Methodology for Efficient Design of Continuous/Discrete-Events Co-Simulation Tools

G. Nicolescu, H. Boucheneb, L. Gheorghe, F. Bouchhima
Ecole Polytechnique Montréal, Computer Science Department
gabriela.nicolescu@polymtl.ca

Keywords: Co-Simulation, Heterogeneous Systems, Discrete Event Simulation, Continuous Simulation, Simulink, SystemC

Abstract

Continuous and discrete components may be integrated in diverse embedded systems ranging across defense, medical, communication, and automotive applications. The global validation of these systems requires new validation techniques, the main challenge being the definition of global simulation models able to accommodate the different concepts specific to continuous and discrete models. This paper presents a generic methodology for the efficient design of continuous/discrete-events co-simulation tools. Before the implementation stage, the methodology proposes several steps enabling the gradual formal definition of the simulation interfaces functionality and their internal architecture.

1. INTRODUCTION

Systems on chip are becoming complex not only in terms of components density but also in terms of heterogeneity. These systems may be found in various domains like defense, medical, electronic, communication, and automotive. A recent ITRS study concludes that heterogeneity is "a form of *diversity* that arises with respect to system-level SOC integration" and the design specification and validation are extremely challenging, particularly with respect to complex operating contexts. Continuous/discrete systems, with direct applications in mixed-signal and RF domains, are presented as main system drivers [1].

The co-simulation represents currently one of the most popular validation techniques for heterogeneous systems. It allows joint simulation of existing heterogeneous components with different execution models. This technique was successfully applied for hardware/software discrete systems [12], but very few applied it for continuous/discrete systems. The main challenge in the definition of co-simulation tools for continuous/discrete systems is due to the heterogeneity of concepts manipulated by the discrete and the continuous components. In the case of validation tools, several execution semantics have to be taken in consideration in order to perform global simulation. The most important are the following:

- in discrete-events models, *time* represents a global notion for the overall system and advances discretely when passing by time stamps of events, while in continuous models, the time is a global variable involved in data computation and it advances by integration steps that may be variable.

- in discrete-events models, *processes* are sensitive to events while in continuous models processes are executed at each integration step.

Therefore, the global validation of continuous/discrete systems requires simulation interfaces providing synchronization models for the accommodation of the heterogeneous aspects cited above. This implies a complex behavior for the simulation interfaces, their design being time consuming and an important source of error. These interfaces play also an important role in the accuracy and the performance of the global simulation. Consequently, new validation tools able to generate automatically simulation interfaces for the global simulation of continuous/discrete systems are mandatory. The key issue is the rigorous definition of the behavior and the architecture of simulation interfaces [12], [13].

This paper presents a generic methodology for the efficient design of continuous/discrete co-simulation tools. Before the implementation stage, the methodology proposes several steps enabling the gradual formal definition of the simulation interfaces functionality and internal architecture.

2. BACKGROUND

2.1. Global execution model for continuous/discrete systems

A continuous/discrete system and its corresponding global execution model are illustrated in Figure 1. As shown in the figure, three types of basic elements compose the global execution model:

- The execution models of the different components constituting the heterogeneous system;
- The simulation bus;
- The simulation interfaces.

The *simulation bus* is in charge of interpreting the interconnections between the different components of the system.

The *simulation interfaces* enable the communication of different components through the simulation bus. Their role is to adapt each component to the simulation bus. Consequently, they are in charge of:

- Adaptation of continuous and discrete execution models;
- Adaptation of different simulators to the simulation bus – in order to guarantee the transmission of information between simulators executing the different components of the heterogeneous systems.

The simulation interfaces have to provide efficient synchronization models for the adaptation of the enumerated concepts. The synchronization can be generally defined as coordination with respect to time. The interfaces have, by consequence, a very complex behavior. Their automatic generation is very desirable, since their design is time consuming and an important source of errors.

The *simulation backplane* is the element of the global execution model that guarantees the synchronization and the communication between the different components of the system. It is composed of the above mentioned simulation interfaces and the simulation bus.

