# Discrete-Event Modeling and Simulation

## A Practitioner's Approach

Gabriel A. Wainer

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742


Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-1-4200-5336-4 (Hardcover)



**Library of Congress Cataloging-in-Publication Data**

Wainer, Gabriel A.
Discrete-event modeling and simulation : a practitioner's approach / Gabriel A. Wainer.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4200-5336-4 (hardcover : alk. paper)
1. Computer simulation. 2. Discrete-time systems. I. Title. II. Series.

QA76.9.C65W35 2009
003'.3--dc22

2008039739

**Visit the Taylor & Francis Web site at**
**http://www.taylorandfrancis.com**

**and the CRC Press Web site at**
**http://www.crcpress.com**

# 2 Introduction to the DEVS Modeling and Simulation Formalism

## 2.1 INTRODUCTION

As discussed in our previous chapter, the theory of differential equations has a long history, while modeling techniques for discrete-event systems have only appeared recently. Discrete-event simulation techniques were developed hand in hand with the creation of the computer. Consequently, the first discrete-event modeling and simulation (M&S) approaches were tightly coupled to the computer hardware and languages, while formal modeling techniques with a solid mathematical background are more recent. Discrete-event systems specification (DEVS) M&S theory is one of such techniques that were based on systems theory concepts [1–3]. Although in this chapter we will briefly introduce some of the basic ideas behind this theory of modeling and simulation, the reader should refer to Zeigler [1]; Zeigler, Praehofer, and Kim [4]; and Zeigler [5] to understand the details behind the mathematical background of this technique. Our goal here is to provide the basis to discuss the application examples introduced in the following chapters. As we will briefly discuss here, DEVS also provides a formal background for a common methodology for modeling both discrete and continuous worlds (the interested reader should consult Zeigler et al. [4] and Cellier and Kofman [6].

According to DEVS theory, the system of interest is seen as a source of behavioral data for the study within a given experimental frame (EF). The EF is a restricted set of the elements observed in the system and the conditions under which they are observed. These data are used to create an abstract representation of such a system (a model). Using a set of instructions, rules, or mathematical equations, the model tries to replicate the behavior of the system of interest under the experimental conditions. Figure 2.1 shows these basic entities in M&S and their relationships [4].

This separation of concerns and the use of a formal mechanism to describe each of the components allowed DEVS methodology to improve the creation of models and the execution of simulations. The formal specifications provide means for mathematical manipulation and it permit independence of the language chosen to implement the models.

The model represents a simplified version of reality and its structure. The model is built considering the conditions of experimentation of the system of interest, including the work conditions of the real system and its application domain. Thus, the model is restricted to the experimental framework under which it was developed (which influences its construction, experimentation tasks, and validation).

The model is subsequently used to build a simulator (i.e., a device capable of executing the model's instructions) generating the model's behavior. When we implement the model, there is an extra reduction in the precision. If the model is implemented in a computer, the programming languages and the computer used are limited, including the duration of the simulation, memory availability, precision of the variables involved, etc.

Figure 2.1 also shows the two fundamental relationships discussed in Chapter 1: verification and validation. Validation links the model with the real system within the experimental frame (i.e.,

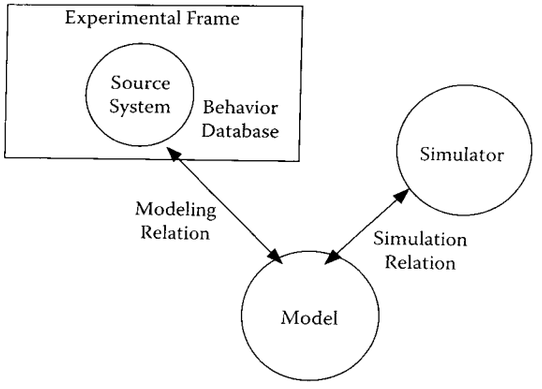


**FIGURE 2.1** Basic entities in M&S and their relationships. (From Zeigler, B. P. et al. 2000. *Theory of modeling and simulation*, 2nd. ed. New York: Academic Press.)

how well the model's behavior agrees with the system's behavior under the conditions specified in the experimental frame). Verification links a simulator and its model, dealing with the accuracy of the simulator to execute the instructions of the model (this can be related to the precision of the computer, the simulation algorithms, etc.).

DEVS was created for modeling and simulating discrete-event dynamic systems (DEDS); thus, it defines a way to specify systems whose states change either upon the reception of an input event or due to the expiration of a time delay. In order to attack the complexity of the system under study, the model is organized hierarchically (i.e., it is organized in a way such that every element is higher than its precedent), and higher-level components of the system are decomposed into simpler elements. The second tool used to attack complexity is information hiding, through the provision of a modular interface for each of the models.

The separation between model and simulator and the hierarchical and modular nature of the formalism have enabled carrying out of formal proofs on the different entities under study. One of them is the proof of composability of the subcomponents (including legitimacy and equivalence between multicomponent models). The second is the ability to conduct proofs of correctness of the simulation algorithms, which results in simulators rigorously verified. The proofs are based on formal transformations (*morphisms*) between each of the representations, trying to prove the equivalence between the entities under study at different levels of abstraction [4,7–9]. For instance, we can prove that the mathematical entity *simulator* is able to execute correctly the behavior described by the mathematical entity *model*, which represents the system under the experimental framework (which can also be represented formally).

Different mechanisms are used to prove this, including the mathematical manipulation of the abstraction hierarchy, observation of the I/O trajectories (in order to ensure that different levels of specification correctly describe the system's structure), and decomposition concepts (DEVS is closed under composition, which means that a composite model integrated by multiple components is equivalent to an atomic component).

## 2.2 THE DEVS FORMALISM

A real system modeled using DEVS can be described as a composition of *atomic* and *coupled* components [1,4]. Here, we will use the definition of *DEVS with ports* [4] (instead of *classic* DEVS [1]). An *atomic* model is specified as

$$M = <X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta> \tag{2.1}$$



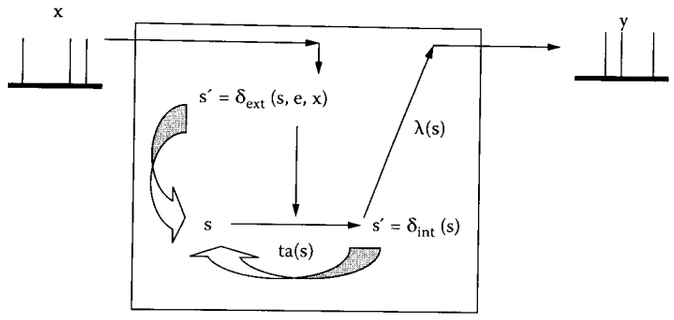


**FIGURE 2.2** DEVS atomic model semantics.

where

$X = \{(p,v) | p \in IPorts, v \in X_p\}$ is the set of *input* events, where *IPorts* represents the set of input ports and $X_p$ represents the set of values for the input ports;
$Y = \{(p,v) | p \in OPorts, v \in Y_p\}$ is the set of *output* events, where *OPorts* represents the set of output ports and $Y_p$ represents the set of values for the output ports;
$S$ is the set of sequential *states*;
$\delta_{ext}: Q \times X \rightarrow S$ is the *external* state transition function, with $Q = \{(s,e) | s \in S, e \in [0, ta(s)]\}$ and $e$ is the elapsed time since the last state transition;
$\delta_{int}: S \rightarrow S$ is the *internal* state transition function;
$\lambda: S \rightarrow Y$ is the *output* function; and
$ta: S \rightarrow R_0^+ \cup \infty$ is the *time advance* function.

