

A SystemC/Simulink Co-Simulation Framework for Continuous/Discrete-Events Simulation

F. Bouchhima^{1,2}

M. Brière¹

G. Nicolescu¹

M. Abid²

E. M. Aboulhamid³

¹Ecole Polytechnique, Montreal, Canada; ²National Engineer School of Sfax, Sfax, Tunisia

³Université de Montréal, Montreal, Canada

Email: faouzi.bouchhima@polymtl.ca

ABSTRACT

The increasing complexity of continuous/discrete systems makes their simulation and validation a demanding task for the design of heterogeneous systems. The global validation of these systems requires new techniques offering high abstraction levels and simulation accuracy from a time point of view. The main challenge is the time synchronization and the accommodation of different concepts specific to continuous and discrete models.

This paper proposes a co-simulation approach that relies on Simulink for the continuous simulation and SystemC for the discrete simulation. It is based on more than one synchronization model. The synchronization and the communication are assured by co-simulation interfaces. The article also introduces the CODIS tool for the automatic generation of co-simulation instances composed of models and co-simulation interfaces. Experimental results are presented for an illustrative discrete/continuous application.

1. INTRODUCTION

Modern systems like mixed-signal systems and real-time controllers integrate discrete and continuous components. Despite the efforts made by Electronic Design Automation (EDA) industry and academia, research in the continuous/discrete simulation area is still behind its discrete counterpart. Some difficulties in the definition of new CAD tools for continuous/discrete systems are (1) the heterogeneity of concepts manipulated by the discrete and the continuous components [1] and (2) the need of continuous/discrete communication and synchronization.

The most common approach for the mixed-signal design is to extend classical HDLs, originally designed to model discrete systems, to the continuous domain. Illustrative examples are SystemC-AMS [2] and the more popular VHDL-AMS [3] and Verilog-AMS [4]. Therefore, the developers will have to drop well established and powerful languages/tools used for the continuous domain (like Matlab/Simulink [5] or Modelica [6]) with all the existing expertise, models and libraries. Although VHDL-AMS and Verilog-AMS are certainly very useful to support mixed-

signal design, they are not efficient enough at high level of abstraction. The simulation of these languages generally uses a signal solver to resolve differential equations of the analog part, based on the Newton-Raphson algorithm, unsuitable for nonlinear systems resolution.

Another approach is the co-simulation. Although it decreases the simulation speed, it allows the designer to describe each, the continuous and the discrete model, in a specific and appropriate language (such as SystemC [7] or VHDL for the discrete model and Simulink or Spice for the continuous model).

This work uses SystemC and Matlab/Simulink language. They are popular and widely known by the modeling and simulation community. Simulink offers several libraries in automotive, power electronics, etc. and seven solvers designed for *stiff* (appeared in nonlinear systems) and *nonstiff* problems, which can provide an excellent accuracy. SystemC is a standardized modeling language intended to enable system level design and intellectual property integration at multiple abstraction levels, for systems containing both software and hardware components. However, the use of different languages and simulators makes the communication and the synchronization more difficult. This problem is eased by using co-simulation interfaces. These interfaces have a great influence on the accuracy and the performance of the global simulation. Their automatic generation is very important, since their design is time consuming and an important source of errors.

This paper proposes a new co-simulation approach for continuous/discrete systems modeling and simulation. The synchronization is based on the canonical synchronization algorithm [8]. This model was adapted for the co-simulation and it was enhanced to minimize the interaction between both simulators. The synchronization and the communication are assured by co-simulation interfaces. We show here how to control these interfaces execution order and their interaction with both models. The article also presents the CODIS tool for the automatic generation of co-simulation instances. The global simulation architecture and the solutions for the interfaces implementation have already been detailed in [1].

2. RELATED WORK

In [9], the authors propose a co-simulation environment based on Xyce (SPICE simulator) and SAVANT (parallel VHDL simulator). They focus only on signal conversion and data exchange between simulators. In [10], a simulation framework based on VHDL and ELDO is proposed. For the synchronization, the authors used the lock-step technique. Nexus-PDK [11] supports co-simulation of cycle accurate C/C++ and Handel-C models with SystemC, MATLAB/Simulink, and VHDL/Verilog simulators. All the integrated models are discrete.

