11

Discrete Event Simulation

Preview

This chapter explores a new way of approximating differential equations, replacing the time discretization by a quantization of the state variables. We shall see that this idea will lead us to discrete event systems in terms of the DEVS formalism instead of difference equations, as in the previous approximations.

Thus, before formulating the numerical methods derived from this approach, we shall introduce the basic definitions of DEVS. This methodology, as a general discrete event systems modeling and simulation formalism, will provide us the tools to describe and translate into computer programs the routines that implement a new family of methods for the numerical integration of continuous systems.

Further, the chapter explores the principles of quantization–based approximations of ordinary differential equations and their representation as DEVS simulation models.

Finally, we shall briefly introduce the QSS method in preparation for the next chapter, where we shall study this numerical method in more detail.

11.1 Introduction

In previous chapters, we studied many different methods for the simulation of continuous systems. In spite of their differences: explicit *vs.* implicit methods, fixed–step *vs.* variable–step, fixed–order *vs.* variable–order, all of these algorithms had something fundamental in common: given time t_{k+1}, a polynomial extrapolation is performed for the purpose of determining the values of all state variables at that time instant.

In this chapter, we shall pursue an entirely different idea. Rather than asking ourselves, what value a particular state variable assumes at any given point in time, we shall ask the question, at what time the state variable will deviate from its current value by more than ΔQ. Hence we wish to find the smallest time step, h, such that $x(t_k + h) = x(t_k) \pm \Delta Q$.

Evidently, such an integration algorithm will naturally be a variable–step method. During time periods, when the state variable changes its value slowly, the algorithm will compute using large step sizes, whereas it will use small step sizes, whenever the state variable exhibits a large either positive or negative gradient.

It should be remarked that, when $\mathbf{x}$ is a state vector, the resulting value of h will be different for each component of $\mathbf{x}$. Then, we have two possibilities: we can either choose the smallest of these values as the next central step size, h, or we can use different values of h_i for different components of the state vector, x_i, leading to an *asynchronous simulation*, in which each state variable possesses its own simulation time.

The former of these two alternatives can be combined with any of the previously introduced integration algorithms. It simply represents a novel way of performing step–size control. It is an interesting concept, but shall not be pursued further in this chapter, as it doesn't really introduce any new challenges.

The latter idea looks much more revolutionary. At first glance, the resulting methods would consist in a sort of combination of multi–rate and variable–step algorithms.

Up to this point, all of the methods we saw can be described by *difference equations*. Such a representation makes no sense in a method, in which each component evolves following its own values of h_i.

A first consequence of this remark has to do with linearity. Given a linear time–invariant system:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{11.1}$$

numerical integration using any of the previously introduced integration methods leads to a linear difference equation:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} \tag{11.2}$$

If we allow each component to follow its own step size, we not only lose the representation as a difference equation, but we also sacrifice linearity. Consequently, we can no longer hope to be able to draw a stability domain, as we have gotten accustomed to throughout this book.

¿From a system–theoretic point of view, we can say that all of the methods that we have looked at until now discretize time. In other words, the resulting simulation codes (i.e., the models executed by the simulation program) are always *discrete–time systems*. When we talk about discrete–time systems, we refer to systems that change their states synchronously in time.

Our proposed approach produces algorithms that are entirely different, as they operate in a completely asynchronous fashion. New problems arise that shall have to be dealt with. We shall need to discuss both *stability* and *accuracy* of these algorithms in a new light, as our previously used techniques break down in the context of these algorithms. Also, we shall need to discuss *synchronization mechanisms*, a problem that we had not encountered so far. As most state equations depend on more than one state variable, the values of which are now known at different time points, we shall need to analyze, how we can synchronize the state variables for the purpose of computing state equations under controlled accuracy conditions.

Yet, these new algorithms not only cause new difficulties. They also offer important simplifications and potential savings.

A first simplification relates to the handling of state events. As we mentioned in Chapter 9, the integration method must evaluate the discontinuous states at event times. Since those event times usually do not coincide with the discrete time instants prescribed by the integration method, we had to modify the method to hit the events with a given accuracy. This requirement implied adding iteration techniques that not only complicated the algorithms but also increased the number of computations per elapsed simulation time. Moreover in the context of real–time simulation, we may not be able to afford those iterations without losing the race against time. We shall learn that the newly proposed algorithms do not call for the iteration of state events, and therefore, may be better suited for the simulation of discontinuous systems, especially in the context of real–time simulation.

Another potential simplification results in the case of large systems of ODEs, as they arise when discretizing hyperbolic partial differential equations using the method–of–lines approach. Hyperbolic PDEs frequently lead to shock waves that travel through space with time. Consequently at any point in time, the gradients of these waves will be steep at some point in space, whereas they are flat in all other regions. Using a synchronous integration algorithm, the step size of all states will have to be adjusted to the steepest gradient, so that small step sizes will have to be used on all differential equations at all times. In contrast, the newly proposed algorithms will enable us to use large step sizes on most state variables most of the time.

11.2 Space Discretization: A Simple Example

Returning to our idea of designing integration methods, in which the steps are ruled by changes in states rather than in time, we shall introduce an example that shows some of the basic principles of these integration techniques.

Consider the first order system:

$$\dot{x}_a(t) = -x_a(t) + 10 \cdot \varepsilon(t - 1.76) \tag{11.3a}$$

where $\varepsilon(t)$ represents the unit step function, i.e., $\varepsilon(t-1.76)$ describes a unit step taking place at $t = 1.76$.

We shall consider the initial condition

$$x_a(t_0 = 0) = 10 \tag{11.3b}$$

If we try to simulate this model using a fixed–step method with a step size of $h = 0.1$, which would be appropriated in accordance with the rate, at which the single state variable changes its value, the time step would

occur at an instant of time that does not coincide with the discrete time instants prescribed by the integration algorithm.

Let us now consider what happens with the following *continuous–time system*:

$$\dot{x}(t) = -\text{floor}[x(t)] + 10 \cdot \varepsilon(t - 1.76) \tag{11.4a}$$

or:

$$\dot{x}(t) = -q(t) + 10 \cdot \varepsilon(t - 1.76) \tag{11.4b}$$

where $q(t) \triangleq \text{floor}[x(t)]$ denotes the integer part of the positive real–valued variable $x(t)$.

Although the system defined by Eq.(11.4) is nonlinear and does not satisfy the analytical properties that we like (it is highly discontinuous), it can be easily solved.

When $0 < t < 1/9$, we have $q(t) = 9$ and $\dot{x}(t) = -9$. During this interval, $x(t)$ decreases linearly from 10.0 to 9.0. Then, during the interval $1/9 < t < 1/9 + 1/8$, we have $q(t) = 8$ and $\dot{x}(t) = -8$. During this time interval, $x(t)$ decreases linearly from 9.0 to 8.0.

Continuing with this analysis, we find that $x(t)$ reaches a value of 3.0 at time $t = 1.329$. If no time event were to occur, $x(t)$ would reach a value of 2.0 at time $t = 1.829$. However at time $t = 1.76$, when $x = 2.138$, the input changes, and from that moment on, we have $\dot{x}(t) = 8$. The variable $x(t)$ increases its value again linearly with time, until it reaches a value of 3.0 at time $t = 1.8678$.

The real–valued $x(t)$ variable continues to grow, until the system reaches $x(t) = q(t) = 10$, at which time the derivative $\dot{x}(t)$ becomes zero, and the system will not change its state any longer.

Figure 11.1 shows the trajectories of $x(t)$ and $q(t)$.

We completed this simulation using 17 steps and, ignoring the round–off problems, we obtained the *exact solution* of Eq.(11.4). All computations required to obtain this solution were trivial, because the state derivative *remains constant* in between event times, which enabled us to compute the real–valued variable $x(t)$ analytically.

The solution $x(t)$ and the solution of our original system of Eq.(11.3), $x_a(t)$, are compared in Fig.11.2.

The solutions of the original and the quantized system are definitely related to each other. By replacing the state variable $x(t)$ by $q(t) = \text{floor}[x(t)]$ on the right hand side of a first–order differential equation, we found an explicit method to simulate the quantized model.

The question naturally arises, whether we might not be able to generalize this approach to n^{th}–order systems by quantizing all state variables on the right hand side of all state equations. Unfortunately, we are not ready to answer this question yet. To this end, we shall first need to explore the

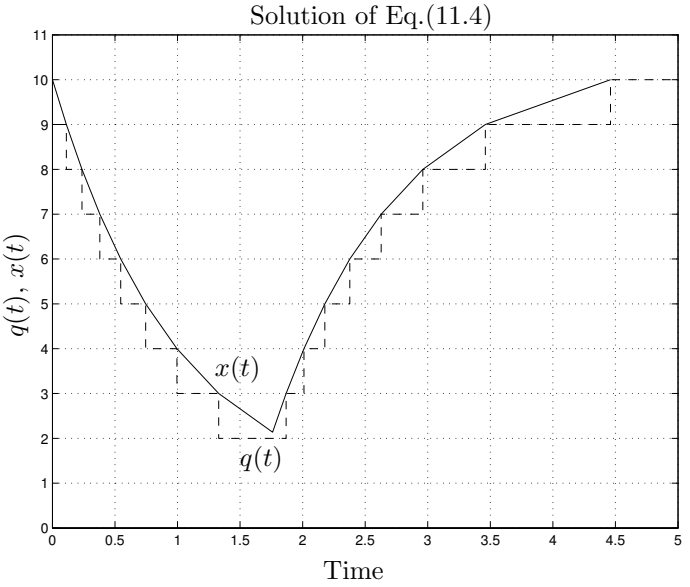

FIGURE 11.1. Variable trajectories of the system of Eq.(11.4).

