Chapter 6

Anatomy of a Continuous/Discrete System Execution Model for Timed Execution of Heterogeneous Systems

Faouzi Bouchimma[1], Luiza Gheorghe[1], Mohamed Abid[2], Gabriela Nicolescu[1]
[1]*Ecole Polytechnique de Montreal,* [2]*ENIS, Tunis*

1. Introduction

Modern embedded systems like micro-electro-mechanical systems (MEMS), mixed-signal systems and real-time controllers integrate discrete and continuous components. These systems can be found in various domains such as health, communication, as well as the defence, and automotive industries. The design process of these systems needs to be improved, International Technology Roadmap for Semiconductor (ITRS) announcing a "*shortage of design skills and productivity* arising from a lack of training and poor automation with a need for basic design tools" as one of the most daunting challenges in the domain [ITR06].

Designers currently build the different discrete and continuous components by using powerful existing tools specialized for an application domain (e.g. SystemC or VHDL for the electronic digital part which is discrete, Matlab/Simulink for the mechanical part which is continuous) and they are generally not too fond of changing their tools. New CAD tools, enabling the global execution of continuous/discrete systems, are consequently mandatory. These tools must be based on global execution models which are independent from the specification languages and simulators and provide several implementations enabling integration of different existing tools in order to exploit their features.

One of the main difficulties in the definition of new CAD tools for continuous/discrete (C/D) systems pertains to the heterogeneity of the concepts manipulated by the discrete and continuous components. In the case of validation tools, the key issue consists of defining global execution models, accommodating several of the discrete and continuous execution semanticists. Co-simulation

G. Nicolescu and A.A. Jerraya (Eds.), Global Specification and Validation of Embedded Systems – Integrating Heterogeneous Components, 75–108.

(as defined in Chapter 4) is currently the most popular validation technique for heterogeneous systems. This technique was successfully applied for hardware/software discrete systems [Val95], but rarely applied to C/D systems. It allows a joint simulation of heterogeneous components with different simulation models.

This chapter proposes the anatomy of a C/D global execution. This model is based on a generic architecture integrating several synchronization models, which provide adequate accuracy/performance compromises. These models result from a discrete and continuous model study and the concepts involved in continuous/discrete simulation such as time and event management.

The remainder of this chapter is organized as follows: the following section introduces continuous and discrete simulations as well as the main concepts pertaining to such models. Section 4.2 addresses time distribution in timed simulation models and defines a time model for the global execution model. Section 3.3 discusses event management in continuous/discrete systems, defines new synchronization models and a generic simulation model for continuous/discrete systems execution. Section 5 proposes implementation solutions for the defined execution model. The application of the execution model for the validation of three heterogeneous systems is discussed in Section 6. Section 7 presents the experimentations and concludes the chapter.

2. Continuous Simulation Model vs. Discrete Simulation Model

Discrete and continuous systems are characterized by different physical properties and modeling paradigms. For example, mechanical systems and analog circuits respectively, are usually modeled by analog equations derived from Newton's and Kirchhoff's laws while discrete system models are based on mathematical logic resulting from Boolean logic and arithmetic expressions.

2.1 Discrete Systems Modeling and Simulation

This section introduces the concurrent processes formalism used in discrete systems modeling and presents their execution model.

Discrete systems are commonly modeled by concurrent processes describing their behavior using Boolean and/or algebraic expressions. These processes are generally grouped into modules according to their functionalities within the system. Modules are connected by signals through input/output ports.

Discrete model execution is based on *events*. An event represents an occurrence or happing of significance to a process. The process may wait for an event or any set of events or it may (request to) receive asynchronous notification that an event has occurred. For instance, modifying a signal value at a given moment causes an event. In this case, the event is represented by a couple (signal value,

time of occurrence). Events represented only by their time of occurrence (e.g. clock event) are called *pure events*.

Processes are considered event-sensitive when events trigger executions. If several processes are sensitive to one or several events (with the same time of occurrence) then, these processes must be executed in parallel. However, executions often occur on sequential machines which can only execute a single instruction at a time, therefore one process.

Thus, this type of execution cannot really parallelize processes. The solution consists of emulating parallelism, which is based on a simple yet effective strategy: in order to execute each process "as if" the parallelism were real, it is necessary that its environment (its inputs) does not change when executing other processes. Thus, the process execution order loses its importance and everything takes place as if a parallel execution occurred. This requires that shared variables (signals) between processes keep their values until the execution of all parallel processes finishes.

Processes preserve the sequential aspects. The same instructions and control structures are found with C, Pascal and Ada languages. The only exception concerns the signal assignment: expressions assigning variables to signals are considered parallel expressions. Each assignment expression can thus be seen as a process. An example of two processes with Signals A, B, and C appears below.

Process 1
A <= B **and** C;
B <= Init **when** Select = '1' **else** C;
C <= A **and** B;

Is equivalent to:

Process 2
C <= A **and** B;
A <= B **and** C;
B <= Init **when** Select = '1' **else** C;

During simulation, the simulator must maintain *a timer* and associate a notification time for each event. Its main role consists of maintaining the event order in a global queue according to their notification times. A simulation cycle is performed at each discrete time. Within a simulation cycle, the event with the first ordered time stamp in the event queue is processed and the processes sensitive to this event are executed. This may generate other events that trigger the execution of other processes. Once all events with a time stamp identical to the current time are treated, the simulator advances its time to the value associated to the nearest discrete scheduled event and starts a new simulation cycle.

Process execution does not advance the simulator local time. Consequently, within the same simulation cycle the "cause" and the "effect" events will share the same time of occurrence, which violates the causality principle. To address this problem, the simulator uses a virtual time interval, called *delta*, the duration of which equals zero. The role of a delta-cycle is to order "simultaneous" events within a simulation cycle, i.e. identifying which event caused another. Thus, event "causes" and "effects" are consistently differed by a delta. Simulation cycles are composed of several deltas. The simulator uses a delta counter set at zero before launching a simulation cycle. If the processes executed at the beginning of the simulation cycle generate events, then the simulator annotates these events with a time stamp indicating the "real" current time with an additional delta. If there are no processes to execute at the current time ("real" current time plus zero delta), the simulator increments the delta counter (current time equals the "real" current time plus one delta) and executes the processes which are sensitive to the indicated events.

Although discrete event simulators can virtually execute any discrete system, their concept of rigorous event order may not be necessary. In fact, simulator variants have been proposed to increase discrete system performance:

- Synchronous systems simulation does not require global event sorting as signals can process events only as the clock ticks. Within a clock cycle, events can be totally or partially ordered, or entirely unordered, depending on the model [Cha96; Cha96; Cha96].

- Data flow model simulation solely uses partial ordering of events. Events associated to the different signals may not have ordering relationships. The advantage of this method: it avoids over-specifying a design when complete orders are not required [Cha96; Cha96]. In [Pat04], this solution makes it possible to improve a discrete event scheduler by increasing synchronous data flow simulation performance.

2.2 Continuous Systems Modeling and Simulation

This section introduces the computation models used for continuous systems modeling. Their simulation model is also presented.

Modeling continuous systems. A continuous system is generally described using ordinary differential equations (ODEs). An ODE refers to an equality involving a function and its derivatives. ODEs can be written in the form (1) as a system of first-order ordinary differential equations. Higher order ODEs can be reduced to a system of first-order equations. Although higher order equations solved directly are sometimes more efficient, few tools are yet available for this purpose [Gup85].

$$\dot{y} = \frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0 \qquad \text{where } y \text{ is a vector} \tag{1}$$

The form (1) is called explicit ODE. Another ODE form is the fully implicit ODE:

$$f(x, y, \dot{y}) = 0 \tag{2}$$

Most fully implicit ODEs can be written as [Gup85]:

$$M(x, y)\ \dot{y} = f(x, y) \quad \text{where } M \text{ is a matrix.} \tag{3}$$

The form (3) is called linearly implicit ODE. The inversion of the matrix M converts this form to the conventional form (1).

In the case of continuous systems, the form (1) becomes:

$$\dot{x} = \frac{dx}{dt} = f(x, u, t), \quad x(t_0) = x_0 \tag{4.1}$$

$$y = g(x, u, t) \tag{4.2}$$

where, t is the time, u is the inputs vector, x is the state variables vector, and y is the output vector.