Modelica is a language for the design of heterogenous systems. It provides a set of libraries for several application domains. However, the concept of discrete events is difficult to manipulate in this language. In [12] the authors presented a mixed-signal simulation framework to simulate an analog to digital data converter. The framework includes C++ mixed-signal modules. They implement a virtual clock for the scheduling of the analog blocks to avoid multiple executions of them due to the SystemC scheduler. In [13] a framework which supports signal processing-dominated applications is proposed. The synchronization between the synchronous dataflow and linear continuous time is using fixed time step.

In summary, a number of attempts were proposed for mixed-signal simulation. They require the abandon of well established efficient tools for specific domains. Moreover, most of the proposed approaches are application-specific extensions [9] [11] [13].

Comparing with the presented related work, our contributions are (1) proposing a co-simulation approach integrating powerful simulators for continuous and discrete domains (2) adapt the canonical synchronization model for an accurate co-simulation and enhanced it to minimize interactions between simulators and (3) implementing generic simulation interfaces that can be automatically generated using the new CODIS tool.

3. SYNCHRONIZATION IN THE CO-SIMULATION FRAMEWORK

One of the most important difficulties in continuous/discrete simulation is the time synchronization between the event-driven discrete simulation and the numerical integration in the continuous solver. Synchronization is a key issue that influences the accuracy and the simulation speed.

The events exchanged between the discrete and the continuous models composing a continuous/discrete system are:

- The continuous model may send a *state event*. It is an unpredictable and pure event, whose time stamp depends on its state variables (e.g. a zero-crossing event, a threshold overtaking event, etc.).

- The discrete model sends events that we classify in two types of events: (1) the *signals update events* caused by the change of its output discrete signals and (2) the *sampling events* which are pure events sent to the continuous model to indicate the sampling instants of its output continuous signals.

We highlight that for signal update events, the data path is from the discrete model to the continuous model and for the sampling events it is from the continuous model to the discrete model.

Given the nature of the discrete model, we define here three synchronization modes: *the full synchronization mode*, *the predictable events mode* and *the events-driven mode*. The synchronization models are generic, so in the rest of this section we will not specify a particular simulator.

3.1. Generic synchronization model

Figure 1 gives an overview of the first synchronization model, which respects the canonical algorithm [8]. In this algorithm, the continuous simulator (CS) must shake hands with the discrete simulator (DS) at each discrete event time and the discrete simulator (DS) must take into account the state events generated by the continuous simulator. In the following, we explain the adaptation of this algorithm for the co-simulation.

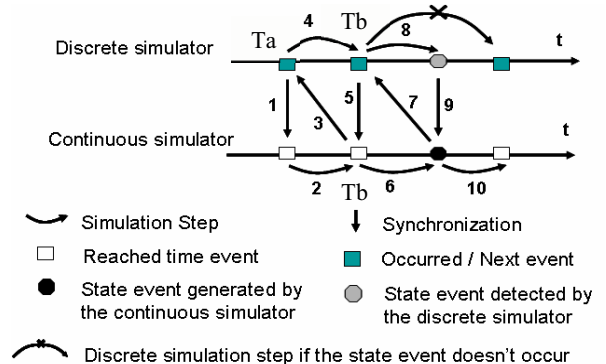


Figure 1 Generic continuous/discrete synchronization model

Assuming that the DS and the CS are synchronized at time T_a , the DS executes all processes sensitive to the current notified events (with zero time) and updates signals (with zero time). Then it sends to the CS the time stamp of its next output event (T_b , next event) and before advancing its time, it switches the simulation context toward the CS (arrow 1). The latter computes signals by resolving system's equations (at each integration step) until it reaches with accuracy (without going beyond) the time sent by the DS (T_b , reached time event). Two cases are possible:

1. T_b is a sample event time stamp. In this case the CS updates the signals with the values calculated at this time and switches the simulation context to the DS (arrow 3). The DS will advance to the sample event time stamp (arrow 4) and restarts the cycle.
2. T_b is a signal update event time stamp. In this case, the CS switches the simulation context to the DS (which

will advance to the indicated event time stamp), computes signals and sends their values and the next event time stamp. Finally, it switches the context to the CS. The last one proceeds to this time with the new signals values, and the cycle restarts (arrows 5, 6).

