roduction to

ger, F. J. San-
ulation Lan-

Simulation.

ns. Ellis Hor-

. PWS-KENT

on Cellular

Performance

g, New York,

nway's New
3 (1970).
r Simulation

ılysis, 2nd ed.

x Systems: An
l Univ. Press,

., Englewood

d Continuous
l, MA, 1986.
nery, Nurner-
2.
1990.
mata. World

4 Introduction to Discrete Event System Specification (DEVS)

4.1 Introduction

This chapter introduces the DEVS formalism for discrete event systems. DEVS exhibits, in particular form, the concepts of systems theory and modeling that we have introduced already. DEVS is important not only for discrete event models, but also because it affords a computational basis for implementing behaviors that are expressed in the other basic systems formalisms—discrete time and differential equations. Particular features of DEVS are also present in more general form in the systems theory that we will see in the next chapter. So starting with DEVS also serves as a good introduction to this theory and the other formalisms that it supports.

In this chapter, we start with the basic DEVS formalism and discuss a number of examples using it. We then discuss the DEVS formalism for coupled models, also giving some examples. The DEVS formalism that we start with is called Classic DEVS because after some 15 years, a revision was introduced called Parallel DEVS. As we will explain later, Parallel DEVS removes constraints that originated with the sequential operation of early computers and hindered the exploitation of parallelism, a critical element in modern computing. Finally, we discuss how DEVS is implemented in an object-oriented framework.

4.2 Classic DEVS System Specification

A *discrete event system specification* (*DEVS*) is a structure

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

where

X is the set of input values
S is a set of states,
Y is the set of output values
$\delta_{int}: S \rightarrow S$ is the *internal transition* function
$\delta_{ext}: Q \times X \rightarrow S$ is the *external transition* function, where
  $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$ is the *total state* set
  e is the *time elapsed* since last transition

$\lambda: S \to Y$ is the output function
$ta: S \to R^+_{0,\infty}$ is the set positive reals with 0 and ∞

The interpretation of these elements is illustrated in Fig. 1. At any time the system is in some state, *s*. If no external event occurs, the system will stay in state *s* for time *ta*(*s*). Notice that *ta*(*s*) could be a real number as one would expect. But it can also take on the values 0 and ∞. In the first case, the stay in state *s* is so short that no external events can intervene—we say that *s* is a *transitory* state. In the second case, the system will stay in *s* forever unless an external event interrupts its slumber. We say that *s* is *a passive* state in this case. When the resting time expires, i.e., when the elapsed time, $e = ta(s)$, the system outputs the value, $\lambda(s)$, and changes to state $\delta_{int}(s)$. Note that output is only possible just before internal transitions.

If an external event $x \in X$ occurs before this expiration time, i.e., when the system is in total state (*s*,*e*) with $e \le ta(s)$, the system changes to state $\delta_{ext}(s,e,x)$. Thus, the internal transition function dictates the system's new state when no events occurred since the last transition. The external transition function dictates the system's new state when an external event occurs—this state is determined by the input, *x*, the current state, *s*, and how long the system has been in this state, *e*. In both cases, the system is then is some new state *s′* with some new resting time, *ta*(*s′*), and the same story continues.

The preceding explanation of the semantics (or meaning) of a DEVS model suggests, but does not fully describe, the operation of a simulator that would execute such models to generate their behavior. We will delay discussion of such simulators to later chapters. However, the behavior of a DEVS is well defined and can be depicted as in Fig. 2. Here the *input trajectory* is a series of events occurring at times such as t_0 and t_2. In between these event times may be those such as t_1, which are times of internal events. The latter are noticeable on the *state trajectory*, which is a steplike series of states that change at external and internal events (second from top). The *elapsed time trajectory* is a sawtooth pattern depicting the flow of time in an elapsed time clock that is reset to 0 at every event. Finally, at the bottom, the *output tra-*

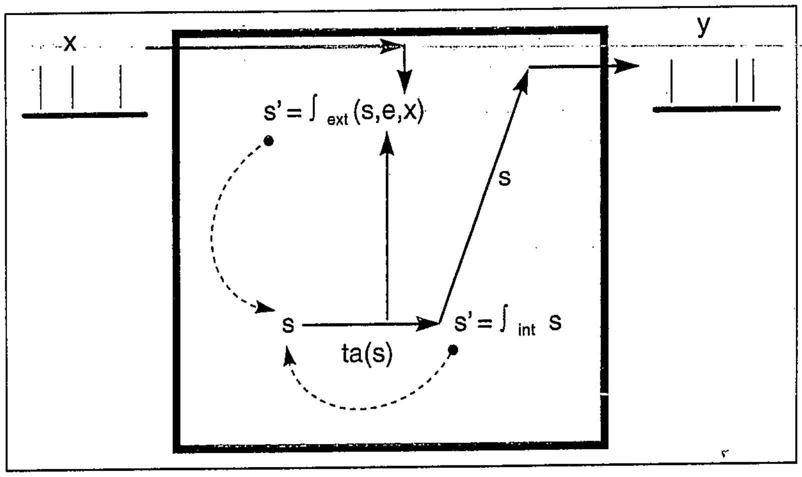


Figure 1 DEVS in action.