Figure 2.2 shows an informal depiction of DEVS atomic models.

At any given moment, a DEVS model is in a state $s \in S$. In the absence of external events, it remains in that state for a lifetime defined by $ta(s)$. When $ta(s)$ expires, the model outputs the value $\lambda(s)$ through a port $y \in \gamma$, and it then changes to a new state given by $\delta_{int}(s)$. A transition that occurs due to the consumption of time indicated by $ta(s)$ is called an internal transition. On the other hand, an external transition occurs due to the reception of an external event. In this case, the external transition function determines the new state, given by $\delta_{ext}(s, e, x)$, where $s$ is the current state, $e$ is the time elapsed since the last transition, and $x \in X$ is the external event that has been received.

The time advance function can take any real value between 0 and $\infty$. A state for which $ta(s) = 0$ is called a *transient* state (which will trigger an *instantaneous* internal transition). In contrast, if $ta(s) = \infty$, then $s$ is said to be a *passive* state, in which the system will remain perpetually unless an external event is received (can be used as a termination condition).

A DEVS *coupled model* is composed of several atomic or coupled submodels. It is formally defined by

$$CM = <X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, select> \tag{2.2}$$

where

$X = \{(p,v) | p \in IPorts, v \in X_p\}$ is the set of *input* events, where *IPorts* represents the set of input ports and $X_p$ represents the set of values for the input ports;
$Y = \{(p,v) | p \in OPorts, v \in Y_p\}$ is the set of *output* events, where *OPorts* represents the set of input ports and $Y_p$ represents the set of values for the output ports;
$D$ is the set of the component names and for each $d \in D$;
$M_d$ is a DEVS basic (i.e., atomic or coupled) model;

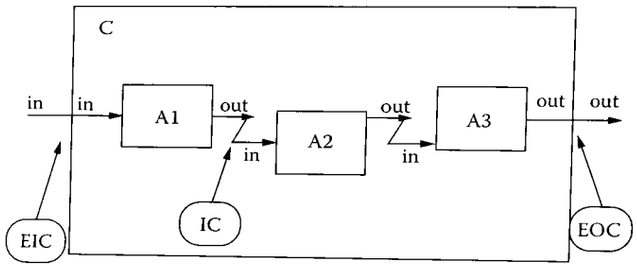


**FIGURE 2.3** A coupled model.

*EIC* is the set of external input couplings, $EIC \subseteq \{ ( (Self, in_{Self}), (j, in_j) ) | in_{Self} \in IPorts, j \in D, in_j \in Iports_j \}$;

*EOC* is the set of external output couplings, $EOC \subseteq \{ ((i,out_i), (Self,out_{Self})) | out_{Self} \in OPorts, i \in D, out_i \in OPorts_i \}$;

*IC* is the set of internal couplings, $IC \subseteq \{ ((i,out_i), (j,in_j)) | i,j \in D, out_i \in OPorts_i, in_j \in IPorts_j \}$; and

*select* is the tiebreaker function, where $select \subseteq D \rightarrow D$, such that, for any nonempty subset $E$, $select(E) \in E$.

Figure 2.3 shows an example of a DEVS coupled model with three subcomponents, A1–A3. These basic models are interconnected through the corresponding I/O ports presented in the figure. The models are connected to the external coupled model through the EIC and EOC connectors. Keep in mind that A1–A3 are *basic* models (i.e., they can be atomic or coupled components).

The model depicted in Figure 2.3 can be formally defined as

$$C = <X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, select> \quad (2.3)$$

where

$X$ = {(in, v)|in ∈ *IPorts*, $v \in \boldsymbol{R}$};
$Y$ = {(out, v)|out ∈ *IPorts*, $v \in \boldsymbol{R}$ };
$D$ = {A1, A2, A3};
$M_d$ = { $M_{A1}$, $M_{A2}$, $M_{A3}$ };
*EIC* ⊆ { ( (Self, in), (A1, in) ) }; (or *EIC* ⊆ { ( (C, in), (A1, in) ) } );
*EOC* ⊆ { ( (A3, out), (Self, out)) }; (or *EOC* ⊆ { ( (A3, out), (C, out)) } );
*IC* ⊆ { ( (A1, out), (A2, in)); ( (A2, out), (A3, in)) };
*select* = {A3, A1, A2}.

The coupled model definition presented shows the specification of the three components A1–A3 and their internal/external couplings. Coupled models group several DEVS into a composite model that can be regarded, due to the *closure property*, as a new DEVS model. The closure property guarantees that the coupling of several class instances results in a model of the same class, allowing hierarchical construction [4].

Because multiple subcomponents can be scheduled for an internal transition at the same time, ambiguity could arise. In our example, if A1 executes its output/internal transition first, producing an output that maps into an external event for A2 (which is also scheduled for an internal transition at the same time), then it is not clear which transition this second component should execute first. There are two alternatives for this:

(a) to execute the external transition first and then the internal transition, with $e = ta(s)$; or
(b) to execute the internal transition first, followed by the external transition, with $e = 0$.



The *select* function provides a simple way to solve this ambiguity. The function defines an ordering over all the components of the coupled model so that only the first model to execute in the case of simultaneous internal events can be chosen. In our example, A1 is executed before A2; thus, we execute the external transition first.

A different definition of coupled models (that we will be using later in this and other chapters) is

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select > \quad (2.4)$$

where

$X$ is the set of input events;
$Y$ is the set of output events;
**D** is an index for the components of the coupled model;
$\forall i \in D$, $\mathbf{M}_i$ is a basic DEVS model;
$\mathbf{I}_i$ is the set of influencees of model i and $\forall j \in I_i$;
$\mathbf{Z}_{ij}$ is the i to j translation function, where $Z_{ij}$: $Y_i \rightarrow X_j$

Figure 2.4 shows an example of the execution of a DEVS atomic model (which, due to the closure under coupling property, could also be representing the I/O trajectories of a DEVS coupled model).