The continuous model may generate a state event. In this case, the CS indicates its presence, sends its time stamp to the DS and switches the simulation context (arrow 7). The DS has to be able to consider this event by advancing the local time to its time stamp and to execute the processes that are sensitive to it (since it is an external event).

3.1.1. Full Synchronization (FS) mode

In section 3.1, the DS must provide, at each mixed simulation cycle, its next event time stamp. Sampling events occur each sampling period, so their time stamps are usually known. The only difficulty is with signals update events which can be unknown in advance. However, the next discrete time is always known. In this case, if the next time is a multiple of a sampling period then the DS sends it as a sampling event time stamp else it sends it as a signals update event time stamp. Thus, if the next time is a sample event and a signals update event time stamp (the discrete model consumes and produces data at the same time), then it will be sent to the CS only as a sample event time stamp. This situation can arise (1) if the sample event and the signals update event are caused by parallel processes or (2) due to the instantaneous execution of discrete events processes. The second case is to exclude because discrete processes must create computational delay between input and output data. Thus, from a modeling point of view, only the first case is accepted. The solution consists in checking, at each sampling event time stamp, if the inputs of the continuous model are modified. If it is the case, these signals are updated with the new values after a delay equal to the next integration step of the continuous solver. This requires the control of this next step which should be smaller than the next discrete step and sufficiently small to preserve the timing of the global model.

This model creates overhead: the synchronization is performed at each step of the discrete simulator, but it is necessary only when this step coincides with a signals update event time stamp. However, the key point of this synchronization model is that it allows an accurate consideration of the state event without any need of rollback. With this model we define the *Full synchronization (FS)* mode.

3.1.2. Predictable Events (PE) mode

An important optimization can be made to minimize simulators interaction. This is possible when the signals update events are periodic events, which is the case for most RTL designs. Within this mode, the time stamps of these events and sampling events are placed and sorted in a special queue. In order to know the time stamp of the next output event, the queue is consulted to take out the smallest

time. This time is verified if it is a sampling or a signals update event time stamp and sent to the CS. With this model we define the *Predictable events (PE)* mode.

3.2. Events-Driven synchronization model

The synchronization model is given by Figure 2. It is the most adequate synchronization model if the continuous model never generates a state event, since it eliminates unnecessary synchronization between simulators. It consists in running the DS in advance until the discrete model generates an event (sampling or signals update event), then it sends this event (with its time stamp) to the CS and switches the simulation context. As shown by Figure 2, if the continuous model generates a state event then the DS must backtrack. With this model we define the *events-driven (ED)* mode.

We implemented this model with state events consideration for control systems, where the discrete model represents a software component specified at the IA (Instruction Accurate) abstraction level. The states events model external interruptions. For the discrete model, the output signals computation takes several and unpredictable number of discrete steps. Thus, using the FS mode creates an important unnecessary synchronization points and the PE mode cannot be used since the signals update events cannot be known in advance.

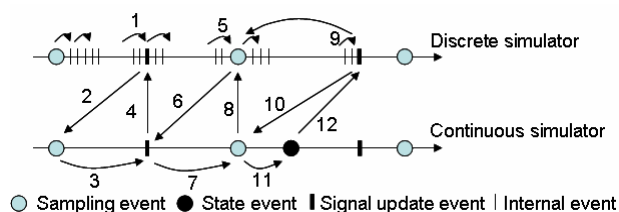


Figure 2. Synchronization model for events-driven mode

In order to roll back time (see Figure 2), the checkpoints-based technique [14] gives an effective solution allowing light-rollbacks asking for reduced memory resources which are in our case the data segment in the memory, processor registers, and I/O signals values.

4. THE SIMULATION INTERFACES IN THE CO-SIMULATION FRAMEWORK

Figure 3 shows the continuous and the discrete models with the simulation interfaces required for the mixed simulation.