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

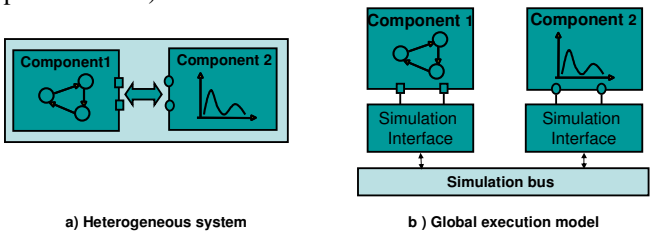

Figure 1. Continuous/discrete heterogeneous systems and its corresponding execution model

2.2. Co-Simulation Tool – Basic Principle

A co-simulation tool is an instrument for the creation of simulation instances. In case of the global validation of continuous/discrete systems, the simulation environment must be able to receive as input a heterogeneous specification and to generate automatically the simulation bus and the simulation interfaces required for the global execution of the system.

This type of environment requires a conceptual foundation defining the different basic elements and the algorithms that are used. The conceptual definition of a simulation environment is based on two main aspects:

- The *definition of the execution model* - the definition of the simulation bus and simulation interfaces;
- The *definition of a methodology* for the automatic generation of simulation instances.

These two aspects are related: the definition of the simulation bus and simulation interfaces may facilitate the automatic generation of simulation instances. These two definitions have direct consequences on the efficiency of the simulation environment.

Figure 2 shows an overview of the flow for automatic generation of execution models. At the entry in the flow we have the continuous and discrete models composing the heterogeneous system. These models are realized using specific tools (i.e. Simulink® [9] or Modelica™ [10] for the continuous models and SystemC, VHDL, for the discrete models). The different elements composing the simulation bus and the simulation interfaces are stored in the simulation library. The flow also includes a block for the automatic generation of the execution models. This block gets at the input the two simulators and the components from the simulation library and generates the execution model by connecting the modules, the simulation interfaces and the simulation bus.

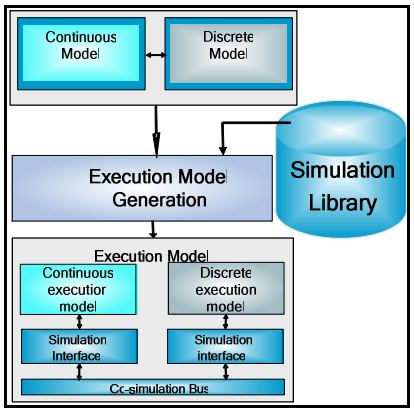

Figure 2: Overview of a flow for automatic generation of execution models

2.3. Requirements for an efficient co-simulation tool

The main characteristics required for the execution models in order to facilitate their automatic generation and to provide the efficient global validation are:

2.3.1. Flexibility

A flexible co-simulation tool is a tool that is able to be adapted to the modifications occurring during the design flow:

- Modifications of the external environment (e.g. the modification of the functional constraints);
- Modifications of the utilization mode (e.g. the modification of the communication protocols);
- Technology modifications (e.g. the replacement of a given simulator by another one).

2.3.2. Modularity and scalability

An efficient co-simulation tool must provide also modularity and scalability. Thus, it must enable the independent handling of the different components of the systems. It must also enable the evolution of the system complexity. The main advantages of using such a tool are:

- The possibility to validate the different solutions for the realization of a given system;
- The possibility to validate a system when new components or new functionalities have to be added;
- The validation of systems and their environment.

2.3.3. Accuracy

An efficient co-simulation tool model must allow designers the choice of the accuracy level. Depending on the accuracy level, two types of validation may be defined:

- Timed validation takes into account the time notion and enables to evaluate the performances and the respect of timing constraints of a system;
- Functional validation enables the validation of the functionality of a system. In this case, only the data transfers are considered without taking into account the time required for their transfer.