Initially, the model is in state $s_0$, and an internal transition is scheduled for $ta(s_0)$. Time advances (i.e., the elapsed time variable $e$ moves forward in continuous time) and, at time $t_0$ (before the scheduled time is consumed), the model receives the internal input $X_0$. Consequently, the external

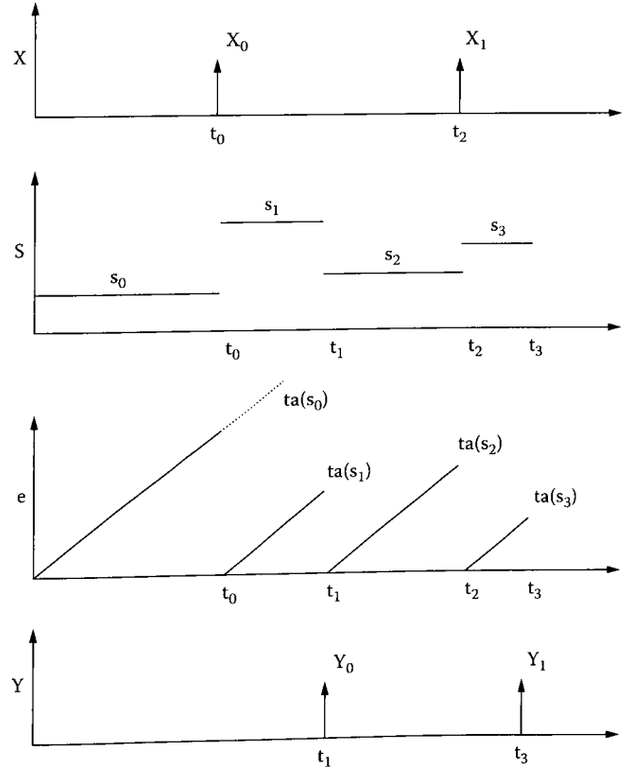


**FIGURE 2.4** Sample I/O trajectories.

transition function is triggered, and the model changes to $s_1$, which has an associated time advance of $ta(s_1)$ (at this point, the elapsed time $e$ is reset to 0). In this case, no inputs are received before the internal event, and when time $t_1 = ta(s_1)$ arrives, we activate the output function (which generates the output $Y_0$). We then execute the internal transition function (which will make the model change to the state $s_2$ and schedule the next internal transition at $ta(s_2) = t_2$). At time $t_2$, we simultaneously receive the external input $X_1$ (in order for this to happen, the *select* function at the parent coupled model must have triggered an influencer, which executed its output function before our atomic model). Thus, we consume the input event $X_1$, triggering the external transition function (which makes the model change to $s_3$ with $ta(s_3) = t_3$). At this time, we trigger both the output function (which generates the output $Y_1$) and the internal transition function. This cycle is repeated until the end of the simulation.

## 2.3 A DEVS MODEL EXAMPLE

The Generator, Processor, Transducer (GPT) model presented in this section has been widely used as a "Hello, world!" example for DEVS modeling and simulation [4]. The structure of the model is introduced in Figure 2.5.

The top-level model *GPT* is a simple coupled model that is composed of two basic components: *generator* and *QPT*. *Generator* is an atomic model that creates jobs to be processed (at random times) and sends them through the *out* output port. *QPT* is a coupled model consisting of two main atomic components: a *processor* that consumes the jobs received (and informs that they are ready through the *out* output port) and a *transducer* in charge of calculating statistics. When a new job arrives through the *arrived* input port, the *transducer* computes the arrival time; when the job finishes, its end time arrives through the *solved* port, and we can use this information to compute metrics. In this case, we have also included a *queue* model, which is used as a buffer for the arriving jobs before they are processed. Based on Figure 2.5, we can define the coupled model for this example as

$$M_{GPT} = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, select \rangle \quad (2.5)$$

where

$X$ = $\varnothing$;
$Y$ = {(cpuUsage, $\boldsymbol{R}_0^+$); (Throughput, $\boldsymbol{R}_0^+$) };
$D$ = { Generator, QPT };
$M_d$ = { $M_{Generator}$, $M_{QPT}$ } (where $M_{Generator}$ is an atomic model and $M_{QPT}$ a coupled one);
$EIC$ = $\varnothing$;
$EOC \subseteq$ { ( (QPT, cpuUsage), (Self, cpuUsage) ) ; ( (QPT, Throughput), (Self, Throughput) ) };

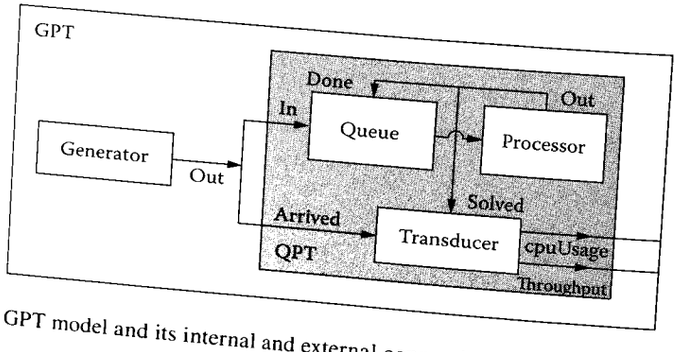


**FIGURE 2.5** The GPT model and its internal and external connections.



$IC \subseteq$ { ( (Generator, out), (QPT, in)); ( (Generator, out), (QPT, arrived)) }; and
*select* = {QPT, Generator }.

The QPT coupled model can be defined as

$$M_{QPT} = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, select \rangle \quad (2.6)$$

where

$X$ = {(in, $N$); (arrived, $N$) };
$Y$ = {(cpuUsage, $\boldsymbol{R}_0^+$); (Throughput, $\boldsymbol{R}_0^+$) };
$D$ = { Queue, Processor, Transducer};
$M_d$ = { $M_{Queue}$, $M_{Processor}$, $M_{Transducer}$ };
$EIC$ = { ( (Self, in), (Queue, in) ) ; ( (Self, arrived), (Transducer, arrived) ) };
$EOC \subseteq$ { ( (Transducer, cpuUsage), (Self, cpuUsage) ); ( (Transducer, Throughput), (Self, Throughput) ) };
$IC \subseteq$ { ( (Queue, out), (Processor, in)); ( (Processor, out), (Queue, done)); ( (Processor, out), (Transducer, solved)) };
*select* = {Processor, Queue, Transducer }.

The coupled model definitions show the structure of the whole model, but then we need to define the behavior for each atomic model, which should use the following descriptions:

- **Generator** generates new tasks transmitted through an output port. The output value represents a task identifier (a positive integer uniquely used during the simulation process). The period used to create a new process is generated at random (with probability distributions chosen during the configuration of the experiment).
- **Processor** simulates the tasks' execution delays. A new task ID (a positive integer number) is received through an input port, and the processor remains busy until the processing is finished. Then it sends the process identifier through an output port. The processing time is generated using random numbers with exponential distribution.
- **Queue** is a buffer that stores task IDs (positive integer numbers). When an ID is received through the *done* input port, the buffer must transmit a stored job (if available). The queue uses a nonpreemptive first in, first out (FIFO) policy. A *stop* input port is used to deactivate/reactivate the queue, allowing control flow by higher-level models (this port has not been used in the example introduced in Figure 2.5).
- **Transducer** records metrics and computes statistics. Two measures are considered: throughput (tasks executed per time unit) and CPU usage (average time of tasks waiting in the ready queue). The transducer accepts job IDs on the *arrived* input port and records the time for arrival of the job. Jobs processed must be forwarded to the transducer's *solved* input port so that the transducer can record the time when the job was finished and calculate the elapsed time. This value is subsequently used for calculating throughput and CPU usage.