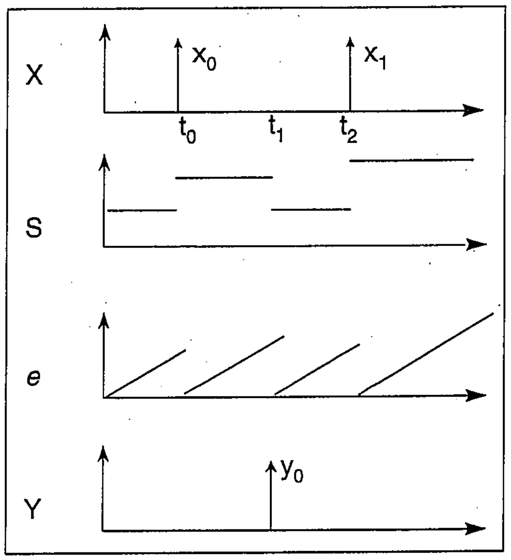


Figure 2 DEVS trajectories.

jectory depicts the output events that are produced by the output function just before applying the internal transition function at internal events. Such behaviors will be illustrated in the next examples.

4.2.1 DEVS Examples

We will present a number of models and discuss their behaviors to illustrate practical construction of DEVS models. Table 1 lists the models to be discussed. As will be detailed later, there are two classes of models in any object-oriented implementation of DEVS. *Atomic* models are expressed in the basic formalism. *Coupled* models are expressed using the coupled model specification—essentially providing component and coupling information.

We start with models that have only a single input and a single output, both expressed as real numbers.

Exercise: Write a simulator for any DEVS model that has scalar real values for its inputs, states, and outputs. Pending discussion of DEVS simulators in Chapter 8, you can use the event scheduling approach of Chapter 3 in which the external events in an input trajectory are all prescheduled on the event list. The DEVS is started in a state *s* and an internal event notice is placed in the event list with a time advance given by ta(*s*). This notice must be rescheduled appropriately every time an internal or external transition is processed.◆

Passive

The simplest DEVS to start with is one that literally does nothing. Appropriately called *passive*, it does not respond with outputs no matter what the input trajectory is. Illustrated in Fig. 3, the simplest realization of this behavior is:

$$DEVS = \left(X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta\right)$$

Table 1 Examples of DEVS models

Models	**I/O behavior description**
Atomic	
Classic DEVS	
passive	Never generates output
storage	Stores the input and responds with it when queried
generator	Outputs a 1 in a periodic fashion
binaryCounter	Outputs a 1 only when it has received an even number of 1's
nCounter	Outputs a 1 only when it has received a number of 1's equal to *n*
Infinite counter	Outputs the number of input events received since initialization
processor	Outputs the number it received after a processing time
ramp	Output acts like the position of a billiard ball that has been hit by the input
Parallel DEVS	
multiple input processor with queue	Processor with queue with the capability to accept multiple simultaneous inputs
Coupled	
Classic DEVS	
pipeSimple	Pipeline—processes successive inputs in assembly line fashion
netSwitch	Outputs the input it receives after a processing time (internally input is alternated between two processors)
Parallel DEVS	
pipeSimple	Same as before except for handling of event collisions
netSwitch	Revised Switch and coupled model handling simultaneous inputs on different ports
experimental frame	Generator and transducer encapsulated into a coupled model
Hierarchical DEVS	
frame-processor	Experimental frame coupled to processor (could be an port compatible model e.g., processor with buffer, netSwitch, etc)

where

$$X = \mathrm{R}$$

$$Y = \mathrm{R}$$

$$S = \{\text{“passive”}\}$$

$$\delta_{ext}(\text{“passive”}, e, x) = \text{“passive”}$$

$$\delta_{int}(\text{“passive”}) = \text{“passive”}$$

$$\lambda(\text{“passive”}) = \varnothing$$

$$ta(\text{“passive”}) = \infty.$$

The input and output sets are numerical. There is only one state that is "passive." In this state, the time advance given by *ta* is infinite. As already indicated, this means the system will stay passive forever unless interrupted by an input. However, in this model, even such an interruption will not awaken it, since the external transition function does disturb the state.

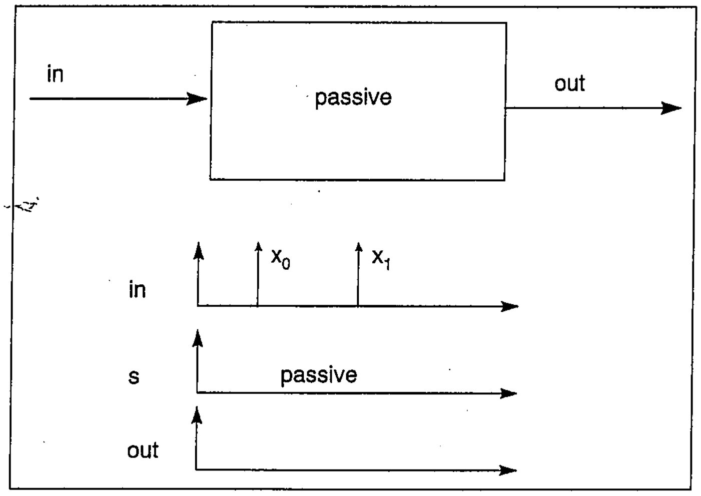


Figure 3 Passive DEVS.

The specifications of the internal transition and output functions are redundant here since they will never get a chance to be applied.