3. METHODOLOGY FOR THE DESIGN OF CONTINUOUS/DISCRETE CO-SIMULATION TOOLS - OVERVIEW

This section introduces a methodology for the design of generic continuous/discrete co-simulation tools. To enable the design of flexible, modular, scalable and accurate co-simulation tools, this methodology presents several steps that are independent of the simulation tools used for the continuous and discrete components of the system. During these generic steps, designers work in a conceptual framework, expressing the functionality and the internal structure of simulation interfaces using existing formalisms and temporal logic. After the rigorous definition of the required functionality for simulation interfaces, the designer will start the steps related to the implementation.

The main steps of the proposed methodology are:

1. Definition of the operational semantics for the synchronization in continuous/discrete 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.

These steps will be detailed in the next sections.

4. DEFINING THE OPERATIONAL SEMANTICS FOR SYNCHRONIZATION IN CONTINUOUS/DISCRETE GLOBAL EXECUTION MODELS

The first step of the methodology for co-simulation tools design is the definition of the operational semantic for the synchronization in continuous/discrete global execution models. An operational semantics is a mathematical definition of the communication/synchronization relation between the continuous and the discrete simulators.

In a continuous/discrete heterogeneous system, we find two distinct models:

- A continuous model where the computation is realized in the continuous domain;
- A discrete model where the computation is realized in cycles.

For a rigorous synchronization, each simulator involved in a continuous/discrete co-simulation must consider the events coming from the external world and it must reach accurately the time stamps of these events. We refer to this as *events detection*. These time stamps are synchronization and communication points between the continuous and the discrete simulator. Therefore, the continuous simulator must detect the next discrete event (*time event*) scheduled by the discrete simulator [11]. These events may be clock events, timed notified events, events because of the `wait` function. This detection requires the adjustment of the integration steps for the continuous simulator (see Figure 3).

The discrete simulator must detect the *state events*. A *state event* is an unpredictable event, generated by the continuous simulator, whose time stamp depends on the values of the state variables (ex: a zero-crossing event or a threshold overtaking event). This implies the control of the discrete simulator advancement in time: instead of advancing with a normal simulation step, the simulator has to advance precisely to the time step of the state event (see Figure 3).

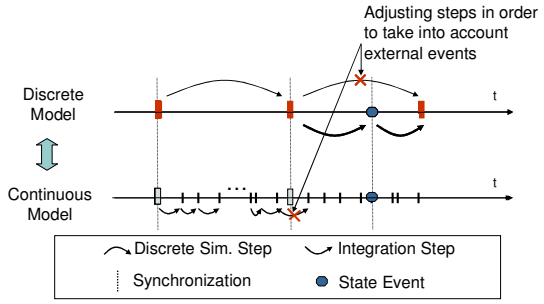

Figure 3 : Continuous/Discrete Synchronization

The operational semantic of the synchronization model in continuous/discrete global execution models may be defined starting from the DEVS (Discrete EVent System Specification) formalism [6] (a model-based formalism) and using the rules of process algebra. The DEVS formalism allows a dynamic representation of extended systems. It provides an abstract simulation mechanism and atomic modules to build complex simulations. It separates the modeling and the simulation. The time advances on a continuous base. The formalism is based on the system theory: we have a system, a time base, inputs, states, outputs, given the current state and the inputs, established functions can be used to determine the next state and the outputs. The complete operational semantic for continuous/discrete synchronization using the DEVS formalism is described in [4].

5. DISTRIBUTING SYNCHRONIZATION FUNCTIONALITY TO SIMULATION INTERFACES

The second step of the methodology for the design of the co-simulation tools consists of the distribution of the synchronization functionality to the simulation interfaces.

The behavior of the discrete domain interface can be described by a few processing steps detailed in Figure 4. The interface is in charge of:

- the *data* exchanged between the simulators (send/receive);
- the *time stamps* of the next events;
- the *state events* consideration;
- the context switch to the continuous interface.

The behavior of the continuous domain interface can also be described by a few processing steps detailed in Figure 5. This interface handles:

- the *data* exchanged between the simulators (send/receive),
- the *time stamps* of the next events;
- the indication (to the discrete interface) of the occurrence of a *state event;*
- the context switch.