Thus, a *state space* completely specified by the equations (4.1) and (4.2) is obtained. Formula 4.1 gives the set of *state equations* with its initial conditions and 4.2 indicates the set of *output equations*. Assuming there are n state variables, m input variables, and r output variables, these equations can be written in scalar for as a set of n state equations and r output equations:

$$n \text{ state equations} \begin{cases} \dot{x}_1 = f_1(x_1(t), \ldots\ldots, x_n(t),\ u_1(t), \ldots\ldots,\ u_m(t),\ t),\ x_1(t_0) = x_{10} \\ . \\ . \\ \dot{x}_n = f_n(x_1(t), \ldots\ldots\ldots, x_n(t),\ u_1(t), \ldots\ldots,\ u_m(t),\ t),\ x_n(t_0) = x_{n0} \end{cases}$$

$$r \text{ output equations} \begin{cases} y_1 = g_1(x_1(t), \ldots\ldots, x_n(t),\ u_1(t), \ldots\ldots,\ u_m(t),\ t) \\ . \\ . \\ y_r = g_r(x_1(t), \ldots\ldots, x_n(t),\ u_1(t), \ldots\ldots,\ u_m(t),\ t) \end{cases}$$

Linearity. The nature of the functions f and g in equations (4.1) and (4.2) consist of classifying *linear* and *nonlinear* systems. The system is said to be linear if its functions are both linear. In this case, the model given by (4.1) and (4.2) is reduced to the following system:

$$\dot{x}(t) = A(t)\, x(t) + B(t)\, u(t) \tag{5.1}$$

$$y(t) = C(t)\, x(t) + D(t)\, u(t) \tag{5.2}$$

$A(t)$ (n, n), $B(t)$ (n, m), $C(t)$ (r, n) and $D(t)(r, m)$ are matrices with n, m and, r are specified above.

Time-Invariant Systems. For time-invariant systems, functions f and g do not explicitly depend on time. In this case, equations (4.1) and (4.2) become:

$$\dot{x}(t) = f(x(t),\ u(t)) \tag{6.1}$$

$$y(t) = g(x(t),\ u(t)) \tag{6.2}$$

Assuming this time-invariance property, for linear system whose matrices $A(t)$, $B(t)$, $C(t)$ and $D(t)$ are all constant, we obtain:

$$\dot{x} = A\,x + B\,u \tag{7.1}$$

$$y = C\,x + D\,u \tag{7.2}$$

Differential-Algebraic Equations. If the set of equations describing the continuous system consists of both algebraic and differential equations, the equations are called differential-algebraic equations (DAE). The equations may be written as

$$\begin{aligned} &F_1(x, y, z, \dot{y}) = 0, \quad y(x_0) = y_0 \\ &F_2(x, y, z) = 0, \end{aligned} \tag{8}$$

where F_1 is a set of N equations and F_2 of M equations, with $N \geq M$.

Example. Figure 6.1 illustrates a simple continuous system, an RLC circuit.

The second-order differential equation describing this circuit is given by equation (9):

$$V_{in} = LC\frac{d^2V_{out}}{dt} + RC\frac{dV_{out}}{dt} + V\,out \tag{9}$$

To solve this equation numerically, it must be rewritten as an equivalent system of first-order equations:

Making the assumption expressed by equation (10) and combining it with equation (9), the following system of first order equations is obtained:

$$\begin{aligned} y_1 &= V_{out} \\ y_2 &= \dot{V}_{out} \end{aligned} \tag{10}$$

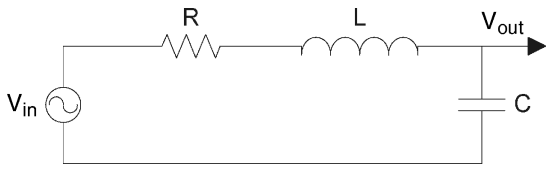

Figure 6.1. RLC circuit

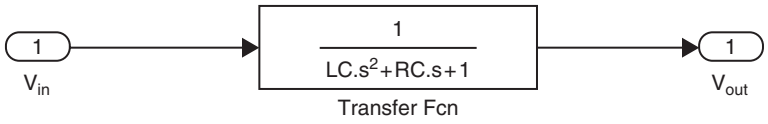

Figure 6.2. Modelling RLC circuit as a block, using the transfer function

$$\begin{cases} \dot{y}_1 = y_2 \\ \dot{y}_2 = 1/LC\,(V_{in} - RC\,y_2 - y_1) \\ V_{out} = y_1 \end{cases} \tag{11}$$

The same circuit can be easily described by the first-order ODE given by Forms 7.1 and 7.2:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1/C \\ -1/L & -R/L \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1/L \end{bmatrix} V_{in} \tag{12}$$

$$V_{out} = [1, 0] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \text{ where } x_1 = V_c \text{ and } x_2 = I_l \text{ are the state variable}$$

The system described by equation (11) and the system specified by equation (12) are equivalent. These systems can be modeled by using text editor or special blocks (the majority of continuous simulators provide special blocks).

Figure 6.2 shows the circuit from Figure 6.1 modeled as a single block specified using its transfer function given by the expression (13):

$$H(s) = \frac{V_{out}(s)}{V_{in}(s)} = \frac{1}{LCs^2 + RCs + 1} \tag{13}$$

Using (12) (13), the circuit can be described using primitive blocks: the adder, the gain and the integrator blocks. The integrator represents the principal block [Cal91] (see Figure 6.3).

Continuous models simulation. Simulating a continuous model requires solving numerically differential and algebraic equations. A widely used class of algorithms discretizes the continuous time line into a set of ordered discrete time instants and computes numerical values of model variables at these ordered time instants. The interval between two consecutive time instants is called the *integration step* and, depending on the algorithm used, this step can be either fixed or variable. As explained in the previous section, continuous systems can be modeled using several blocks. During the simulation, the equations modeling the blocks composing the system are solved at each integration step, according to the order determined by the data dependence rule.

The criteria used to select integration steps are the accuracy, stability, and signal continuity. In the case of accuracy, both fixed and variable integration

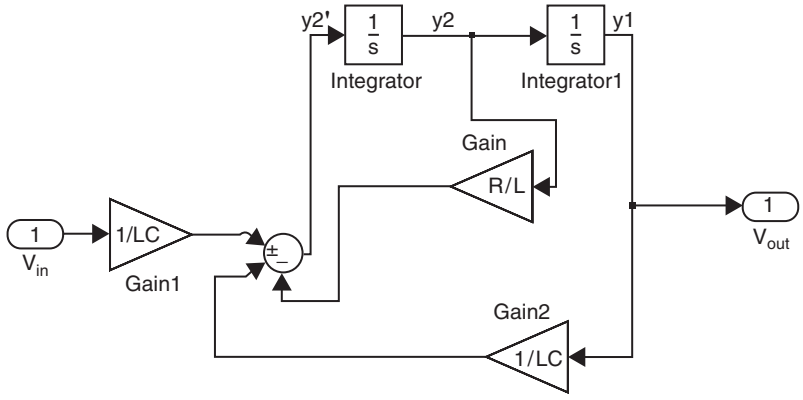

Figure 6.3. Modeling RLC circuit as a set of primitive blocks

steps can be used. However, the variable step algorithm improves the simulation performance. It reduces the step size and increases accuracy when model states change rapidly and it increases the step size to avoid unnecessary steps when the model states change slowly. When continuous models present discontinuities in form of finite jumps and/or stability problems, it is necessary to use:

- Variable step algorithms – to solve discontinuity problems observed at the solutions level, especially when interacting with discrete environments where signals change values discontinuously. In case of discontinuity, the algorithm reduces the step size again and this process can be repeated several times before the trouble spot is passed successfully [Gea84].
- Specific algorithms with a variable step – to resolve *stiff* [1] nonlinear systems of equations were used in the case of mechanical, thermal, and other models. This system exhibits time constants whose values differ by several orders of magnitude. Algorithms not designed for *stiff* problems are inefficient as they control the step size with accuracy rather than stability requirements [Cha96].

2.3 Heterogeneity in Continuous/Discrete Systems

Table 6.1 presents the main concepts characterizing continuous and discrete models: the notion of time, the means of communication and the process activation rules. Considering these concepts is a key issue for composing these two

[1] If, in a certain interval of integration, a numerical method is forced to use a step length which is excessively small in relation to the smoothness of the exact solution in that interval, then the problem is said to be *stiff* in that interval.

Table 6.1. Basic concepts for discrete and continous simulation models

Concepts / Model type	Time	Communication means	Process activation rules
Discrete	• Global notion for all system processes. • Advances discretely according to event time stamps.	Set of events located discretely on the time line	• Processes are sensitive to events. They can be executed in parallel. • At a given discrete time (simulation cycle), the execution order is determined using the delta concept.
Continuous	• Global variable involved in data computation. • Advances by integration steps.	Piecewise-continuous signals	• Processes (blocks) are executed sequentially at each integration step. • The order of execution is determined by using the technique known from static data flow (data dependency).

types of simulation models. Their composition must preserve the accurate data communication between them. This requires time synchronization and signal adaptation.

The next section introduces the time distribution approaches used to distribute time among processes. It also fixes the time distribution model for the global simulation model involving discrete/continuous simulation. This time model will be used when proposing the synchronization models.

