely on an accurate definition of the behavior of the interfaces that is the formal representation of the co-simulation interfaces. Some characteristics of a formalism for the interface generation are: an execution semantics and a set of rules for the automatic interfaces generation and formal properties for analyses and verifications [3].

More efforts are necessary for the global formal representation and for the formal verification in the co-simulation context that has not been thoroughly if at all addressed until now. The ITRS emphasizes that "a more structured approach to verification demands an effort towards the formalization of a design specification" [1]. The global formal representation of a heterogeneous system requires the formal representation of the synchronization model between the domains. This representation also has to clearly define the computation and the communication for the global co-simulation model and verify the behavior of the interfaces given certain restrictions.

This article proposes a generic conceptual framework based on formal representation, for the modeling and the generation of the co-simulation interfaces. The generality of the framework was demonstrated by the co-simulation of a C/D system using two different simulators for the discrete domain model (SystemC and SystemVerilog) while for the continuous domain model we used Simulink®. We present also an event update schema for a discrete simulator integrated in a C/D co-simulation environment.

The article is structured as follows. Section 2 presents the related work. Section 3 details the synchronization model. Section 4 shows the proposed framework for the design of C/D co-simulation tools. Section 5 illustrates the application of formal methods for the co-simulation tools design. Section 6 presents CODIS, a tool designed using the proposed approach. Section 7 discusses the benefits and limitations of our work; Sect. 8 gives our conclusions.

2 Related work

The effectiveness of the co-simulation technique is currently exploited in industry for the validation of Hw/Sw systems [4,5]. This type of validation requires co-simulation of discrete simulators (Instruction Set Simulators or Native Simulators for Sw components and behavioral event-based discrete simulators for Hw components). However, the heterogeneity increases for the modern systems integrating components from different application domains (e.g. electronics, optics, and mechanics) and the C/D co-simulation becomes necessary. This topic is currently addressed by academia and there is no commercial solution enabling automatic generation of the interfaces enabling concurrent simulation of continuous and discrete simulators. The current practice in industry consists in developing ad-hoc C/D co-simulation interfaces that are not formally verified. The obtained interfaces are only appropriate for specific applications or a unique combination of simulation tools.

The research works existing in the C/D systems validation field can be roughly divided into the following classes: simulation-based and formal representation-based approaches.

Some approaches in the simulation-based group 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 [6–11]. This requires the abandonment of well established efficient tools for the continuous domain (e.g. Simulink®).

There are tools such as Ptolemy in which the systems are designed by assembling together different components [12]. HyVisual is a hybrid systems modeler based on Ptolemy [13] that supports the construction of hierarchical hybrid systems for continuous-time dynamical systems and hybrid systems. However, the different sub-systems and components need to be developed in the same environment in order to be compatible. In [14] Lee argues the need of plurality of distinct actor-oriented modeling languages and mechanisms for composing models in these languages, each with an unambiguous, clear semantics. The proposal is based on formal representation, but the formal verification of the simulation models is not considered.

The Functional Mock-up Interface (FMI) is a "tool independent standard to support both model exchange and co-simulation of dynamic models" [15]. FMI for co-simulation provides a common interface for coupling different simulation engines but data exchange is restricted to discrete communication points. However, the standard does not provide semantics for the communication between the models (co-simulation algorithm) or their formal verification.

The formal representation approaches propose different definitions for heterogeneous systems modeling. In [16] Lee and Vincentelli present a formal framework for comparing different models of computation used in heterogeneous models. The authors propose a formal classification framework that makes it possible to compare and express the differences between them. The intent is "to be able to compare and contrast its notions of concurrency, communication, and time with those of other models of computation" [16]. The role of the model of computation in abstracting functionalities of complex heterogeneous systems was presented by Jantsch and Sander in [17]. In [18], the author proposes the formalization of the heterogeneous systems by separating the communication and the computation; however the execution models for the validation of the interfaces between domains were not taken into consideration. DESTECS [19] is a modeling and co-simulation of a continuous/discrete system tool that uses Vienna Development Method (VDM) to express discrete event models and Bond Graphs for the differential equations. The tool uses a direct connection between the continuous time (CT) interface and the operational semantics of the discrete event (DE) (a connection point-to-point) while our tool-independent approach allows for multiple simulators on a co-simulation bus.

Zeigler introduced in [20], a formalism defined for the modeling and the simulation of discrete event systems (discrete event system specifications—DEVS) where the time advances on a continuous time base. DEVS is a formal approach that can be used to build the models, using hierarchy and modularity. It allows the definition of the operational semantics for a system but not its formal verification. Based on this formalism, [21] proposes a tool for the modeling and simulation of hybrid systems using Modelica and DEVS. The models are "created using Modelica notation and a translator converts them into DEVS models" [21]. In [22], a heterogeneous simulation framework using DEVS BUS is proposed. Non-DEVS-compliant models are converted through a conversion protocol into DEVS-compliant models.

The authors present in [23] a modeling library on top of SystemC, targeting heterogeneous embedded system design, based on four models of computation. The approach is based on formal representation, but the formal verification of the interfaces/models is not considered and the framework is tool-dependent. In [24] the authors present how the HetSC and SystemC-AMS specification methodologies can be used together to enable models based on SystemC language. In [25] the authors address the connection of system parts using different models of computation and data types, using convertor channels. However, there is no formal representation and formal verification and their approach is also tool dependent.

Compared with the previous work, the main contribution of this paper is the definition of a generic conceptual framework based on formal representation for the generation of C/D co-simulation interfaces and the efficient design of C/D co-simulation tools. This framework allows for the convergence of the formal verification and the simulation based approaches into a generic environment in the context of global validation of C/D systems

The proposed framework includes the definition of the operational semantics for a C/D synchronization model with respect to the generic canonical synchronization model [26,27] as well as the formal representation and verification of the behavior of C/D co-simulation interfaces. The functionality and the architecture of the co-simulation interfaces with respect to the execution models of typical discrete and continuous simulators and the canonical synchronization model are defined in this framework. In order to be integrated in a co-simulation tool using this approach, the main property required from the continuous and discrete simulators is to provide application programming interfaces (APIs) or the possibility to develop custom APIs that allow for the addition of some functionality for events management. The importance of the interaction with external tools is currently acknowledged by the simulators designers. Therefore, most of the existing simulators feature the required property. Illustrative simulators compatible with our approach are the two popular continuous simulators: Simulink®, Spice and the discrete simulators: VHDL, SystemC and SystemVerilog simulators.

3 Basic concepts

Before giving the details of the proposed framework we present the basic concepts that are used in our work: DEVS [20], timed automata [28,29] and UPPAAL [30].

3.1 Discrete event systems specification (DEVS)

DEVS is a formalism supporting a full range of dynamic system representation, hierarchical and modular model development. The abstraction separates modeling from simulation and provides atomic models that can be used to build complex simulations, as well as defined coupling of components [20]. Its features allow the integration of continuous and discrete-

event models. It also provides all the mechanisms for the definition of an operational semantics for the continuous/discrete synchronization model, the high level representation of the global formal model.

Definition A DEVS is defined [20] as a structure $M =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$, where:

$X = \{(p_d, v_d) | p_d \in InPorts, v_d \in X\ pd\}$ set of input ports and their values in the discrete event domain;
S is a 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:

– *case 1* where we assume that no external events occur before the expiration time, $t_a(s)$. 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. We propose, for the definition of this type of transition the following rule of the form $\frac{\textbf{\textit{Premises}}}{\textbf{\textit{Conclusions}}}$:

$$\frac{e = t_a(s) \wedge s' = \delta_{int}(s)}{(s, e) \xrightarrow{!\lambda(s)} (s', 0)}$$ where ! represents the operator send.

– *case 2* where 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 $\delta_{ext}(s, e, x)$. For this type of transition we propose the following rule:

$$\frac{e \leq t_a(s) \wedge s' = \delta_{ext}(s, e, x)}{(s, e) \xrightarrow{?x} (s', 0)}$$ where ? represents the operator receive.

The internal transition function dictates the system's new state when no external events occurred since the last transition while the external transition function dictates the system's new state when an external event occurs—this state is determined by the input x, the current state s and how long the system has been in this state, e. In both cases the system passes into a new state s' with a new expiration time $t_a(s')$.

This formalism is used to define each module composing the model; modules that will be integrated into a discrete event system using the formalism for parallel DEVS coupled models [20]. The resulting model is generic, the input and output ports of the coupled DEVS model are configurable in terms of name, number, data type and sampling period.

3.2 Timed automata

A timed automaton [28] 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

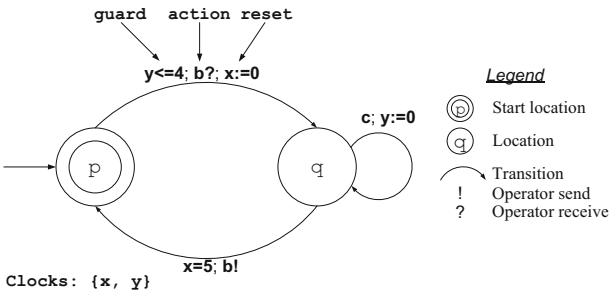

Fig. 2 Example of a timed automaton

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 an 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 [28]. There are two types of clock constraints: constraints associated to transitions and constraints associated to locations. A transition can be taken when the clocks' values satisfy the guard labeled on it. Figure 2 shows an example of a timed automaton. The constraints associated to locations are called invariants and they specify the amount of time that may be spent in a location. The invariant "true" for a location means there are no constraints for the time spent in the location.