Storage

In contrast to the passive DEVS, the next system responds to its input and stores it forever, or until the next input comes along. This is not very useful unless we have a way of asking what is currently stored. So there is a second input to do such querying. Since there is only one input port in the current DEVS model, we let the input value of zero signal this query. As the first part of Fig. 4 shows, within a time, *response_time,* of the zero input arrival, the

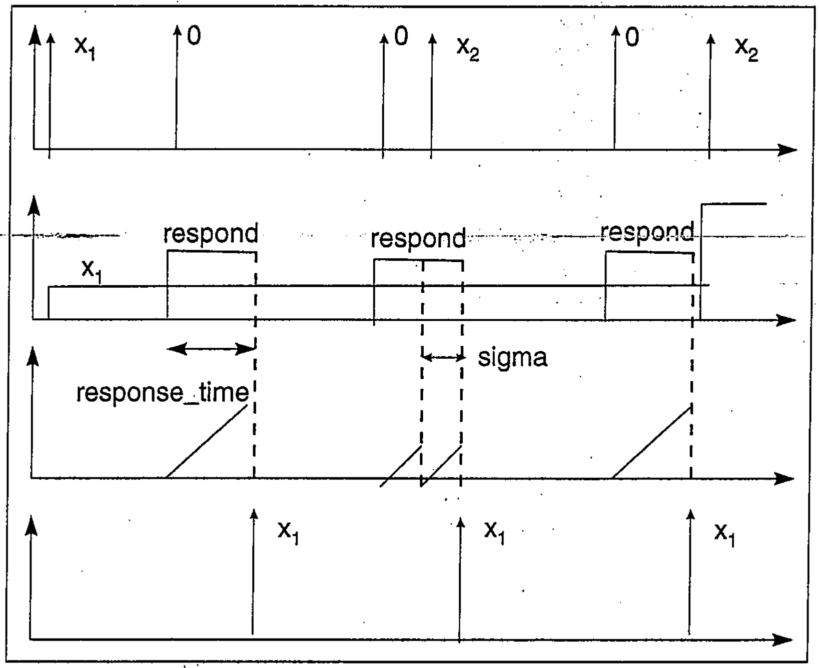


Figure 4 Trajectories for storage.

DEVS responds with the last stored nonzero input. To make this happen, the Storage is defined by:

$$DEVS_{response\_time} = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$$

where

$$X = \mathrm{R}$$

$$Y = \mathrm{R}$$

$$S = \{\text{"passive"}, \text{respond"}\} \times R_0^+ \times R - \{0\}$$

$$\delta_{ext}(\text{"passive"}, \sigma, \text{store}, \text{e}, \text{x}) = \begin{cases} (\text{"passive"}, \sigma - \text{e}, \text{x}) & \text{if x} \,!\!\neq 0 \\ (\text{"respond"}, \text{response\_time}, \text{store}) & \text{if x} = 0 \end{cases}$$

$$\delta_{ext}(\text{"respond"}, \sigma, \text{store}, \text{e}, \text{x}) = (\text{"respond"}, \sigma - \text{e}, \text{store})$$

$$\delta_{int}(\text{"respond"}, \sigma, \text{store}) = (\text{"passive"}, \infty, \text{store})$$

$$\lambda(\text{"respond"}, \sigma, \text{store}) = \text{store}$$

$$ta(\text{phase}, \sigma, \text{store}) = \sigma.$$

There are three state variables: *phase* with values {"passive","respond"}, σ, having positive real values, and *store* having real values other than zero. We need the active "respond" phase to tell when a response is underway. As for σ, the definition of *ta* shows that it stores the time advance value. In other words, σ is the time remaining in the current state. Note that when an external event arrives after elapsed time, *e*, σ is reduced by *e* to reflect the smaller time remaining in the current state. When a zero input arrives, σ is set to *response_time* and the "respond" phase is entered. However, if the system is in phase "respond" and an input arrives, the input is ignored as illustrated in the second part of Fig. 4. When the *response_time* period has elapsed, the output function produces the stored value and the internal transition dictates a return to the passive state. What happens if, as in the third part of Fig. 4, an input arrives just as the response period has elapsed? Classic DEVS and Parallel DEVS differ in their ways of handling this collision, as we shall show later.

Note that we subscript the DEVS with *response_time*, to indicate that the latter is a parameter of the model—it needs to be specified before execution but it doesn't change subsequently, under the model's own actions.

Generator

Although the store reacts to inputs, it does not do anything on its own. A simple example of a proactive system is a *generator*. As illustrated in Fig. 5, it has no inputs but when started in phase "active," it generates outputs with a specific period. The generator has the two basic state variables, *phase* and σ. Note that the generator remains in phase "active" for the *period*, after which the output function generates a "one" output and the internal transition function resets the phase to "active" and σ to *period*. Strictly speaking, restoring

/S)

happen, the

if x ! ≠ 0

if x = 0

,"respond"},
han zero. We
erway. As for
lue. In other
ıen an exter-
:t the smaller
:s, σ is set to
the system is
illustrated in
ɔsed, the out-
on dictates a
t of Fig. 4, an
EVS and Par-
ıll show later.
icate that the
ɔre execution
tions.

s own. A sim-
n Fig. 5, it has
ts with a spe-
phase and σ.
d, after which
ınsition func-
ing, restoring