3. Time Distribution Approaches

When designing hardware or software embedded systems, time is considered an important factor. In most cases, it must be considered with a high level of accuracy and measured with physical units. Timed simulation models assure that these requirements are fulfilled by providing a physical time model. Three approaches are proposed to calculate and distribute time among system processes [Jan04]: the local time counter approach, the time stamps approach

and the absent event approach. An overview of these approaches appears below.

3.1 Local Timer Approach

This approach considers a local time counter for the entire model. In order to obtain the global time, each process accesses this counter. In a typical usage, a time-out period is set by the process and the time counter emits an event when the time period has expired. This approach is typically used by programming languages supporting real-time process modeling such as SDL-RT [SDL06], and others derived from C. Generally, the time is not defined by the language; it is supplied by a physical source of time, a timer or a quartz oscillator counter.

A correct functionality requires that all the processes share the same global time value at a given time instance. This is obvious for systems where all processes use the same clock (e.g. single processor systems). However, synchronization is required with distributed processes (e.g. multiprocessors) using various sources of time. Most hardware languages consider time as an integral part supplied by a virtual counter. A model composed of several processes described by different languages requires a synchronization mechanism for local time. Such synchronization is mandatory for accurate event exchanges between distributed processes. Figure 6.4 illustrates two processes P and P′ with their local times T and T′ respectively. If $T \neq T'$, then an incorrect event exchange will occur: event e is emitted by P at time T and consumed by P' at time $T' \neq T$ and vice versa.

3.2 Time Tag Approach

In this approach, there is no shared source of time. Each event exchanged between two processes is annotated with a time tag that represents the global time of the event occurrence. A process receives information on time only via explicit events on its input ports. It is clear that two processes will be synchronized if they receive two events characterized by the same time of occurrence. For instance, the two processes P and P′ illustrated in Figure 41 are synchronized if $t = t'$.

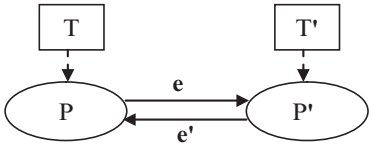

Figure 6.4. Distributed processes exchanging events

This approach increases process interdependency and eliminates unnecessary synchronization. However, it lacks total knowledge of the global time while not connected to a clock source process. Therefore, situations which require complete global time knowledge cannot be modeled with this approach.

3.3 The Absent Event Approach

In this approach, the concept of absent event enables time information communication. The time line is divided into discrete time slots and, at every time slot, all processes send events to all output signals. If no useful events are generated by a process at a given time instance, an absent event is emitted instead. The absent event value is distinct from all other possible values in the system. Its role is to inform the other processes about the time elapsed in the process that emitted it. Thus, all of the system processes are aware of the global time.

3.4 Summary

In conclusion, the first approach associated with synchronized processes is the most flexible. Although the second approach is interesting, as it preserves process independence properties, it remains applicable to a restricted number of models (used especially in data flow models). However, a time source could be added to this latter approach, but it would make it equivalent to the first approach. Finally, contrary to the third approach, where the time line is divided into discrete time slots, the first and the second approach can be used naturally for continuous time models.

4. Time Distribution Model Involved in Continuous/Discrete Execution Model

Continuous time and discrete execution models are timed models, where time is an explicit component integrated by the simulators. The continuous simulation model uses the local time counter-approach, while the discrete simulation model (discrete events) uses this approach with an additional time tag approach. At all simulation instants, both models can access the local time of the simulators on which they are executed. Thus, each process can annotate its emitted events with its own time stamps.

The global execution model cannot be characterized by a local time as this requires a tight synchronization between the continuous and discrete simulators. The approach based on event tags can be used for this type of global execution model. Thus, the events exchanged between the two simulators convey the time information. This avoids the need of a complete synchronization for each simulation time step. Using the event tag approach, a simulator performs synchronization only to accurately reach the time instants given by the event tags (stamps).

5. Global Execution of Heterogeneous Continuous/Discrete Systems

Continuous and discrete models interact via events. The instances of consuming and emitting these events must be considered with respect to causality and correct event exchanges. This section addresses event management in continuous/discrete systems co-simulation. We also propose a global simulation model based on accurate synchronization models.

5.1 Event Management in Continuous/Discrete Systems

In continuous/discrete systems, an event occurs when signal values change or once a variable exceeds a given threshold. In the first case, the event is defined by the couple (value, time stamp) while in the latter, it consists of a pure event solely defined by its time stamp. The events exchanged between the continuous and discrete models composing a continuous/discrete system are:

- The discrete model (DM) sends two types of events:
  - The *signals update events* caused by the modification of its discrete output signals
  - The *sampling events* which are pure events sent to the continuous model to indicate the sampling instants of its output continuous signals
- The continuous model (CM) can send *state events*. State events are pure and unpredictable events whose time stamps depend on the CM state variables (e.g. a zero-crossing event, a threshold overtaking event).

It is important to notice that the data path goes from the DM to the CM for signal update events, while it travels in the reverse direction (from CM to DM) for the case of sampling events.

Event exchanges must respect the causality principle during systems execution. A model is considered causal when causes precede effects. Here is one of the definitions for "causality": "The output of a process at time t should not depend on inputs that are later than t" [Liu02].

This section presents a study of event exchanges in the case of continuous/discrete global execution, where we examine compliance with causality and discuss incorrect situations that occur in event exchanges. Such situations are illustrated in Figure 6.5 and discussed below.

In Figure 6.5a, the DM generated an event at the time t_j while the CM was at the time t_i, where $t_i < t_j$. Since its time is continuous, the CM proceeds from this time and generates a state event at time t_k. This state event is caused by the consumed event at time t_j ($t_k < t_j$), violating the causality principle. In this case,

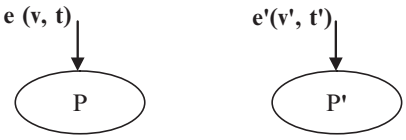

Figure 6.5. Processes synchronized by the time stamps approach

a possible solution consists of controlling the CM in order to take into account an event sent by the discrete model solely once its local time matches the time stamp of this event (each event is sent with a time stamp). Hence, even if a state event is generated at time t_k, the causality principle is not violated as this event was not caused by a discrete event not yet taken into account. However, this state event sent to the DM will have a time stamp that is inferior to the local time of the discrete simulator. This situation is illustrated on Figure 6.5c. In this case, in order to take into account the state event, the DM must be able to perform a rollback to a required time stamp. For most simulators, this rollback represents a daunting task since significant memory resources are required [Ree04]. To avoid the rollback, the CM local time must be consistently superior or equal to the discrete model local time. This solution is very effective. However, as further explained in Section 5.2, it may be preferable to run the DM before the CM for improved performance.

In Figure 6.5b, the CM generates a state event at time t_j. The DM is at the time t_i, where $t_i < t_j$. This case also presents a causality violation since the DM can generate an event at time t_k, where $t_k < t_j$. Note that the DM may generate an event regardless of the state event and this new event can impact the CM state variables and can subsequently cause a state event cancellation. The solution remains in controlling the DM to provide the time of each of its next output events. Then, the CM accurately reaches this time instant and sends its possible state events without any risks. This takes care of the causality problems.

Figure 6.5d illustrates another case where causality is violated. The CM is at time t_i and the DM generates an event at time $t_j < t_i$. In this case, an initial solution consists of using a continuous simulator that can perform rollbacks. However, this situation can be avoided when adopting the previously mentioned strategy: running the DM before the CM. Thus, the CM cannot exceed the time of the following input event from the DM. Indeed, it has to proceed until this time and then stop. Consequently, the DM must always provide the time stamp of its next event. This solution requires a DM able to predict the time stamp of its next output event.

This discussion shows that preserving causality is an important requirement to be considered when defining interfaces for the global execution model of a continuous/discrete model.

5.2 Synchronization Models

This section introduces a generic synchronization model for the global execution of continuous/discrete systems. An alternative synchronization model allowing the minimization of the interaction between the continuous and the discrete simulator is also presented.

Generic continuous/discrete synchronization model. Figure 6.6 gives an overview of the generic synchronization model, where the continuous simulator runs before the discrete simulator.