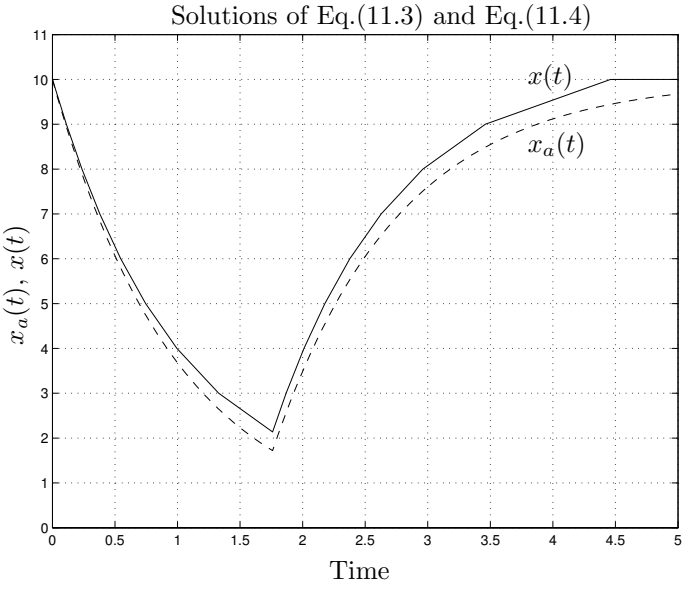

FIGURE 11.2. State trajectories of the systems of Eq.(11.3) and Eq.(11.4).

discrete nature of the system of Eq.(11.4) and introduce some tools for its representation and simulation.

11.3 Discrete Event Systems and DEVS

The simulation of a differential equation system using any of the methods we studied in previous chapters led us to a set of difference equations of the form:

$$\mathbf{x}(t_{k+1}) = \mathbf{f}(\mathbf{x}(t_k), t_k) \tag{11.5}$$

where the difference $t_{k+1} - t_k$ can be either constant or variable, and the function $\mathbf{f}$ can be explicitly or implicitly defined. As a consequence, the simulation program contained an iterative code that advances the time in accordance with the next step size. In other words, those simulation methods produce *discrete–time models* of simulation.

The system of Eq.(11.4) can be viewed as a simulation model, because it can be exactly simulated with only 17 steps. However, it does not fit the format of Eq.(11.5). The problem here is the asynchronous way, in which it deals with the input change at time $t = 1.76$.

Evidently, we are confronted with a system that is discrete in some way, which however belongs to a different class than the systems characterized as discrete–time systems. As we shall see soon, our new approximation can be represented by a *discrete event system*.

Many popular discrete event formalisms have been defined that some of our readers may already be familiar with. These include the *state automata*, *Petri nets*, *event graphs*, and *state charts*. However, none of these representations is suitable for dealing with our system in a general situation. These graphical languages are limited to systems with a finite number of states. Fortunately, there has been found a more general discrete event system formalism, called DEVS, that offers the support that we were looking for.

DEVS, which stands for *Discrete EVent System specification* [11.14, 11.11], was introduced by Bernard Zeigler in the mid seventies. DEVS allows to represent all systems, whose input/output behavior can be described by sequences of events under the condition that the state undergoes a finite number of changes within any finite interval of time.

In our context, an *event* is the representation of an instantaneous change in some part of a system. It can be characterized by a value and a time of occurrence. The value can be a number, a vector, a word, or in general, an element of a given set.

The trajectory defined by a sequence of events assumes the value ϕ (or *No Event*) for all time instants except for those, when there are events. In those instants, the trajectory takes the value corresponding to the event. Figure 11.3 shows an event trajectory that takes the value x_2 at time t_1, then the value x_3 at time t_2, etc.

A DEVS model processes an input event trajectory and, according to that trajectory and its own initial conditions, provokes an output event trajectory. This input/output behavior is depicted in Fig.11.4.

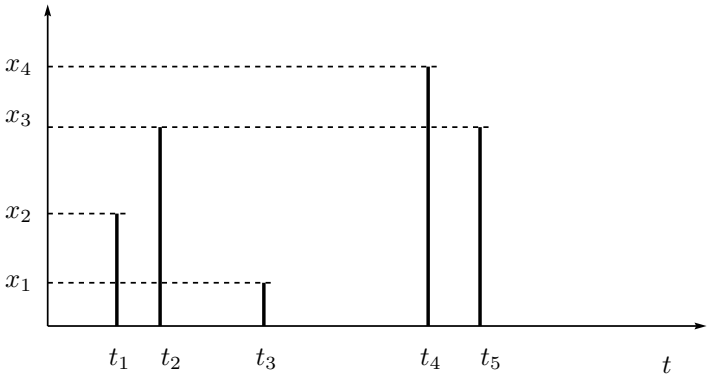

FIGURE 11.3. An event trajectory.

FIGURE 11.4. Input/output behavior of a DEVS model.

The behavior of a DEVS model is expressed in a way that is quite common in automata theory. This kind of representation consists in enumerating some sets and functions that define the system dynamics in accordance with certain rules. Since the rules are always the same in a given formalism, they are not mentioned in each model.

Following this idea, an *atomic* DEVS model is defined by the following structure:

$$M = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta)$$

where:

- X is the set of input event values, i.e., the set of all possible values that an input event can assume;
- Y is the set of output event values;
- S is the set of state values;
- δ_{int}, δ_{ext}, λ and ta are functions that define the system dynamics.

Figure 11.5 illustrates the behavior of a DEVS model.

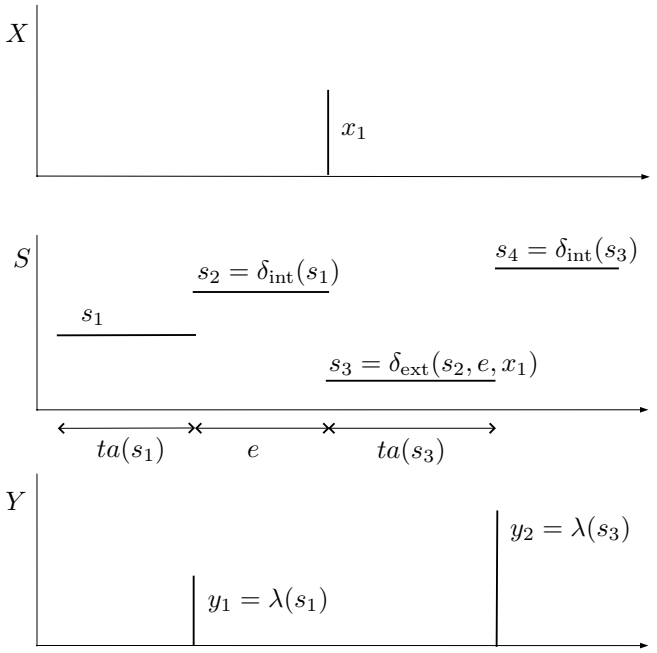

FIGURE 11.5. Trajectories in a DEVS model.

Each possible state s ($s \in S$) has an associated *time advance* calculated by the *time advance function* $ta(s)$ ($ta(s) : S \to \mathbb{R}_0^+$). The time advance is a non–negative real number, determining how long the system remains in a given state in absence of input events.

Thus, if the state adopts the value s_1 at time t_1, after $ta(s_1)$ units of time (i.e., at time $t_1 + ta(s_1)$), the system performs an *internal transition*, taking it to a new state s_2. The new state is calculated as $s_2 = \delta_{\text{int}}(s_1)$. Function δ_{int} ($\delta_{\text{int}} : S \to S$) is called the *internal transition function*.

When the state changes its value from s_1 to s_2, an output event is produced with the value $y_1 = \lambda(s_1)$. Function λ ($\lambda : S \to Y$) is called the *output function*. In this way, the functions ta, δ_{int} and λ define the autonomous behavior of a DEVS model.

When an input event arrives, the state changes instantaneously. The new state value depends not only on the value of the input event, but also on the previous state value and the elapsed time since the last transition. If the system assumes the state value s_2 at time t_2, and subsequently, an input event arrives at time $t_2 + e < ta(s_2)$ with value x_1, the new state is calculated as $s_3 = \delta_{\text{ext}}(s_2, e, x_1)$. In this case, we say that the system performs an *external transition*. Function δ_{ext} ($\delta_{\text{ext}} : S \times \mathbb{R}_0^+ \times X \to S$) is called the *external transition function*. No output event is produced during an external transition.

Let us consider the following simple example: A system receives positive numbers in an asynchronous way. After it received a number x, it generates an output event with the number $x/2$ after $3 \cdot x$ time units. A DEVS model that correctly represents this behavior is the following:

$$
\begin{aligned}
&M_1 = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ where} \\
&X = Y = S = \mathbb{R}^+ \\
&\delta_{\text{int}}(s) = \infty \\
&\delta_{\text{ext}}(s, e, x) = x \\
&\lambda(s) = s/2 \\
&ta(s) = 3 \cdot s
\end{aligned}
$$

Observe that the state can assume a time advance equal to ∞. When this occurs, we say that the system is in a passive state, since it will no longer change its state, unless and until it receives an input event.