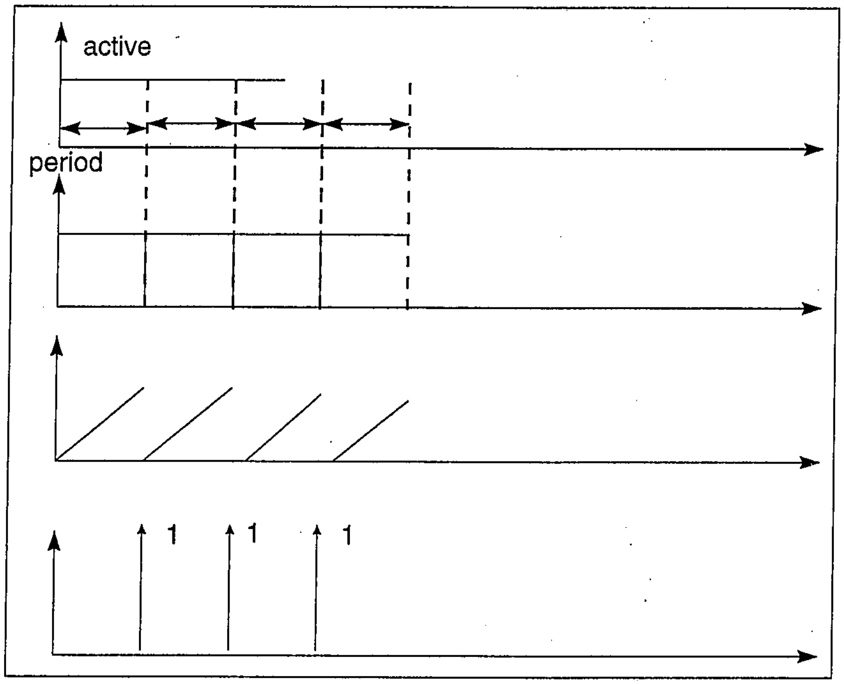


Figure 5 Generator trajectories.

these values isn't necessary unless inputs are allowed to change them (which they aren't in this model).

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

where

$$X = \{\}$$
$$Y = \{1\}$$
$$S = \{\text{“passive”}, \text{“active”}\} \times R^{+}$$
$$\delta_{int}(\text{phase}, \sigma) = (\text{“active”}, \text{period})$$
$$\lambda(\text{“active”}, \sigma) = 1$$
$$ta(\text{phase}, \sigma) = \sigma$$

Binary Counter

In this example, the DEVS outputs a "one" for every two "one"s that it receives. To do this it maintains a count (modulo 2) of the "one"s it has received to date. When it receives a "one" that makes its count even, it goes into a transitory phase, "active," to generate the output. This is the same as putting *response_time* := 0 in the storage DEVS.

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

where

$$X = \{0.1\}$$
$$Y = \{1\}$$
$$S = \{\text{“passive”}, \text{“active”}\} \times R_0^+ \times \{0.1\}$$
$$\delta_{ext}(\text{“passive”}, \sigma, \text{count}, e, x) =$$
$$\begin{array}{ll} (\text{“passive”}, \sigma, -e, \text{count} + x) & \text{if count} + x < 2 \\ (\text{“active”}, 0, 0) & \text{otherwise} \end{array}$$
$$\delta_{int}(\text{phase}, \sigma, \text{count}) = (\text{“passive”}, \infty, \text{count})$$
$$\lambda(\text{“active”}, \sigma, \text{count}) = 1$$
$$ta(\text{phase}, \sigma, \text{count}) = \sigma$$

Exercise: An nCounter generalizes the binary counter by generating a “one” for every n “one”s it receives. Specify an nCounter as an atomic model in DEVS.◆

Exercise: Define a DEVS counter that counts the number of non-zero input events received since initialization and outputs this number when queried by a zero valued input.◆

Processor

A model of a simple workflow situation is obtained by connecting a generator to a processor. The generator outputs are considered to be jobs to do and the processor takes some time to do them. In the simplest case, no real work is performed on the jobs, only the times taken to do them are represented.

$$DEVS_{processing\_time} = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta),$$

where

$$X = \mathrm{R}$$
$$Y = \mathrm{R}$$
$$S = \{\text{“passive”}, \text{“active”}\} \times R_0^+ \times \mathrm{R}$$
$$\delta_{ext}(\text{phase}, \sigma, \text{job}, e, x) =$$
$$\begin{cases} (\text{“busy”}, \text{processing\_time}, \text{job}) & \text{if passive} = \text{“passive”} \\ (\text{phase}, \sigma - e, x) & \text{otherwise} \end{cases}$$
$$\delta_{int}(\text{phase}, \sigma, \text{job}) = (\text{“passive”}, \text{processing\_time}, \text{job})$$
$$\lambda(\text{“busy”}, \sigma, \text{job}) = \text{job}$$
$$ta(\text{phase}, \sigma, \text{job}) = \sigma$$

Here, if the DEVS is passive (idle), when it receives a job, it stores the job and goes into phase “busy.” If it is already busy, it ignores incoming jobs. After a period, *processing_time,* it outputs the job and returns to “passive.”

unt + x < 2

rwise

erating a "one"
omic model in

non-zero input
: when queried

ecting a gener-
.o be jobs to do
ɜst case, no real
hem are repre-

/e = "passive"

ɜe

it stores the job
incoming jobs.
ɪs to "passive."