Assuming that the continuous and the discrete simulators are synchronized at the A time instant, the discrete simulator executes (without modifying time) all processes sensitive to the current notified events and updates signals (without modifying its time). Then it sends the time stamp of its next output event (point B, Figure 6.6) to the continuous simulator. Before increasing its time, it switches the simulation context towards the continuous simulator (arrow 1, Figure 6.6). The latter computes signals by solving differential equations until it accurately reaches the time sent by the discrete simulator (point C, Figure 6.6). Two cases are possible:

- This time corresponds to the occurrence time of a sample event. In this case, the continuous simulator updates the signals with the values calculated at this time and switches the simulation context to the discrete simulator (arrow 3). The latter will advance to the occurrence time of the stamp event (Figure 6.6, arrow 4) and restarts the cycle.
- This corresponds to the occurrence time of a signal update event. In this case, the continuous simulator stops temporally allowing the discrete

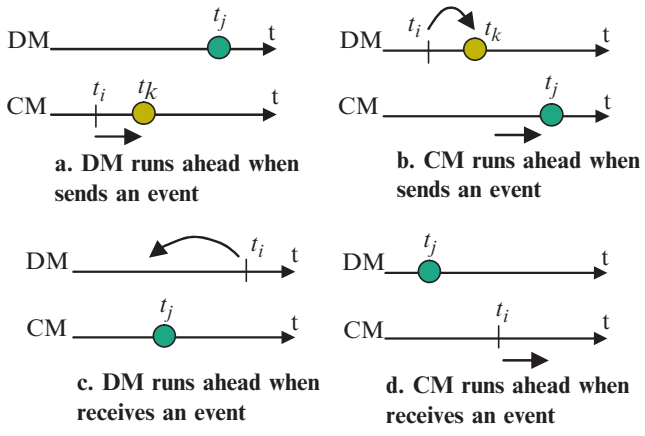

Figure 6.6. CM/DM events exchange

simulator to advance to the indicated event time stamp, to compute signals and send their values and the next event time stamp. Finally, it stops temporally allowing the continuous simulator to resume its execution. The latter proceeds to this time with the new signal values and the cycle starts over (Figure 43, arrows 5 and 6).

The continuous model may generate a state event. In this case, the continuous simulator indicates its presence, sends its time stamp to the discrete simulator and the simulation context is switched (Figure 6.6, arrow 7). The discrete simulator must be able to consider the state event by advancing the local time to the event time stamp and by executing the processes that are sensitive to it (since it is an external event).

One of the key features of this synchronization model is that it eliminates the need for rollbacks. As mentioned above, this requires that the discrete simulator can predict the time stamp of the next event corresponding to the following synchronization instant. Sampling events occur for each sampling period, so their time stamps are easily predictable. The difficulty is given by the signal update events.

Depending on the DM behavior, there are two modes which respect the presented synchronization model:

- The first mode, the *Predictable Events mode (PE*) can be used when signal update events are predictable (e.g. periodic events). In this mode, event time stamps and sampling events are placed and sorted in a special 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.

- The second mode, the *full synchronization mode* (FS) can be used when signal update 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.

Synchronization model for systems including unpredictable signal update events. In software models, signal computation requires several discrete steps. Generally, predicting the total number of steps is very difficult. In this case, the synchronization overhead of the FS mode may be inconvenient because the number of unnecessary synchronization steps will be much superior to the number of useful synchronization steps. Also, the PE mode cannot be used since signal update events are not periodic. The solution consists of running the discrete simulator in advance until it generates and sends an event to the

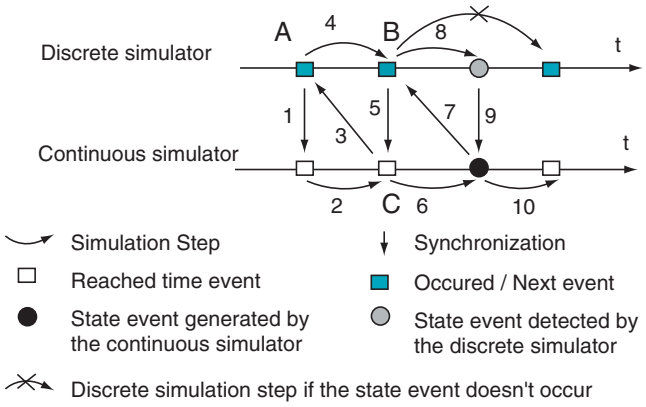

Figure 6.7. Generic continuous/discrete synchronization model

continuous simulator (with its time stamp). The synchronization model appears in Figure 6.7. This model defines a new mode which is the *unpredictable events mode* (UE).

Section 5.1 reported that, in this case, the causality problem can be solved by considering event time stamps. The difficulty is that the CM can generate a state event, requiring the discrete simulator to backtrack (Figure 6.6c). This model is recommended if the CM never generates state events as it eliminates all unnecessary synchronization. However, if the CM generates state events, rollbacks are necessary.

We suggest the use of this model, enhanced with state event consideration for control systems, where the DM represents a software component specified at the ISA abstraction level. State events model external interruptions. In this case, the checkpoint-based technique [Fle95] provides an effective solution, allowing light-rollbacks which require reduced memory resources. Indeed, only a backup of memory data segment, processor registers as well as input and output signal values will be made for each output discrete event time stamps used as checkpoints. In the case where the CM generates a state event, the discrete simulator performs a light-rollback toward the time stamp of the previous output event, restores the saved data, initializes the time counter with this time stamp and starts over with this time, taking into account the state events. The new counter replaces the discrete simulator local time and becomes the time source whose unit equals the period of the processor clock.

Summary. Table 6.2 shows the possible synchronization modes that can be used according to the continuous and the discrete model. The FS mode can usually be used but the overhead created by this mode may not be acceptable

Table 6.2. Synchronization modes depending on the continous and discrete models

Continuous Model / Discrete Model	Predictable Events	Unpredictable Events
State Events	FS, PE	FS, UE with state event considerations (ISA level)
No State Events	FS, UE	

(e.g. at the ISA level). If the CM does not generate state events, then the UE mode is recommended since it avoids unnecessary synchronization.

5.3 Global Execution Model

To run continuous and discrete simulators with respect to the presented synchronization models, new simulation interfaces must be added to allow detecting:

- State events generated by the continuous model and the consideration of these events by the discrete simulator
- Discrete simulator events (by the continuous simulator)
- The end of the discrete simulation cycle and event sending

A generic architecture for the global continuous/discrete execution model is illustrated in Figure 6.8. Continuous and discrete execution models communicate through a *co-simulation bus* via *simulation interfaces* (as introduced in Chapter 4). For both models, these interfaces implement two main *layers*: *the synchronization layer* and *the communication layer.*

Communication is assured by two layers: “data exchange” and “signal conversion and data exchange”. Data exchange consists of reading or writing signal values that connect both models. Signal conversion consists of converting continuous signals to discrete signals and vice versa. The synchronisation layer will be detailed in the following section.

5.4 The Synchronization Layer

This layer consists of six sublayers (Figure 6.8). Their task consists of supplying mechanisms that can address the three difficulties presented in Section 5.3. Their role and response to each difficulty is explained in the following sections.

State event detection and consideration. Most continuous simulators provide adequate mechanisms to detect the state events generated by the continuous model. Once detected, this event is sent by the “detection and sending of state events” layer. The discrete model “State events consideration” layer must be

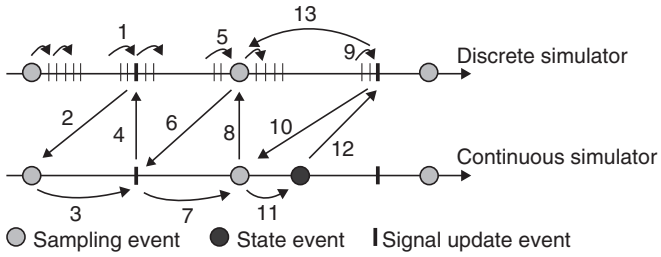

Figure 6.8. Handling of unpredictable updates

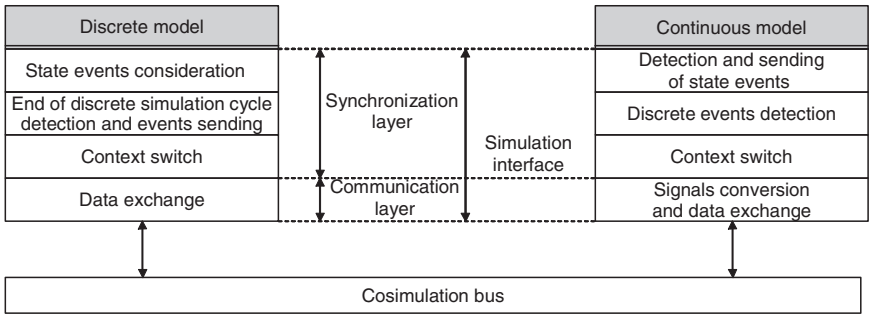

Figure 6.9. Global Execution for accurate continuous-discrete simulation model

able to take this into account. Then, the next discrete time must reflect the event time stamp rather than the internal event time stamp scheduled by the discrete simulator (Figure 6.6). In the case of the UE mode (Figure 6.7), this synchronization layer must initialize the new timer (see Section 6.5.2.2) with the last sampling event time stamp (light-rollback) and to restore the saved data.