Let us analyze what happens with the model M_1 when it receives an input event trajectory. Consider for instance that input events occur at times $t = 1$, $t = 3$, and $t = 10$ with the values 2, 1, and 5, respectively. Suppose that initially we have $t = 0$, $s = \infty$ and $e = 0$. Then, the following behavior would be observed:

time $t = 0$:
$s = \infty$
$e = 0$
$ta(s) = ta(\infty) = \infty$

time $t = 1^-$:
$s = \infty$
$e = 1$

time $t = 1$:
$s = \delta_{\text{ext}}(s, e, x) = \delta_{\text{ext}}(\infty, 1, 2) = 2$

time $t = 1^+$:
$s = 2$
$e = 0$
$ta(s) = ta(2) = 6$

time $t = 3^-$:
$s = 2$
$e = 2$

time $t = 3$:
$s = \delta_{\text{ext}}(s, e, x) = \delta_{\text{ext}}(2, 2, 1) = 1$

time $t = 3^+$:
$s = 1$
$e = 0$

$ta(s) = ta(1) = 3$

time $t = 6$:
$\lambda(s) = \lambda(1) = 0.5$
$s = \delta_{\text{int}}(s) = \delta_{\text{int}}(1) = \infty$

time $t = 6^+$:
$s = \infty$
$e = 0$
$ta(s) = ta(\infty) = \infty$

time $t = 10^-$:
$s = \infty$
$e = 4$

time $t = 10$:
$s = \delta_{\text{ext}}(s, e, x) = \delta_{\text{ext}}(\infty, 4, 5) = 5$

time $t = 10^+$:
$s = 5$
$e = 0$
$ta(s) = ta(5) = 15$

time $t = 25$:
$\lambda(s) = \lambda(5) = 2.5$
$s = \delta_{\text{int}}(s) = \delta_{\text{int}}(5) = \infty$

time $t = 25^+$:
$s = \infty$
$e = 0$
$ta(s) = ta(\infty) = \infty$

According to the above model, when a new state arrives through an input event before the previous state has expired, the system assumes the new state value and forgets the previous one. In the above example, this happens at time $t = 3$. A different scenario might require that arriving input events are to be ignored, while the system is busy. This modified behavior can be generated using the following DEVS model:

$$
\begin{aligned}
&M_2 = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ where} \\
&X = Y = \mathbb{R}^+ \\
&S = \mathbb{R}^+ \times \mathbb{R}_0^+ \\
&\delta_{\text{int}}(s) = \delta_{\text{int}}(z, \sigma) = (\infty, \infty) \\
&\delta_{\text{ext}}(s, e, x) = \delta_{\text{ext}}(z, \sigma, e, x) = \tilde{s} \\
&\lambda(s) = \lambda(z, \sigma) = z/2 \\
&ta(s) = ta(z, \sigma) = \sigma
\end{aligned}
$$

where:

$$\tilde{s} = \begin{cases} (x, 3 \cdot x) & \text{if } z = \infty \\ (z, \sigma - e) & \text{otherwise} \end{cases}$$

In this new model, we included the variable σ in the state s. People working routinely with DEVS almost always introduce the variable σ, set equal to the time advance, as this generally facilitates the modeling task.

11.4 Coupled DEVS Models

As mentioned before, DEVS is a general formalism that can be used to describe highly complex systems. However, the representation of a complex system based only on transition and time advance functions is rather difficult. The reason is that in those functions we have to imagine and describe all possible situations in the system.

Complex systems can usually be thought of as the coupling of simpler systems. Through the coupling, the output events of some subsystems are converted to input events of other subsystems. The DEVS methodology guarantees that the coupling of atomic DEVS models defines new DEVS models, i.e., DEVS is closed under coupling. For this reason, complex systems can be represented by DEVS in a hierarchical fashion [11.11].

There are two different ways, in which DEVS models may be coupled. The first approach is the most general one. It uses *translation functions* between subsystems. The second approach is based on the use of input and output ports. We shall use the latter approach, since it is simpler and more adequate in the context of continuous system simulation.

The use of ports requires adding to the input and output events a new number, word, or symbol, representing the port, through which the event is arriving. It suffices to enumerate the connections describing the couplings between different systems. An *internal connection* involves an input and an output port belonging to subsystems. However in the context of hierarchical coupling, there also exist connections from the output ports of subsystems to the output ports of the system. These are called *external output connections*. There also exist connections from the input ports of the systems to input ports of subsystems. These are referred to as *external input connections*.

Figure 11.6 shows a coupled DEVS model N that is the result of coupling the models M_a and M_b. There, the output port 1 of M_a is connected to the input port 0 of M_b. This connection can be represented by the pair $[(M_a, 1), (M_b, 0)]$. Other connections are $[(M_b, 0), (M_a, 0)]$, $[(N, 0), (M_a, 0)]$, $[(M_b, 0), (N, 1)]$, etc. According to the closure property of DEVS, the model N can itself be used in exactly the same way, as an atomic DEVS model would be used, and it can be coupled with other atomic and/or coupled models.

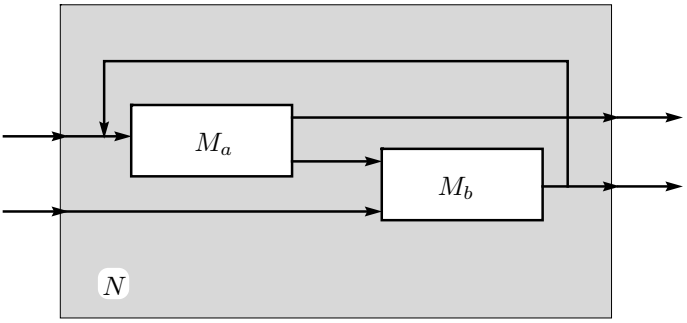

FIGURE 11.6. Coupled DEVS model.

Note that the input and output ports are numbered using integer numbers starting from 0. The DEVS methodology allows using any word to identify a port. However in the context of this book, we shall always use integer numbers starting from 0, because we shall work with a software tool that defines the ports in this fashion [11.7].

Consider for example a system that calculates a static function $f(u_0, u_1)$, where u_0 and u_1 are real–valued piecewise constant trajectories generated by other subsystems. We can represent piecewise constant trajectories by sequences of events, if we relate each event to a change in the trajectory value. Using this idea, we can build the following atomic DEVS model:

$$
\begin{aligned}
&M_3 = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ where} \\
&X = Y = \mathbb{R} \times \mathbb{N}_0 \\
&S = \mathbb{R}^2 \times \mathbb{R}_0^+ \\
&\delta_{\text{int}}(s) = \delta_{\text{int}}(u_0, u_1, \sigma) = (u_0, u_1, \infty) \\
&\delta_{\text{ext}}(s, e, x) = \delta_{\text{ext}}(u_0, u_1, \sigma, e, x_v, p) = \tilde{s} \\
&\lambda(s) = \lambda(u_0, u_1, \sigma) = (f(u_0, u_1), 0) \\
&ta(s) = ta(u_0, u_1, \sigma) = \sigma
\end{aligned}
$$

where:

$$
\tilde{s} = \begin{cases} (x_v, u_1, 0) & \text{if } p = 0 \\ (u_0, x_v, 0) & \text{otherwise} \end{cases}
$$

Here, each input and output event includes an integer number, indicating the corresponding input or output port. In the input events (cf. the definition of $\tilde{s}$ in function δ_{ext}), the port p can be either 0 or 1. In the output events (cf. function λ), the output port is always 0.

Now, if we want to represent another system that calculates the function $f[f(u_0, u_1), u_2]$, we must couple two models of the M_3 class with a connection from the output port of the first subsystem to the input port

0 of the second subsystem. The system output must be taken from the second model. Thus, calling the subsystems A and B, respectively, and the overall system N, the connections can be expressed as: $[(A,0),(B,0)]$, $[(N,0),(A,0)]$, $[(N,1),(A,1)]$, $[(N,2),(B,1)]$, and $[(B,0),(N,0)]$.

The DEVS methodology uses a formal structure for representing coupled DEVS models with ports. The structure includes the subsystems, the connections, the system input and output sets, and a *tie–breaking function* to govern the occurrence of *simultaneous events*. The connections are divided into three sets: one set composed by the connections between subsystems (internal connections), another set that contains the connections from the system to the subsystems (external input connections), and a final set that lists the connections from the subsystems to the system (external output connections).

The use of the aforementioned tie–breaking function can be avoided with *Parallel–DEVS*, which is an extension of the DEVS formalism that allows dealing with simultaneous events.

We shall not develop these latter concepts, neither the coupled DEVS formal structure nor the parallel–DEVS formalism, any further, since our aim is not the introduction of the complete DEVS methodology here. We are only interested in using DEVS as a tool for continuous system simulation. For a more complete coverage of DEVS methodology, we refer the reader to Zeigler's book [11.11].

11.5 Simulation of DEVS Models

One of the most important features of DEVS is that very complex models can be simulated in an easy and efficient manner.

DEVS models can be simulated with a simple ad–hoc program written in any language. In fact, the simulation of a DEVS model is not much more complicated than that of a discrete–time model.

A basic algorithm that may be used for the simulation of a coupled DEVS model can be described by the following steps:

(a). Identify the atomic model that, according to its time advance and elapsed time, is the next to perform an internal transition. Call the system d^*, and let t_n be the time of the aforementioned transition.

(b). Advance the simulation clock t to $t = t_n$, and execute the internal transition function of model d^*.

(c). Propagate the output event produced by d^* to all atomic models connected to it through its output ports, while executing the corresponding external transition functions. Then, return to step (a) above.