The functionality of each of these models must be described using the formal specification for DEVS. For instance, the queue model can be formally described as

$$\text{Queue} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (2.7)$$

$X$ = {(in, $N$); (stop, N); (done, $N$) };
$S$ = { state $\in$ {active, passive}, preparationTime, timeLeft $\in \boldsymbol{R}_0+$, queue $\in$ {jobid $\in N$}* };
$Y$ = {(out, $N$); };

```
δext (s, e, x) {
    if ( x.port == in ) {                          // A new job has arrived
      add (x.value, s.queue);                      // Add it to the queue
      if ( sizeOf (s.queue) == 1 )                 // The queue was empty
        state = active; ta(state) = preparationTime ;   // The arrived job must be executed.
    }
    if( x.port == done ) {                         // A job has finished
      delete_first (s.queue);                      // Delete it from the queue
      if( !empty(s.queue) )                        // Take the next element in the queue
        state = active; ta(state) = preparationTime ;   // This job must be executed
    }
    if(x.port == stop )                            // Stop the transmision: buffer overflow
      if( state == active && x.value != 0) {       // The queue was active
        timeLeft = preparationTime – e ;           // Record the time left to execute
          state = passive; ta(state) = infinity;   // Deactivate the queue: passivate.
        }
        else                                       // Reactivate the queue
          if( state == passive && x.value == 0 )
            state = active; ta(state) = timeLeft ;      // Use the time left before being stopped
}

δ ( s ) {
    sendOutput(time, out, first (queue));
}

δint ( s ) {
    passivate();
}
```

The transducer model collects timing information of the jobs arriving in the system and their departure times. Using these events, it computes the number of jobs processed per time unit (*throughput*) and the level of utilization of the CPU (*cpuUsage*). The model can be formally defined as

$$\text{Transducer} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta > \tag{2.8}$$

```
X = {(arrived, N); (solved, N); (done, N) };
S = state ∈ {active, passive}, procCount ∈ N, cpuLoad, frequence ∈ R0+, unsolvedQ
    ∈ {jobid ∈ N}* };
Y = {( throughput, R0+); (cpuUsage, R0+); };

δext (s, e, x) {
  cpuLoad += (time – lastChange) * size_of(unsolvedQ) ; // Average load
  if( x.port == arrived ) unsolvedQ[ x.value ] = time;  // Store information about task
  if( x.port == solved ) {                              // Task ended: erase
    which = find( x.value, unsolved ) ;
    procCount ++ ;
    erase( which ) ;
  }
}
```



```
λ ( s ) {
    sendOutput(time, throughput, procCount/time );
    sendOutput(time, cpuUsage, cpuLoad/time );
}

δint ( s ) {
    passivate();
}
```

This model will be used later for varied examples throughout the book.

### Exercise 2.1

Modify the *queue* model to implement a LIFO strategy.

### Exercise 2.2

Modify the *queue* model to implement a priority-based strategy. Job numbers also represent the priority of the job; thus, every arriving job should be located in the right position in the queue (using the ID number).

### Exercise 2.3

Write the model specification for the processor model. Include two different versions: one without preemption and one with preemption (i.e., a newly arriving job will stop the execution of the current job and will start the new one).

### Exercise 2.4

Compute the input/output trajectories for the queue model for the following three jobs: (1, 0.3 s), (2, 5.1 s), (3, 10.6 s), where the first number is the job ID and the second the arrival time.

### Exercise 2.5

Compute the input/output trajectories for each atomic model and the whole coupled model using the same input trace used in Exercise 2.4.

### Exercise 2.6

Use the internal transition function in the queue model to represent a faulty buffer. Every time the size of the queue is a multiple of 13, one element in the queue is deleted.

### Exercise 2.7

Modify the mechanism for computing the CPU load in the transducer. In this case, use an accumulator to keep track of the total use of the CPU (i.e., add all the time between arrival/departure of jobs) instead of the average used in the original version.

### Exercise 2.8

Add a new model, *ControlFlow*, which will stop or reactivate the queue model according to its internal state. A random number is used to decide when the queue should be stopped. The internal transition function will generate a random number using a normal distribution with average 5 and standard deviation 3. If the number generated is larger than 9, then the output function will generate a "stop" signal for the queue. Then, if the number generated is smaller than 8, the queue will be reactivated. Write a formal specification for this model and modify the corresponding coupled model.

## 2.4 DEVS WITH SIMULTANEOUS EVENTS (PARALLEL DEVS)

As seen in the previous section, whenever two models are scheduled for state transitions at the same time, a DEVS coupled model will pick the one specified by the *select* function to execute first. This tie-breaking strategy is rigid. Let us suppose that model A2 in Figure 2.3 represents vehicles going into an intersection, and A3 represents vehicles inside the crossing area. According to the *select* function definition, A3 has the highest priority (which tries to free traffic in the crossing area before allowing new vehicles in the intersection). If now we want to be able to represent collisions, we would need to give priority to A2; however, this is not possible in the current specification, which will free space in the crossing first (making it more difficult to represent the collision situation). In addition, *select* introduces serialization in the execution of components when many interconnected atomic models are imminent (which could be executed in parallel in a multiprocessor environment).

*Parallel DEVS* (or PDEVS) is an extension to DEVS that provides a more flexible way of dealing with these ambiguities [10]. Atomic models provide an additional *confluent* function to specify collision behavior for events that might be scheduled simultaneously and a mechanism for receiving multiple external events at the same time and processing them together. An atomic PDEVS model is defined as

$$M = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle \tag{2.9}$$

where

$X = \{(p,v) | p \in IPorts, v \in X_p\}$ is the set of *input* events, where *IPorts* represents the set of input ports and $X_p$ represents the set of values for the input ports;

$Y = \{(p,v) | p \in OPorts, v \in Y_p\}$ is the set of *output* events, where *OPorts* represents the set of input ports and $Y_p$ represents the set of values for the output ports;

$S$ is the set of sequential states;

$\delta_{ext}$: $Q \times X^b \rightarrow S$ is the external state transition function;

$\delta_{int}$: $S \rightarrow S$ is the internal state transition function;

$\delta_{con}$: $Q \times X^b \rightarrow S$ is the confluent transition function;

$\lambda$: $S \rightarrow Y^b$ is the output function;

$ta$: $S \rightarrow R_0^+ \cup \infty$ is the time advance function, with $Q = \{(s, e) | s \in S, 0 \le e \le ta(s)\}$ the set of total states.

PDEVS models use bags (multisets) of events for receiving inputs and collecting outputs ($X^b$, $Y^b$) instead of a single event. This allows multiple events to be processed simultaneously. Because external input events received by the component are added to the bag, external transition functions can combine the functionality of a number of external transitions into a single one, and simultaneous events (like the departure of a vehicle and a collision in the intersection) can be treated simultaneously. Also, PDEVS allows a better way to deal with collisions: the model specification includes a confluent transition function ($\delta_{con}$). When a collision between the internal and external functions occurs, the confluent function determines the new state of the model.

The semantics of PDEVS for internal/external transition functions is similar to DEVS. If one or more external events $X^b = \{x_1 \ldots x_n / x_i \in X\}$ occur before $ta(s)$ expires (i.e., while the system is in total state $(s, e)$ with $e < ta(s)$), the new state will be given by the model's external transition function, $\delta_{ext}(s,e,X^b)$. If the external events $X^b$ are received when $e = ta(s)$, the new state of the model will be given by the confluent function ($\delta_{con}$). If multiple components in a coupled model are imminent, all their outputs are first collected and mapped to their influences in parallel. Then the corresponding transition function is executed for every model.