The process shown in Fig. 2 starts at a 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 clocks. The actions are used for synchronization and are expressed by a (action) [29]. 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" [29]. In [28] the authors define the state as the tuple: `(l, v)` where `l` is the location and `v` is the *clock valuation*. The *clock valuation* is a function defined on the set of the clocks with values in the real positives including 0 (v vector reflects the actual value of each 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.

The transition showing the time passing is $(l, v) \xrightarrow{t} (l', v')$ if and only if:

$$\begin{cases} v' = v + t \\ \forall t' \in [0, t], (v + t') \textit{ verifies } Inv(l) \end{cases} \tag{1}$$

where `Inv(l)` is the invariant in the location `l, l=l', v'=v + t` showing that for all clocks `x, v'(x)=v(x)+t`. An invariant is a clock constraint that specifies the time that may be spent in a location. It is used to force a transition.

For the discrete transitions (p,v) $\xrightarrow{g;a;r}$ (q,v′), v′ has to satisfy the invariant of q. v′ is obtained from v by resetting the clocks indicated by the reset r.

Compared to other formalism, timed automata have the following characteristics that make them desirable for our formal model.

- the ease and the flexibility of systems' modeling [31].
- a whole range of verification techniques becomes available [31].
- it possesses the adequate expressivity in order to model time constrained concurrent systems.

Our formal model needs to support concurrency between C/D systems thus it was represented as a parallel composition of several timed automata with no constraints regarding the time spent in the locations.

In order to model, validate and check our model we used UPPAAL [30]. UPPAAL [30] is an integrated tool environment for modeling, simulation and verification of timed automata that 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 here simulators), local and global variables, and channels.

The main advantage of UPPAAL is that the product automaton[1] is computed on-the-fly during verification. This reduces the computation time and the required memory space.

The co-simulation requires rigorous concurrent models. An efficient formalization of such models can be realized using multiple automata incorporated into a single one. UPAAL is a well established tool offering formal verification of system modeled as product automata. Therefore, this tool was selected in our approach.

4 C/D synchronization model

Discrete and continuous simulation models present different communication means, time notions and process activation rules. While the behavior of each of these systems was widely presented in the literature (and will be revised in the first part of this section), the synchronization model in the co-simulation context was less shown. The key issue for composing these two types of models is to provide the adaptation of the specific domain concepts.

A linear continuous system is described by the equations:

$$\begin{cases} \frac{dxc}{dt} = A_c xc(t) + B_c u(t). \\ y(t) = C_c xc(t) + D_c u(t) \end{cases} \tag{2}$$

where xc 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 simulation of continuous model, described by differential and algebraic equations, requires solving numerically these equations. A widely used class of algorithms discretizes the continuous time line into an increasing set of discrete time instants, and computes numerically values of state variables at these ordered time instants.

[1] The product automaton is defined as multiple automata in a system, incorporated into a single one. The product automaton creates a new state for all possible states of each automaton [30].

The discrete system is described by the equations [32]:

$$\begin{cases} xd(t_{(k+1)}) = f(xd(t_k), u(t_k), t_k) \quad \text{with} \quad x(t_0) = x_0 \\ y(t_k) = g(xd(t_k), u(t_k), t_k) \end{cases} \tag{3}$$

where x_d is the discrete state vector, u the input signal vector, y the output signal vector [32].

For the linear discrete systems, (3) becomes:

$$\begin{cases} xd(t_{(k+1)}) = A_d xd(t_k) + B_d u(t_k) \\ y(t_k) = C_d xd(t_k) + D_d u(t_k) \end{cases} \tag{4}$$

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

For the model presented in this paper, the time interval is $[t_k, t_{k+1}]$. The input signal vector for the continuous domain is the output signal vector from the discrete domain and vice versa. The simulation of discrete models is based on *events* [32]. At each simulation cycle, the first event with the smallest time stamp (that is the time of occurrence of the event) is processed and the processes sensitive to this event are executed. This may generate other events causing execution of other processes. Once all events with discrete time stamp equal to the current time have been treated, the simulator advances the time to the nearest discrete scheduled event.

The events exchanged between the discrete and the continuous simulators are:

- *discrete events* (from the discrete simulator)—timed events that are scheduled by the discrete simulator. They are defined by the couple (value, time of occurrence).
- *state events* (from the continuous simulator)—events that are triggered by a condition that depends on the values of the state or other dynamic variables in the simulation such as zero passing or a threshold crossing. They are generated by the continuous simulator, their time stamp depends on the values of state variables [33].

Both discrete events and state events can be unpredictable.

When stepping ahead in time, a discrete event simulator must consider the time stamps of the events sourced from the interfaces to other simulators (the continuous time simulator in this case), and the continuous time simulator must consider the new state (boundary) conditions fixing state values at certain time stamps imposed from their interface to other simulators (the discrete time simulator in this case). The response of one simulator at events generated by the other simulator is called here events detection. The time stamps are the synchronization and communication points between the different simulators involved in a global simulation. Depending on the Discrete Model (DM) behavior the following synchronization modes, without roll back are possible:

- A loose synchronization that can be used when discrete events are predictable (e.g. periodic events). In this case, event time stamps and sampling events are placed and sorted in a queue. In order to obtain the time stamp of the subsequent output event, the queue is consulted to find the minimum time stamp. Then, the type event (sampling or update signal) is verified and the information is sent to the continuous simulator.
- A conservative synchronization that can be used when the events are unpredictable. In this mode, the discrete simulator sends its next discrete time (always known) which may correspond to the time stamp of a signal update event. The synchronization overhead specific to this mode depends on the DM computation granularity.

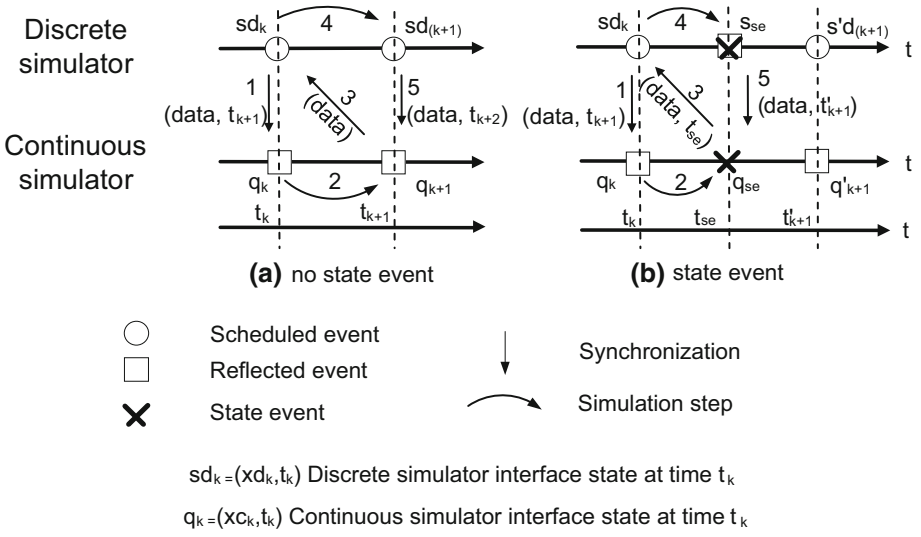

Fig. 3 Synchronization model in C/D co-simulation interfaces

Both synchronization modes can be used during the same simulation:

- If the timed events are predictable and no unpredictable event is triggered by the continuous model during part of the simulation, the simulator works in loose synchronization mode, in order to avoid overhead.
- If unpredictable events are triggered by the continuous model the simulator would change to conservative synchronization.

In the work presented here, in order to provide accuracy and avoid roll back we consider a conservative synchronization mode. In order to avoid the roll back, there is a sequentialization of the execution of the simulators. The discrete simulator is always with one step behind the continuous simulator therefore, the discrete simulator will always detect the state events from the continuous simulator.

Figure 3 presents the synchronization model in the C/D interfaces without (Fig. 3a) and with state event (Fig. 3b).

For a rigorous synchronization, the discrete domain has to detect the events generated by the continuous domain and the continuous simulator must detect the scheduled events from the discrete domain. The simulators have to be controlled by the co-simulation interfaces in order to provide these functionalities.

At the time t_k the discrete simulator is in the state (xd_k, t_k) with xd_k the the location and t_k the k-th discrete time (that can be seen also as the kth event in the queue of events in the discrete domain). The discrete simulator executes all the processes sensitive to this event and sends to the continuous simulator the C/D coherence data, the next synchronization time t_{k+1} and switches the context from the discrete to the continuous simulator without advancing the time (arrow 1 in Fig. 3a, b).

The continuous simulator that is in state (xc_k, t_k) receives the data and the time from the discrete simulator and starts the simulation of the continuous model. The simulation will stop when the time equals t_{k+1}, (sent by the discrete simulator) or when a state event was generated.

The behavior of the continuous interface can be described by the following transition state (arrow 2 in Fig. 3a, b):

$$(xc_k, t_k) \rightarrow \begin{cases} (xc_{k+1}) & \text{if} \quad t = t_{k+1} \quad (5) \\ (se, t_{se}) & \text{if} \quad t < t_{k+1} \quad (6) \end{cases}$$

where tc is the time in the continuous domain, the state (xc_{k+1}, t_{k+1}) is the state of the continuous simulator when no state event was generated in the time interval $[t_k, t_{k+1}]$. The state events are, as were defined in Sect. 4 of this article, events generated by the continuous simulator at times that are not known beforehand. 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_k (arrow 3 in Fig. 3a, b). The event taken into consideration is 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 (and consequently a synchronization point).