One of the simplest ways for implementing these steps is by writing a program with a hierarchical structure equivalent to the hierarchical structure of the model to be simulated. This is the method developed in [11.11], where a routine called *DEVS–simulator* is associated with each atomic DEVS model, and a different routine called *DEVS–coordinator* is related to each coupled DEVS model. At the top of the hierarchy, there is a routine called *DEVS–root–coordinator* that manages the global simulation time. Figure 11.7 illustrates this simulation technique for a coupled DEVS model:

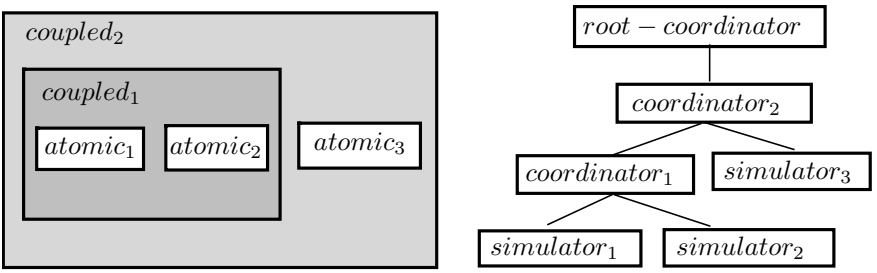

FIGURE 11.7. Hierarchical model and simulation scheme.

The simulators and coordinators of consecutive layers communicate with each other through messages. The coordinators send messages to their children, triggering the execution of their transition functions. When a simulator goes through a transition, it calculates its next state and, when the transition is internal, sends the output value to its parent coordinator. The simulator state coincides with its associated atomic DEVS model state.

When a coordinator executes a transition function, it sends messages to some of its children, triggering the execution of their own transition functions. When an output event produced by one of its children has to be propagated outside the coupled model, the coordinator sends a message to its own parent coordinator, carrying the output value.

Each simulator or coordinator has a local variable t_{n_i}, indicating the time instant, when its next internal transition is scheduled to occur. In a simulator, the value of that variable is calculated using the time advance function of the corresponding atomic model. In a coordinator, it is calculated as the minimum t_{n_i} of its children. Thus, the t_{n_i} of the coordinator at the root of the tree is the time instant, at which the next event of the entire system will occur. The root coordinator is responsible for advancing the global time t to that value.

At the beginning of the simulation, a message of initialization is sent from the root to the leaves of the tree structure.

The following pseudo–code corresponds to a simulator associated with a generic atomic model. There, the i–message, $*$–message, and x–message represent the initialization, internal transition, and input message, respectively. These messages are sent from a parent to its children. Similarly, the

y–message is an output message, sent from a child to its parent.

DEVS–simulator
  variables:
    t_l // time of last event
    t_n // time of next event
    s // state of the DEVS atomic model
    e // elapsed time in the actual state
    $y = (y.value, y.port)$ // current output of the DEVS atomic model
  when i–message (i, t) is received at time t
    $t_l = t - e$
    $t_n = t_l + ta(s)$
  when $*$–message $(*, t)$ is received at time t
    $y = \lambda(s)$
    send y–message (y, t) to parent coordinator
    $s = \delta_{\text{int}}(s)$
    $t_l = t$
    $t_n = t + ta(s)$
  when x–message (x, t) is received at time t
    $e = t - t_l$
    $s = \delta_{\text{ext}}(s, e, x)$
    $t_l = t$
    $t_n = t + ta(s)$
end DEVS–simulator

The routine corresponding to a coordinator can be written as follows:

DEVS–coordinator
  variables:
    t_l // time of last event
    t_n // time of next event
    $y = (y.value, y.port)$ // current output of the DEVS coordinator
    D // list of children
    IC // list of connections of the form $[(d_i, port_x), (d_j, port_y)]$
    EIC // list of connections of the form $[(N, port_x), (d_j, port_y)]$
    EOC // list of connections of the form $[(d_i, port_x), (N, port_y)]$
  when i–message (i, t) is received at time t
    send i–message (i, t) to all children
    $d^* = \arg[\min_{d \in D}(d.t_n)]$
    $t_l = t$
    $t_n = d^*.t_n$
  when $*$–message $(*, t)$ is received at time t
    send $*$–message $(*, t)$ to d^*
    $d^* = \arg[\min_{d \in D}(d.t_n)]$
    $t_l = t$
    $t_n = d^*.t_n$
  when x–message $((x.value, x.port), t)$ is received at time t
    $(v, p) = (x.value, x.port)$
    for each connection $[(N, p), (d, q)]$
      send x–message $((v, q), t)$ to child d
    $d^* = \arg[\min_{d \in D}(d.t_n)]$
    $t_l = t$
    $t_n = d^*.t_n$

```
    when y–message ((y.value, y.port),t) is received from d*
        if a connection [(d*, y.port), (N, q)] exists
            send y–message ((y.value, q),t) to parent coordinator
        for each connection [(d*, y.port), (d, q)]
            send x–message ((y.value, q),t) to child d
end DEVS–coordinator
```

Finally, the root coordinator executes the following routine:

```
DEVS–root–coordinator
    variables:
        t // global simulation time
        d // child (coordinator or simulator)
    t = t0
    send i–message (i,t) to d
    t = d.tn
    loop
        send *–message (*,t) to d
        t = d.tn
    until end of simulation
end DEVS–root–coordinator
```

There are many other possibilities for implementing a simulation engine for DEVS models. The main problem with the routines outlined is that, due to their hierarchical structure, we may observe a significant traffic of messages passing from higher to lower layers of the architecture. All of these messages and their corresponding computational time can be avoided if a flat simulation structure is being used. Hierarchical DEVS simulation architectures can be converted to flat DEVS simulation architectures fairly easily [11.4]. In fact, most of the software tools mentioned before implement the simulation based on a flat code.

Although the implementation of the pseudo code shown above is fairly straightforward, practical models are usually composed of many subsystems, and therefore, ad–hoc programming of all of these models may become very time–consuming.

In recent years, a number of different software tools for the simulation of DEVS models have been developed. Some of these tools offer software libraries, graphical user interfaces, and a variety of other facilities that are designed to support the user in the modeling task.

A number of software packages for DEVS simulation have been placed in the public domain, including DEVS–Java [11.13], DEVSim++ [11.5], DEVS–C++ [11.1], CD++ [11.10], and JDEVS [11.2].

In this book, we shall focus on PowerDEVS [11.7, 11.8, 11.9], a software that –in spite of being a general–purpose DEVS simulator– is a software environment that was specifically conceived for facilitating the simulation of continuous systems.

As we already mentioned, this textbook is not geared towards a general

course on DEVS, and we do not expect the reader to become an expert on DEVS. Our aim is to provide enough information about DEVS, such that a PowerDEVS user will understand enough of the underlying principles to be able to make use of PowerDEVS as a tool for continuous system simulation.

PowerDEVS is a tool that was designed with many different kinds of users in mind, ranging from mere beginners, who don't know anything about either DEVS or C++ programming, to experts in both domains.

PowerDEVS offers a convenient *graphical user interface* that permits creating coupled DEVS models using the typical drag and drop tools. A number of DEVS atomic model definitions have been predefined and stored in a PowerDEVS model library.

Atomic models can be easily created and modified using an *atomic model editor*, where the user has to define the transition, output, and time advance functions using C++ syntax.

11.6 DEVS and Continuous System Simulation

In the first example of section 11.4, we saw that piecewise constant trajectories can be represented by sequences of events. This simple idea constitutes the basis for the use of DEVS in the simulation of continuous systems.

In that example, we also showed that a DEVS model can represent the behavior of a static function with piecewise constant input trajectories. The problem is that the continuous system trajectories are usually far from being piecewise constant. However, we can alter the system, such that it exhibits these kinds of trajectories. In fact, that is what we did to the system of Eq.(11.3), where we used the floor function to convert it to the system of Eq.(11.4).

We can split Eq.(11.4) in the following way:

$$\dot{x}(t) = d_x(t) \tag{11.6a}$$

$$q(t) = \text{floor}[x(t)] \tag{11.6b}$$

and:

$$d_x(t) = -q(t) + u(t) \tag{11.7}$$

where $u(t) = 10 \cdot \varepsilon(t - 1.76)$.

We can represent this system using the block diagram of Fig.11.8.

As we mentioned before, the subsystem of Eq.(11.7) –being a static function– can be represented by the DEVS model M_3 presented in Section 11.4.

The subsystem of Eq.(11.6) is a dynamic system having a piecewise constant input trajectory $d_x(t)$ and a piecewise constant output trajectory $q(t)$. We can represent it exactly using the following DEVS model:

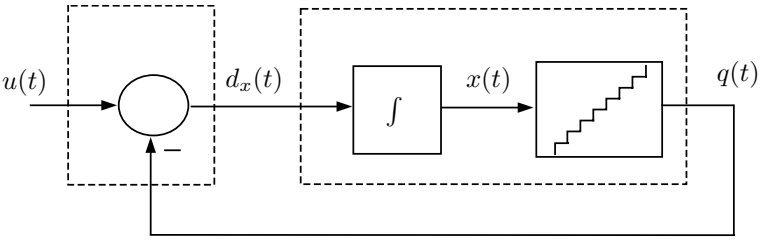

FIGURE 11.8. Block diagram representation of Eqs.(11.6–11.7).

$$
\begin{aligned}
&M_4 = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ where}\\
&X = Y = \mathbb{R} \times \mathbb{N}\\
&S = \mathbb{R}^2 \times \mathbb{Z} \times \mathbb{R}_0^+\\
&\delta_{\text{int}}(s) = \delta_{\text{int}}(x, d_x, q, \sigma) = (x + \sigma \cdot d_x, d_x, q + \text{sign}(d_x), \frac{1}{|d_x|})\\
&\delta_{\text{ext}}(s, e, x) = \delta_{\text{ext}}(x, d_x, q, \sigma, e, x_v, p) = (x + e \cdot d_x, x_v, q, \tilde{\sigma})\\
&\lambda(s) = \lambda(x, d_x, q, \sigma) = (q + \text{sign}(d_x), 0)\\
&ta(s) = ta(x, d_x, q, \sigma) = \sigma
\end{aligned}
$$

where:

$$
\tilde{\sigma} = \begin{cases} \frac{q+1-x}{x_v} & \text{if } x_v > 0 \\ \frac{q-x}{x_v} & \text{if } x_v < 0 \\ \infty & \text{otherwise} \end{cases}
$$

Now, if we want to simulate the system of Eqs.(11.6–11.7) using PowerDEVS, we can translate the generic DEVS atomic models, M_3 and M_4, into corresponding PowerDEVS atomic models.