For *SystemC*, the interfaces are implemented as SystemC modules. They are classified in:

SC_inter_In implements the input communication function and ensures synchronization with input data and state events. It can be viewed as a sampler circuit and can be auto clocked or have an external clock supplied by the discrete model. The interface has two types of signals:

- Data signals of type `sc_signal` or `sc_fifo`. If the discrete model input ports are bits vectors then the interface add functionality converting double data to bit vector data.
- State events signals, which have a boolean type (bit). Each time the CS sends a state event, the corresponding state event signal is set to "1".

`s` : number of state events signals
`n` : number of continuous model data output signals
`m` : number of discrete model output signals

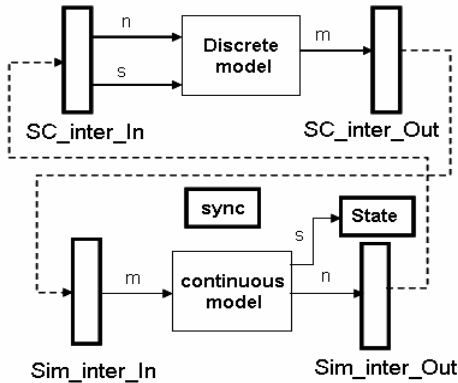


Figure 3. Continuous and Discrete models integrated by means of simulation interfaces

SC_inter_Out implements the output communication function and ensures synchronization with output data. If the discrete model output ports are bits vectors then the interface add functionality that converts bits vector data to double data.

For *Simulink*, the interfaces are functional blocks programmed in C++ using S-Functions. They are classified into four types:

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 type 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 type implements the synchronization function that creates break points which must be reached accurately by a solver (a variable step solver). These points are the time stamps of the received events. When an event is received, this interface makes its next activation time equal to this event time stamp. Once this time stamp is reached, the `Sim_inter_In` or the `Sim_inter_Out` are executed to synchronize with the event and switches the context (see Figure 5). After resuming execution the interface `Sync` is executed to set its next activation time equal to the new received event. The interface is executed at $t = 0$ to fix its first activation time.

4.1. The execution order and interaction between the Simulink model and its simulation interfaces

The Sync interface must be the last in the static execution order list. This is usually possible by assigning to it the lowest priority relative to other model blocks. The others interfaces have priority affected by Simulink with respect to the data dependency rule. For better explanation, we give in Figure 4 an example of Simulink model.

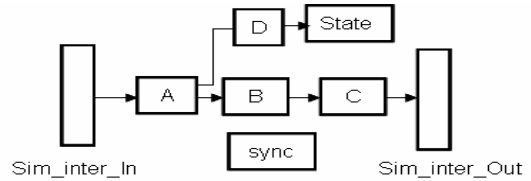


Figure 4 Example of Simulink model

The execution order may be `Sim_inter_In`, `A`, `D`, `State`, `B`, `C`, `Sim_inter_Out` and `Sync`. At each integration step these blocks are executed in the indicated order. The execution of an interface involves the execution of its graph given by the

Figure 5. In this figure, the execution of SystemC sets the Next break time equal to the next discrete time or to the next event time stamp.

CSw: Context Switch; SC: SystemC; SE: sampling event
 SUE: signals update event; Now = current time