Exercise: Draw typical input/state/elapsed time/output trajectories for the simple processor.◆

Ramp

As we shall see later, DEVS can model systems whose discrete event nature is not immediately apparent. Consider a billiard ball. As illustrated in Fig. 6, struck by a cue (external event), it heads off in a direction at constant speed determined by the impulsive force imparted to it by the strike. Hitting the side of the table is considered as another input that sets the ball off going in a well-defined direction. The model is described by

$$DEVS_{step\_time} = \left(X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta\right),$$

where

$$X = \mathrm{R}$$
$$Y = \mathrm{R}$$
$$S = \{\text{"passive, "active}\} \times R_0^+ \times \mathrm{R} \times \mathrm{R}$$
$$\delta_{ext}(\text{phase}, \sigma, \text{position}, \text{input}, \text{e}, \text{x}) = (\text{"active"}, \sigma - \text{e}, \text{x}, \text{position} + \text{e} * \text{input})$$
$$\delta_{int}(\text{phase}, \sigma, \text{input}, \text{position}) = (\text{"active"}, \text{step\_time}, \text{input}, \text{position} + \sigma * \text{input})$$
$$\lambda(\text{phase}, \sigma, \text{input}, \text{position}) = \text{position} + \sigma * \text{input}$$
$$ta(\text{phase}, \sigma, \text{input}, \text{position}) = \sigma$$

The model stores its input and uses it as the value of the slope to compute the position of the ball (in a one-dimensional simplification). It outputs

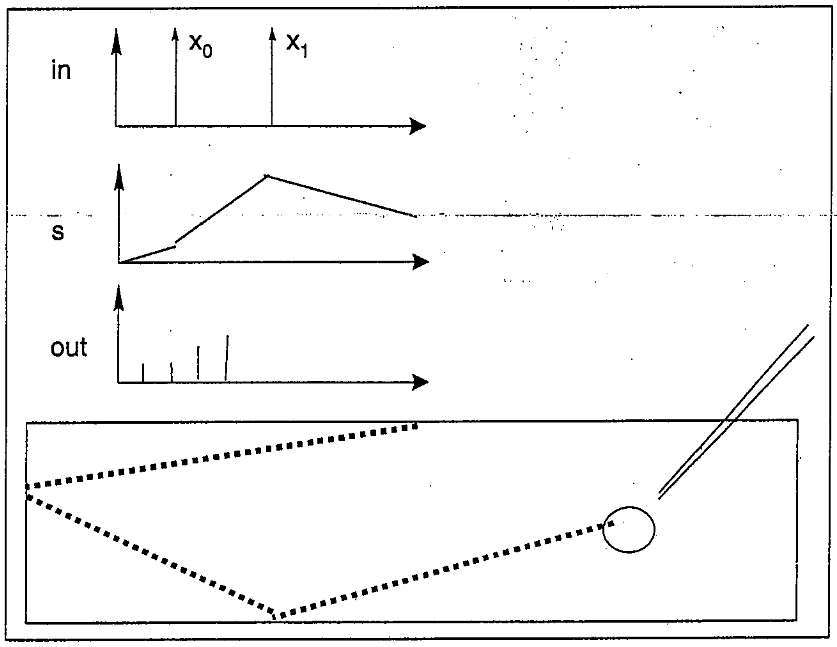


Figure 6 Ramp trajectories.

the current position every *step_time*. If it receives a new slope in the middle of a period, it updates the position to a value determined by the slope and elapsed time. Note that it outputs the position prevailing at the time of the output. This must be computed in the output function since it always called before the next internal event actually occurs. Note that we do not allow the output function to make this update permanent. The reason is that the output function is not allowed to change the state in the DEVS formalism. (A model in which this is allowed to happen may produce **unexpected and hard-to-locate** errors because DEVS simulators do not guarantee correctness for nonconforming models.)

4.2.2 Classic DEVS With Ports

Modeling is made easier with the introduction of input and output ports. For example, a DEVS model of a storage naturally has two input ports, one for storing and the other for retrieving. The more concrete DEVS formalism with port specifications is as follows:

$$DEVS = \left(X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta\right)$$

where

$X = \{(p,v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values

$Y = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values

S is the set of *sequential* states

δ_{ext}: $Q \times X \to S$ is the *external state transition function*

δ_{int}: $S \to S$ is the *internal state transition function*

λ: $S \to Y$ is the output function

ta: $S \to R^{+}_{0,\infty}$ is the *time advance function*

$Q := \{(s,e) \mid s \in S, 0 \le e \le ta(s)\}$ is the set of *total states.*

Note that in classic DEVS, only one port receives a value in an external event. We shall see later that parallel DEVS allows multiple ports to receive values at the same time.

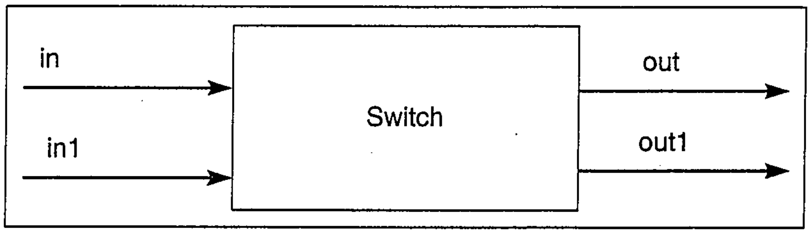


Figure 7 Switch with input and output ports.

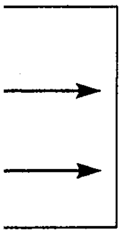
in the middle
the slope and
ne time of the
always called
not allow the
s that the out-
formalism. (A
expected and
antee correct-

put ports. For
ports, one for
rmalism with

l values

nd values

in an external
orts to receive



Switch

A switch is modeled as a DEVS with pairs of input and output ports, as shown in Fig. 7. When the switch is in the standard position, jobs arriving on port "in" are sent out on port "out", and similarly for ports "in1" and "out1." When the switch is in its other setting, the input-to-output links are reversed, so that what goes in on port "in" exits at port "out1," etc.