A PowerDEVS atomic model corresponding to M_3 looks, in the atomic model editor, as follows:

ATOMIC MODEL STATIC1
State Variables and Parameters:
float u[2],$sigma$; //states
float y; //output
float inf; //parameters

Init Function:
$inf = 1e10$;
$u[0] = 0$;
$u[1] = 0$;
$sigma = inf$;
$y = 0$;

Time Advance Function:
return $sigma$;

Internal Transition Function:
$sigma = inf$;

External Transition Function:
float xv;
xv = *(float*)($x.value$);
$u[x.port] = xv$;
$sigma = 0$;

Output Function:
$y = u[0] - u[1]$;
return Event(&y,0);

The conversion of the DEVS model M_3 to the corresponding PowerDEVS model STATIC1 is straightforward.

Similarly, the DEVS model M_4 can be represented in PowerDEVS as follows:

ATOMIC MODEL NHINTEGRATOR

State Variables and Parameters:
float X, dX, q, $sigma$; //states
//we use capital X because x is reserved
float y; //output
float inf; //parameters

Init Function:
va_list $parameters$;
va_start($parameters$, t);
X = va_arg($parameters$, double);
$dX = 0$;
q =floor(X);
$inf = 1e10$;
$sigma = 0$;
$y = 0$;

Time Advance Function:
return $sigma$;

Internal Transition Function:
$X = X + sigma * dX$;
if ($dX > 0$) {
$sigma = 1/dX$;
$q = q + 1$;
}
else {
if ($dX < 0$) {
$sigma = -1/dX$;
$q = q - 1$;

```
        }
    else {
        sigma = inf;
    };
};
```

External Transition Function:

```
float xv;
xv = *(float*)(x.value);
X = X + dX * e;
if (xv > 0) {
    sigma = (q + 1 - X)/xv;
    }
else {
    if (xv < 0) {
        sigma = (q - X)/xv;
        }
    else {
        sigma = inf;
    };
};
dX = xv;
```

Output Function:

```
if (dX == 0) {y = q;} else {y = q + dX/fabs(dX);};
return Event(&y,0);
```

Again, the translation was fairly direct. However, we added a few new items to the init function. The first two lines are automatically included by the atomic model editor, when a new model is being edited. They declare a variable *parameters*, where the graphical user interface puts the model parameters. In our case, we defined the initial condition in variable X as a parameter. The third line in the init function just takes the first parameter of the block and places it in X.

Then in the graphical user interface, we can just double click on the block and change the value of that parameter (i.e., we can change the initial condition without changing the atomic model definition).

The other change with respect to model M_4 is also related to the inclusion of initial conditions. At the beginning of the simulation, we force the model to provoke an event, so that the initial value of the corresponding quantized variable q becomes known to the rest of the system. We shall write more about this topic in the next chapter.

The coupled PowerDEVS model generated using the graphical model editor is shown in Fig.11.9.

In that model, beside from the atomic models STATIC1 and NHINTEGRATOR, we included three more blocks: a STEP block that produces an event with value 10 at time $t = 1.76$, and two additional models that save and display the simulation results. These last two models are being

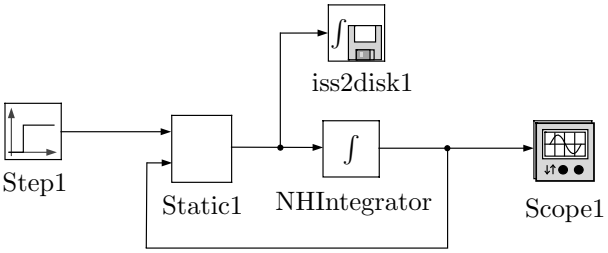

FIGURE 11.9. Coupled PowerDEVS model of Eqs.(11.6–11.7).

included with the standard PowerDEVS library.

Using the PowerDEVS model of Fig.11.9, we obtained the data plotted in Fig.11.1. The plot of that figure was generated by MATLAB, using the data saved in an appropriate format on a file by the PowerDEVS block iss2dsk.

The subsystem of Eq.(11.6) corresponds to the integrator together with the staircase block in the block diagram of Fig.11.8. It is equivalent to DEVS model M_4, represented by the NHINTEGRATOR model in PowerDEVS.

This is, what Zeigler called the *quantized integrator* [11.12, 11.11]. There, the function floor acts as a *quantization function*. A quantization function maps real–valued numbers onto a discrete set of real values.

A system that relates its input and output by any type of quantization function shall henceforth be called a *quantizer*. Thus, our staircase block is a particular case of a quantizer with uniform quantization.

A quantized integrator is an integrator concatenated with a quantizer that may employ either a uniform or a non–uniform quantization scheme.

Similarly, model M_3 models a static function with its input vector in $\mathbb{R}^2$. The corresponding STATIC1 PowerDEVS model implements a particular case of such a static function, namely the function: $f(u_0, u_1) = u_0 - u_1$. A DEVS model for generic static functions with their input vector in $\mathbb{R}^n$ can easily be built and programmed in PowerDEVS (cf. Hw.[H11.4] for the general linear case).

In the same way, we can obtain a DEVS model representation of general quantized integrators that can be employed, whenever their input trajectories are piecewise constant, and it is also evident that we can build a generic DEVS model of an arbitrary static function, as long as its input trajectories are piecewise constant.

Thus, if we have a general time–invariant system[1]:

[1]We shall use x_a to denote the state variables of the original system, so that $x_a(t)$ is the analytical solution.

$$\begin{aligned} \dot{x}_{a_1} &= f_1(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \\ \dot{x}_{a_2} &= f_2(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \\ &\vdots \\ \dot{x}_{a_n} &= f_n(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \end{aligned} \tag{11.8}$$

with piecewise constant input functions $u_j(t)$, we can transform it into:

$$\begin{aligned} \dot{x}_1 &= f_1(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\ \dot{x}_2 &= f_2(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\ &\vdots \\ \dot{x}_n &= f_n(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \end{aligned} \tag{11.9}$$

where $q_i(t)$ is related to $x_i(t)$ by some quantization function.

The variables q_i are called *quantized variables*. This system of equations can be represented by the block diagram of Fig.11.10, where **q** and **u** are the vectors formed by the quantized variables and input variables, respectively.

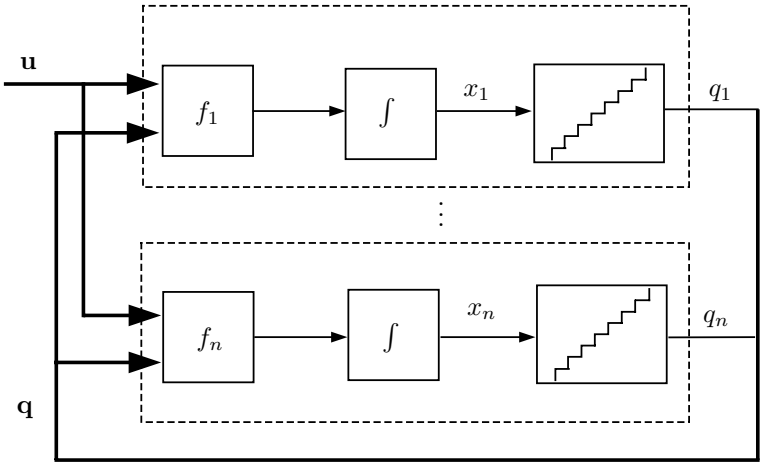

FIGURE 11.10. Block diagram representation of Eq.(11.9).

Each subsystem in Fig.11.10 can be represented by a DEVS model exactly, since all of the subsystems are composed either by a static function or by a quantized integrator. These DEVS models can then be coupled, and, due to the aforementioned closure under coupling property, the coupled system also forms a DEVS model.

Thus, when a system is modified by adding quantizers to the outputs of all integrators, the resulting system is equivalent to a coupled DEVS model that can be simulated, assuming that all of the input functions are piecewise constant as well.

This idea formed the first approximation to a discrete event–based method for continuous system simulation. With this method, we can simulate exactly –ignoring round–off errors– the system of Eq.(11.9), which seems to be a reasonable approximation to that of Eq.(11.8), while avoiding any kind of time discretization. The time discretization was replaced by the quantization of the state variables.

Unfortunately, there is a problem with the *legitimacy* of the resulting DEVS model. A DEVS model is said to be legitimate if it cannot perform an infinite number of transitions in a finite interval of time [11.11].

Although it can be easily verified that the subsystems in Fig.11.10 are legitimate, the legitimacy property is not closed under coupling.

In fact, this simulation method will lead to illegitimate DEVS models in most cases. The simulation of an illegitimate DEVS model gets stuck, when the number of state transitions per time unit grows to infinity.

The reason for the illegitimacy of the DEVS model is related to the solution of Eq.(11.9). There, the trajectories of $q_i(t)$ are not necessarily piecewise constant. Sometimes, they can exhibit an infinite number of state changes within a finite time interval, which produces an infinite number of events in the corresponding DEVS model. Due to this problem, we cannot claim that the use of a simple quantization in the state variables constitutes a general method for the simulation of continuous systems.