In the case described by Eq. (5), after switching the context, the discrete simulator will advance to the time t_{k+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 interface the discrete simulator sends the C/D state coherence data (that is synchronization data and, eventually, a result of a process execution) and the time of the next scheduled event t_{k+2} (also the next synchronization point) and the cycle restarts (arrow 4 in Fig. 3a).

Equation (6) describes the case where a state event occurred. The continuous simulator sends not only the data but also the time when the state event occurred t_{se} (arrow 3 in Fig. 3b). The discrete simulator advances until this time (t_{se}) where it executes all the processes sensitive to the event and recalculates the time of the next scheduled event t'_{k+1} (arrow 5 in Fig. 3b). According to the state event the time stamp of the next scheduled event in the discrete simulator can stay the same (t_{k+1}) or can change (t'_{k+1}) as a consequence of the event triggered by the continuous simulator. This time stamp takes any value bigger than t_{se} but smaller than t_{k+1}. The discrete interface switches the context to the continuous simulator and sends the data and the time stamp of the next scheduled event that is a synchronization point between the two simulators. The advantage of this model is that it avoids any need of roll-back even if one or multiple state events were generated.

Figure 4, inspired by [32] presents the event update schema for a discrete simulator integrated in a C/D co-simulation environment. In [32] the authors proposed the event update schema for a purely discrete event system. Figure 4 extends this schema with the interaction in terms of communication/synchronization (through the events exchanged) between the discrete and the continuous simulators.

The elements used in this representation respect the definitions introduced in [32]:

- The discrete simulator maintains a list of the feasible events at the current state named Scheduled Event List L = $\{(xd_k, t_k)\}$ with k=1,2,3,...n. The list is ordered on the smallest-scheduled-time-first basis [32].
- The queue of events is ordered by the events lifetimes, from the smallest to the largest. The lifetime v_k is the length of the time interval between two successive events [32].

The Scheduled Event List (SEL) that is the list we use for the discrete events and their time stamps is reordered each time the context is switched from the continuous domain to the discrete domain. It is possible that some events in the discrete domain will become unfeasible

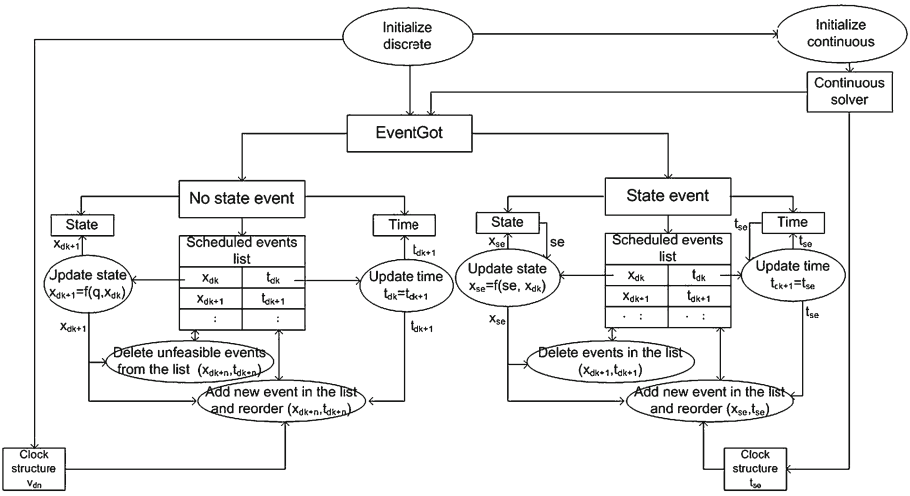

Fig. 4 The event update schema for the discrete simulator

as a consequence of an unpredictable state event so they have to be deleted from the list. There are two possible behaviors of the scheduler (corresponding to the two branches in Fig. 4), both of them depending on the behavior of the continuous domain:

- When no state event was generated in the continuous domain.
- When a state event was generated in the continuous domain.

In both cases State is initialized to a given value x_0 and the simulation time- Time is initialized to 0. The Clock Structure is a set of clock sequences, one for each event. The following steps are executed (see Fig. 4):

Step1 - First entry in the list (x_{dk}, t_k) is removed from the SEL
Step2 - Time is updated to a new time:
t_{k+1}, if no state event.
$t_{se} < t_{k+1}$, if state event.
Step3 - State is updated according with the transition function:
$x_{dk+1}=f(q,x_{dk})$ with q the data from the continuous domain, if no state event.
$x_{se}=f(se,x_{dk})$ with se the data from the continuous domain, if state event.
Step4 – If no state event - the events that became infeasible after the data is received from the continuous domain are deleted from the SEL [32].
If state event - the event is added in the list always as the next entry to be removed from the list. The events that became infeasible as a consequence of the detection of a state event (which is an unpredictable event) are deleted from the list [32].
Step5 - New feasible events that are a consequence of the data and/or state events received from the continuous domain are added to the SEL.
Step6 - The SEL is reordered on the smallest-scheduled-time-first basis.

The procedure repeats with step 1 for the new list. If no state event occurs, the clock structure is controlled by the discrete domain; the events queue is reordered by the discrete kernel. In the case when a state event occurs, the clock structure is controlled by the continuous solver, the time of the state event is sent by the continuous domain and the first consequence is the re-start of the discrete simulator at a time t_{se}, before the expected time t_{k+1}.

5 Generic conceptual framework based on formal representation

This section introduces a conceptual framework, formal representation based, for the design of C/D co-simulation tools. The framework presents several steps that are independent of the simulation tools used for modeling of the continuous and discrete components of the system. During these steps, the co-simulation interfaces are defined; their functionality and the internal structure of co-simulation interfaces are expressed using existing formalisms and temporal logic. The framework is composed by the following steps:

1. Definition of the operational semantics for the synchronization in C/D global execution models.
2. Distribution of the synchronization functionality to the co-simulation interfaces.
3. Formalization and verification of the co-simulation interfaces behavior.
4. Definition of the library elements and the internal architecture of the co-simulation interfaces.

This structure leads to the rigorous definition of the internal architecture of the interfaces that is the skeleton for the library elements implementation using existing simulation tools. The rigorous definition of the required functionality for co-simulation interfaces is naturally followed by the implementation of the interfaces components in an interfaces library (by tool designers) and the generation of the global execution model. During this stage (implementation), the developers will:

5. Analyze the existing simulators for the discrete, respectively continuous domain.
6. Implement the library elements specific to different simulation tools, using these simulators.
7. Validate the implementation.

Figure 5 presents the proposed conceptual framework in the context of the generation of global execution models completed with the implementation stage that is not the main focus of the work presented is this article. The first four steps will be detailed in the following subsections.

5.1 Definition of the operational semantics for the synchronization in C/D global execution models

The first step of the framework is the definition of the operational semantics for the synchronization in C/D global execution models. An operational semantics gives a detailed description of the system's behavior in mathematical terms. This model serves as a basis for analysis and verification. The description provides a clear language independent model that can serve as a reference for different implementations.

The operational semantics for C/D systems requires the rigorous representation of the interaction between simulators (communication/synchronization, data exchanged) as well as their high level and dynamic representations.

In order to demonstrate the approach we show here the definition of the operational semantics of the co-simulation interfaces with respect to the synchronization model presented in Sect. 4 using DEVS. The operational semantics for the C/D synchronization model is given by the set of rules presented in Table 1. The notations used in this table are:

- DataDC—Data sent by the discrete simulation interface (DSI) to the continuous simulation interface (CSI) that is also the data received by CSI from DSI.
- DataCD—Data sent by CSI to DSI that is also the data received by DSI from CSI.

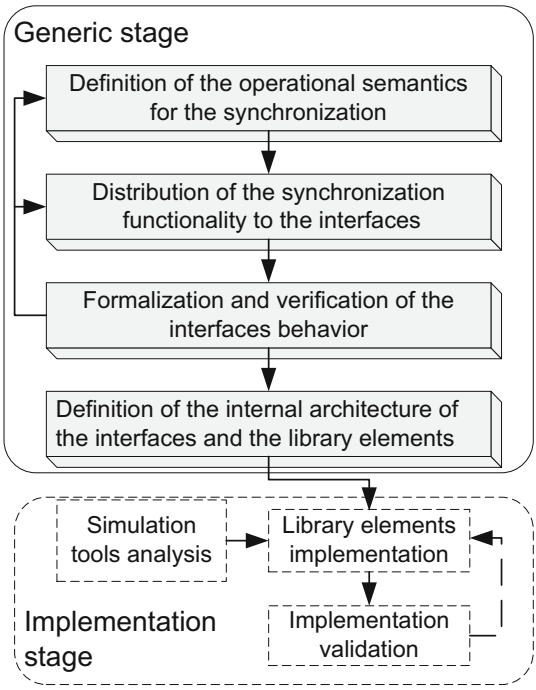

Fig. 5 Conceptual framework integrated in the global co-simulation model (for co-simulation tools design)

In Table 1, DataDC is the output function from the discrete domain interface λ(sd), and DataCD is the output function from the continuous domain interface λ(q). The semantics of the global variable `flag` is related to the context switch between the continuous and discrete simulators. When `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.

In order to introduce the rules, we detail here the first rule, corresponding to the arrow 1 in Fig. 3a, b. 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 (δ_{ext}) for the continuous model.