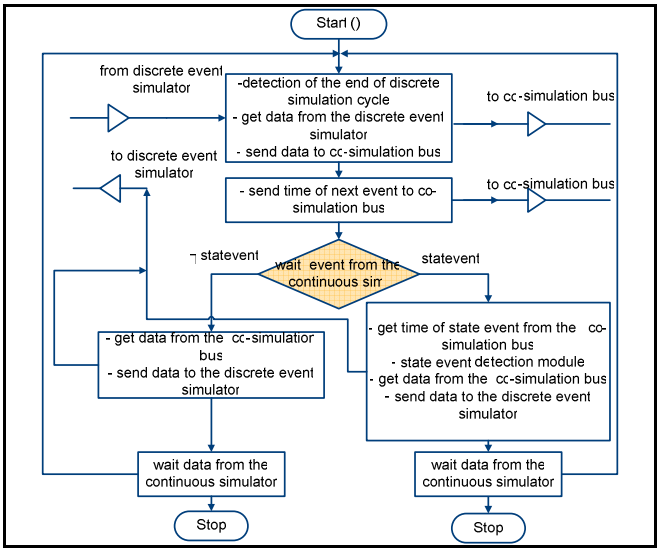

Figure 4: Flowchart for the discrete domain interface

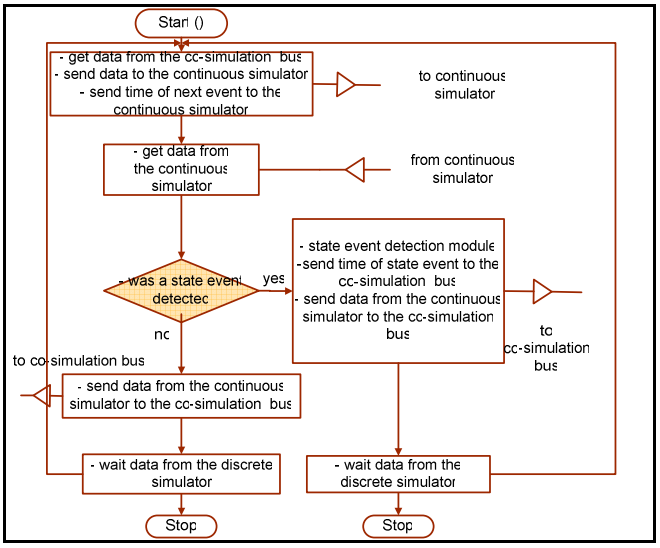

Figure 5: Flowchart for the continuous domain interface

6. FORMALIZATION AND VERIFICATION OF SIMULATION INTERFACES BEHAVIOR

The formalization and verification of the simulation interfaces can be realized based on different formalisms and temporal logics. Temporal logics exploitation is justified by the functionality of the simulation interfaces; their main role is the synchronization, therefore their temporal behavior is a key aspect. Considering that the 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 (ex. the values of the variables characterizing the system) also change. The

 ISBN 1-56555-311-X

temporal logic handles formalization where the properties evolve over the time. In general, the temporal logic uses:
- Propositions that describe the states (i.e. elementary formulae and logical connectors);
- Temporal operators that allow the expression of the properties of the states successions (called executions).

The differences between these logics are in terms of temporal operators and objects on which they are interpreted (such as sequences or state trees) [7].

The most commonly used logics are linear temporal logic (LTL), computation tree logic (CTL* and CTL) (both of them untimed temporal logics) and their timed extensions TCTL and MITL.
- CTL* allows the use of all temporal and branching operators but the properties verification is very complex. For this reason, most of the tools actually used allow the verification of fragments of CTL* only.
- LTL is a fragment of CTL* that excludes the trajectory quantifiers. In this case only the trajectory predicates are considered. LTL does not provide a means for considering the existence of different possible behaviors starting from a given state (sequential) [7].
- CTL is also a fragment of CTL* and it is obtained when every occurrence of a temporal operator is immediately preceded by a branching operator. In the case of CTL we have state trees.
- TCTL is a timed temporal logic that is an extension of CTL obtained by subscribing the modalities with time intervals specifying time restrictions on formulae.