We can observe this problem in the system of Eqs.(11.6–11.7) by changing the input function to $u(t) = 10.5 \cdot \varepsilon(t - 1.76)$. The trajectories until time $t = 1.76$ are exactly the same as those shown in Fig.11.1. Once the step has been applied, the trajectory starts growing a bit faster than shown in Fig.11.1. When $x(t) = q(t) = 10$, the state derivative doesn't become zero, however. Instead, the trajectory continues to grow with a slope of $\dot{x}(t) = 0.5$. Then after 2 more time units, we obtain $x(t) = q(t) = 11$. At this point in time, the slope becomes negative. $x(t)$ now decreases with a slope of $\dot{x}(t) = -0.5$. Thus, $q(t)$ immediately returns to 10, the state derivative becomes again positive, and $x(t)$ starts growing again. We obtain a cyclic behavior with infinite frequency.

This anomalous and annoying behavior can also be observed in the resulting DEVS model. When the DEVS model corresponding to the integrator performs an internal transition, it produces an output event that represents the change in $q(t)$. This event is propagated through the internal feed-back loop (cf. Figs. 11.8–11.9), and produces a new external transition in the integrator that changes the time advance to zero. Consequently, the integrator undergoes another internal transition, and the cycle continues forever.

The reader may wonder why we introduced a method that only works in a very few cases. Yet, we had a very good reason for doing so. It turns out that, by adding only a small and very simple modification to the method, a general simulation method can indeed be designed that is based on the previously introduced simulation approach, yet avoids the illegitimacy prob-

lem that has plagued us so far. This new method is called *quantized state systems method* (*QSS method* for short) [11.6], and we shall dedicate the final part of this chapter to introducing this new simulation algorithm. The study of its theoretical and practical properties as well as some of its extensions shall be left to the next and final chapter of this book.

11.7 Quantized State Systems

If we try to analyze the infinitely fast oscillations in the system of Eqs.(11.6–11.7), we can see that they are caused by the changes in $q(t)$. An infinitesimally small variation in $x(t)$ can produce, due to the quantization, a significant oscillation with an infinitely fast frequency in $q(t)$.

A possible solution might consist in adding some delay after a change in $q(t)$ to avoid those infinitely fast oscillations. However, adding such delays is equivalent, in some way, to introducing time discretization. During the delays, we lose control over the simulation, and we have to deal with the problems associated with discrete–time algorithms once again.

A different solution is based on the use of hysteresis in the quantization. If we add hysteresis to the relationship between $x(t)$ and $q(t)$, the oscillations in $q(t)$ can only be produced by *large* oscillations in $x(t)$ that cannot occur instantaneously, as long as the state derivatives remain finite.

Therefore, before introducing the QSS method formally, we shall define the concept of a *hysteretic quantization function*.

Let $Q = \{Q_0, Q_1, ..., Q_r\}$ be a set of real numbers, where $Q_{k-1} < Q_k$ with $1 \leq k \leq r$. Let Ω be the set of piecewise continuous real–valued trajectories, and let $x \in \Omega$ be a continuous trajectory. Let b be a mapping $b : \Omega \to \Omega$, and let $q = b(x)$, where the trajectory q satisfies:

$$q(t) = \begin{cases} Q_m & \text{if } t = t_0 \\ Q_{k+1} & \text{if } x(t) = Q_{k+1} \quad \wedge \quad q(t^-) = Q_k \quad \wedge \quad k < r \\ Q_{k-1} & \text{if } x(t) = Q_k - \varepsilon \quad \wedge \quad q(t^-) = Q_k \quad \wedge \quad k > 0 \\ q(t^-) & \text{otherwise} \end{cases} \tag{11.10}$$

and:

$$m = \begin{cases} 0 & \text{if } x(t_0) < Q_0 \\ r & \text{if } x(t_0) \geq Q_r \\ j & \text{if } Q_j \leq x(t_0) < Q_{j+1} \end{cases}$$

Then, the map b is a hysteretic quantization function.

The discrete values Q_i are called *quantization levels*, and the distance $Q_{k+1} - Q_k$ is defined as the *quantum*, which is usually constant. The width of the hysteresis window is ε. The values Q_0 and Q_r are the lower and upper saturation values. Figure 11.11 shows a typical quantization function with uniform quantization intervals.

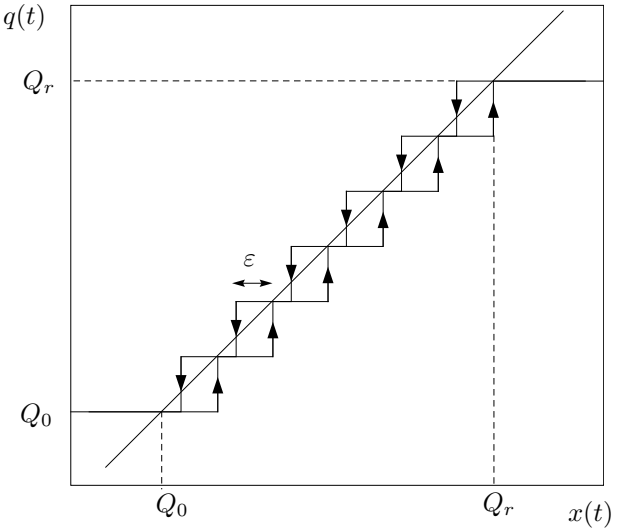

FIGURE 11.11. Quantization function with hysteresis.

Now, we are ready to define the QSS method:

> Given a system such as that of Eq.(11.8), the QSS method transforms the system to a system similar to that of Eq.(11.9), where the variables $x_i(t)$ and $q_i(t)$ are related by hysteretic quantization functions. The resulting system is called a *quantized state system (QSS)*.

In [11.6], it is shown that the quantized and state variable trajectories of Eq.(11.9) are always piecewise constant and piecewise linear, respectively. Hence a QSS can be simulated exactly by a legitimate DEVS model.

A legitimate DEVS model can be built as the coupling of subsystems corresponding to static functions and *hysteretic quantized integrators*.

The hysteretic quantized integrators are quantized integrators, where the simple memoryless quantization functions have been replaced by hysteretic quantization functions. This is equivalent to replacing Eq.(11.6b) by Eq.(11.10) in the system of Eq.(11.6).

With this modification, the hysteretic quantized integrator constituted by Eq.(11.6a) and Eq.(11.10) can be represented by the DEVS model:

$$
\begin{aligned}
&M_5 = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ where} \\
&X = Y = \mathbb{R} \times \mathbb{N} \\
&S = \mathbb{R}^2 \times \mathbb{Z} \times \mathbb{R}_0^+ \\
&\delta_{\text{int}}(s) = \delta_{\text{int}}(x, d_x, k, \sigma) = (x + \sigma \cdot d_x, d_x, k + \text{sign}(d_x), \sigma_1)
\end{aligned}
$$

$$\delta_{\text{ext}}(s, e, x_u) = \delta_{\text{ext}}(x, d_x, k, \sigma, e, x_v, p) = (x + e \cdot d_x, x_v, k, \sigma_2)$$
$$\lambda(s) = \lambda(x, d_x, k, \sigma) = (Q_{k+\text{sign}(d_x)}, 0)$$
$$ta(s) = ta(x, d_x, k, \sigma) = \sigma$$

where:

$$\sigma_1 = \begin{cases} \dfrac{Q_{k+2} - (x + \sigma \cdot d_x)}{d_x} & \text{if } d_x > 0 \\ \dfrac{(x + \sigma \cdot d_x) - (Q_{k-1} - \varepsilon)}{|d_x|} & \text{if } d_x < 0 \\ \infty & \text{if } d_x = 0 \end{cases}$$

and:

$$\sigma_2 = \begin{cases} \dfrac{Q_{k+1} - (x + e \cdot d_x)}{x_v} & \text{if } x_v > 0 \\ \dfrac{(x + e \cdot d_x) - (Q_k - \varepsilon)}{|x_v|} & \text{if } x_v < 0 \\ \infty & \text{if } x_v = 0 \end{cases}$$

The QSS method then consists in choosing the quantization levels (Q_0, Q_1, ..., Q_r) and the hysteresis width ε to be used in each state variable. This choice automatically defines DEVS models of the M_5 class for each resulting hysteretic quantized integrator. Representing the static functions $f_1, \ldots, f_n$ with different DEVS models similar to M_3 and coupling them, the system of Eq.(11.9) can be exactly simulated (ignoring round–off problems). As mentioned above, the resulting coupled DEVS model is legitimate, and the simulation will consume a finite amount of time.

Figure 11.12 shows the block diagram representation of a generic QSS.

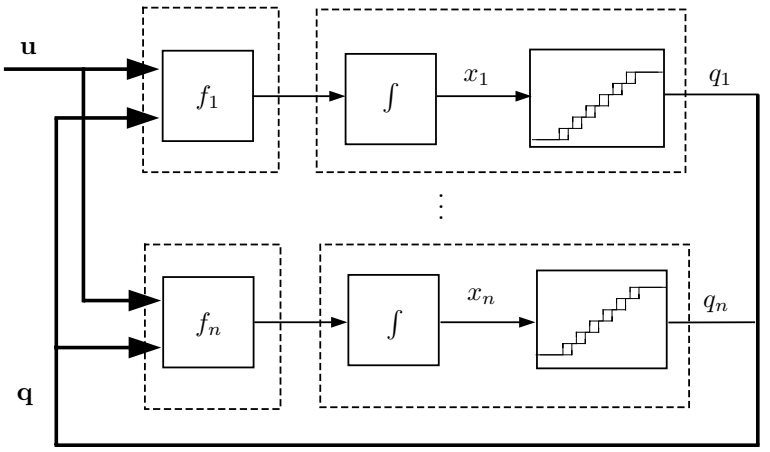

FIGURE 11.12. Block diagram representation of a QSS.