In PDEVS, coupled models are defined as in DEVS, without the need for a *select* function. Formally, a coupled model is defined as



$$CM = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC \rangle \tag{2.10}$$

where definitions for the set of input and output events ($X$ and $Y$), components ($D$ and $M_d$), and couplings (*EIC*, *EOC*, and *IC*) follow the specifications of DEVS coupled models presented earlier in this chapter.

## 2.5 DYNAMIC STRUCTURE DEVS

The definitions of DEVS presented in the previous sections consider a model with a static structure (i.e., invariant in time). Nevertheless, in many cases it is useful to allow modeling of the dynamic adaptation to dynamic changes in the environment. The only way of doing this with DEVS is by having multiple models defined and a selector model to activate the right one at any time. Dynamic structure systems instead focus on the possibility of changing the system structure dynamically according to the system's real requirements. Dynamic structure DEVS allows addressing some of these issues and supports structural changes on three levels:

1. System level: the structural change happens between coupled models (i.e., a new link between two coupled models is added).
2. Component level: the structural change happens within a coupled model but including two or more atomic models (i.e., a new link is added between two atomic models).
3. Subcomponent level: the structural change only happens within a single atomic model (i.e., the external transition function changes).

There are two popular dynamic structure DEVS definitions, namely, dynamic structure discrete event (DSDE) [11–13] and dynDEVS [14,15]. DSDE divides the models into two groups: *basic* and *network* models. The basic models are atomic structure units (cannot be split); network models are coupled components composed of multiple basic interconnected structure models (which can include structural changes). A *network executive* is a modified basic model that is used to conduct the changes in network models by storing all possible states for structural changes and their corresponding component sets. The two parts are associated together through an index function in the network executive. A DSDE network is defined as

$$DSDEN_N = (X_N, Y_N, \chi, M_\chi) \tag{2.11}$$

where

$X_N$ is the network input value set;

$Y_N$ is the network output value set;

$\chi$ is the name of the network executive; and

$M_\chi$ is the model of the network executive $\chi$, which is a modified basic model and is defined by

$$M_\chi = (X_\chi, s_{0,\chi}, S_\chi, Y_\chi, \gamma, \Sigma^*, \delta ext_\chi, \lambda_\chi, \delta int_\chi) \tag{2.12}$$

Here,

$X_\chi$, $S_\chi$, $Y_\chi$, $\delta ext_\chi$, $\lambda_\chi$, and $\delta int_\chi$ are defined as in DEVS;

$\gamma$: $S_\chi \rightarrow \Sigma^*$ is the structure function; and

$\Sigma^*$ is the set of network structures.

If $s_\alpha \in S_\chi$ is a partial state of the network executive, then $\gamma(s_\alpha) = \Sigma_\alpha = (D, \{M_i\}, \{I_i\}, \{Z_i\})_\alpha$ is a network structure (equivalent to a coupled model), where D is the set of component names associated with the executive partial state $s_\alpha$ for all $i \in D$; $M_i$ is the model of the component $i$ for all $i \in D \cup \{\chi, N\}$; $I_i$ is the set of influencees of i for all $i \in D \cup \{\chi\}$; and $Z_i$ is the input function of the component i and $Z_N$ is the network output function.

As we can see, the structure function provides a mapping between a partial state of the network and a new network structure, permitting us to carry out structural changes.

The dynDEVS formalism does not introduce an extra component to conduct dynamic structural changes. Instead, two kinds of dynamic DEVS models are included: dynDEVS (atomic) and dynNDEVS (coupled). The dynDEVS models atomic components are defined as

$$dynDEVS = df <X, Y, minit, M(minit)> \quad (2.13)$$

where

$X$, $Y$ are the input/output sets;

$minit \in M(minit)$ is the initial model; and

$M(minit)$ is the least set having structure $\{ <S, sinit, \delta_{ext}, \delta_{int}, \rho\alpha, \lambda, ta> \}$.

A dynDEVS model can be interpreted as a set of DEVS models with the same interface plus a transition function that determines which DEVS model succeeds the previous one. It includes an initial state and a dynamic reconfiguration function ($\rho\alpha$), which will be in charge of structural changes. A model's state space, internal and external transition, output, time advance, and model transition functions are subject to change during simulation.

dynNDEVS models are coupled structural components defined as

$$dynNDEVS = df <X, Y, ninit, N(ninit)> \quad (2.14)$$

where

$X$, $Y$ are the input/output sets;

$ninit \in N(ninit)$ is the start configuration; and

$N(ninit)$ is the least set having the structure $\{<D, \rho_N, \{dynDEVS_i\}, \{I_i\}, \{Z_{ij}\}, Select>\}$.

The dynNDEVS model is similar to a coupled model, but it now includes the dynamic configuration function $\rho_N$.

Both of the preceding formalisms introduce new structure transition functions to conduct structural changes. In DSDE, the structural changes are carried out by $\chi$ (the network executive) and the structure function $\gamma$ (which maps the network structure state set $S_\chi$ and the network structure models' set $\Sigma^*$). The centralized network executives make sure that the structure transition is executed sequentially without any conflicts between structural change functions of the models. In dynDEVS, agents associated with the models conduct structural changes. $\rho_\alpha$ and $\rho_N$ are structure transition functions in dynDEVS and dynNDEVS models, respectively, which execute structural changes concurrently and independently.

We will show how to apply these concepts to a model of an automated manufacturing system (AMS) consisting of a flow shop for manufacturing cars. The system consists of dedicated stations that perform tasks on products being assembled and conveyors that transport the products to or from those workstations. The structure of the model is presented in Figure 2.6. As we can see, the flow shop consists of five parts:

1. *Conveyor belts* are used to transport products between the different stations (a conveyor is composed of an engine and sensors).
2. The *control unit* is in charge of controlling the movement of the conveyors according to the production cycle provided by a scheduler.
3. The *scheduler* (SCH) is in charge of the production cycle organization, and it programs the control unit to execute the production cycle on both conveyors.
4. The *display controller* displays the current status of the whole AMS system (a SCADA system).



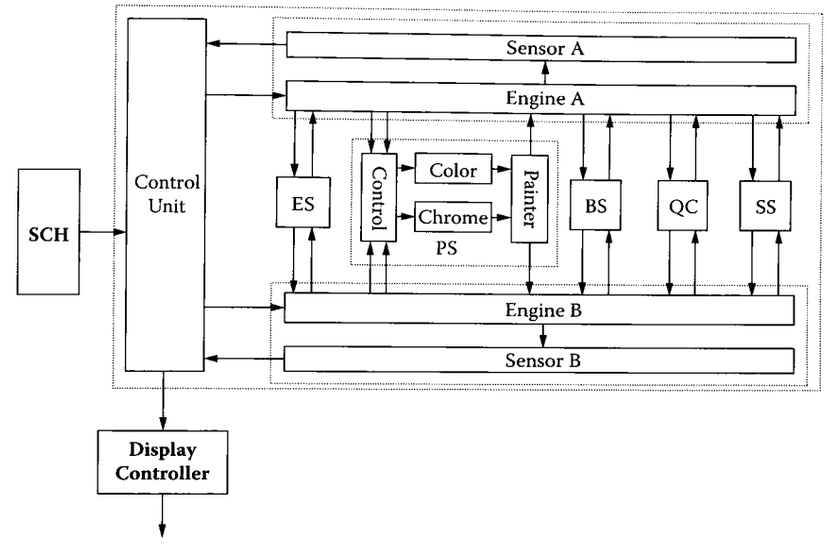


**FIGURE 2.6** An automated manufacturing system model.

5. Each *workstation* is in charge of a different task and its quality control. Vehicles being manufactured are delivered to each workstation (in order to be served step by step) by the conveyors. The Engine Station (ES) is in charge of assembling engine parts, the Painting Station (PS) undertakes the painting and special painting tasks, the Baking Station (BS) is in charge of baking (which takes place after painting), QC serves as a Quality Center to evaluate the quality of the vehicles for the whole plant, and the Storage Station (SS) distributes the vehicles to their corresponding warehouses.