The discrete model is initially in the total state (s_{dk}, e_{dk}), this means it is in the state s_{dk} from the time e_{dk}. In this state, the discrete simulator performs the following actions: (1) sends the data and the value of its next time stamp (this action is expressed by !(DataDC, $t_a(s_{dk})$) and (2) switches the co-simulation context to the continuous model (this action is expressed by `flag = 0`). For the same rule, the continuous model is in state q_k and performs the following actions: (1) receives the data and the value of the time stamp from the discrete simulator (expressed by ?(DataDC, $t_a(s_{dk})$) and (2) sets the global variable `synch` to 0 (action expressed by `synch=0`) in order to respect the premise of the rule corresponding to the arrow 4.

The actions expressed by this rule are executed by the discrete simulator when the context will be switched to it.

5.2 Distribution of the synchronization functionality to the co-simulation interfaces

Based on the operational semantics, we can now define the synchronization functionality between the continuous and the discrete simulators. This functionality is provided by the

Table 1 Operational semantics for continuous/discrete synchronization model

Rule	Arrows in Fig. 3
$\dfrac{synch{=}1 \ \wedge \ flag{=}1 \wedge q_k{=}\delta_{ext}(q_k)}{(sd_k,ed_k) \xrightarrow{!(DataDC,t_a(sd_k));flag:=0} (sd_k,ed_k);q_k \xrightarrow{?(DataDC,t_a(sd_k));synch:=0} q_k}$	Arrow 1 Fig. 3a, b
$\dfrac{flag{=}0 \wedge \neg stateevent \ \wedge q_{k+1}{=}\delta_{\text{int}}(q_k)}{q_k \xrightarrow{\delta_{\text{int}}} q_{k+1} \xrightarrow{!DataCD;flag:=1} q_{k+1}}$	Arrows 2 and 3 in Fig. 3a
$\dfrac{synch{=}0 \ \wedge \ flag{=}1 \wedge \neg stateevent \ \wedge \ sd_{k+1}{=}\delta_{ext}(sd_k)}{(sd_k,ed_k) \xrightarrow{t_a(sd_k):=t_a(sd_k)-ed_k} (sd_k,t_a(sd_k)) \xrightarrow{?DataCD,\delta_{int}(sd_{k+1});\lambda(sd_{k+1});synch=1} (sd_{k+1},0)}$	Arrow 4 in Fig. 3a
$\dfrac{flag{=}0 \ \wedge \ \neg stateevent \wedge \ q_{k+1}{=}\delta_{\text{int}}(q_k)}{q_k \xrightarrow{\delta_{\text{int}}} q_{se} \xrightarrow{!DataCD;flag:=1} q_{se}}$	Arrows 2 and 3 in Fig. 3b
$\dfrac{synch{=}0 \ \wedge \ flag{=}1 \ \wedge \ \neg stateevent \ \wedge \ sd_{k+1}{=}\delta_{ext}(sd_k,t)}{(sd_k,ed_k) \xrightarrow{?t_{se}} (sd_k,t_{se}) \xrightarrow{?DataCD,\delta_{\text{int}}(s_{se});\lambda(s_{se});synch=1} (sd_{se},0)}$	Arrow 4 in Fig. 3b

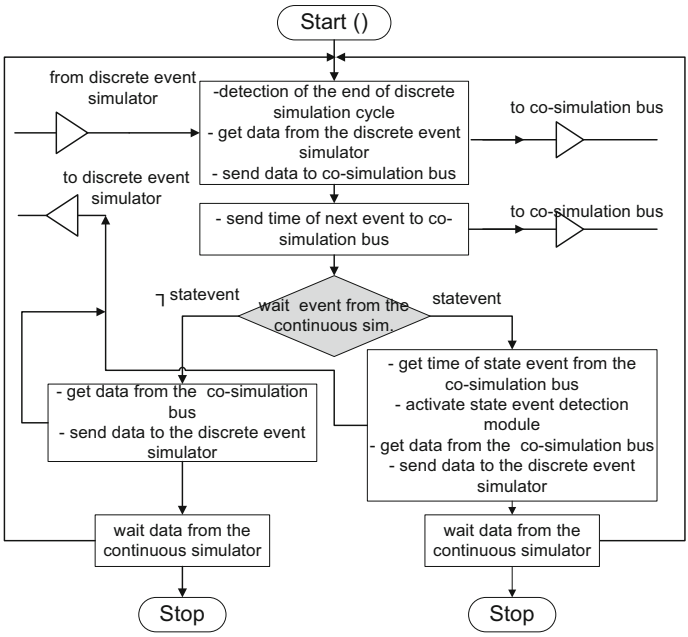

Fig. 6 Flowchart for the discrete domain interface

interfaces that are the link between the different execution models and the co-simulation bus (see Fig. 1). They are each in charge of a part of the synchronization between the two models. To ensure the system's flexibility, the synchronization functionality has to be distributed to the co-simulation interfaces. Moreover, each computation step has to be thoroughly specified.

The behavior of the discrete domain interface can be described by a few processing steps detailed in Fig. 6:

– exchanging data between the simulators.
– sending the time stamps of the next events.
– the state events consideration.
– the context switch to the continuous interface.

The flowchart allows for the definition of the operational semantics specifically for the discrete interface. The representation can also be done using DEVS for coherence purposes. From the rules we can trace the state graph of the DSI for the canonical synchronization model as shown in Fig. 7. The dashed lines represent internal transitions and the corresponding states and the plain lines represent external transitions and the corresponding states.

The behavior of the continuous domain interface can also be described by a few processing steps (see Fig. 8):

– exchanging data between the simulators.
– sending the time stamps of the next events.
– the indication (to the discrete interface) of the occurrence of a state event.
– the context switch to the discrete interface.

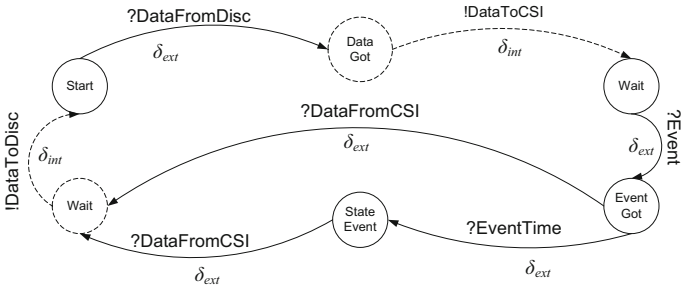

Fig. 7 State graph of the DSI for the canonical synchronization model represented using DEVS

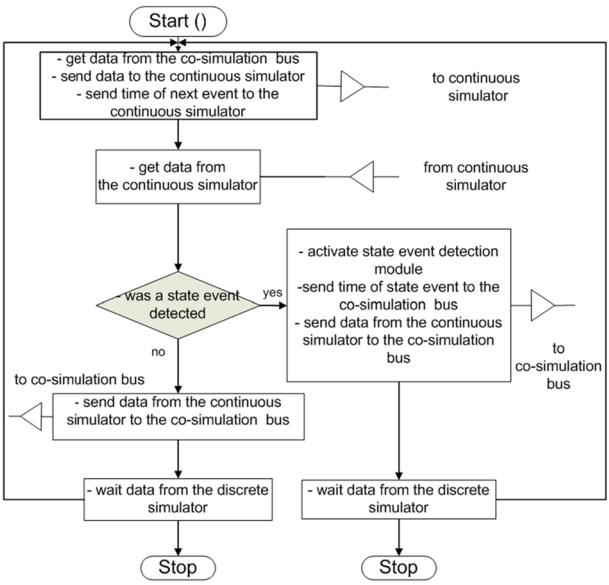

Fig. 8 Flowchart for the continuous domain interface

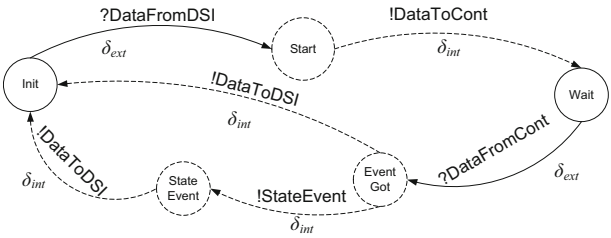

Fig. 9 State graph of the CSI represented using DEVS

Same as for the discrete interface we can trace the state graph of the DSI for the canonical synchronization model as shown in Fig. 9. The dashed lines represent internal transitions and the corresponding states and the plain lines represent external transitions and the corresponding states.

5.3 Formalization and verification of the co-simulation interfaces behavior

After the distribution of the synchronization functionality to the interfaces, the resulting state graphs are interface specific and show the behavior of each interface. The formalization and verification of the co-simulation interfaces behavior stage can be roughly divided in two or three steps: formalization (that can be the formal specification of the heterogeneous system), the validation by simulation and/or the formal verification. The validation can be done either through simulation or formal verification. For our work we need to validate through formal verification because our goal was to check system's properties for a broad class of inputs. The validation was also done by simulation because the tool we used (UPPAL) allowed it.

The two main techniques that can be used for the formal verification of the interfaces are model checking and theorem proving [34].

Considering that our system is dynamic, it is necessary to use a formalism that allows the expression of dynamic properties (the state of a system changes and by consequence the properties of the state also change). The temporal logic handles formalization where the properties evolve over the time and in general uses:

- Propositions that describe the states (i.e. elementary formulas and logical connectors).
- Temporal operators that allow the expression of the properties of the states successions (called executions).

For our formal model, the properties that need to be checked are the connection (branching) properties expressed using Computation Tree Logic (CTL), which is an untimed temporal logic or its timed extension TCTL logics. Figure 10 shows the global formal model for the validation of the C/D simulation interfaces with respect to the synchronization model presented in Sect. 4. The input is the simplified formal representation of the continuous and the discrete models while the interfaces are the timed automata presented here. This model is further used for the simulation and formal verification. We can see here the similarities between the global co-simulation model shown in Fig. 1 and the global formal model used in our application. The next subsections focus on the detail the formalization and the formal verification of the co-simulation interfaces.