A possible approach to formalize systems behavior is using the DEVS formalism with LTL [6]. An approach using timed automata and TCTL for continuous/discrete systems is presented in [5]. A timed automaton [2] can be seen as classical finite state automata with clock variables and logical formulae on the clock (temporal constraints). In order to validate the formal model we need to define and verify a broad range of properties that characterize the system. The formalism has to allow the verification of correctness, liveness and safety properties. Examples of key properties for continuous/discrete simulation interfaces are the deadlock, the correct answer to a change in the behavior of one of the simulators as well as the synchronization between the two interfaces (correctness of events detection).

Model checking is a successful technique for verifying automatically temporal properties of concurrent systems. In general, model checkers work by checking if a given property holds for every reachable state, and produce counterexamples in case the property does not hold. To attenuate the state explosion problem, which is inherent to the model checking technique, the model checkers use, in general, a symbolic method possibly combined with an on-the-fly method. The symbolic methods aim at representing very large sets (sometimes infinite sets) of states concisely and manipulating them, as inseparable blocks of states. In the on-the-fly methods, the property is checked while the graph of reachable states (or blocks of states) is being generated. The generation of reachable states is stopped as soon as the truth value of the property is determined.

7. DEFINING THE LIBRARY ELEMENTS OF THE INTERNAL ARCHITECTURE OF SIMULATION INTERFACES FOR THE CO-SIMULATION FRAMEWORK LIBRARY

After the formalization of the simulation interfaces behavior, it is important to define their internal architecture and the library elements for the co-simulation framework. We give in Figure 6 the hierarchical representation of the global 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 continuous/discrete simulation interface required for the global simulation.

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

The bottom hierarchical level includes the atomic modules of the domain specific simulation interface. These atomic components implement basic functionalities of the synchronization model and represent the elements that form the co-simulation library.

For the continuous domain, the simulation interface is composed of the following modules:
- a module for the State Event indication and Time Sending (SETS);
- a module for Detecting Events from the Discrete simulator (DED);
- a module for the Context Switch (CS);
- a module for the Signal Conversion and Data Exchange (SCDE).

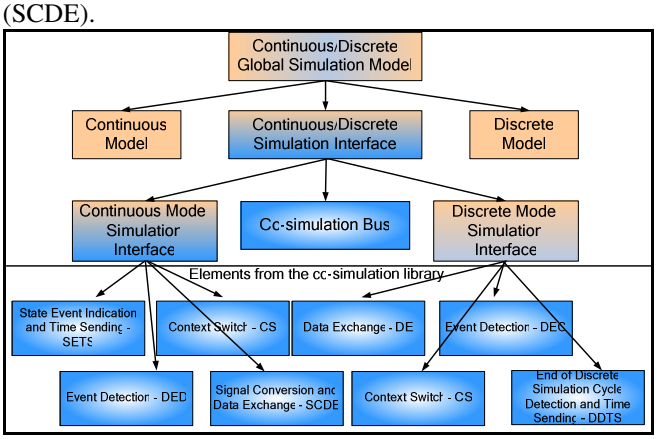

Figure 6 : Hierarchical representation of the generic architecture of co-simulation model

The discrete domain simulation interface is composed of the following modules:
- a module for the Detection of the end of Discrete Simulation cycle and for the Time Sending (DDTS);
- a module for Detecting Events from the Continuous simulator (DEC);
- a module for the Context Switch (CS);
- a module for the Data Exchange (DE).

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

8. SIMULATION TOOLS ANALYSIS FOR THEIR INTEGRATION IN CO-SIMULATION FRAMEWORK

The previous steps that describe the gradual formal definition of the simulation interfaces and the required library elements are independent on 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:

- Detection of state events;
- Setting break points during differential equation solving;
- On-line update of the breakpoints settings;
- 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).