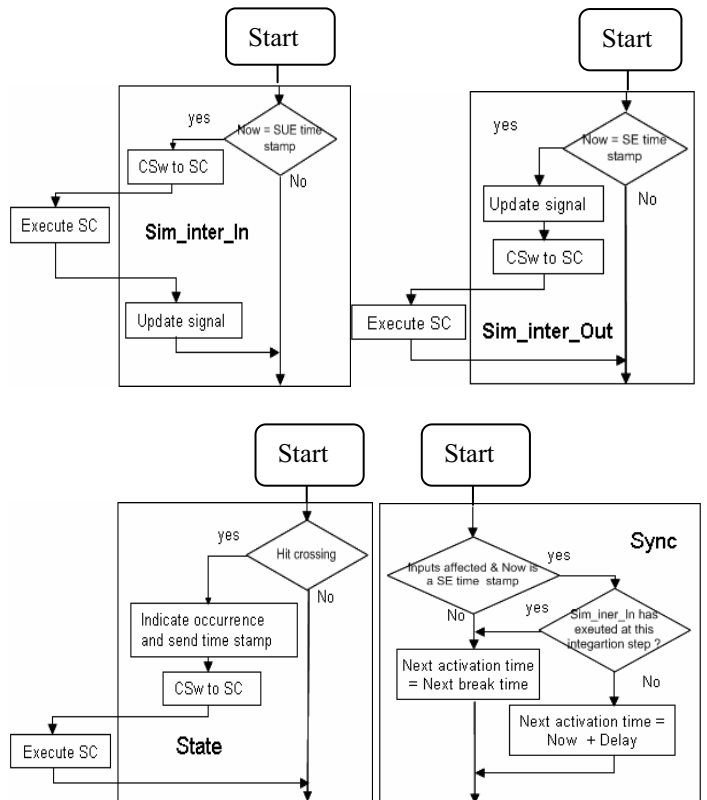


Figure 5 Simulink interfaces

4.2. The execution and interaction between the interfaces and SystemC model

Figure 6 illustrates the SystemC scheduler with the added synchronization functionalities given by the co-simulation interfaces. These functionalities are located in steps 6 and 7.

1. *Initialization Phase* – Execute all processes (except SC_THREADS) in an unspecified order.
2. *Evaluate Phase* – Select a process that is ready to run and resume its execution.
3. If there are still processes ready to run, go to step 2.
4. *Update Phase* – Execute any pending calls to update() resulting from request_update() calls made in step 2.
5. If there are pending delayed notifications, determine which processes are ready to run due to the delayed notifications and go to step 2.
6. If Mode = FS then Send the next discrete time to Simulink and Switch context to Simulink, Else
 If Mode = PE then send the next signals update event or sampling event time stamp and Switch context to Simulink.
 Else (mode = ED) if signals update event flag = "1" or sampling event then send the current time and Switch context to Simulink
7. If state event then add a timed event with time stamp equal to the state event time stamp to the scheduler queue.
8. If there are no more timed notifications, simulation is finished.

Figure 6 Extending the SystemC scheduler with co-simulation interfaces functionalities

5. THE CODIS TOOL

CODIS (COntinuous DIcrete Simulation) is a tool which can automatically produces co-simulation instances for continuous/discrete systems simulation using SystemC and Simulink simulators. This is done by generating and providing co-simulation interfaces and the co-simulation bus. Figure 7 gives an overview of the automatic generation flow of the co-simulation instances. The inputs in the flow are the continuous model in Simulink and the discrete model in SystemC which are respectively schematic and textual. The output of the flow is the co-simulation instance.

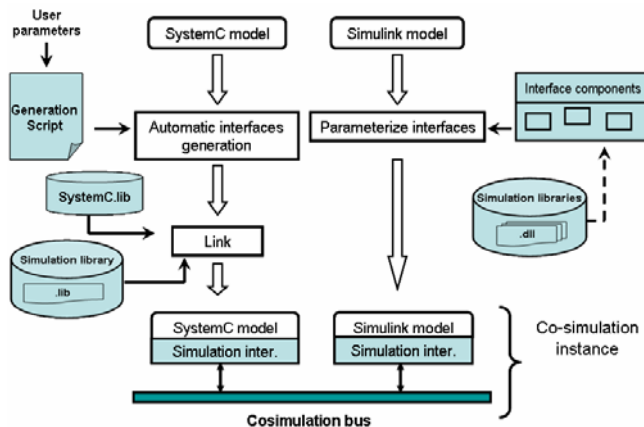


Figure 7 Automatic generation flow of the co-simulation instances

Simulink interfaces can be parameterized starting with their dialog box. The parameters of Sim_inter_In and Sim_inter_Out interfaces are the number of input and respectively output ports. State interface has as parameter the state event number. These interfaces 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. Before the simulation, the functionalities of these blocks are loaded by Simulink from the .dll libraries (Figure 7).