During system execution, the structure of the AMS could be affected in order to adapt to the changes of external environment. Two kinds of system adjustments are considered:

- Workstation duty shifts: workstations have different working capacity during day and night.
- Workload in the PS workstation: this station is in charge of color or chrome painting. Vehicles might need color painting or both color and chrome painting. Painting selection is determined by the "control" model residing in the PS workstation.

Figure 2.7 shows the case of dynamic reconfiguration in the ES workstation due to duty shifts. ES and ES′ represent the engine workstation during day and night, respectively, and they can be considered as two structural states of the basic model $\varsigma$. $Z_{es,0}$ and $Z_{es,1}$ represent the input functions of ES and ES′; $Z_{\varsigma,0}$ and $Z_{\varsigma,1}$ represent the output functions of the structural model $\varsigma$. $\chi$ is the network executive described in DSDE.

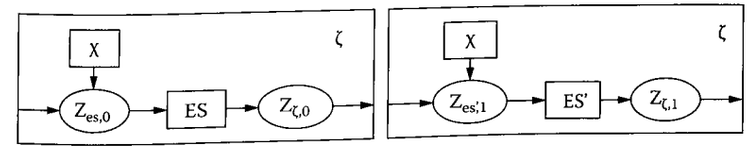


**FIGURE 2.7** ES structure layout during daytime and nighttime.

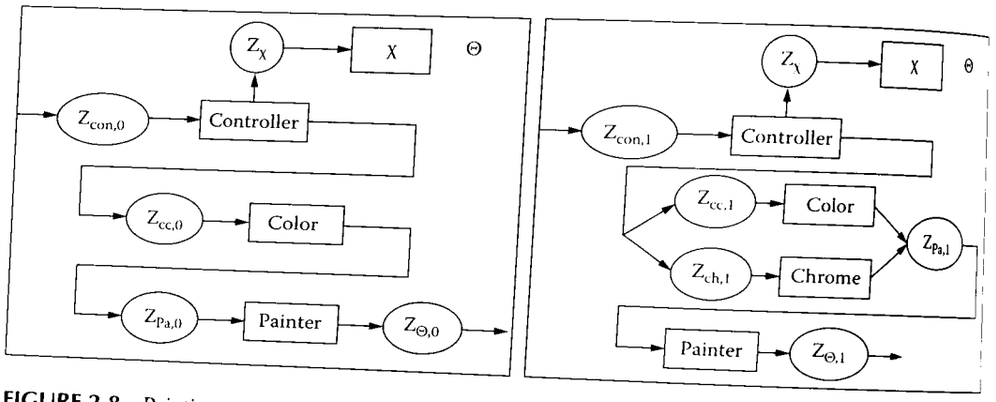


**FIGURE 2.8** Painting mode I in PS workstation and painting mode II in PS workstation.

The PS workstation is a coupled model including four atomic models: controller, color, chrome, and painter. The atomic model "chrome" is an optional component. Painting selection is determined by the "Controller." Figure 2.8 shows the two structural states of the network model Θ. $Z_{i,\alpha}$ (i = controller, color, chrome, and painting; $\alpha = 0,1$) is the input function of the atomic models and $Z_{\Theta,\alpha}$ is the output function of the network model Θ. $Z_\chi$ is the input function of the network executive $\chi$.

Generally, the autos on the conveyor are painted with specific colors. Therefore, only the atomic model "color" is needed in the PS. Assume that the current bulk of autos on the conveyor needs to be painted both color and chrome. The atomic model "chrome" should be added into PS automatically.

**EXERCISE 2.9**

Write the dynDEVS specification for the previous example, based on [14,15].

**EXERCISE 2.10**

Write a static DEVS specification for all the previous examples.

## 2.6 QUANTIZED DEVS

As mentioned in Chapter 1, the first existing modeling techniques focused on modeling the continuous behavior of the dynamic systems, using various kinds of differential equation formalisms. The evolution of state variables for dynamic systems is described via state equations modeled using differential equations. Ordinary differential equations (ODEs) are described as

$$\dot{x} = f(x,t) \tag{2.15}$$

with no algebraic constraints for the vector of differential variables. The simplest state-space models are represented by ODEs:

$$\dot{x} = f(x,u,t) \tag{2.16}$$

where $\boldsymbol{x}$ represents the state variables vector and $\boldsymbol{u}$ represents the inputs vector.

Differential algebraic equations (DAEs) are constructed as a set of differential equations with additional algebraic constraints in the form

$$f(\dot{x},x,u,t) = 0 \tag{2.17}$$



where

$x \in \boldsymbol{R}^n$ is a vector of *differential* variables;
$u \in \boldsymbol{R}^m$ is a vector of *algebraic* variables;
$t \in$ R is an *independent* variable; and
$f \in R^{2n+m+1} \rightarrow R^{n+m}$ is the set of DAEs.

As discussed in Chapter 1, continuous systems simulation is mainly solved by approximating the set of differential equations describing the system and finding consistent initial conditions. There is a wide variety of ODE solvers—for example, *forward Euler* (explicit method), *backward Euler* (implicit method), and *Runge–Kutta* [6]. For DAEs, if the equations can be transformed to a set of ODEs, a simulator can numerically approximate the equations using any ODE solver. If the transformation is not possible, a DAE solver can be used (e.g., *DASSL* and *implicit Runge–Kutta*). In DAEs, the simulator might have to differentiate the equations a very large number of times in order to get an ODE in all the state variables (because the ODE's index is equal to zero). Constraints (dependencies among variables that cannot be chosen freely) are usually hidden in these high-index DAEs. Several algorithms for index reduction and finding hidden constraints can be found in the literature, including *Gear and Petzold*, *Bachmann*, and *Pantelides* algorithms for index reduction, etc. [16–18].

Most of the techniques just mentioned have traditionally been simulated by discretizing the time domain and solving the equations over each discrete time interval. However, a few years ago a new approach for continuous systems simulation based on the discrete event paradigm was introduced. Discrete event methods in general and DEVS in particular present several advantages in contrast to the classical discrete time techniques:

- Computational times reduction: for a given accuracy the number of calculations can be decreased.
- Complex model definition in a hierarchical modular fashion: DEVS allows specification of complex systems in a hierarchical way.
- Hybrid systems modeling and simulation: DEVS provides a theory to develop a uniform approach to model and simulate systems with continuous and discrete components.