Simulink is an illustrative example of continuous simulator enabling the presented control functionalities. Moreover, these functionalities can be added in generic library blocks and a given Simulink model may be prepared for the co-simulation by parameterization and addition of these blocks. Some details on the implementation of Simulink blocks will be given in Section 9.

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

- Detection of the end of the simulation cycle
- Insertion of new events (*state events*) in the scheduler's queue. This must be done before the advancement of the simulator time
- 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 these characteristics. SystemC 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 the co-simulation. In this way the co-simulation overhead may be minimized. However, the addition of simulation interfaces is more difficult than in Simulink because the specifications in SystemC are textual and a code generator is required in order to facilitate the addition of simulation interfaces. More details on the implementation of SystemC library elements are given in Section 9.

9. IMPLEMENTING LIBRARY ELEMENTS SPECIFIC TO DIFFERENT SIMULATION TOOLS (ILLUSTRATION FOR SIMULINK AND SYSTEMC)

The last step of the methodology for the design of co-simulation tools for continuous/discrete systems is the implementation of the library elements that are specific to different simulation tools. We present here the implementation for the validation of continuous/discrete systems where we used SystemC [8] for the discrete simulation models and Simulink [9] for the continuous simulation models.

9.1. State Event Detection in SystemC

The modules responsible for solving the *state event detection* are (as represented in Figure 6): the "State Event indication and Time Sending" (SETS) module for the continuous domain and the "Event Detection" (DEC) module in the discrete domain interface.

For the detection of a state event by the discrete simulator, a pure event (without value) has to be introduced in the SystemC queue. The time stamp of this event is equal to the time of the state event occurrence. The event insertion must be made before the SystemC time advancing, otherwise SystemC has to roll-back to take it into account. Since the scheduler passes the processor to the user processes only after the time advancement, this insertion required the modification of SystemC scheduler. This modification consists in the creation of a set of events which can be notified by the scheduler in the case of state events presence.

Figure 7 illustrates with an example where *My_Method* is sensitive to the `et_mat0` *state event*. In case of a state event, Simulink indicates its presence to the SystemC scheduler, which notifies the event (here `et_mat0`), associated by the user to the state event, with a time equal to the time stamps sent by Simulink.

```
#define et_mat0
      sc_get_curr_simcontext()->et_mat[0]
/* this definition is in the file defining
environment     variables     added     for
heterogeneous simulation */

My_module.h
/* ex. of module header file */
...
SC_CTOR(My_Module)
{  //creation of et_mat0 event
   //associated with state event
   et_mat0 = new sc_event;
   //make My_Method sensitive to et_mat0,
   //as consequence of state event
   SC_METHOD(My_Method);
   sensitive(et_mat0); ... }
```

Figure 7 : Sensitivity at state events in SystemC models

9.2. Detection of Events coming from discrete simulator in Simulink

The *detection of the next event* in the discrete simulator by the continuous simulator is realized in the module "Event Detection" (DED) that is part of the continuous interface. The Simulink simulator must step ahead until the detection of next SystemC discrete event (without going beyond). As a consequence, Simulink has to adjust its integration steps to satisfy the criteria of resolution (precision, continuity and stability) and to detect this event. Since Simulink does not provide possibility to control the integration steps (variable steps in this case) directly, it is difficult to guarantee the accuracy for the detection of the next event of the discrete simulator.

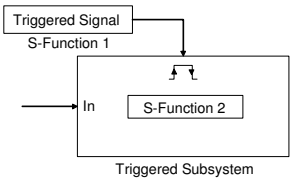

Figure 8 : Implementation for detection events from the discrete simulation model

To cope with this difficulty, we used the "`triggered subsystem`" component from the Simulink library and an S-function. Its role is to trigger via an activation signal the execution of the next layers of the simulation model. When the value of this signal passes from zero to a positive value (or the opposite case) the execution of the next sub-layers is started. The generation of this signal is assured by an S-Function called *triggered signal* (Figure 8). The time mode used in this S-function allows us to choose its next time execution equal to the next discrete time of SystemC. In this case, Simulink adjusts the integration steps to satisfy the criteria of resolution and to detect accurately the execution time of the *triggered signal* (also the time stamp of next event for SystemC). This one will be executed to generate the activation signal for the next layers of the simulation interface.