5.3.1 Formalization of the co-simulation interfaces

Figure 11 shows the formal model (using timed automata) for the discrete domain interface. The model has only one initial location (marked in Fig. 11 by a double circle) Start.

In [35] the authors present the transition from DEVS model to timed automata. In our approach the timed-automata model completes the DEVS graph shown in Fig. 11 with the addition of the timing evolution notions. Thus, the transition labels may include operation for the time variable update.

For instance, during the transition from the DataFromCont state to the Start state, the value of the NextTime variable is assigned to the t_d 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.

The discrete interface will change location from Start to NextTimeGot following the transition:

$$Start \xrightarrow{DataFromDisc?} NextTimeGot$$

This is an external transition realized with zero time and it is triggered by the receiving of the data (that is also a synchronization between the discrete simulator and the interface) from

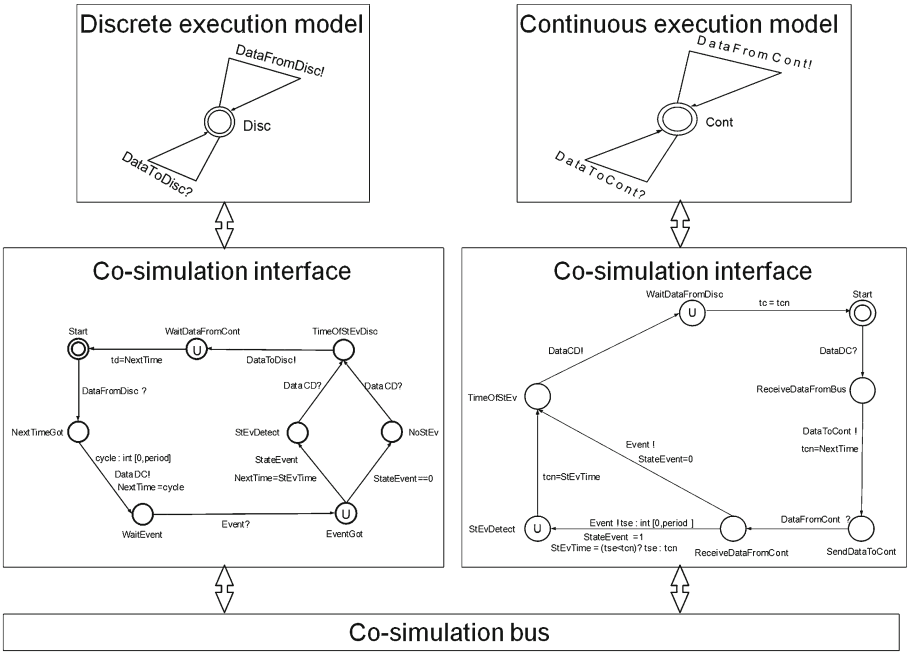

Fig. 10 Global formal model for the validation of C/D interfaces behavior

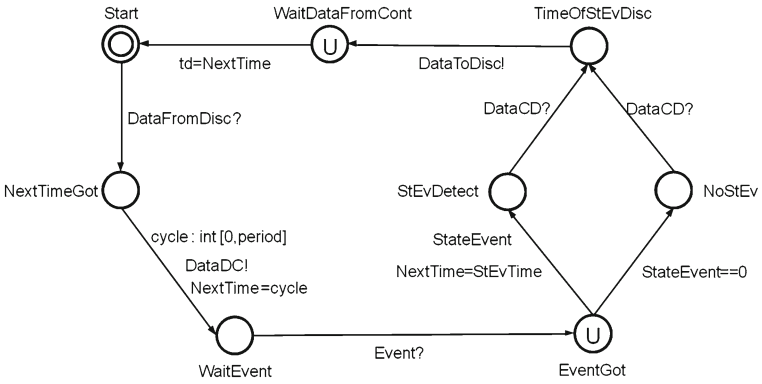

Fig. 11 The discrete domain interface model

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:

$$NextTimeGot \xrightarrow{DataDC!,NextTime=cycle,cycle:int[0,period]} WaitEvent$$

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 DataDC!). The variable NextTime is the time of the next event in the discrete domain. This variable takes, in this mode, the value `cycle`. For an accurate co-simulation we assume the sampling

intervals are not equidistant, therefore, cycle takes random values in an interval defined here as `[0, period]`. 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?. Once arrived to EventGot location, the occurrence of a state event in the continuous domain is considered. EventGot is an urgent location (as defined in the previous section). Two cases are possible:

(1) When no state event was generated by the continuous domain, the location changes from EventGot to NoStEv. The transition

$$EventGot \xrightarrow{StateEvent==0} NoStEv$$

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:

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

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 will reach the next location: TimeOfStEvDisc. In both cases the model performs synchronization (DataCD?). 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.

The graph presented in Fig. 9 with the addition of the time notion allows for the representation of the continuous domain interface using timed automata (see Fig. 12) [35]. This formal model also has only one initial location (marked in Fig. 12 by a double circle) Start.

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

$$Start \xrightarrow{DataDC?} ReceiveDataFromBus$$

This is also an external transition realized with zero time and it is triggered by the receiving of the data from the discrete interface (DataDC?) 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

$$ReceiveDataFromBus \xrightarrow{DataToCont!,tcn=NextTime} SendDataToCont$$

The value NextTime, the time of the next event (occurred/scheduled event) in the discrete simulator is assigned to t_{cn}, the next time scheduled by the discrete simulator which will

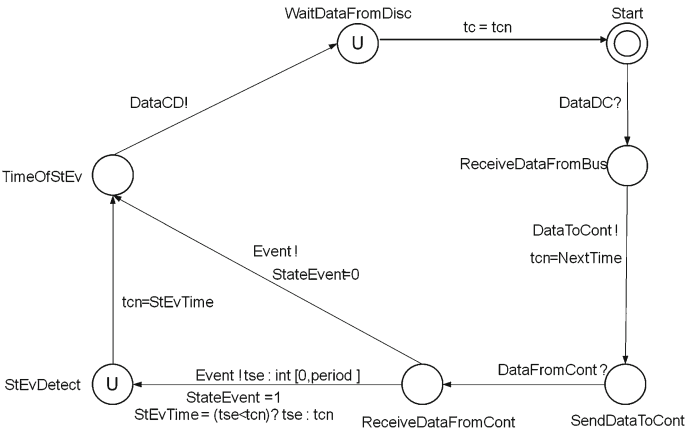

Fig. 12 The continuous domain interface model

be transferred to the continuous time simulator. In our model, the synchronization on this transition is between the continuous domain interface and the continuous domain simulator. The interface sends to the simulator data received from the discrete domain interface and the time of the next event in the discrete domain.

The system changes the location from SendDataToCont to ReceiveDataFromCont following the synchronization transition:

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

During this transition the continuous interface receives data from the continuous simulator and the time of the state event if a state event occurred. 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:

$$ReceiveDataFromCont \xrightarrow{Event!StateEvent=0} TimeOfStEv$$

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:

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

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:

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

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 (DataCD!) between the continuous interface and the continuous simulator. The next location is Start, the continuous time variables is initialized on this channel ($t_c = t_{cn}$) and the cycle restarts.

5.3.2 Formal model simulation and formal verification

The UPPAAL tool allows for the validation of the system's expected behavior regarding the synchronization, the communication and conflicts. Our goal was to verify all the possible dynamic executions of our model.

The formal verification consists of checking properties of the system for a broad class of inputs [36]. In our work we checked properties that fall into three classes:

- Safety properties—the system does not get into an undesirable configuration, (i.e. deadlock, etc) [34,37].
- Liveness properties—some desired configuration will be visited eventually or infinitely (i.e. expected response to an input, etc.) [34,37].
- Reachability properties—the system always has the chance of reaching a given situation (some particular situation can be reached) [36].

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

P0 Deadlock freedom (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[] not 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 [36].

Definition A state is in deadlock if it is impossible that the model evolves to a successor state neither by waiting some time nor by a transition between locations, i.e. there are no enabled transitions.

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 continuous/discrete heterogeneous systems. We defined a liveness property in order to check this behavior that is stated as follows:

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

P2 No state event detected in discrete domain if no state event generated 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.

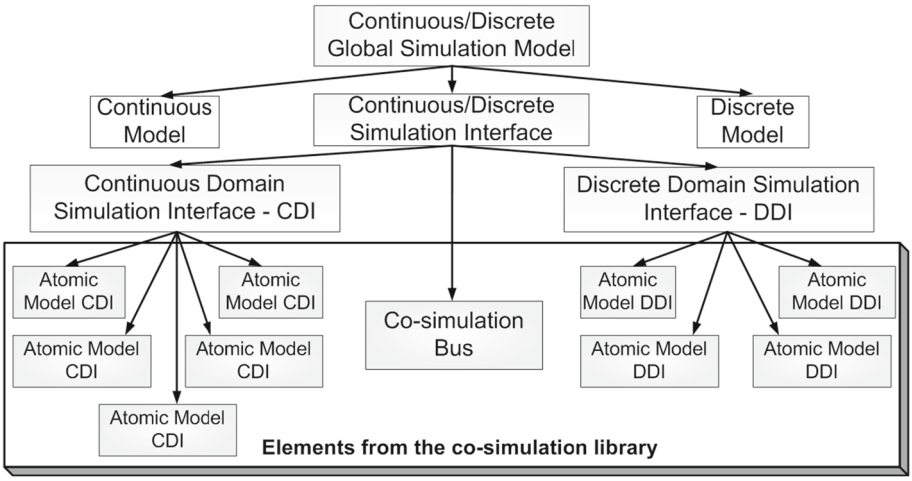

Fig. 13 Hierarchical representation of the generic architecture of the co-simulation model

P3 Synchronization between the interfaces (reachability 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 are at the same time stamp (and by consequence are synchronized).