The hysteretic quantized integrator M_5 can be implemented in PowerDEVS as follows:

ATOMIC MODEL HINTEGRATOR
State Variables and Parameters:
float X, dX, q, $sigma$; //states
float y; //output
float $epsilon$, inf; //parameters

Init Function:
va_list $parameters$;
va_start($parameters$, t);
dq = va_arg($parameters$, double);
$epsilon$ = va_arg($parameters$, double);
X = va_arg($parameters$, double);
$dX = 0$;
q =floor$(X/dq) * dq$;
$inf = 1e10$;
$sigma = 0$;

Time Advance Function:
return $sigma$;

Internal Transition Function:
$X = X + sigma * dX$;
if $(dX > 0)$ {
$sigma = dq/dX$;
$q = q + dq$;
}
else {
if $(dX < 0)$ {
$sigma = -dq/dX$;
$q = q - dq$;
}
else {
$sigma = inf$;
};
};

External Transition Function:
float xv;
xv = *(float*)($x.value$);
$X = X + dX * e$;
if $(xv > 0)$ {
$sigma = (q + dq - X)/xv$;
}
else {
if $(xv < 0)$ {
$sigma = (q - epsilon - X)/xv$;
}
else {
$sigma = inf$;
};
};

$dX = xv$;

Output Function:
if ($dX == 0$) {$y = q$;} else {$y = q + dq * dX/$fabs(dX);};
return Event(&y,0);

Here we changed some things with respect to model M_5. In this model, we used a uniform quantum ΔQ, and we replaced variable k (the index of the quantization levels) by q (the quantized variable).

It has now become clear how the QSS method can be applied to systems such as those of Eq.(11.8). We only have to build a block diagram in PowerDEVS using atomic models such as HINTEGRATOR and STATIC1.

However, we must not forget that the result that we obtain is the solution of Eq.(11.9). Thus, the accuracy of the simulation will be connected to the similarity between this system and the original system of Eq.(11.8).

Taking into account that the only difference between both systems is the presence of the quantization functions, we expect that the error depends on the size of the quantization intervals. As we shall explain in the next chapter, this is indeed the case, and this dependence will provide us with a rule for choosing the quantization levels and the hysteresis width.

Let us illustrate the method by means of a simple example. Consider the second order system:

$$\begin{aligned} \dot{x}_{a_1}(t) &= x_{a_2}(t) \\ \dot{x}_{a_2}(t) &= 1 - x_{a_1}(t) - x_{a_2}(t) \end{aligned} \tag{11.11}$$

with initial conditions:

$$x_{a_1}(0) = 0, \quad x_{a_2}(0) = 0 \tag{11.12}$$

We shall use a uniform quantum $Q_{k+1} - Q_k = \Delta Q = 0.05$ and a hysteresis width of $\varepsilon = 0.05$ for both state variables.

Thus, the resulting quantized state system:

$$\begin{aligned} \dot{x_1}(t) &= q_2(t) \\ \dot{x_2}(t) &= 1 - q_1(t) - q_2(t) \end{aligned} \tag{11.13}$$

can be simulated using a coupled DEVS model, composed by two atomic models of the M_5 class, corresponding to the quantized integrators, and two atomic models similar to M_3 that calculate the static functions $f_1(q_1, q_2) = q_2$ and $f_2(q_1, q_2) = 1 - q_1 - q_2$. Figure 11.13 represents the coupled system.

Observe that, due to the fact that function f_1 does not depend on variable q_1, there is a connection that is not necessary. Moreover, taking into account that $f_1(q_1, q_2) = q_2$ the subsystem F_1 can be replaced by a direct connection from QI_2 to QI_1. These simplifications can reduce considerably the computational cost of the implementation.

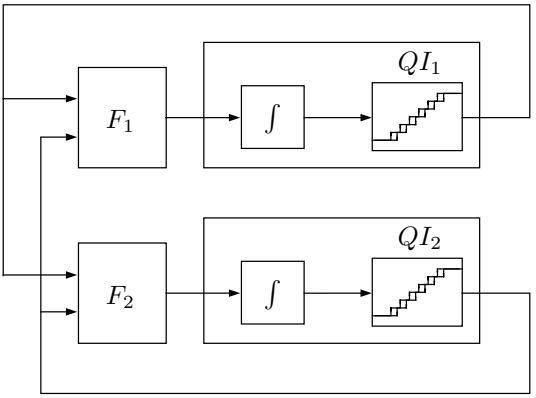

FIGURE 11.13. Block diagram representation of Eq.(11.13).

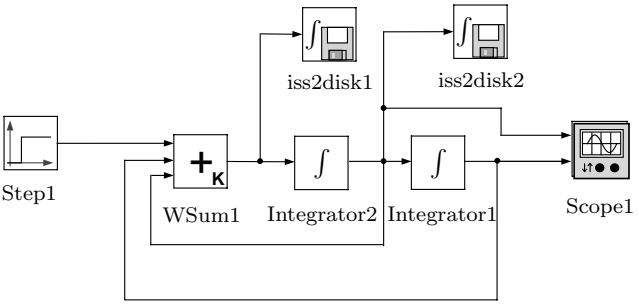

FIGURE 11.14. PowerDEVS model.

In fact, when drawing the PowerDEVS block diagram, we automatically make these simplifications (cf. Fig.11.14).

In the PowerDEVS model of Fig.11.14, there appears a new atomic block that calculates a weighted sum. The reader should be able to imagine, what this block does, and what the hidden DEVS model may look like (cf. Hw.[H11.4]).

The simulation results are shown in Fig.11.15. The first simulation was completed using 30 internal transitions at each quantized integrator, which gives a total of 60 steps. We can see in Fig.11.15 the piecewise linear trajectories of $x_1(t)$ and $x_2(t)$, as well as the piecewise constant trajectories of $q_1(t)$ and $q_2(t)$.

The presence of the hysteresis can be easily noticed where the slope of a state variable changes its sign. Near those points, we can observe different values of q for the same value of x.

The simplifications we mentioned in the connections can be applied to general systems, where few of the static functions do depend on all of the state variables. In this way, the QSS method can exploit the structural properties of the system to reduce the computational burden. When the

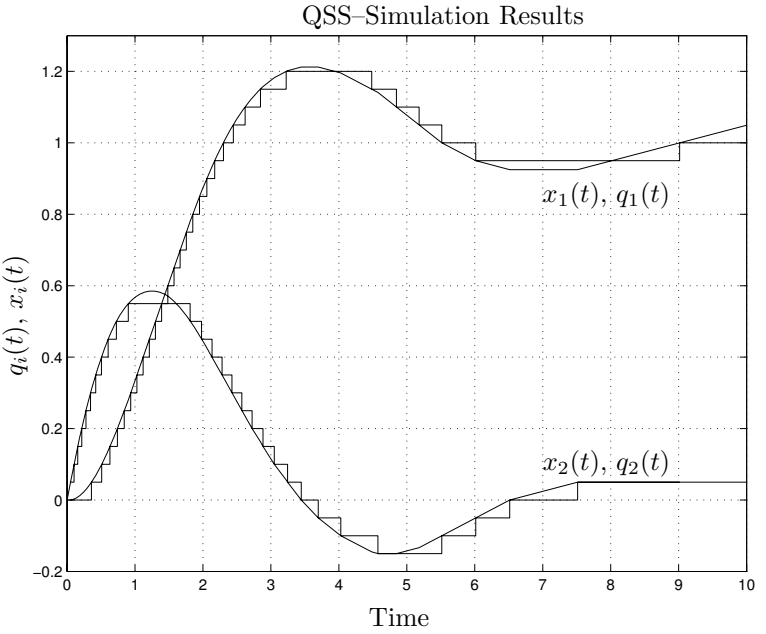

FIGURE 11.15. Trajectories of the system of Eq.(11.13).

system is *sparse*, QSS simulations are particularly efficient, since each step involves calculations at few integrators only.

Discrete–time algorithms can also exploit sparsity properties. However, these techniques require specific sparse matrix algorithms to do so. In the QSS method, the exploitation of sparsity is an intrinsic property.

11.8 Summary

In this chapter, we studied the basic principles of discrete event simulation under the DEVS formalism and their applications to continuous systems simulation.

We introduced the concept of state variable quantization and, based on this concept, we showed how to build a DEVS model that exactly represents the dynamics of general time–invariant continuous systems with quantization in their state variables. We saw that the use of simple memoryless quantization can produce illegitimacy in the DEVS model. We then demonstrated that the use of hysteretic quantization solves this problem. Making use of these techniques, we introduced the QSS method that allows the simulation of general time–invariant continuous system.

In the next chapter, we shall show and discuss the main theoretical properties and practical applications of the QSS method and its extensions. For

now, it suffices to mention that the DSS method and its extension provide efficient simulations of discontinuous systems and sparse problems, and that these techniques are of particular interest in the context of real–time simulation.

We might mention further that quantization–based methods are not the only possible discrete event approaches to continuous system simulation. A different idea is based on the event representation of trajectories and the definition of GDEVS [11.3]. However, this *solution–based* approximation requires the knowledge of the continuous system response to some particular input trajectories, which is not available in most cases. For this reason, GDEVS does not constitute a general continuous simulation method.

For this reason, we shall not introduce GDEVS in this book. Yet, it should be acknowledged that some of the ideas behind GDEVS were used in the design of the second–order accurate QSS2 method that shall be introduced in the next chapter.