Detecting events from discrete model. The continuous simulator must move forward until the discrete model event is detected (without missing it). This may not correspond with its discretization time. The "discrete event detection" layer (Figure 6.9) must force the continuous simulator, to adjust its integration steps which detect the event and satisfy the resolution criteria (accuracy, continuity, and stability), when approaching the event time stamp coming from the discrete model Figure 6.10 illustrates this phenomenon. It is also possible to use a fixed integration step that can be changed to a variable step when the simulator comes near a discrete event time stamp.

Detecting the end of discrete simulation cycle. Most discrete events simulators (such as the VHDL [IEE99] and SystemC [Sys03]) use the delta concept.

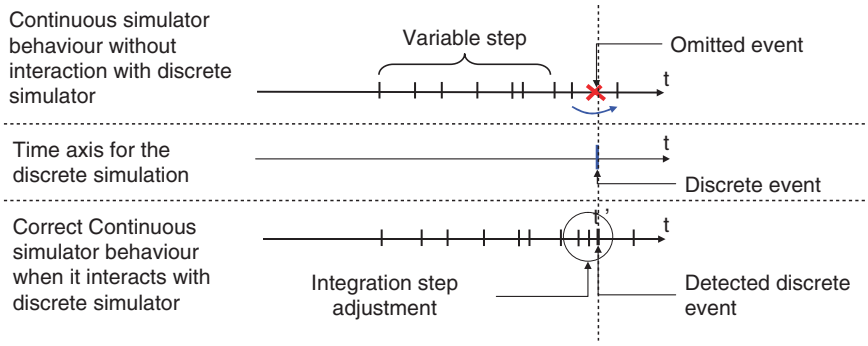

Figure 6.10. Detecting events from the discrete model

The problem of detecting the end of the discrete simulation cycles appears for this type of simulator.

For these simulators, delta-cycles are performed during delta time spans. They essentially contain two phases: an evaluation phase to execute processes and an update phase to update signals that were modified at the evaluation step. At a given discrete time, an unpredictable number of delta cycles may occur (Figure 6.11; Section 2.1) until the simulated model stabilizes: no signals to change, or in a general way, no more zero-delayed events to consider at the current time. Then, the discrete simulator increases its local time to the value of the following discrete time (next event time stamp). To guarantee that the context switch layer transfers the simulation control to the continuous model only once at a given discrete time, the discrete signals that connect both models have been stabilized, the "end of discrete simulation cycle detection and event sending" layer (Figure 6.9) becomes necessary.

6. Implementing the Global Execution Model

This section presents solutions to implement the synchronization and the communication layers for SystemC (used as an example of discrete event simulator) and Simulink (as an example of continuous simulator). To better illustrate, first, here is a brief presentation of SystemC and Simulink simulators.

6.1 SystemC

SystemC [Sys03] is a standardized modeling language intended to facilitate system level design and intellectual property exchange at multiple abstraction levels, for systems containing both software and hardware components. The SystemC simulator includes an effective and relatively simple scheduler. As indicated, the SystemC scheduler uses the delta-cycle concept (see Section 2.1). Its task consists of determining the order of the execution process by considering

sensitivity lists and events in its global queue. The last one is ordered according to the time stamps of these events. The first element in the queue thus represents the next event to occur. Events are classified into two types: zero-delayed and timed events. The time stamp of a timed event translates into a next "real" time. The time stamp of a zero-delayed event consists of two components: the current "real" time plus the sum of the number of deltas: both components are used to order the events in the queue.

6.2 Matlab/Simulink

Highly popular and widely known by the modeling and simulation community, Simulink [Mat06] offers several libraries in the automotive and power electronics sectors, etc. and seven solvers designed for stiff (appearing in nonlinear systems) and nonstiff problems, which can provide the utmost accuracy. Simulink solvers subdivide the simulation time span into major and minor integration steps, where a minor integration step represents a subdivision of a major integration step. Solvers produce results for each major integration step, using the resolution results at the minor integration steps to improve result accuracy at major integration steps.

The order in which blocks are updated is critical for result validity. The data dependence rule is used during the initialization phase in order to statically determine the order of block activation. In fact, if block outputs are depend on its inputs, they must be updated after the blocks that drive their inputs (e.g. adder or gain computing block). This approach is called *direct-feedthrough*. All of the other blocks are called *nondirect-feedthrough* (e.g. integrator block). To assure a valid update order, Simulink uses the following rule: *nondirect-feedthrough* blocks can be executed, first in no particular order, followed by *direct-feedthrough* blocks in an order which respects the above-mentioned rule.

6.3 Implementing the Synchronization Layer

The following section details the implementation of the different synchronization sublayers in the global simulation model.

Implementing discrete event detection sublayer. Simulink does not make it possible to control integration variable steps in a direct manner. Consequently, it is difficult to guarantee accuracy to detect discrete events (the discrete event time stamp).

To cope with this difficulty, the "discrete event detection" sublayer was implemented in a special S-function. The last one is devoid of input or output ports. Its role is to create breakpoints that must be reached accurately by the solver (without going beyond). The time mode used in this S-function (VARIABLE_SAMPLE_TIME) allows for choosing its next time execution equal to

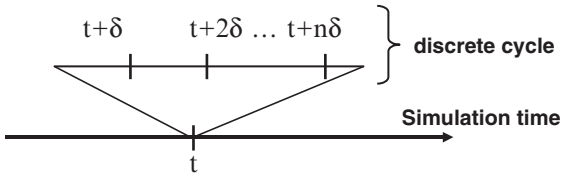

Figure 6.11. A simulation cycle is composed of an unpredictable number of delta cycles

```
/* Function: mdlGetTimeOfNextVarHit */
#define MDL_GET_TIME_OF_NEXT_VAR_HIT
static void mdlGetTimeOfNextVarHit(SimStruct *S)
 {

        /*time of the next breakpoint to consider by the solver */
        double NextBreakpointTime;

        /* read the sample event time stamp in memory*/
        double SamplingEventTime= *((double*)lpMapAddress + 200);

        /* read the signal update event time stamp in memory*/
        double UpdateEventTime =  *((double*)lpMapAddress + 100);
        if(SamplingEventTime < UpdateEventTime)
                NextBreakpointTime = UpdateEventTime;
        else NextBreakpointTime = SamplingEventTime;

        /* set the next breakpoint*/
        ssSetTNext(S, NextBreakPointTime);
 }
```

Figure 6.12. Function creating breakpoints

the next discrete event sent by the SystemC synchronization layer. In this case, Simulink adjusts the integration steps to satisfy the resolution criteria and to accurately reach the time execution of this S-function (which is the time stamp of the SystemC event). Once the sublayer detects that the event is reached, it sends (sampling event) or receives (signal update event) data. The code given in Figure 6.12 illustrates the function (member of the "discrete event detection" sublayer) that creates breakpoints.

Detection the end of discrete simulation cycle and events sending sublayers.

To guarantee that SystemC sends data and transfers the simulation control to Simulink only after discrete signals have been stabilized, detecting the end of the discrete simulation cycle is necessary. Since SystemC does not provide such mechanism, a modification to its scheduler was made to detect the discrete simulation cycle end and to switch the simulation context to Simulink. This functionality was added to the *simulate()* function in the *sc_simcontext* class of

the SystemC scheduler. This function essentially contains the simulation loop. Figure 6.12 shows the pseudo-code that provides a part of this function and indicates the end of the discrete cycle location.

Synchronization sublayers for state event cases. In the case of state events, there are two synchronization sublayers: the "state event consideration" sublayer (in the discrete model synchronization layer) and the "detection and sending of state event" sublayer (in the continuous model synchronization layer).

To detect state events, the "detection and sending of state events" layer (Simulink side) adds a "*Hit Crossing*" component from the Simulink library. This component compares the input signal to the hit crossing the offset value. If the signal increases, falls or remains at the offset value, the block output is set to "1". Once a state event is detected, the indicated layer communicates its presence by setting a special flag and sending its time stamp.

For the "state event consideration" layer (SystemC side), the solution consists of inserting a pure event (without value) into the SystemC simulator queue whose time stamp is equal to the time of the state event occurrence. The event insertion must occur before the discrete simulator increments its timer (Figure 6.7, between arrows 3 and 4). Otherwise, it must backtrack in order to take it into account. Figure 6.13 locates this insertion point of such an event. The scheduler modification solely consists of creating a set of events which can be notified in the case of state event presence. Their notification causes the execution of the SEC_Method in the code below.

For the "state event consideration" layer (the part implemented by the SystemC interface), the usual syntax to create processes sensitive to events was used (see the code below). From the designer point of view, a user process which is

1. *Initialization Phase* – Execute all processes (except SC_CTHREADs) 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 events* **or** *sampling events* **time stamp and Switch context to Simulink.**
Else (mode = UE) if *signals update events* **flag = "1" then Switch context to Simulink**
7. If state event, then add to the scheduler queue a timed event with time stamp equal to the state event time stamp.
8. If there are no more timed notifications, the simulation is finished.