Definition Invariantly both processes in the Start location (initial state) imply the time in the continuous t_c is equal with the time in the discrete t_d.

P4 Causality principle (liveness property)

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. The causality is already verified by P3 for scheduled events. However, when a state event is generated by the continuous domain, the discrete domain has to 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 t_c is equal to the time in the discrete t_d.

5.4 Definition of the internal architecture of the co-simulation interfaces

The formalization of the co-simulation interfaces behavior step is naturally followed by the definition of their internal architecture. This definition eases the generation of the co-simulation interfaces. We present in Fig. 13 the hierarchical representation of the global co-simulation model used in our approach.

At the top hierarchical level, the global model is composed of the continuous and discrete models and of the C/D co-simulation interface required for the global co-simulation.

The second hierarchical level of the global co-simulation model includes the domain specific co-simulation interfaces and the co-simulation bus in charge of the data transfer between these interfaces.

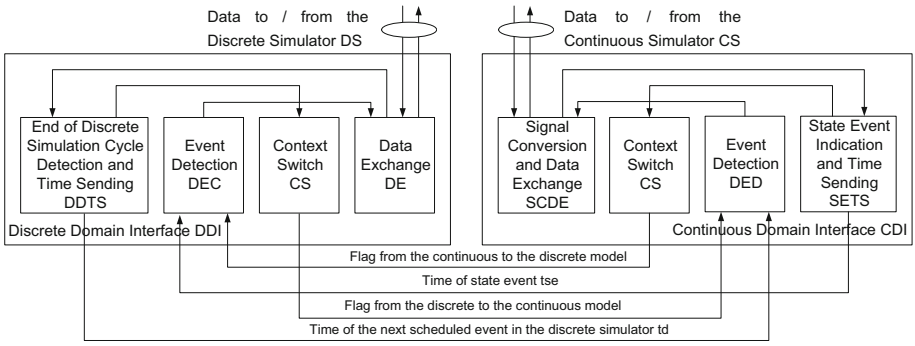

Fig. 14 Internal architecture of the C/D co-simulation interface

The bottom hierarchical level includes the elements from the co-simulation library that are the atomic modules of the domain specific simulation interface. These atomic components implement basic functionalities of the synchronization model. The overall continuous/discrete co-simulation interface is formally defined using DEVS formalism. The interface is composed by as a set of coupled models that are: the continuous domain interface (CDI), the discrete domain interface (DDI) and the co-simulation bus.

The specific functionalities of the interfaces were presented in Sect. 5.2. In term of internal architecture the blocks assuring these features are:

- For the Continuous Model Simulation Interface:

– The State Event Indication and Time Sending block (SETS).
– The Signal Conversion and Data Exchange block (SCDE).
– The Event Detection block (DED).
– The Context Switch block (CS).

- For the Discrete Model Simulation Interface:

– The End of Discrete Simulation Cycle Detection and Time Sending block (DDTS).
– The Data Exchange block (DE).
– The Event Detection block (DEC).
– The Context Switch block (CS).

These atomic modules are forming the co-simulation library and the co-simulation tools enable their parameterization and their assembly in order to generate a new co-simulation instance.

Figure 14 presents the atomic modules interconnection in each domain specific simulation interface. The block that is the co-simulation bus was omitted for a clearer figure, but we show the signals and the interactions between the interfaces.

The modules connect with each other through ports. A port can be defined as a module's point of interaction with other modules. The interactions are limited to sending and receiving data (where time is part of the data). Each module has at least one port. Two ports are connected when the ports that communicate are compatible but they have different directions. Depending on the data flow direction a module can have a set of Input ports and a set of Output ports.

A port can be defined by a tuple {abstraction, type, number, direction, execution type}: *portname = {level, type, number, direction, execution}* where:

– *level* ∈ *{abstraction levels set}*

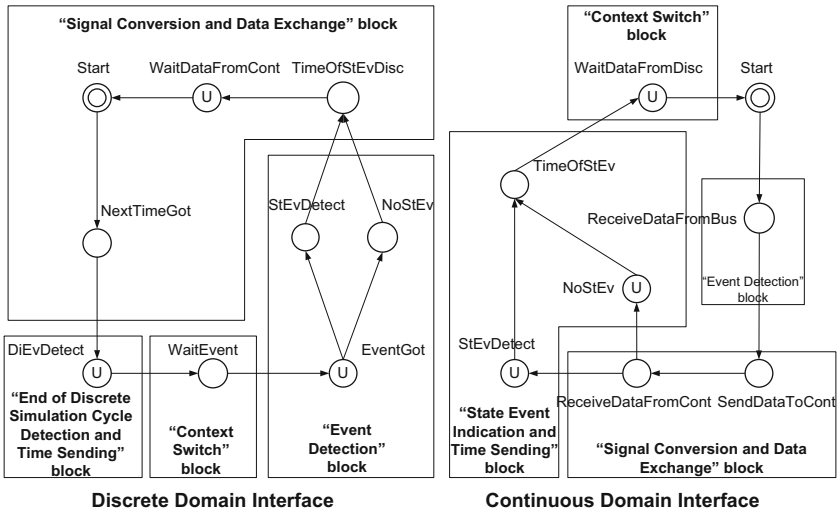

Fig. 15 Formal model and the internal architecture of the co-simulation interface

- *type* ∈ *{port type set}*
- *number* ∈ $N\backslash\{0\}$
- *direction* ∈ *{Input ports set}* ∨ *{Ouput ports set}*
- *execution* ∈ *{continuous, discrete}*

Two ports are *compatible* when there is a correspondence 1->1, 1->N or N->1 between them, the type of the port that sends is the same as or less than the type of the receiving port(s). If the send port has a declared type, it can also send types that are less than the declared type (i.e if the type is declared *double*, it can send *float*) `outType<=inType` [38]. In terms of ports compatibility, the following situations can be found:

- *covariance* converting from narrow to wide [39].
- *contravariance* converting from wide to narrow [39].
- *invariance* not able to convert.

The architecture of the discrete domain interface and the continuous domain interface are also formally defined as a set of coupled modules. Formal descriptions for DDI and CDI respect the coupled module DEVS formalism [20]. Each element of the structure follows the concepts presented in Sect. 5.1 and applied for the overall continuous/discrete co-simulation interface. The modules are implemented in a generic way.

5.5 Discussion

There is a tight connection between the internal architecture of the C/D co-simulation interfaces (shown in Fig. 14) and formal representation of the continuous and discrete co-simulation interfaces using UPPAAL (and shown in Figs. 10 and 11). Each block that provides a functionality in the internal architecture of the interface corresponds to at least one location in the domain specific formal model.

Figure 15 shows the correspondence between the blocks that represent the internal architecture of the co-simulation interface and the locations in the formal model. The mapping

of the formal model onto the blocks that form the interfaces helps with the definition of the elements that are necessary for the library.

6 Application of the conceptual framework to the design of A C/D co-simulation tool

The steps of the proposed methodology presented in Sect. 5 describe the gradual refinement from the operational semantics to the definition of the internal architecture of the co-simulation interfaces. They are independent of the different simulation tools and specification languages used generally for the specification/execution of the continuous and discrete sub-systems.

The considerations presented in the previous steps of the methodology show that specific functionalities are required for the co-simulation of continuous and discrete modes. Therefore, the integration of a simulation tool in the co-simulation environment demands their analysis. Thus, in the case of continuous simulator integration in the co-simulation tool, this simulator has to provide APIs enabling the following controls:

- State events detection.
- Setting break points during differential equation solving.
- On-line update of the breakpoints settings.
- Mechanism for sending processing results and information for synchronization (i.e. the time step of the state event) to the discrete simulator. This implies generally the possibility to integrate C-code and Inter-Process Communications (IPC).

After the analysis of the existing tools we found that Simulink® [40] is an illustrative example of a continuous simulator enabling the control functionalities presented here. These functionalities can be added in configurable library blocks and a given Simulink® model may be prepared for the co-simulation by parameterization and addition of these blocks. These blocks are manipulated like all other components of the Simulink® library. They contain input/output ports compatible with all model ports that can be connected directly using Simulink® signals.

For the integration of a discrete simulator in the co-simulation tool, the simulator has to enable the addition of the following functionalities:

- End of simulation cycle detection.
- Insertion of new events (state events) in the scheduler's queue. This must be done before the advancement of the simulator time.
- Mechanism for sending processing results and information for synchronization to the continuous simulator (i.e. the time stamp of its next discrete event).

Several discrete simulators present the characteristics required for the integration in the co-simulation tool. SystemC [41] is an illustrative example. Since it is open source, SystemC enables the addition of the presented functionalities in an efficient way—the scheduler can be modified and adapted for co-simulation. In this way, the co-simulation overhead may be minimized. In order to increase the simulation performance, part of the synchronization functionality has been implemented at the scheduler's level, which is a part of the state event management and the end of the discrete cycle detection (detects that there are no more delta cycles at the current time).