These techniques are based on a theory of quantized DEVS (QDEVS) [19]. The basic idea is shown in Figure 2.9(a). We discretize the space of the state variables using a fixed value called the *quantum* size. Thus, a state change will be informed only if it crosses the threshold defined by the quantum. As we can see, a continuous curve is now represented by the crossings of an equally spaced set of boundaries, separated by the quantum size, converting the continuous signal into a discrete-event version (in which the signal is piecewise constant). This operation reduces substantially the frequency of message updates while potentially incurring error (like any other numerical method).

The *QSS* (*quantized state systems*) formalism developed by Kofman [20] allows continuous systems simulation based on a combination of QDEVS and hysteresis. This approach constitutes the

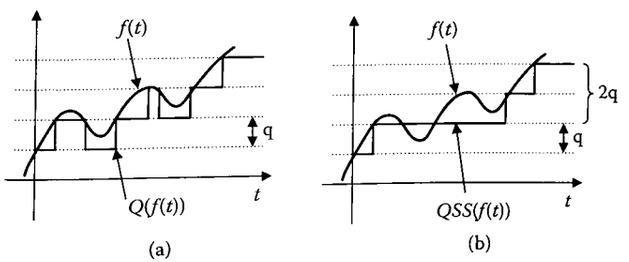


**FIGURE 2.9** (a) Signal quantization; (b) quantization function with hysteresis.

first general method for ODE integration using discrete event theory, and it has been proved that ODE systems can be approximated by a legitimate DEVS model. The main difference with QDEVS is that the quantization function is combined with hysteresis (e.g., the quantum size is changed to its double when there are direction changes on the values, as seen Figure 2.9(b)). This means that if a value changes its direction with respect to the last threshold value, the next value will have to change two regions to be transmitted. This eliminates the problem of possibly infinite numbers of transitions performed by a model in a finite interval (a legitimacy problem in DEVS models).

In Kofman [20] it was proven that the original system and the resulting QSS have similar trajectories. Some properties of the original system, like equilibrium points and stability, are preserved on the simulation model. It was also shown that the solution of the simulation model converges to the solution of the original system when the discretization goes to zero, allowing the method to be implemented with an arbitrary small error [6].

**EXERCISE 2.11**

Let $y = 3e^x + 1$. Define a simulation algorithm that will "plot" this function starting with $x_0 = -10$ and will use a time step of $h = 0.5$. Run a desk test of the simulation, plotting the results (use any programming language, spreadsheet, or pen-and-pencil solution). Repeat the exercise with $h = 1$.

**EXERCISE 2.12**

Invert the function in Exercise 2.11 so that now we can obtain $x$ as a function of $y$. Define a simulation algorithm that will "plot" this function starting with $y_0 = 2.00013$ and a quantum size of $q = 1$. Run a desk test of the simulator, plotting the results (use any programming language, spreadsheet, or pen-and-pencil solution). Repeat the exercise with $q = 20$.

**EXERCISE 2.13**

Write an algorithm for a function that, given the last two values computed, will determine if there was a difference of more than five units between the two values.

**EXERCISE 2.14**

Combine the function in Exercise 2.13 with the algorithm introduced in Exercise 2.11 in order to find differences larger than five units.

**EXERCISE 2.15**

Define the models of Exercises 2.12 and 2.13 as DEVS atomic models. Combine them as coupled models.

As we can see from these exercises, quantization requires a fundamental shift in thinking about the system as a whole. Instead of determining what value a dependent variable will have (its state) at a given time, we must determine at what time a dependent variable will enter a given state—namely, the state above or below its current state.

## 2.7 GENERALIZED DEVS (GDEVS)

Another approach recently applied to deal with continuous systems modeling based on discrete-event specifications is the *GDEVS* (*Generalized Discrete Event Specification*) formalism [21]. GDEVS uses polynomials of arbitrary degree to represent the piecewise input–output trajectories of a discrete event model.

GDEVS uses a new definition for the concept of event. The target real-world system is modeled through piecewise *polynomial segments* translated into piecewise constant trajectories. A *coefficient*



*event* is thus considered as an instantaneous change of at least one of the values of the coefficients defining the piecewise polynomial trajectory of the variable under study. An event is a list of coefficient values defining the polynomial that describes the trajectory of the variable.

A piecewise continuous polynomial segment is one that is defined over a continuous time base $\omega < t_0, t_n > \rightarrow \mathbf{A}$, as follows [21]:

- There is a finite number of elements $\{t_1, \ldots, t_{n-1}\}$, $\forall i \in [1, n-1]$, and $t_i$ is associated with a constant valued $n$-tuple $(a0_i, \ldots, an_i)$. $\forall\ t \in <t_k, t_l>$, where $t_k, t_l \in \{t_1, \ldots, t_{n-1}\}$; we define $\omega(t) = a0_i + a1_i\, t + \ldots + an_i\, t^n$.
- $\omega_{<t0,tn>} = \omega_{<t0,t1>} / \omega_{<t1,\,t2>} / \ldots / \omega_{<t-1,\,tn>}$, where / represents the left concatenation operator over segments.

For an individual segment $\omega_{<t_i,\,t_j>}$ of order $n$, its coefficient value is defined by $(a0, \ldots, an)$, where $a0$ is the value of the segment at time $t_1$ (named the "intercept"), and every $ai$ is the i-gradient.

Figure 2.10 shows the continuous function $f(x) = \sin(2\pi x) + \cos(e^x)$, a polynomial approximation of order 1 (i.e., the function is approximated by $a1x + a2$), and by $a$ events of order one ($a1$) (i.e., by a piecewise constant function, as in DEVS).

For a given piecewise polynomial segment, a coefficient event is defined by an instantaneous change in at least one of the values of the coefficients of the polynomial. For the piecewise polynomial segment $w_{<t0,\,tn>}$, there exists an event at time $t_i$ if the values of the coefficients $(a0_k, \ldots, an_k)$ over $<t_k, t_i$ and those of the coefficients $(a0_i \ldots, an_i)$ over $t_i, t_j>$ satisfy the condition that there exists an $l$ such that $a1_k \neq a1_i$.

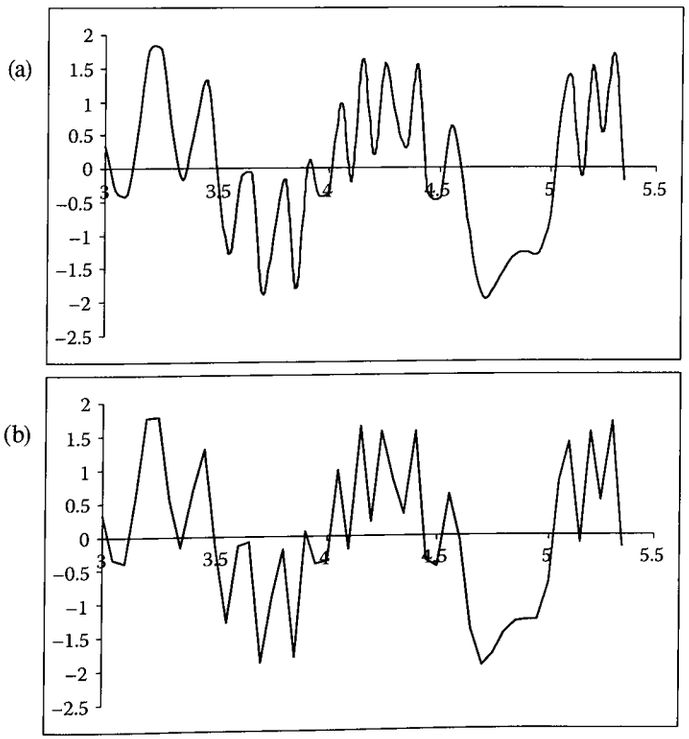


**FIGURE 2.10** GDEVS approximation of a continuous signal: (a) continuous segment; (b) linear segment; (c) piecewise segment.

