HYBRID SYSTEM SIMULATION WITH SIMEVENTS

Christos G. Cassandras [*]

[*] *Center for Information and Systems Engineering*
Boston University, Brookline, MA 02446 USA

Michael I. Clune and Pieter J. Mosterman [**]

[**] *The MathWorks*
3 Apple Hill Drive, Natick, Mass. 01760 USA

Abstract: A new simulation product for hybrid systems is described, which combines a time-driven component and an event-driven component (SimEvents). Some of the key issues arising in designing such simulation environments are also discussed. *Copyright © 2006 IFAC*

1. INTRODUCTION

A *Hybrid System* (HS) is often defined as a system that combines continuous with discrete state variables (Levine and (Eds.), 2005). More important, however, is the fact that a HS combines time-driven dynamics (associated with processes modeled through differential or difference equations) with event-driven dynamics (modeled though state automata, Petri nets or other modeling frameworks for *Discrete Event Systems* (DES) (Cassandras and Lafortune, 1999)). Traditionally, simulators (such as Simulink® (MathWorks, 2001)) employ a time-driven execution mechanism, while drastically different ways are employed for DES. SimEvents (MathWorks, 2005) is designed to simulate DES, but is embedded in Simulink and equipped with functionality that enables an effective co-existence of time-driven and event-driven components making up a HS.

2. ARCHITECTURE

Figure 1 highlights the main functional components of the overall architecture. As a DES simulation engine, SimEvents is driven by an Event Calendar where all future events to occur are listed in ascending order of their scheduled time. SimEvents always processes the first event in this list and updates the DES state accordingly. When such an event takes place, the Cooperative Event Driver is responsible for translating it into a Simulink signal. The Data Exchange module passes this signal on to Simulink so that it may trigger a time-driven process or update various model parameters. Conversely, as a time-driven process evolves under the control of Simulink, it may generate events in the form of level-crossing points (from above, from below, or either) that the Data Exchange module appropriately translates so they may be processed by SimEvents blocks. The most challenging aspect of coordinating time-driven and event-driven dynamics is that of proper timing. In the architecture of Fig. 1, the system "clock" is maintained by Simulink and the Cooperative Event Driver is responsible for ensuring consistency between Simulink blocks and SimEvents blocks, which interact with the Event Calendar. Note that when a pure DES is simulated, the only interaction between SimEvents and Simulink is a simple link to the system clock through the Cooperative Event Driver which ensures that the sample times applied are consistent with times in the Event Calendar.

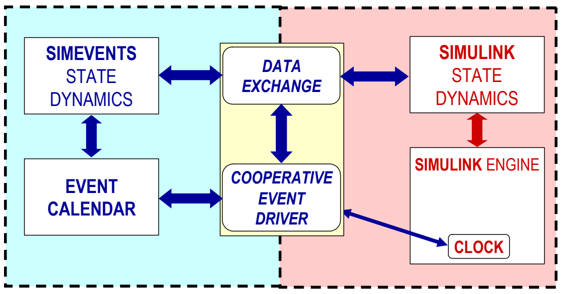

Fig. 1. SimEvents and Simulink collaborative functionality

3. SIMEVENTS FUNCTIONALITY

In Simulink, communication across blocks is based on signals. In SimEvents, it is based on both signals and entities. The "entity" concept is motivated from the view of a DES as an environment consisting of "users" and "resources": users request resources in order to perform various tasks, occupy these resources for a certain amount of time, and then relinquish them so that other users may access them. Examples of users are messages in a communication network and parts in a manufacturing system. Examples of resources are switches in a network and machines in a factory. A typical hybrid system scenario arises when an entity accessing a resource initiates a physical process (thus, defining an event in SimEvents), which is carried out until some termination condition is satisfied (defining another event in Simulink). Based on this approach, SimEvents consists of a number of libraries containing blocks with different system functionalities. The main libraries are the following:

1. **Generators**: Blocks that generate entities, or function calls (i.e., events that call Simulink blocks), or random variates.

2. **Queues**: Blocks where entities can be temporarily stored, while waiting to access a resource.

3. **Servers**: Blocks that model various types of resources.

4. **Routing**: Blocks that control the movement of entities as they access queues and servers.

5. **Gates**: Blocks that control the flow of entities by enabling/disabling access of entities to certain blocks.

6. **Event Translation**: Blocks that enable communication between SimEvents and Simulink by translating events into function calls.

7. **Attributes**: Blocks that assign and modify data to entities. Various control actions are then made based on the values of these data, allowing blocks to differentiate between entities they process.

8. **Subsystems**: These allow a combination of blocks to be executed upon occurrence of specific events (not upon Simulink sample times).

9. **Timers and Counters**: Blocks that measure event occurrence times or time elapsing between events, and blocks that count occurrences of particular event types. These data are supplied to standard display or scope blocks in Simulink or specialized scopes designed specifically for SimEvents.

4. SOME DESIGN PROBLEMS IN HYBRID SYSTEM SIMULATION

We limit ourselves to three key problems that are ubiquitous in combining time-driven and event-driven systems.