Following the proposed steps, CODIS a co-simulation tool allowing continuous/discrete co-simulation (see Fig.16), was implemented. The tool uses Simulink® [40] for the modeling of the continuous execution model and SystemC [41] for the modeling of the discrete execution model.

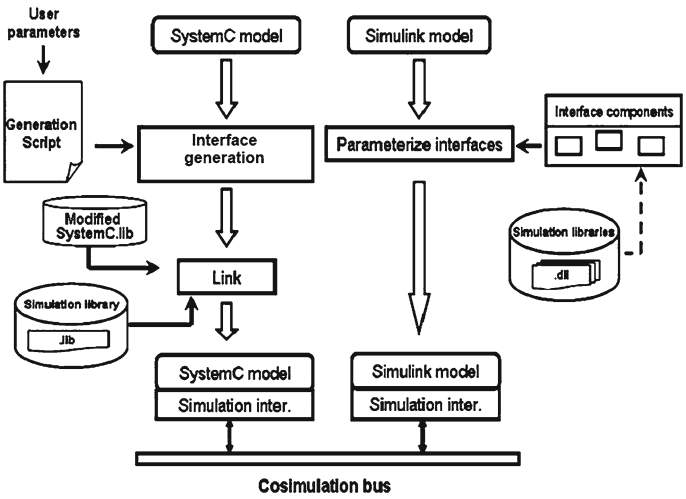

Fig. 16 CODIS framework—co-simulation flow

The co-simulation interfaces that are specific for each domain are generated by selecting components, from a co-simulation library. The inputs in the flow are the continuous model in Simulink® and the discrete model in SystemC which are schematic and respectively textual models. The output of the flow is the global co-simulation model instance.

6.1 Organization of the simulation libraries for SystemC and Simulink®

The Simulink® interfaces are functional blocks programmed in C++ using S-Functions. These blocks are manipulated like all other components of the Simulink® library. They contain input/output ports compatible with all model ports that can be connected directly using Simulink® signals. The user starts by dragging the interfaces from the interface components library into his model's window, then parameterizes them and finally connects them to the inputs and the outputs of his model. The parameters of the interfaces are the number of input and respectively output ports, their type, and the number of state events. Before the simulation, the functionalities of these blocks are loaded by Simulink® from the .dll libraries.

The Simulink® interfaces are:

- **Sim_inter_In** interface provides the input communication function and synchronization with signals update events.
- **Sim_inter_Out** interface implements the output communication function and provides synchronization with the sampling events sent by SystemC.
- **State** interface implements synchronization functions used to send the state events once detected and to synchronize with SystemC. For the detection we use the Hit Crossing component from Simulink® library.
- **Sync** interface implements the synchronization function that creates break points which must be reached accurately by a solver (a variable step solver).

Figure 17 shows an example of simulation interfaces utilization.

For SystemC, in order to increase the simulation performances, a part of the synchronization functionalities have been implemented at the scheduler's level which is a part of the state

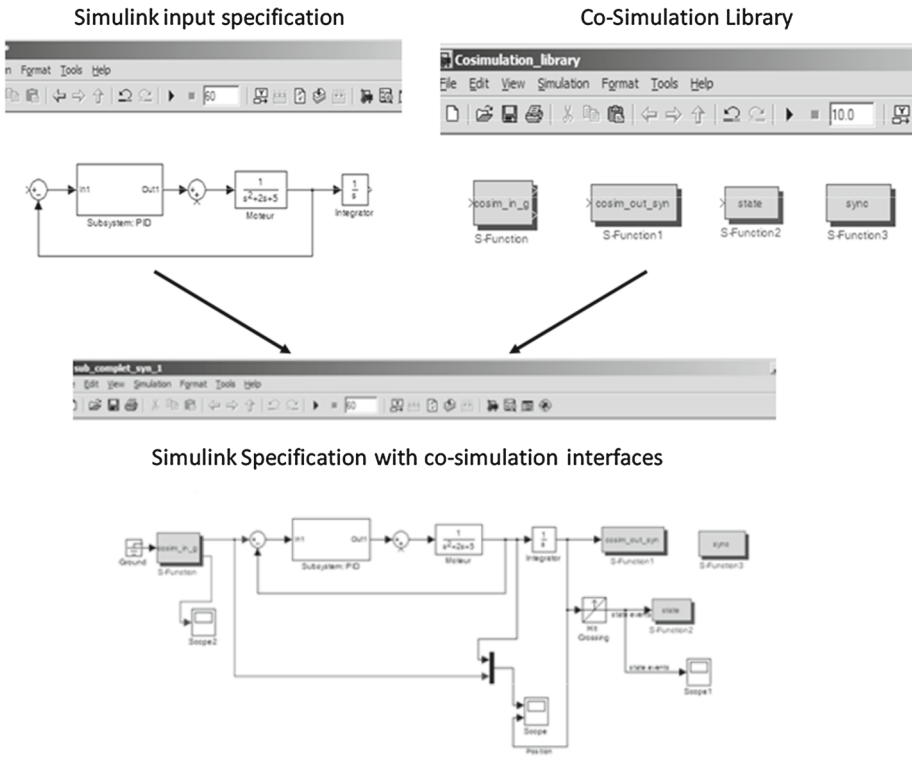

Fig. 17 Extending a Simulink® model with simulation interfaces

event management and the end of the discrete cycle detection (detects that there are no more delta cycles at the current time). The SystemC interfaces are:

- **SC_inter_In** implements the input communication function and ensures synchronization with input data and state events.
- **SC_inter_Out** implements the output communication function and provides synchronization with the sampling events sent by SystemC.

For SystemC, the library blocks are state event management blocks and communication blocks. The interfaces are generated by a script generator that has as input user-defined parameters. The interface parameters are: the names, the number and the data type of the discrete model inputs ports, and the sampling periods. Once the interfaces are generated, their connection is realized within the sc_main function that connects the interfaces to the user model. The model is compiled and the link editor calls the library from SystemC and a static library.

Figure 18 presents a screen capture of the co-simulation; both models are running and the developers can use the different features for debugging.

In the development of CODIS, the steps of the conceptual framework we proposed in this paper are "hidden" in different stages of the co-simulation flow. The "definition of the library elements and the internal architecture of the co-simulation interfaces" step represents the foundation for the generation of the co-simulation library and implicitly for the co-simulation interfaces generation. The "definition of the operational semantics", the "distribution of the

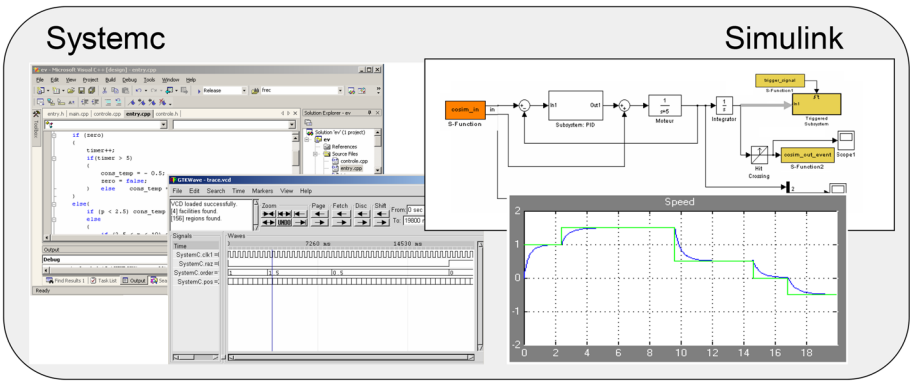

Fig. 18 SystemC/Simulink® co-simulation

synchronization functionality" as well as their behavior play an important role at the output flow with the behavior of the co-simulation interfaces and the synchronization model. This tool was detailed in [27].

6.2 Analysis

In order to have quantitative information regarding the performances of our approach we measured the simulation interfaces overhead. In our case study the functionalities of the continuous and discrete sub-systems were very simple (a few very simple computations). Considering that the communication is the main factor contributing to the co-simulation overhead, the objective was to define an application where the communication between the two subsystems is critical. The Simulink® integration step adjustment when detecting a SystemC event overhead is maximum 5 % of total simulation time. The overhead caused by inter process communication represents about 20 % of the total simulation time. The SystemC synchronization overhead does not exceed 0.2 % of the total simulation time. This cost depends of the number of state events generated by the continuous simulator and the synchronization steps required by an application. The presented overhead was measured in the case where we have a state event generated by the continuous simulator during each discrete step, therefore the synchronization is performed at each step of the discrete simulator. We measured the overhead after 10^6 synchronization points.

The CODIS framework was used to implement a glycemia level regulator that was detailed in [42].

In order to show that the proposed formal framework is generic, after the analysis of the discrete domain simulators, we used SystemVerilog for the discrete domain implementation of the glycemia level regulator while for the continuous domain we used the Simulink® model already tested with SystemC-Simulink® implementation. The SystemVerilog-Simulink® co-simulation replicates the results obtain with the SystemC-Simulink® co-simulation.

7 Discussion

Table 2 presents the advantages and the limitations of our contribution, compared with the existing approaches. In this table, the three pluses (+++) notation refers to a very good

Table 2 Advantages and limitations of the proposed approach

Approach	Requirement				
	Interfaces generation	Modularity/ scalability	Formal verif. of C/D interf.	Validation perform.	Flexibility
Existing simulation-based approaches (e.g. VHDL-AMS, SystemC-AMS...)	+++	++	N/A	+++	++
Existing formal representation—based approach	N/A	+++	+++	+	++
Proposed approach	+++	+++	+++	++	+++

approach; the two pluses (++) notation is used for a good approach while the one plus (+) refers to a fair approache. We can note that the same approach can be very good, good or fair according to the different metrics given in the columns of the table.