Figure 6.13. SystemC enhanced scheduler

```
#define et_mat0
sc_get_curr_simcontext()->et_mat[0]

/* this definition is in the file defining environment variables added for
heterogeneous simulation */
InterfaceIn.h
...
sc_out <sc_bit> StateEventPort;
SC_CTOR(InterfaceIn)
{
  ...
//creation of et_mat0 event associated with //state event
et_mat0 = new sc_event; // et_mat0 will be notified by SC scheduler

//make SEC_Method sensitive to et_mat0,
//as consequence to the state event
SC_METHOD(SEC_Method);
sensitive(et_mat0);

... }
InterfaceIn.cpp
......
Void InterfaceIn::SEC_Method()
{
StateEvPort.write(~StateEvPort.read());
}
```

Figure 6.14. "State event consideration" layer in SystemC interface

sensitive to a state event must be marked sensitive to the input signal (reserved for this state event) coming from the SystemC interface. The following code in Figure 6.14 illustrates an example of the "state event consideration" layer in the SystemC interface.

In the case of state events, Simulink indicates its presence to the SystemC scheduler, which notifies the event (here *et_mat [0]*) associated with it by the event time stamp. This notification causes the execution of the SEC_Method process (see Figure 6.14).

6.4 Implementing the Communication Layer

To ensure "data exchange" between simulators, a shared memory created by the file Mapping API from Windows was used. It has a defined structure composed of data ports which connect signals, flag ports and time ports to exchange event time stamps.

For "signal conversion", a sampler was used to adapt continuous signals to discrete components. A signal generator (zero-order hold) was also used to extrapolate events points in order to adapt discrete signals to continuous

components. By nature, the shared memory can play the role of zero-order hold, consequently no component was added. The sampler consists of sampling signal values at synchronization points. In this case, its role is limited to reading signal values once the Simulink synchronization layer detects a sampling event from SystemC.

6.5 Discussion

The presented global simulation model remains independent from languages and environments. However, the modification of the discrete simulator was necessary during the implementation phase and discrete simulators must allow these modifications. If discrete simulators provide mechanisms to detect the end of the simulation cycle, modifications are unnecessary and commercial simulators can be used. For instance, in the case of the ModelSim VHDL simulator simulation interfaces are implemented as foreign-language interface (FLI) functions [Mod06]. In order to solve detection discrete cycles (i.e. switch simulation context only after discrete signals are stabilized), the "context switch" layer can be implemented by a VHDL process with a MTI_PROC_POSTPONED priority. Postponed processes (when triggered) run once at the end of the discrete cycle for which they are scheduled after all other processes. They can schedule an event in zero delay [Mod06].

7. CODIS a Co-Simulation Tool for Continuous/Discrete Systems

COntinuous DIscrete Simulation (CODIS) is a consistent tool which can automatically produce global simulation model instances for discrete/continuous systems simulation using SystemC and Simulink simulators. This is done by generating and providing interfaces which implement the proposed simulation model layers. It respects the presented synchronization models and offers various options for a most adequate mode. Figure 6.15 gives the overview of the flow of global simulation model generation. The inputs in the flow are the continuous model in Simulink and the discrete model in SystemC which are schematic and textual respectively. The flow output is the global simulation model.

7.1 Simulink Interfaces

Simulink interfaces are classified into four types. They do not change if the synchronization mode changes and can be parameterized from their dialog box. The interface types are:

Inter_in implements the communication layer (input function), the "context switch" layer and a part of the "discrete events detection" layer which is responsible for detecting the passage of the solver by the discrete event time stamps

(signal update events) and synchronizing them. Its parameters consist of the number of input signals.

Inter_out implements the communication layer (output function), the "context switch" layer and a part of the "discrete events detection" layer which is responsible for detecting the passage of the solver by the discrete event time stamps (sampling events) and synchronizing them. Its parameters are identical to those of the Inter_in.

Inter_state implements "detection and sending of state events" as well as the "context switch" layer. Its parameters are the state event numbers.

Sync implements the remainder portion of the "discrete events detection" layer. It is responsible for creating break points which the solver (a variable step solver) must reach with accuracy. These break points are the time stamps of the received events (signal update events and sampling events). It does not have any parameters.

The Simulink interfaces are functional blocks programmed in C++ using S-Functions. These blocks are manipulated like any other components of the Simulink library. They contain input/output ports compatible with all model ports that can be connected directly using Simulink signals. Users start by dragging the interfaces from the interface components library into their models windows. These items are then parameterized and finally connected to the inputs and outputs of users' models. Before the simulation, the functionalities of these blocks are loaded by Simulink from the .dll libraries (Figure 6.15).

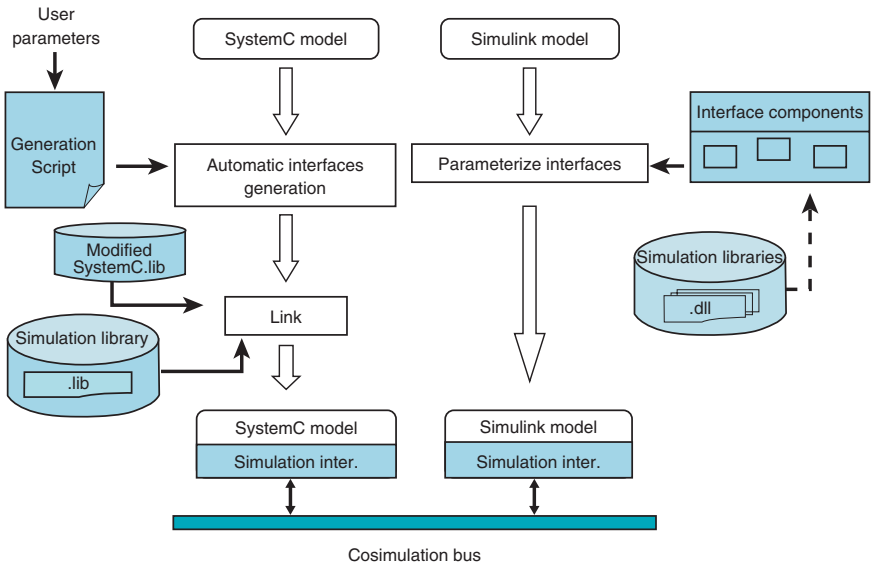

Figure 6.15. Flow of automatic generation of the simulation interfaces

7.2 SystemC Interfaces

For SystemC, as indicated above, some of the synchronization functionalities have been implemented at the scheduler level (which is a part of the state event management and the detection of the end of the discrete simulation cycle).

The interfaces are classified in the following fashion:

InterIn implements the communication layer (input function), a part of the "state event consideration" layer and the "context switch" layer (UE mode). It ensures synchronization with input data thanks to the sampling clock events (intern events). It can be viewed as a sampler circuit and contains its own parameters: (1) the names, numbers and data type of input ports, (2) the sampling periods, and (3) the mode used (e.g. FS, PE).

InterOut implements the output communication function and additional synchronization functionalities in the case of the UE mode. Its parameters are: (1) the names, number, and data type of output ports; and (2) the mode used.

Figure 6.16 presents an example of the "InterOut" interface with the UE Mode:

The interfaces are automatically generated by a script generator which is equipped with parameters defined by the user's input. Once the interfaces are generated, their connection is made within the function *sc_main*. The model is compiled and the link editor calls the SystemC library and a static library, called a "simulation library" (see Figure 6.15).

8. Experimentations

To analyze the capabilities of the continuous/discrete simulation model and its implementation, two illustrative examples were used: an engine controller used to activate a manipulator arm, and a sigma/delta converter. For both examples, the discrete part was modeled using SystemC and the continuous part using Simulink.

8.1 The Arm Controller

To regulate the engine speed, a closed-loop proportion, integral, derivative (PID) controller was used. The engine control is provided by a *discrete controller* providing speed orders calculated according to the arm position. The arm advances first at a progressive speed, then constantly and finally the speed is reduced. It runs at a constant speed when it returns to its initial position (Figure 6.18). The *continuous submodel* consists of the PID model, a sensor, an engine model and an integrator. The position sensor ("*Hit Crossing*" component) was used to signal the arrival of the arm in the desired position (state event), see Figure 6.17.