Finally, the reader might notice that this chapter does not offer a broad basis of references and bibliographic pointers. The reason for this is simply that discrete event simulation of continuous systems is a fairly recently developed topic. In fact, the first references that we know of are from the late nineties. This implies that these methods are not yet completely developed and optimized, which makes them a fertile field for research.

11.9 References

[11.1] Hyup Cho and Young Cho. *DEVS–C++ Reference Guide*. The University of Arizona, 1997.

[11.2] Jean Baptiste Filippi, Marielle Delhom, and Fabrice Bernardi. The JDEVS Environmental Modeling and Simulation Environment. In *Proceedings of IEMSS 2002*, volume 3, pages 283–288, Lugano, Switzerland, 2002.

[11.3] Norbert Giambiasi, Bruno Escude, and Sumit Ghosh. GDEVS: A Generalized Discrete Event Specification for Accurate Modeling of Dynamic Systems. *Transactions of SCS*, 17(3):120–134, 2000.

[11.4] Kihyung Kim, Wonseok Kang, and Hyungon Seo. Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non–Hierarchical One. In *Proceedings of Annual Simulation Symposium*, 2000.

[11.5] Tag Gon Kim. *DEVSim++ User's Manual. C++ Based Simulation with Hierarchical Modular DEVS Models*. Korean Advanced Institute of Science and Technology, 1994. Available at http://www.acims.arizona.edu/.

[11.6] Ernesto Kofman and Sergio Junco. Quantized State Systems: A DEVS Approach for Continuous System Simulation. *Transactions of SCS*, 18(3):123–132, 2001.

[11.7] Ernesto Kofman, Marcelo Lapadula, and Esteban Pagliero. PowerDEVS: A DEVS–based Environment for Hybrid System Modeling and Simulation. Technical Report LSD0306, LSD, UNR, 2003. Submitted to *Simulation*. Available at http://www.fceia.unr.edu.ar/lsd/powerdevs.

[11.8] Esteban Pagliero, Marcelo Lapadula, and Ernesto Kofman. PowerDEVS. Una Herramienta Integrada de Simulación por Eventos Discretos. In *Proceedings of RPIC03*, volume 1, pages 316–321, San Nicolas, Argentina, 2003.

[11.9] Esteban Pagliero and Marcelo Lapadula. Herramienta Integrada de Modelado y Simulación de Sistemas de Eventos Discretos. Diploma Work. FCEIA, UNR, Argentina, September 2002.

[11.10] Gabriel Wainer, Gastón Christen, and Alejandro Dobniewski. Defining DEVS Models with the CD++ Toolkit. In *Proceedings of ESS2001*, pages 633–637, Marseille, France, 2001.

[11.11] Bernard Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation. Second edition.* Academic Press, New York, 2000.

[11.12] Bernard Zeigler and Jong Sik Lee. Theory of Quantized Systems: Formal Basis for DEVS/HLA Distributed Simulation Environment. In *SPIE Proceedings*, pages 49–58, 1998.

[11.13] Bernard Zeigler and Hessam Sarjoughian. *Introduction to DEVS Modeling and Simulation with JAVA: A Simplified Approach to HLA-Compliant Distributed Simulations.* Arizona Center for Integrative Modeling and Simulation. Available at http://www.acims.arizona.edu/.

[11.14] Bernard Zeigler. *Theory of Modeling and Simulation.* John Wiley & Sons, New York, 1976.

11.10 Bibliography

[B11.1] Christos Cassandras. *Discrete Event Systems: Modeling and Performance Analysis.* Irwin and Aksen, 1993.

[B11.2] Ernesto Kofman. *Discrete Event Simulation and Control of Continuous Systems.* PhD thesis, Universidad Nacional de Rosario, Rosario, Argentina, 2003.

11.11 Homework Problems

[H11.1] Achilles and the Tortoise

Consider the second order system:

$$\begin{aligned} \dot{x}_{a_1} &= -0.5 \cdot x_{a_1} + 1.5 \cdot x_{a_2} \\ \dot{x}_{a_2} &= -x_{a_1} \end{aligned} \tag{H11.1a}$$

Apply the memoryless quantization function:

$$q_i = 2 \cdot \text{floor}(\frac{x_i - 1}{2}) + 1 \tag{H11.1b}$$

to both state variables, and study the solutions of the quantized system:

$$\begin{aligned} \dot{x}_1 &= -0.5 \cdot q_1 + 1.5 \cdot q_2 \\ \dot{x}_2 &= -q_1 \end{aligned} \tag{H11.1c}$$

from the initial condition $x_{a_1} = 0$, $x_{a_2} = 2$.

(a). Show that the simulation time cannot advance more than 5 seconds.

(b). Draw the state–space trajectory $x_1(t)$ *vs.* $x_2(t)$.

[H11.2] DEVS Behavior

Using the DEVS model M_2, repeat the simulation *by hand* that was performed with model M_1 on page 527. Use the same input trajectory and compare the evolution obtained with the evolution of M_1.

[H11.3] DEVS Demultiplexer

The use of ports in DEVS gives rise to a difficulty. After an internal transition took place, a model with ports produces an event that carries a value at one specific output port. This is a limitation, because it is not difficult to imagine a situation, in which the event value contains a vector, and each component should be sent to a different sub–model.

This problem does not appear in the general definition of DEVS (coupling without ports). However, even when using ports, the problem can be solved with the addition of a DEVS model that demultiplexes events.

A DEVS demultiplexer receives input events carrying a vector value through its input port, decomposes the vector into individual scalar values, and sends those out immediately through different output ports.

Build a DEVS demultiplexer that receives events with values in $\mathbb{R}^k$. After receiving an event, the model should send k events through its k output ports, carrying the corresponding scalar component values.

[H11.4] Linear Static Function

Obtain a DEVS atomic model of a static function $f_i : \mathbb{R}^n \to \mathbb{R}$ defined as

$$f(u_0, u_1, \cdots, u_{n-1}) = \sum_{k=0}^{n-1} a_k \cdot u_k \tag{H11.4a}$$

where $a_0, \cdots, a_{n-1}$ are known constants.

Then, program the model in PowerDEVS so that the constants and the number of inputs are parameters.

Hint: You will have to limit the number of inputs to a fixed number (10 for instance).

[H11.5] DEVS Delay Function

Consider a function that represents a fixed delay time T,

$$f(u(t)) = u(t - T) \tag{H11.5a}$$

Consider the input $u(t)$ to be piecewise constant, and obtain a DEVS model of this function.

Create a PowerDEVS block of this function, where the delay time T is a parameter.

Hint: Assume that the number of state changes in $u(t)$ during a time period of T is limited to a fixed number (1000 for instance).

[H11.6] Achilles and the Tortoise Revisited

Modify the PowerDEVS atomic model NHINTEGRATOR of page 537, so that the quantizer satisfies Eq.(H11.1b). Then, use this new atomic model, and build the block diagram corresponding to Eq.(H11.1a). Verify by simulation the prediction made in Hw.[H11.1].

We suggest using a final simulation time such as 4.999 for example.

[H11.7] Varying Quantum and Hysteresis

Obtain the exact solution of the system of Eq.(11.11), and then repeat the QSS simulation using the following quantization and hysteresis:

(a). $\Delta Q_1 = \Delta Q_2 = \varepsilon = 0.01$

(b). $\Delta Q_1 = \Delta q_2 = \varepsilon = 0.05$

(c). $\Delta Q_1 = \Delta q_2 = \varepsilon = 0.1$

(d). $\Delta Q_1 = \Delta Q_2 = \varepsilon = 1$

Compare the results, and use them to hypothesize about the effects of the quantization and hysteresis on error and stability.

11.12 Projects

[P11.1] Grouping Models in the QSS Method

The division between quantized integrators and static functions in building the coupled DEVS model that implements the QSS method simplifies considerably the atomic models.

However, this division is not necessary. Indeed, we already mentioned that DEVS is closed under coupling, and therefore, it must be possible to define a unique atomic DEVS model that simulates the entire system. In this way, the number of events is reduced (we do not have to transmit events between components), and the computational efficiency is improved.

Of course, finding this atomic DEVS model may be quite difficult, and even if we find it, we might lose the possibility of implementing the simulation in a parallel fashion.

An intermediate solution for the QSS method, that probably represents the best compromise, consists in grouping each quantized integrator with the static function that calculates its derivative. In this way, the number of events is reduced to less than the one half.

Using this idea, propose an atomic DEVS model that represents simultaneously a static function and a quantized integrator. Program that model in PowerDEVS, and couple two of these models to simulate the system of Eq.(11.11).

Compare the total number of internal and external transitions performed by this coupled DEVS model with that obtained by simulating the model composed of separate quantized integrators and static functions. Compare also the execution time of the two simulations.

Repeat the experiment with other models, and try to determine, under which conditions the grouping of models yields noticeable advantages.

Conclude on the convenience of using grouped models, taking into account the trade–off between simplicity and execution time.

[P11.2] DEVS and Multi–Rate Integration

Build a PowerDEVS model of a *forward Euler integrator*, i.e., a model that receives input events with scalar values f_k and produces scalar output values:

$$x_{k+1} = x_k + h \cdot f_k \tag{P11.2a}$$

where the step–size h is a parameter.

Invoke that model multiple times together with the static function model to simulate some higher–order differential equation models using the FE method in PowerDEVS.

Then, using different values of h for different integrators, perform some *multi–rate integration* experiments.

We suggest that you reproduce the example of Section 10.5, given by Eq.(10.10).

Study the possibility of building integrators corresponding to higher–order algorithms (RK, AB, etc.).

Conclude about the advantages and disadvantages of using DEVS in the context of discrete–time integration algorithms.