Two main existing approaches (discussed also in Sect. 2) are considered:

– The simulation-based approach.
– The different approaches proposing the formal representation of C/D systems.

The following metrics are analyzed:

– The C/D Interface generation metric measures the effort required for the design of C/D interfaces design.
– The Modularity and scalability metric indicates the possibility to enable independent handling of the different components of the C/D systems (ex. addition of new components, validation of different solutions).
– The Formal verification metric refers to the correctness of the C/D interfaces. This correctness can be formally guaranteed or not.
– The Validation performance is directly related to the simulation speed characterizing the global execution models provided by the different approaches.
– The Flexibility metric refers to the ability of the co-simulation models to be adapted to the modifications occurring during the design flow (e.g. environment modification, technology modification).

The existing simulation approaches are very good from the perspective of the simulation interface generation metrics. This is mainly because these approaches do not require any additional effort from the user in order to generate the global simulation model. However, these approaches accept only integration of modules described in a specific language (ex. VHDL-AMS or SystemC-AMS), consequently these approaches are considered good (instead of very good) from the modularity/scalability point of view. This also impacts the flexibility as these approaches accept only certain solvers and their library is less elaborated than the library of well established continuous tools (i.e. Simulink®).

The formal representation-based approaches are powerful in terms of modularity/ scalability and formal verification of the simulation interfaces. However, the validation performance is fair as with the increase of the system's complexity the simulation speed decreases considerably. More so, any change in the system has to be translated in its formal representa-

tion and some changes cannot be expressed formally as only some properties are taken into consideration. Therefore, we consider this having impact on the flexibility of this approach.

Considering the effort required by the C/D interfaces, our approach may be compared with the existing simulation-based approaches. These metrics are not directly related to the formal representation approaches. The main purpose of the approaches using formal representation is to offer a very well defined conceptual framework for interfaces modeling. The users can exploit this in order to formalize the functionalities of the interfaces.

Comparing with the existing simulation-based approaches, our solution improves the scalability and the modularity. This is mainly due to the generic stage offering a formally correct functionality and guiding the implementation of interfaces if simulators need to be added in the design flow.

Unlike the exiting simulation-based approaches, our solution also considers formal verification of C/D interfaces.

As explained in Sect. 6, the integration of several existing simulators implies a cost in validation performance. However, it is important to note that this cost is comparable with the cost already accepted by the designer for the classical Hw/Sw co-simulation. This cost is compensated by the flexibility improvement: the facility to integrate other existing powerful continuous and/or discrete simulators gives the ability to adapt easily to eventual modifications during the design flow (e.g. the technology modification can lead to the necessity to introduce a new simulator in the design flow).

8 Conclusion and future works

This paper proposes a conceptual framework based on formal representation that gives the designers the flexibility and correctness to design C/D co-simulation tools. The main contribution is the representation of the functionality of the co-simulation interfaces, using formal methods for the specification and the validation that is a step in bridging the gap between formalization and the simulation mapping from formal specification to implementation. This methodology was successfully applied for the design of CODIS a co-simulation tool integrating Simulink®, and SystemC simulators.

Our future works concentrate on the formal verification of the composition of the elements of the library in order to create an interface and the analysis of the continuous and discrete models to be integrated in order to verify their compatibility in terms of inputs, outputs and abstraction levels. We are also working on more complex systems where a discrete and multiple continuous systems are taken into consideration.

References

1. ITRS (2010) http://public.itrs.net/
2. Nicolescu G, Sungjoo Y, Bouchhima A, Jerraya AA (2002) Validation in a component-based design flow for multicore SoCs. In: Proceedings of the 15th ISSS (ISSS'02) Kyoto
3. Romitti S, Santoni C, Francois P (1997) A design methodology and a prototyping tool dedicated to adaptive interface generation. In: Proceedings of the 3rd ERCIM workshop on user interfaces for all
4. Cadence http://www.cadence.com
5. Mentor Graphics. Seamless CVE. http://www.mentorg.com/seamless
6. Frey P, O'Riordan D (2000) Verilog-AMS: mixed-signal simulation and cross domain connect modules. In: Proceedings of the BMAS'00, Orlando
7. IEEE Standard VHDL AMS Extensions, IEEE Std 1076.1-1999

8. Marion C, Fanucci L, Iozze F, Forliti M, Rocchi A, Giambastiani A, De Marinis M (2005) VHDL-AMS modelling and system verification flow. Trans Energy Convers 28:189–196
9. Patel DH, Shukla SK (2004) SystemC Kernel-extensions for heterogeneous system modeling. Kluwer Academic Publishers, Dordrecht
10. Vachoux A, Grimm C, Einwich K (2003) Analog and mixed signal modeling with SystemC-AMS. In: Proceedings of the international symposium on circuits and systems
11. Verilog AMS http://www.vhdl.org/verilog-ams/htmlpages/public-docs/lrm/2.3.1/VAMS-LRM-2-3-1.pdf
12. Ptolemy http://ptolemy.eecs.berkeley.edu/
13. Lee EA, Zheng H (2005) Operational semantics of hybrid systems. In: 8th international workshop: computation and control, HSCC, pp 25–53
14. Lee EA (2010) Disciplined heterogeneous modeling. In: Proceedings of the ACM/IEEE 13th international conference on model driven engineering, languages, and systems (MODELS). Springer, New York, pp 273–287
15. Functional Mock-up Intergace https://www.fmi-standard.org/
16. Lee EA, Sangiovanni-Vincentelli A (1996) Comparing models of computation. In: IEEE proceedings of the international conferenceon computer aided design, pp 234–241
17. Jantsch A, Sander I (2005) Models of computation and languages for embedded system design. Kluwer Academic Publishers, Dordrecht
18. Jantsch A (2003) Modeling embedded systems and SoCs: concurrency and time in models of computation (systems on silicon). Morgan Kaufmann Publishers, San Francisco
19. Fitzgerald J, Larsen PG, Verhoef M (eds) (2014) Collaborative design for embedded systems: co-modelling and co-simulation. Springer, Berlin
20. Zeigler BP, Praehofer H, Kim TG (2000) Modeling and simulation - integrating discrete event and continuous complex dynamic systems. Academic Press, San Diego
21. 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
22. Kim YJ, Kim JH, Kim TG (2003) Heterogeneous simulation framework using DEVS-BUS. Simul Soci Model Simul Int 79(1):3–18
23. SH Attarzadeh Niaki SH, Jakobsen MK, Sulonen T, Sander I (2012) Formal heterogeneous system modeling with SystemC. In: Forum on specification and design languages (FDL 2012). Vienna, pp 160–167
24. Herrera F, Villar E, Grimm C, Damm M, Haase J (2008) Heterogeneous specification with HetSC and systemC-AMS: widening the support of MoCs in SystemC. In: Embedded systems specification and design languagtes. Springer, New York
25. Haase J, Damm M, Grimm C, Herrera F, Villar E (2008) Bridging MoCs in SystemC specifications of heterogeneous systems. EURASIP J Embed Syst 2008:371768
26. Ghasemi HR (2005) An effective VHDL-AMS simulation algorithm with event. In: International conference on VLSI design
27. Bouchhima F, Nicolescu G, Aboulhamid EM, Abdi M (2007) Generic discrete-continuous simulation model for accurate validation in heterogeneous systems design. Microelectr J 38:805–815
28. Alur R, Dill D (1990) Automata for modeling real-time systems. In: Proceedings of the 17-th International CALP, vol 443, pp 322–335
29. Bengtsson J, Yi W (1996) Timed automata: semantics, algorithms and tools. Uppsala University, Uppsala
30. Behrmann GF, Behrmann G, David A, Larsen K (2005) A tutorial on UPPAAL. In: Real-time systems symposium, Miami
31. Behrmann G, Fehnker A, Hune T, Larsen K, Pettersson P, Romijn J, Vaandrager F (2001) Minimum-cost reachability for priced timed automata. In: Hybrid systems: computation and control, HSCC 2001
32. Cassandras CG, Lafortune S (2007) Introduction to discrete event systems. Springer, New York
33. Cellier FE (1979) Combined continuous/discrete system simulation languages: usefulness, experiences and future development. In: Proceediings of methodology in systems modelling and simulation conference, Rehovot, pp 201–220
34. Wang F (2004) Formal verification of timed systems: a survey and perspective. In: Proceedings of the IEEE, vol 92, pp 1283–1305
35. Chane F, Giambiasi N, Paillet J-L (2004) From DEVS model to timed automata. In: Proceedings of the international conference on software engineering research and practice, SERP'04
36. Monin J-F (2003) Understanding formal methods. Springer, New York
37. Edwards S, Lavagno L, Lee EA, Sangiovanni-Vincentelli A (1997) Design of embedded systems: formal models, validation, and synthesis. In: Proceedings of the IEEE, vol 85, pp 366–390
38. Claudius Ptolemaeus (2013) System design, modeling, and simulation using Ptolemy II. Ptolemy.org

39. Cardelli L (1984) A semantics of multiple inheritance. Semantics of data types (International Symposium Sophia-Antipolis, June 27–29, 1984). Lecture Notes in Computer Science
40. MathWorks, MATLAB/Simulink. http://www.mathworks.com
41. SystemC LRM http://www.systemc.org
42. Gheorghe L, Bouchhima F, Nicolescu G, Boucheneb H (2008) Semantics for model-based validation of continuous/discrete systems. In: Design automation and test in Europe (DATE'08)