In the following switch DEVS, in addition to the standard *phase* and σ variables, there are state variables for storing the input port of the external event, the input value, and the polarity of the switch. In this simple model, the polarity is toggled between true and false at each input.

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

where

$InPorts = \{\text{"}in\text{"}, \text{"}in1\text{"}\}$, where $X_{in} = X_{in1} = V$(an arbitrary set)

$X = \{(p,v) \mid p \in InPorts\ v \in X_p\}$ is the set of input ports and values;

$\text{OutPorts} = \{\text{"out"}, \text{"out1"}\}$, where $Y_{out} = Y_{out1} = V$(an arbitrary set),

$Y = \{(p,v) \mid p \in OutPorts,\ v \in Y_p\}$ is the set of output ports and values;

$S = \{\text{"passive"}, \text{"active"}\} \times R_0^+ \times \{in, in1\} \times V \times \{true, false\}$

$\delta_{ext}(\text{phase}, \sigma, \text{ inport, store, Sw}, e, (p,v))$

$= (\text{"busy", processing\_time}, p, v, !\text{Sw})$ if phase = "passive" and $p \in \{in, in1\}$

$(\text{phase}, \sigma - e, \text{ inport, store, Sw})$ otherwise

$\delta_{int}(\text{phase}, \sigma, \text{ inport, store, Sw}, e, (p,v)) = (\text{"passive"}, \infty, \text{ inport, store, Sw})$

$\lambda(\text{phase}, \sigma, \text{ inport, store, Sw})$

$= (\text{out}, \text{store})$ if phase = "busy" and Sw = *true* and inport = in

= (~~out1, store~~) if phase = "busy" and ~~Sw = *true* and inport = in1~~

$= (\text{out1}, \text{store})$ if phase = "busy" and Sw = *false* and inport = in

$= (\text{out}, \text{store})$ if phase = "busy" and Sw = *true* and inport = in1

$ta(\text{phase}, \sigma, \text{ inport, store, Sw}) = \sigma$

4.2.3 Classic DEVS Coupled Models

The DEVS formalism includes the means to build models from components. The specification in the case of DEVS with ports includes the external interface (input and output ports and values), the components (which must be DEVS models), and the coupling relations:

$$N = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select),$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values

D is the set of the component names.

Component Requirements

- components are DEVS models, for each $d \in D$,

$$M_d = (X_d, Y_d, S, \delta_{ext}\ \delta_{int}, \lambda, ta) \text{ is a DEVS}$$
$$\text{with } X_d = \{(p,v) \mid p \in IPorts_d, v \in X_p\}$$
$$Y_d = \{(p,v) \mid p \in OPorts_d, v \in Y_p\}.$$

Coupling Requirements

- external input coupling connect external inputs to component inputs,

$$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}.$$

- external output coupling connect component outputs to external outputs,

$$EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}.$$

- internal coupling connects component outputs to component inputs:

$$IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in D, \in OPorts_a, ip_b \in IPorts_b\}.$$

However, *no direct feedback loops are allowed,* i.e., no output port of a component may be connected to an input port of the same component:

$$((d, opd), (e, ipd)) \in IC \text{ implies } d \neq e.$$

(We omit constraints on the interface sets, which are covered in Chapter 5.)

- *Select:* $2^D - \{\} \to D$, the tie-breaking function (used in Classic DEVS but eliminated in Parallel DEVS).

Simple Pipeline

We construct a simple coupled model by placing three processors in series to form a pipeline. As shown in Fig. 8, we couple the output port "out" of the first processor to the input port "in" of the second, and likewise for the second and third. This kind of coupling is called *internal coupling (IC)* and, as in the preceding specification, it is always of the form where an output port connects to an input port. Since coupled models are themselves usable as components in bigger models, we give them input and output ports. The pipeline has an input port "in" that we connect to the input port "in" of the first processor. This is an example of *external input coupling (EIC)*. Likewise, there is an external output port "out" that gets its values from the last processor's "out" output port. This illustrates *external output coupling (EOC)*.

ɪ values

ınd values

nent inputs,

$\text{ʔ}orts_d\}$.

:ternal outputs,

$ƆPorts_d\}$.

nent inputs:

$\in IPorts_b\}$.

ıutput port of a

:omponent:

ɜd in Chapter 5.)

assic DEVS but

essors in series

ıt port "out" of

likewise for the

ıpling (IC) and,

here an output

·mselves usable

ıtput ports. The

port "in" of the

ing (EIC). Like-

es from the last

utput coupling

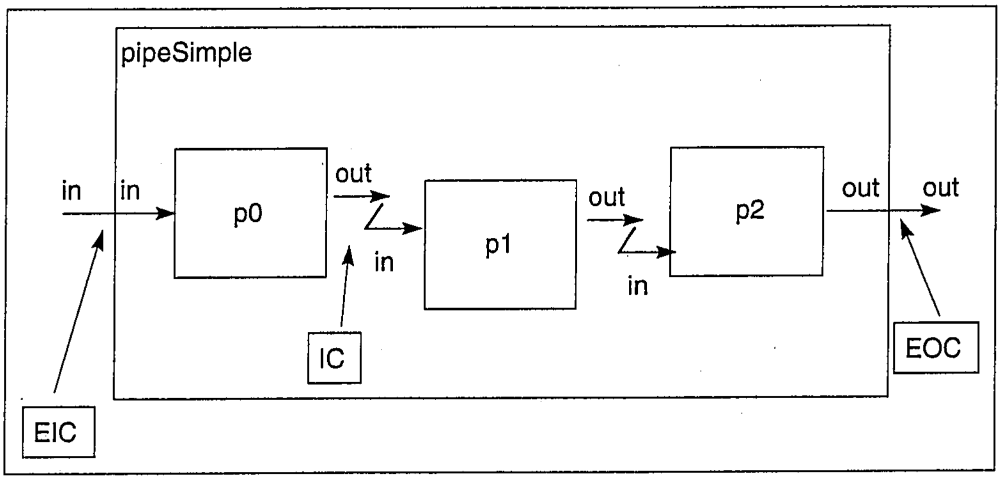


Figure 8 Pipeline coupled model.