1. **Event-driven vs Time-driven Computation**. In a DES or HS setting, the computation of many quantities is not required until a particular event occurs. However, all Simulink computation blocks are executed in a time-driven fashion. This causes an *efficiency* problem as well as a potential *integrity* problem.

To illustrate the efficiency issue, consider two signals x and y and an addition block that generates $z = x + y$. If x and y are constant and only change values at times t_x and t_y respectively, then a time-driven adder needlessly evaluates z at all sample times (as defined by the system clock). The solution provided in SimEvents is to place the adder block in a discrete event subsystem that is triggered only by events occurring at times t_x and t_y.

The integrity issue manifests itself when the period Δ of the sampling mechanism is such that an event, say the one changing x at time t_x, occurs in the interior of an interval $[t, t+\Delta]$. In this case, the value of z over all times in $[t_x, t + \Delta)$ is incorrect. Moreover, if the event at t_x is intended to trigger some other system action, this action can only be taken at time $t + \Delta$. Although most sophisticated simulators are capable of detecting such variable-changing events in their sampling mechanisms, this still imposes a requirement on the timing engine of the simulator instead of making it a process which is naturally triggered by an element of the Event Calendar in Fig. 1.

2. **Declarative vs Imperative Semantics**. A time-driven system naturally employs declarative semantics, i.e., an equation implies a constraint

that the variables involved must satisfy. This fact is crucial when a system includes feedback loops: in Simulink a feedback loop is interpreted as an "algebraic loop" that must be resolved in order for the system simulation to be executed.

In an event-driven system, imperative semantics are used, i.e., an equation implies a strict assignment (which is why the symbol ':=' is sometimes used instead of '='). A feedback loop in this case typically means that an event has been detected in some process at time t, which may change an input variable for that process immediately after t (e.g., disable a process at t as soon as one of its state variables first crosses a given threshold). Combining components from both settings can, therefore, create ambiguities regarding the meaning of loops. In order to resolve this issue, a special SimEvents block termed "Signal Latch" is introduced in such loops to effectively translate a signal from a Simulink block into an event processed by SimEvents blocks, before generating a new signal input to the same Simulink block.

3. **Event concurrency**. In an event-driven system, it is possible for multiple events to occur concurrently. The order in which these events are executed is controlled by means of a "priority" scheme which is part of the underlying DES design. It is also possible for an event to trigger the occurrence of one or more other events, which in turn triggers a number of events, with all this activity taking place in zero time. Such event-ordering capability does not normally exist in a time-driven environment, where events are only defined as level crossings arising in the evolution of a continuous process (although this issue is explicitly addressed in some more advanced simulators, as in (Mosterman, 2002)). This presents a major challenge in a hybrid system setting.

To illustrate this issue, consider a Simulink "enabled subsystem" where some time-driven process $x(t)$ is initiated by an enabling signal, labeled e_1, generated by a SimEvents block event at time t_0 with $x(t_0) = 0$. The subsystem is defined so that the rising edge of an enabling signal $s(t)$ at any time t resets the process to $x(t) = 0$. Further, the process must stop as soon as $x(t) = c$ at some $t_1 > t_0$. This defines a level-crossing event, labeled e_2, which disables this enabled subsystem. The process can subsequently be re-enabled when the next e_1 event takes place. Suppose, however, that, under certain conditions, event e_2 immediately triggers event e_1, i.e., the subsystem is disabled at time t_1 and immediately enabled again. The correct behavior in this case is to re-initialize the process so that $x(t_1^+) = 0$. However, the Simulink timing engine is not designed to process event e_1 followed by event e_2 in zero time; instead, it observes that $s(t_1^-) = s(t_1^+) = 1$ and allows the process to remain enabled with $x(t_1^+) = c$. The only way to resolve this problem is by allowing $s(t)$ to jump from $s(t_1^-) = 1$ to $s(t_1) = 0$ and from $s(t_1) = 0$ to $s(t_1^+) = 1$ again, an operation that basic Simulink blocks are not equipped to perform. In other words, a hybrid system simulator must allow a signal to take multiple values at each point in time. The alternative (without possibly resorting to some advanced functionalities) is to introduce an artificial miniscule delay between the concurrent events e_1 and e_2, allowing the subsystem to process two events at different times.

The broader issue that this problem points to is that of designing a timing mechanism capable of driving differential or difference equations and processing discrete events from an event calendar where concurrent events are possible, hence an ordering scheme is also necessary. This issue boils down to the question "who should control the system clock – the time-driven component or the event-driven component of a hybrid system?"

REFERENCES

Cassandras, C. G. and S. Lafortune (1999). *Introduction to Discrete Event Systems.* Kluwer Academic Publishers.

Levine, W. and D. Hristu (Eds.) (2005). *Handbook of Networked and Embedded Control Systems.* Birkhauser.

MathWorks (2001). *Simulink: A Program for Simulating Dynamic Systems, User Guide.* The MathWorks, Inc.

MathWorks (2005). *SimEvents User's Guide.* The MathWorks, Inc.

Mosterman, P. J. (2002). HyBrSim - a modeling and simulation environment for hybrid bond graphs. *Journal of Systems and Control Engineering* **216**, 35–46. Part I.