For SystemC, the SC_inter_In parameters are: (1) the names, the number and the data type of the discrete model inputs ports, (2) the sampling periods and (3) the used mode. The SC_inter_Out parameters are: (1) the names, the number and the data type of the discrete model outputs ports and (2) the used mode. The interfaces are automatically generated by a script generator that has as input the user defined parameters. The tool generates also the function sc_main (or modifies the existing sc_main) which connects the interfaces to the user model. The model is compiled and the link editor calls the library from SystemC and a static library simulation library (Figure 7).

7. EXPERIMENTAL RESULTS

The application example consists in a bottles filling system.

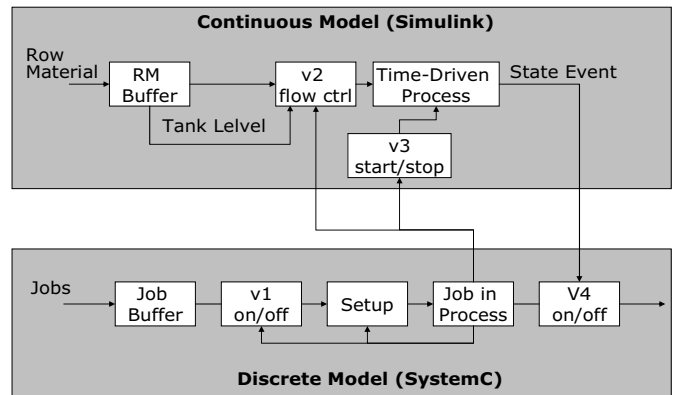


Figure 8. The bottle filling model

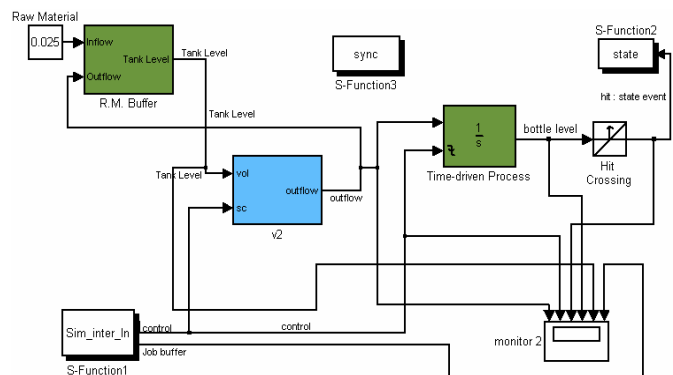


Figure 9. The continuous model overview

Figure 8 illustrates the model of the system that should respect the following specification. The jobs (bottles) are

externally generated where the generation process may be random. These jobs are queued in a “job buffer” (a fifo). If there is no job in “setup” or in “job in process”, the valve v1 is ON and the first job in the “job buffer” proceeds to “setup”. The “setup” process represents the time delay that the system needs to position the bottle at the right spot. After “setup” is completed, the job is placed in “job in process”. When this happens, v2 and v3 are notified so that the actual physical process that defines the job can start. The “time-driven process”, in the continuous model, consists in filling an initially empty bottle with the Raw Material (RM) fluid to a given level (10 lit). It is activated by valve v3 as soon as a job is ready to start. The valve v2 acts as a controller of the bottle filling flow in the “time-driven process”, which can have three values: 0 if there is no job in the “job in process”, the “RM buffer” inflow (0.025 lit/sec) if the last one is empty and 0.033 lit/sec if not. When “time-driven process” is completed (the bottle level reaches 10 lit), it sends a state event to valve v4 to open and let the current job to leave. At this time, “the job in process” opens v1 to accept the next job and after a small delay (the time to react to the state event), it signals v2 to shut off the RM flow and v3 to reset the “time-driven process”. For this application, the “setup” time is set to 60 seconds and the initial “RM buffer” level is set to 3 lit. We give in Figure 9 an overview of the continuous model with the co-simulation interfaces.

7.1. Simulation results

The simulation was performed using the FS synchronization mode, since the continuous model generates state events and the signals update events are not periodic.