In order to evaluate the different synchronization models implemented in the presented simulation tool, the transaction level and ISA level execution

InterOutUE.h

```
#include "systemc.h"
#include "cosim.h"
SC_MODULE(interOut)
{
        sc_in<double> data;   //signals can be double or bit vector
        sc_in<double> data1;
        void send_data();

        SC_CTOR(interOut)
        {
                SC_METHOD(send_data);
                sensitive << data ;
                sensitive << data1;
                dont_initialize();
        }
};
```

InterOutUE.cpp

```
#include "InterOutUE.h"
void interOut :: send_data()
{
        // use the WriteSignalToSimulink function to send the time stamp of
         //signals update events
        WriteSignalToSimulink (sc_simulation_time()/1000000000, 100);

        // possible signal conversion
        .........
// send signals values
WriteSignalToSimulink (data.read(), 0);
        WriteSignalToSimulink (data1.read(), 1);
        // indicate the presence of new events
SwitchContextFlag ();
}
```

Figure 6.16. Excerpt of SystemC interface

models (presented in Chapter 3) were considered for the discrete part of the arm controller. At the first level, timing annotation functions were used to compute the time necessary since the system provides feedback. At this level, the system was simulated using the FS mode. For the ISA level, a SystemC implemented instruction set simulator for the DLX processor was used. The processor frequency was fixed at 4 MHz and the sampling period was fixed at 0.4 s. At this level, the UE mode with state event consideration (see Section 6.5.2.2) and the FS mode were used.

Table 6.3 presents the simulation time for an arm controller simulated during a period of 60 s.

Table 6.3. CPU time for the arm controller simulated for 60 s

Abstraction level for the discrete submodel	ISA level		Transaction level
Synchronization mode	FS	UE	FS
CPU time	25 min	5.31.s	0.26 s

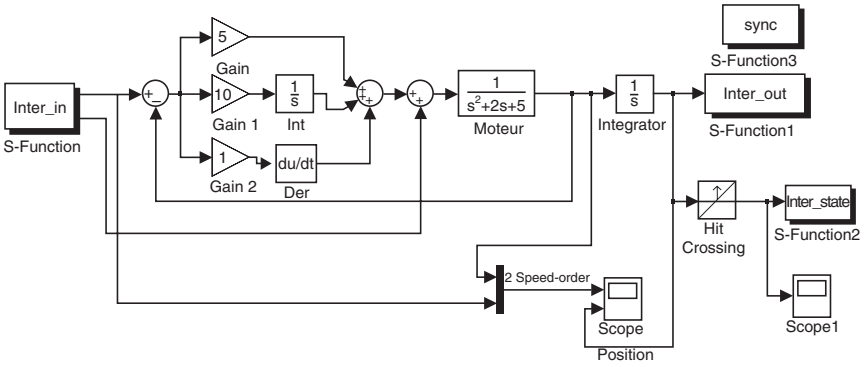

Figure 6.17. The continuous subsystem

Accuracy analysis. The accuracy analysis for the FS mode is presented. In the above example, the arrival of the arm at the desired position is a state event that occurs at the moment 15.4990 s. The *Inter_state* interface (Figure 6.17) signals the presence of this event to SystemC. Then the discrete controller sends a request to reduce the speed to zero. The state event was considered with accuracy by the discrete part and the passage of the order to zero was completed at the state event time stamp. These results are illustrated in Figures 6.18 and 6.19.

Figure 6.19 shows that SystemC scheduler planned an event $e1$at time 15.6 s while the arrival of the state event will force the scheduler to increment its time 15.4990 s. As shown in the figure, the $e1$ event, which is a clock event, will be treated after the state event. Figure 6.18 also shows the synchronization and data exchange accuracy. The continuous submodel sets the order to 1.5 at the time stamp with the value 3.2001 s, the moment it was modified by the discrete controller.

8.2 Δ/Σ Converter

Sigma-delta (Σ/Δ) converter is an over sampling analog-to-digital conversion technique. The analog input is over sampled N times faster than the requested digital output frequency and quantified by one bit, ± 1. The quantified value is fed back to the analog section, refined by an average filter and collected by a digital accumulator. For every Nsample, the converter produces the digital

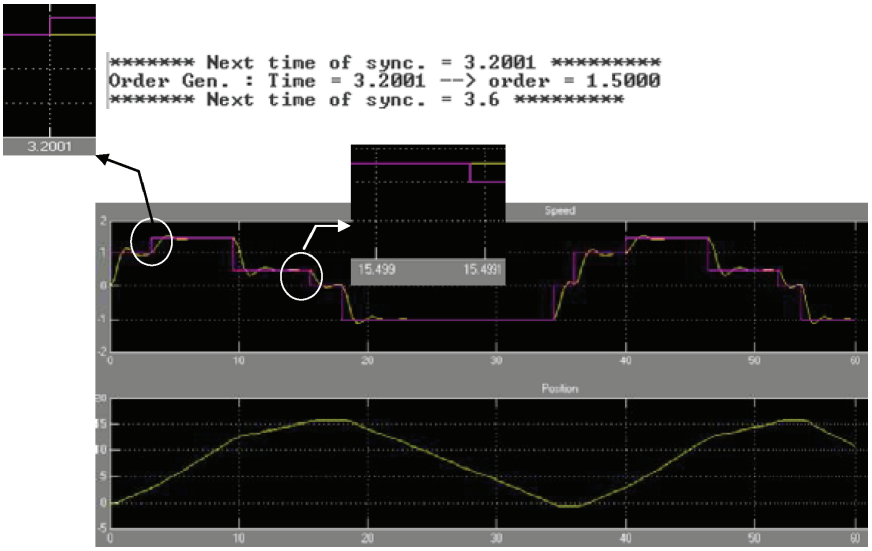

Figure 6.18. Speed, speed order and position (from Scope in Figure 6.17)

```
******* Next time of sync. = 15.6 *********
State event at time 15.4990
Order Gen. : Time = 15.4990 --> order = 0.0000
******* Next time of sync. = 15.6 *********
```

Figure 6.19. State event consideration (discrete controller output)

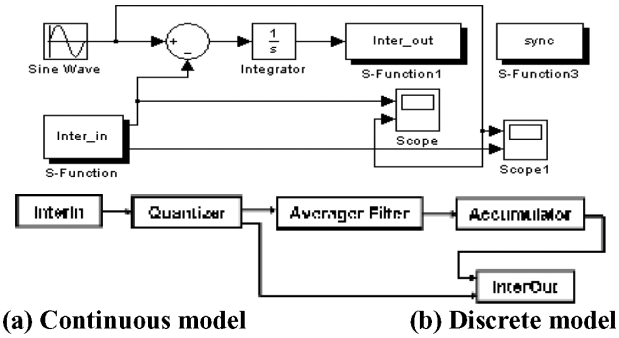

Figure 6.20. Δ/Σ converter overview

output and resets the accumulator (see Figure 6.20). For this example the sampling frequency was 5.12 Khz. Figure 6.21a shows the modulated signal (scope) and Figure 6.21b shows the input and the digital output signals.

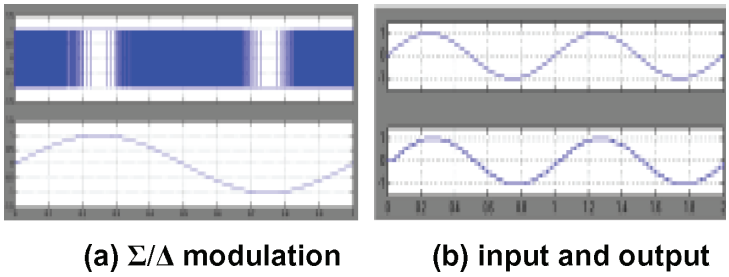

Figure 6.21. Σ/Δ converter signals

Table 6.4. CPU time for Δ/Σ converter

Application	Δ/Σ converter
Simulated time	2s
CPU time	0.8s

Table 6.4 presents time results for the converter simulation. This model simulation was conducted with UE mode since its continuous submodel does not generate state events.

Result discussion. Table 6.3 shows the acceleration of the FS model at communication level vs. the FS model at the ISA level for the discrete part of the arm controller. It is important to note that the discrete model at ISA level presents a better accuracy. However, the accuracy of the data exchange between the continuous and the discrete part is the same.

Table 6.4 also clearly shows the advantage of using the UE mode compared to the FS mode (where the continuous and discrete models are tightly synchronized resulting in unnecessary overhead): a speed-up of about two orders of magnitude was obtained.

8.3 Bottle Filling System

Figure 6.22 illustrates the model of the bottle filling system 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” (an 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 model the delay required forn the positioning of 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,

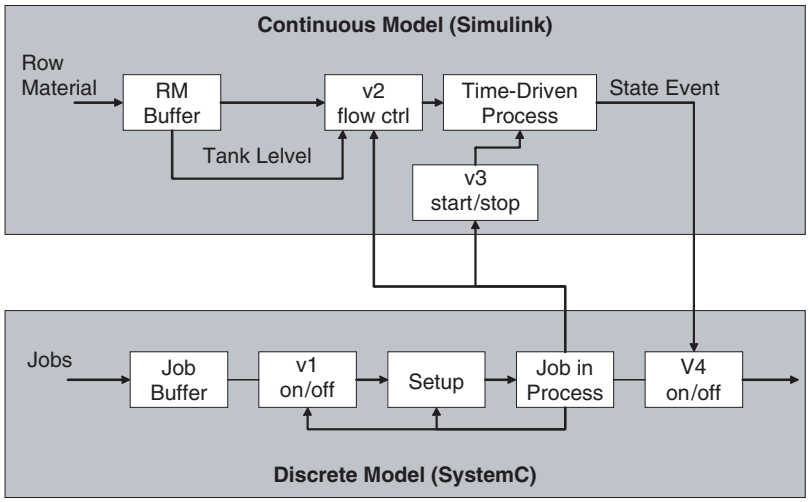

Figure 6.22. The bottle filling model

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 l/s) if the last one is empty and 0.033 l/s if not. When "time-driven process" is completed (the bottle level reaches 10 l), 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 stop the RM flow and v3 to reset the "time-driven process". For this application, the "setup" time is set to 60 s and the initial "RM buffer" level is set to 3 l. We give in Figure 6.23 an overview of the continuous model and its simulation interfaces.