The coupled DEVS specification of the pipeline is

$$N = \left(X, Y, D, \left\{M_d \mid d \in D\right\}, EIC, EOC, IC\right),$$

where

$$InPorts = \{\text{"}in\text{"}\},$$

$$X_{in} = V \text{ (an arbitrary)}$$

$$X = \{(\text{"}in\text{"}, v) \mid v \in V\}$$

$$OutPorts = \{\text{"}in\text{"}\}$$

$$Y_{out} = V$$

$$Y = \{(\text{"}out\text{"}, v) \mid v \in V\}$$

$$D = \{processor0, processor1, processor2\}$$

$$M_{processor2} = M_{processor1} = M_{processor0 = Processor}$$

$$EIC = \{((N, \text{"}in\text{"}), (processor0, \text{"}in\text{"})\}$$

$$EOC = \{((processor2, \text{"}out\text{"}), (N, \text{"}out\text{"})\}$$

$$IC = \{((processor0, \text{"}out\text{"}), (processor1, \text{"}in\text{"})), ((processor1, \text{"}out\text{"}), (processor2, \text{"}in\text{"}))\}$$

$Select(D')$ = the processor in D' with the highest index.

The interpretation of coupling specifications is illustrated in Fig. 9. An external event, x_1 arriving at the external input port of pipeSimple is transmitted to the input port "in" of the first processor. If the latter is passive at the time, it goes into phase "busy" for the processing time. Eventually, the job appears on the "out" port of processor0 and is transmitted to the "in" port of processor1 because of the internal coupling. This transmission from output port to input port continues until the job leaves the external output port of the last processor and appears at the output port of pipeSimple because of external output coupling.

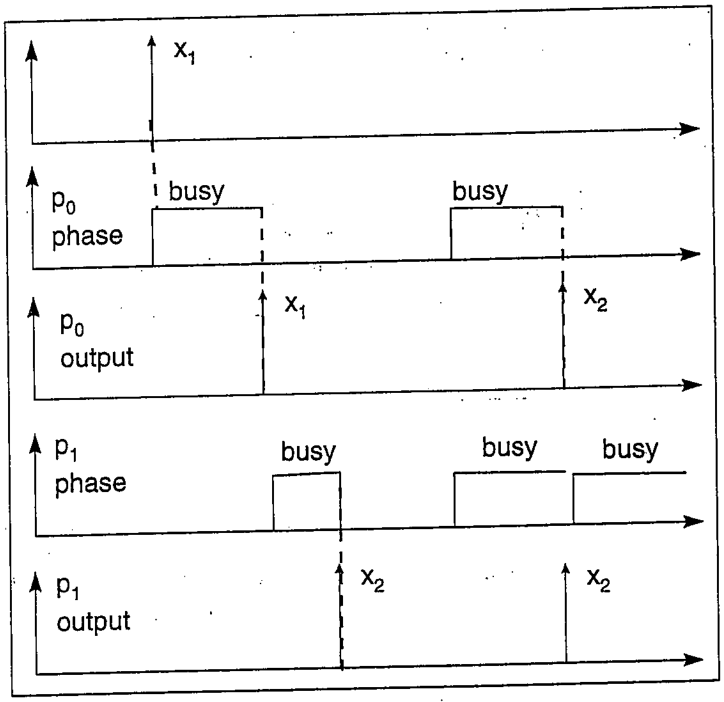


Figure 9 Message transmission and simultaneous events.

Now if the jobs arrive at the same rate that they are processed, the situation shown in the second part of Fig. 9 will arise. The outputs of processor0 and processor1 are generated at the same time with the output of processor0 appearing as the input to processor1. There are now two choices for the modeler to make: (a) should processor1 apply its *internal transition function* to return to phase "passive" in which it can accept the upstream input, or (b) should it apply its *external transition function* first, in which case it will lose the input since it is in phase "busy"? The *Select* function specifies that if both processor0 and processor1 are imminent, then processor1 is the one that generates its output and carries out its internal transition. Thus it enforces alternative (a). In parallel DEVS, we shall see that a similar solution can be achieved without serializing imminent component actions.

Switch Network

A second example of a coupled network employs the *switch DEVS* model defined before to send jobs to a pair of *processors*.