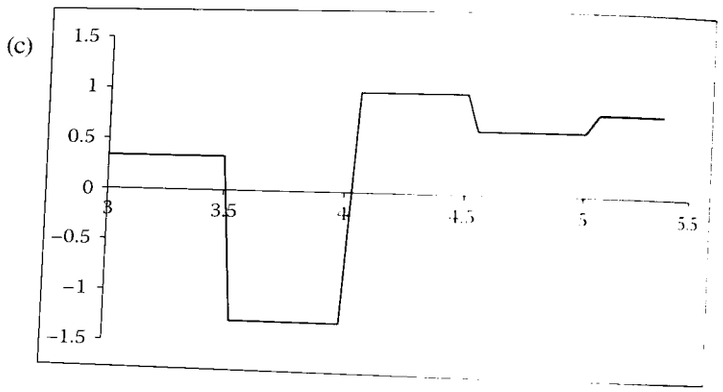


**FIGURE 2.10** (continued)

This approach is *solution based* and requires knowing the continuous system respons... ticular input trajectories and this represents a disadvantage, considering that this information... be available [20].

### EXERCISE 2.16

(a) Write a GDEVS model for the function in Figure 2.10. (b) Write a QDEVS approximation fo... same function. (c) Write a QSS approximation for the same function. (d) Simulate the executio... the three previous models.

### EXERCISE 2.17

Repeat Exercises 2.11–2.15 for the function in Figure 2.10.

## 2.8 SUMMARY

In this chapter, we have introduced the DEVS formalism and different variations. DEVS i... definitions are useful to improve the security and to reduce the development costs of a simul... a formal conceptual model can be validated, improving the error detection process and red... testing time. DEVS models are closed under coupling; therefore, a coupled model is equival... an atomic one, allowing reuse of previously defined models. Each model can be associated wi... experimental framework, allowing the individual testing of components and making integ... testing easier. Likewise, the simulation engines are independent from the modeling frame... which allows having a layered view of modeling and simulation (Figure 2.11).

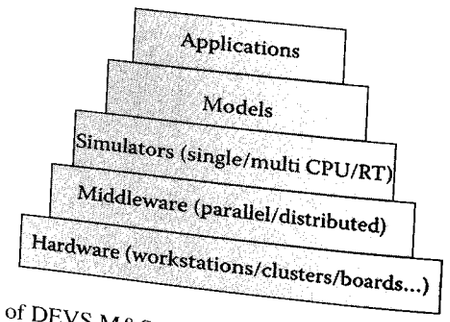


**FIGURE 2.11** A layered view of DEVS M&S.



DEVS, as a discrete event paradigm, uses a continuous time base, which allows accurate timing representation. The hierarchical and modular organization allows describing of multiple layers of a given application. This organization makes the definition of submodels easier, which in turn makes the definition of different levels of abstraction easy. The existence of an internal transition function eases the definition of certain properties. Internal state changes can be captured, describing complex internal interactions in a simple and natural way.

Recently, a theory of DEVS quantized models was developed, and it has been verified when applied to predictive quantization of arbitrary ordinary differential equation models. Quantized models reduce substantially the frequency of message updates. As the information interchange is reduced, the models potentially incur error. In this way, DEVS can be used to express hybrid digital/analog systems. GDEVS also enables the definition of hybrid models, which are expressed in a combined discrete event/differential equation formalism approximated by DEVS. In GDEVS, the accuracy of an analog subsystem is preserved using piecewise polynomial segments. The error introduced in this approximation can be controlled by increasing the order of the polynomials that represent analog signals between successive digital events.

## REFERENCES

1. Zeigler, B. P. 1976. *Theory of modeling and simulation.* New York: Wiley-Interscience.
2. Klir, G. J. 1972. *Trends in general systems theory.* New York: Wiley-Interscience.
3. Zadeh, L. A., and C. A. Desoer. 1963. *Linear system theory: The state space approach.* New York: McGraw–Hill.
4. Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation,* 2nd. ed. New York: Academic Press.
5. Zeigler, B. 1984. *Multifaceted modeling and discrete event simulation.* New York: Academic Press.
6. Cellier, F. E., and E. Kofman. 2006. *Continuous system simulation.* New York: Springer Science+ Business Media.
7. Nutaro, J. 2003. Parallel discrete event simulation with application to continuous systems. PhD thesis, University of Arizona, Tucson.
8. Nutaro, J., and H. Sarjoughian. 2004. Design of distributed simulation environments: A unified system-theoretic and logical processes approach. *Simulation* 80:577–589.
9. Kim, T. G., S. M. Cho, and W. B. Lee. 2000. DEVS framework for systems development: Unified specification for logical analysis, performance evaluation and implementation. In *Discrete event modeling & simulation: Enabling future technologies,* ed. H. S. Sarjoughian and F. Cellier. New York: Springer–Verlag.
10. Chow, A. C., and B. Zeigler. 1994. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. *Proceedings of Winter Simulation Conference,* Orlando, FL.
11. Barros, F. J. 1997. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation* 7:501–515.
12. Barros, F. 1998. Abstract simulators for the DSDE formalism. *Proceedings of Winter Simulation Conference,* Washington, D.C., 407–412.
13. Barros, F. J. 1995. Dynamic structure discrete event system specifications: A new formalism for dynamic structure modeling and simulation. *Proceedings of Winter Simulation Conference,* Arlington, VA, 781–785.
14. Uhrmacher, A. M. 2001. Dynamic structure in modeling and simulation: A reflective approach. *ACM Transactions on Modeling and Computer Simulation* 11:206–232.
15. Uhrmacher, A. M., and J. Himmeelspach. 2004. Processing dynamic PDEVS models. *Proceedings of 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04),* Volenlam, the Netherlands.
16. Pantelides, C. C. 1988. The consistent initialization of differential-algebraic systems. *SIAM Journal of Scientific and Statistical Computing* 9:213–231.
17. Fábián, G. D., D. A. van Beek, and J. E. Rooda. 2000. Substitute equations for index reduction and discontinuity handling. *Proceedings of Third IMACS Symposium on Mathematical Modelling,* Vienna, Austria.
18. Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. 1986. *Numerical recipes.* Cambridge: Cambridge University Press.

19. Zeigler, B. P., DEVS theory of quantization. 1998. Technical report, DARPA contract N6133997K-0007. ECE Dept., University of Arizona, Tucson.
20. Kofman, E. 2003. Quantized-state control. A method for discrete event control of continuous systems. *Latin American Applied Research Journal* 33:339–406.
21. Giambiasi, N., B. Escude, and S. Ghosh. 2000. GDEVS: A generalized discrete event specification for accurate modeling of dynamic systems. *Transactions of the SCS* 17:120–134.