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.

For the results given by Figure 6.24, 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 l/s. Each time the tank level is equal to zero, the outflow is switched to 0.025 l/s. The hit signal represents the state events generated when a bottle level reaches 10 l. The discrete model reacts to these events by switching the control signal to zero. Once a job is presented in "job in process", the DM 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 6.24 that

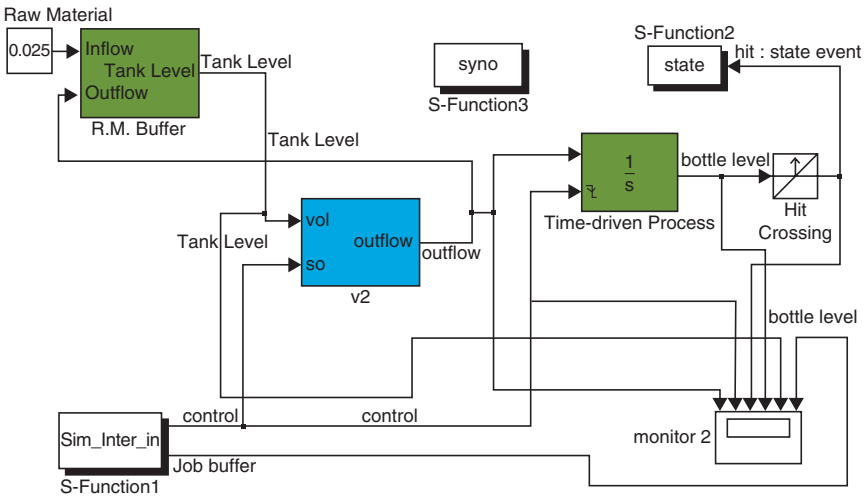

Figure 6.23. The continuous model overview

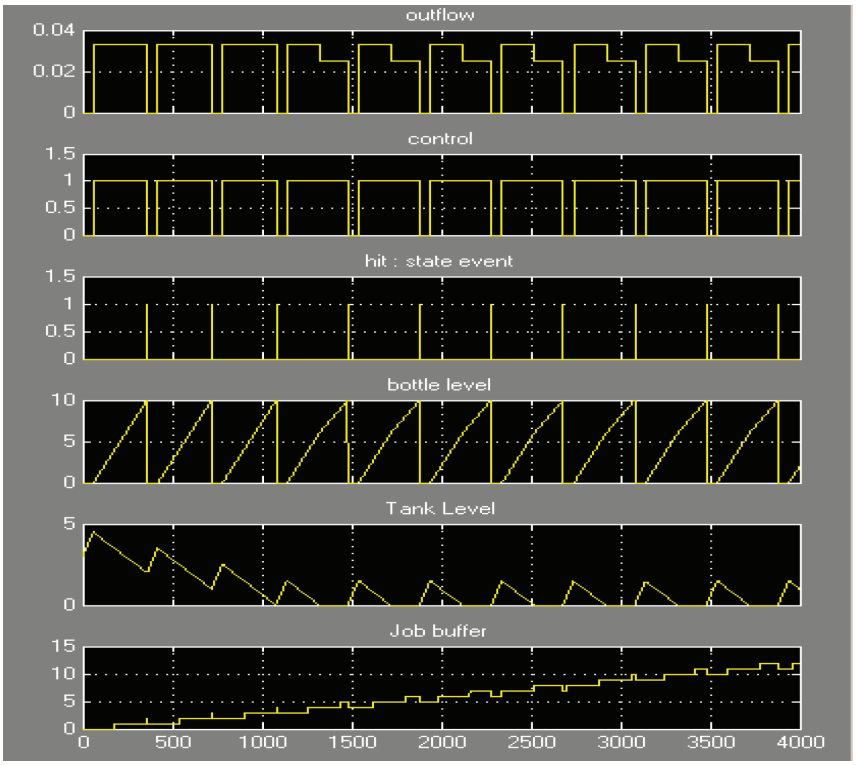

Figure 6.24. Simulation results

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 superior or equal to 400 s. This value depends on the outflow, the inflow and the "setup" time.

9. Conclusion

This chapter presented the anatomy of a discrete/continuous global simulation model. The first section presented the concepts manipulated by both models, the time distribution model as well as event management. In the second part, based on previous studies, several synchronization models that resulted from a deep analysis of synchronization issues with respect to accuracy and performance constraints were introduced. In the third part, the architecture of a generic global simulation model was proposed, providing semantics for the accurate global validation of discrete/continuous systems. It allows for the use of powerful tools in both domains. The global simulation model was implemented by simulation interfaces in order to produce global simulation model instances for discrete-continuous systems simulation using SystemC and Simulink. Finally, to evaluate the proposed simulation model, co-simulation results from two discrete/continuous applications were illustrated.

References

[Bal03] Balarin, F. et al., "Metropolis: An Integrated Electronic System Design Environment", *Computer*, vol. 36, issue 4, April 2003, pp. 45–52.

[Cel06] Celoxica, http://www.celoxica.com/methodology/

[Cal91] Callier F. M., Desoer C. A., *Linear System Theory*, Germany, Springer-Verlag, 1991.

[Cha96] W.T. Chang et al., "Heterogeneous Simulation – Mixing Discrete-Event Models with Dataflow", RASSP special issue of J. on VLSI Signal Processing, 1996.

[Fle95] J. Fleischmann et al., "Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators", Parallel and Distributed Simulation, 1995.

[Fre00] P. Frey et al., "Verilog-AMS: Mixed-Signal Simulation and Cross Domain Connect Modules", Behavioral Modeling and Simulation Workshop, 2000.

[Gea84] C. W. Gear et al., "Solving Ordinary Differential Equations with Discontinuities", *ACM Transaction on Mathematical Software*, vol. 10, 1984, pp. 23–44.

[Gup85] K. Gupta, et al., "A Review of Recent Developments in Solving ODES", Proc. of CSUR, vol. 17, No. 1, 1985.

[IEE99] *IEEE Standard VHDL Analog and Mixed-Signal Extensions*, IEEE Std 1076.1-1999, 23 December 1999.

[ITR06] International Technology Roadmap for Semiconductor Design, 2003.

[Jan04] Jantsch A., *Modeling Embedded Systems and SOCs*, USA, Morgan Kaufmann, 2004.

[Liu02] J. Liu et al., "On The Causality of Mixed-Signal and Hybrid Models", Hybrid Systems: Computation and Control, 2003, pp. 328–342.

[Mat06] Matlab-Simulink, www.mathworks.com

[Mar02] Martin D. E et al., "Integrating Multiple Parallel Simulation Engines for Mixed-Technology Parallel Simulation", Simulation Symposium, 2002.

[Mod06] Modelica, www.modelica.org

[Mod06] ModelSim® SE Foreign Language Interface, version 5.7d.

[Nic02] G. Nicolescu et al., "Validation in a component-based design flow for Multicore SoCs", in Proc. ISSS, 2002.

[Pat04] H. D. Patel et al., "Towards A Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models", Proc. ISVLSI'04.

[Pto06] Ptolemy, http://ptolemy.eecs.berkeley.edu/ptolemyII

[Ree04] D. K. Reed et al., "An Application of Parallel Discrete Event Simulation Algorithms to Mixed Domain System Simulation", DATE'04, pp. 1356–1357.

[SDL06] SDL RT, specification & description language – real time, available at http://www.sdl-rt.org

[Sys03] SystemC LRM, 2003, available at www.SystemC.org

[Tah93] El Tahaway et al., "VHDeLDO: A new mixed mode simulation", DAC Conference, 1993, with EURO-VHDL'93. Proceedings EURO-DAC'93.

[Vac03] A. Vachoux, et al., "Analog and Mixed Signal Modeling with SystemC", Circuits and Systems, ISCAS'03.

[Val95] C. A. Valderrama et al., "A unified model for co-simulation and co-synthesis of mixed hardware/software systems", Proc. 1995 European Conference on Design and Test.