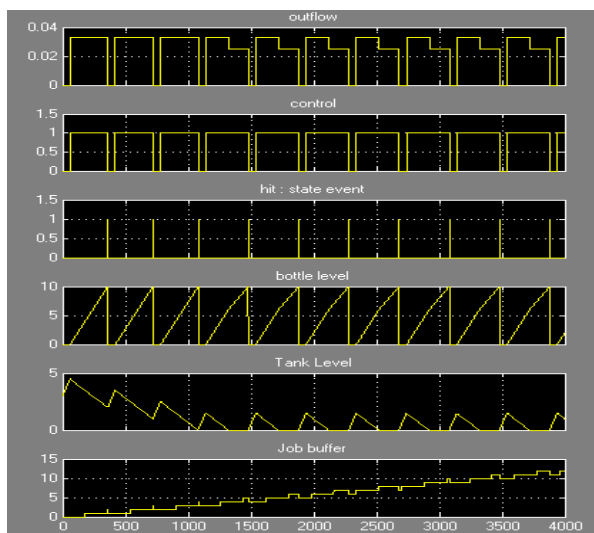


Figure 10. Simulation results

For the results given by Figure 10, the job arrival rate is set to 180 s. In this figure the outflow represents the bottle filling flow. The filling process starts with a 0.033 lit/sec. Each time the tank level is equal to zero, the outflow is switched to 0.025 lit/sec. The hit signal represents the state

events generated when a bottle level reaches 10 lit. The discrete model reacts to these events by switching the control signal to zero. Once a job is presented in “job in process”, the discrete model switches its value to “1 in order to open v2 and v3. The signal ‘Job buffer’, from the discrete model, represents the number of accumulated job in the “job buffer”. This signal is sent to Simulink just for viewing purpose. We remark, in Figure 10, that the number of accumulated jobs is increasing. Our experimentations shows that if jobs are accumulated in the “job buffer” (more than one job) then we risk usually to exceed the buffer capacity. To avoid this situation the job arrival rate must be greater or equal to 400 s. This value depends on the outflow, the inflow and the “setup” time.

8. CONCLUSIONS

This paper proposed a co-simulation framework for continuous/discrete systems, based on generic interfaces. The simulation environments integrated currently in this framework are SystemC (for the discrete parts) and Simulink (for the continuous parts). The framework implements several synchronization modes resulted from a deep analysis of synchronization issues with respect with accuracy and performance constraints of continuous/discrete systems simulation. To evaluate the proposed framework, co-simulation results for a discrete/continuous application were illustrated.

References

- [1] F. Bouchhima et al. “Discrete–Continuous Simulation Model for Accurate Validation in Component-Based Heterogeneous SoC Design”, Rapid System Prototyping 2005.
- [2] A. Vachoux, et al., “Analog and mixed signal modeling with SystemC”, Circuits and Systems, ISCAS’03.
- [3] IEEE Standard VHDL Analog and Mixed-Signal Extensions, IEEE Std 1076.1-1999, 23 Dec. 1999
- [4] P. Frey et al. “Verilog-AMS: Mixed-signal simulation and cross domain connect modules”, Behavioral Modeling and Simulation Workshop, 2000.
- [5] Matlab-Simulink, www.mathworks.com
- [6] Modelica , www.modelica.org
- [7] SystemC LRM, 2003, available at www.SystemC.org
- [8] H.R Ghasemi, “An effective VHDL-AMS simulation algorithm with event”, International Conference on VLSI Design, 2005.
- [9] D. E. Martin, et al., “Integrating multiple parallel simulation engines for mixed-technology parallel simulation”, Simulation Symposium, 2002
- [10] El Tahaway *et al.*, “VHD_cLDO: A new mixed mode simulation”, DAC Conference, 1993.
- [11] Celoxica, <http://www.celoxica.com/methodology/>
- [12] Bonnerud T.E. *et al.* “A mixed-signal, functional level simulation framework based on SystemC for system-on-a-chip applications”, Custom Integrated Circuits, 2001.
- [13] Enwich et al. “SystemC extensions for mixed-signal system design”, Forum Specification, Design Language, 2001.
- [14] J. Fleischmann et al. “Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators”, Parallel and Distributed Simulation, 1995.