9.3. SystemC - End of Discrete Simulation Cycle Detection

To guarantee that SystemC sends data and transfers the simulation control to Simulink only after the stabilization of discrete signals, the *detection of the end of discrete cycle simulation* is necessary. This functionality is provided by the "End of Discrete Simulation Cycle Detection and Time Sending" (DDTS) module. The scheduler in SystemC was modified in order to provide such mechanism, to detect the simulation cycle end and switch context to Simulink. This functionality was added to the `simulate ()` function in the `sc_simcontext` class of SystemC scheduler.

The functions `wait(t)` and `notify(t)` can be used in the SystemC models integrated into our model. They may imply new synchronization points, that are known by Simulink thanks to the time sending mechanism added to the scheduler. These modifications remain transparent for the user and do not change the semantics of SystemC.

Figure 9 shows an example of simulation interfaces utilization.

9.4. Organization of Simulation Libraries for SystemC and Simulink

For Simulink, the 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 Simulink interfaces are:

- Sim_inter_In provides the input communication function and synchronization with signals update events.

- Sim_inter_Out 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).

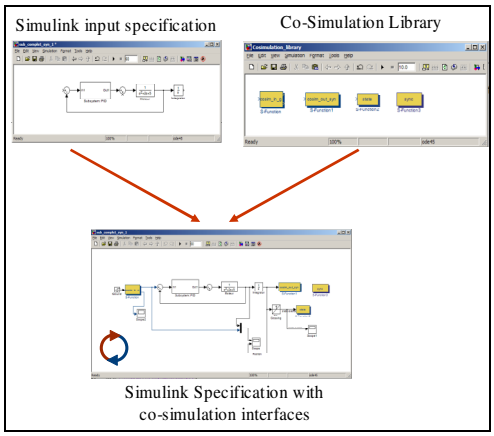

Figure 9 : Extending a given Simulink model with simulation interfaces

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 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 the generation of the co-simulation interfaces for SystemC, the implementation of a script generator was necessary. This script has as input user-defined parameters such as: sampling periods, number and type of ports, synchronization ports. The interfaces are automatically generated by a script generator that has as input the above mentioned user defined parameters. Once the interfaces are generated, their connection is realized within the `sc_main` function.

10. CONCLUSION

This paper proposed a generic methodology for the design of efficient continuous/discrete co-simulation tools. The methodology presents two main stages: (1) a generic stage, defining simulation interface functionality in a conceptual framework and (2) a stage providing implementation for the rigorously defined functionality.

References

[1] International Technology Roadmap for Semiconductor Design (2005), available at http://public.itrs.net/

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

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

[4] Gheorghe, L., et al. "*Formal definitions of simulation interfaces in a continuous/discrete co-simulation tool*", Rapid Systems prototyping, 2006

[5] Gheorghe, L., et al. "*Formalizing Global Simulation Models for Continuous/Discrete Systems*", submitted to the 40-th Annual Simulation Symposium, 2007.

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

[7] Monin, Jean-Francois: "*Understanding Formal Methods*". Springer-Verlag, 2003.

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

[9] Matlab-Simulink ®, www.mathworks.com

[10] Modelica - A unified object-oriented language for physical systems modeling, specifications report, September 1997, version 1.0, www.modelica.org

[11] Cellier, F.E., *Combined Continuous/Discrete System Simulation Languages - Usefulness, Experiences and Future Developmen*t, chapter in Methodology in Systems Modelling and Simulation (B.P. Zeigler et al.), Netherlands, 1979.

[12] Yoo, S, Jerraya, A.A. "Hardware/Software Co-simulation from Interface Perspective", Computers and Digital Techniques, IEE Proceedings-Volume 152, Issue 3, 2005

[13] Jantsch, A.: *"Modeling Embedded Systems and SOC's Concurrency and time in Models of Computation"*. Morgan Kaufmann Publishers. San Francisco, 2004