As shown in Fig. 10, the "out" and "out1" ports of the switch are coupled individually to the "in" input ports of the processors. In turn, the "out" ports of the processors are coupled to the "out" port of the network. This allows an output of either processor to appear at the output of the overall network. The coupled DEVS specification is

$$N = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC),$$

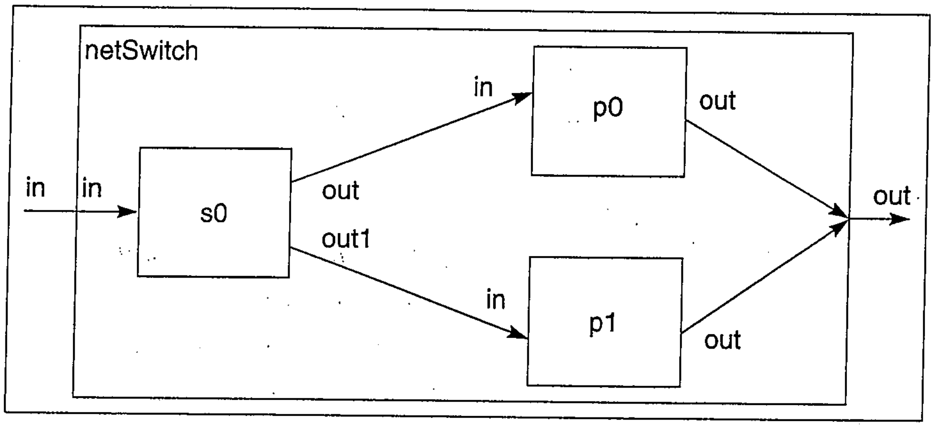


Figure 10 Switch network.

where

$$
\begin{aligned}
&InPorts = \{\text{“}in\text{”}\}\\
&X_{in} = V(\text{an arbitrary set})\\
&X_M = \{(v) \mid v \in V\}\\
&OutPorts = \{\text{“}out\text{”}\}\\
&Y_{out} = V\\
&Y_M = \{(\text{“}out\text{”}, v) \mid v \in V\}\\
&D = \{Switch0, processor0, processor1\}\\
&M_{Switch0} = Switch;\ M_{processor1} = M_{processor0} = Processor\\
&EIC = \{((N, \text{“}in\text{”}), (Switch0, \text{“}in\text{”}))\}\\
&EOC = \{((processor0, \text{“}out\text{”}), (N, \text{“}out\text{”})),\\
&\qquad ((processor1, \text{“}out\text{”}), (N, \text{“}out\text{”}))\}\\
&IC = \{((Switch0, \text{“}out\text{”}), (processor0, \text{“}in\text{”})),\\
&\qquad ((Switch0, \text{“}out1\text{”}), (processor1, \text{“}in\text{”}))\}.
\end{aligned}
$$

4.3 Parallel DEVS System Specification

Parallel DEVS differs from classic DEVS in allowing all imminent components to be activated and to send their output to other components. The receiver is responsible for examining this input and properly interpreting it. Messages, basically lists of port–value pairs, are the basic exchange medium. This section discusses Parallel DEVS and gives a variety of examples to contrast with classic DEVS.

A basic Parallel DEVS is a structure,

$$DEVS = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

where

$X_M = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values

$Y_M = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values

S is the set of *sequential* states

$\delta_{ext}: Q \times X_M^b \to S$	is the *external state transition function*
$\delta_{int}: S \to S$	is the *internal state transition function*
$\delta_{con}: Q \times X_M^b \to S$	is the *confluent transition function*
$\lambda: S \to Y^b$	is the output function
$ta: S \to R_{0,\infty}^+$	is the *time advance function*

$Q: = \{(s,e) \mid s \in S, 0 \le e \le ta(s)\}$ is the set of *total states*.

Note that instead of having a single input, basic Parallel DEVS models have a bag of inputs. A *bag* is a set with possible multiple occurrences of its elements. A second difference is noted—this is the addition of a transition function, called *confluent*. It decides the next state in cases of collision between external and internal events. We have seen examples of such collisions earlier in examining Classic DEVS. There is also a coupled model version of parallel DEVS that matches the assumptions discussed earlier. We return to discuss it after discussing some examples.

4.3.1 Processor With Buffer

A processor that has a buffer is defined in Parallel DEVS as

$$DEVS_{processng\_time} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta),$$

where

$IPorts = \{\text{“in”}\}$, where $X_{in} = V(\text{an arbitrary set})$

$X_M = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values

$OPorts = \{\text{“out”}\}$, where $Y_{out} = V(\text{an arbitrary set})$

$Y_M = \{(p,v) \mid p \in IPorts, v \in Y_p\}$ is the set of output ports and values

$S = \{\text{“passive”}, \text{“busy”}\} \times R_0^+ \times V^+$

$\delta_{ext}(\text{“phase”}, \sigma, q, e, ((\text{“in”}, x1), (\text{“in”}, x2), \ldots, (\text{“in”}, xn)))$

$= (\text{“busy”}, processing\_time, x1, x2, \ldots, xn)$ if phase = “passive”

$= (phase, \sigma - e, q.x1, x2, \ldots, xn)$ otherwise

$\delta_{int}(phase, \sigma, q.v) = (\text{“passive”}, processing\_time, q)$

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$

$\lambda(\text{“busy”}, \sigma, q.v) = v$

$ta(phase, \sigma, q) = \sigma.$

Using its buffer the processor can store jobs that arrive while it is busy. The buffer, also called a queue, is represented by a sequence of jobs, x1 . . . xn in V^+ (the set of finite sequences of elements of V). In Parallel DEVS, the processor can also handle jobs that arrive simultaneously on in its input port. These are placed in its queue and it starts working on the one it has selected to be first. Note that bags, like sets, are not ordered so there is no ordering of the jobs in the input bag. For convenience we have shown the job which is last in the written order in δ_{ext} as the one selected as the one to be processed. Figure 11 illustrates the concurrent arrival of two jobs and the subsequent arrival of a third just as the first job is about to be finished. Note that the confluent function, δ_{con} specifies that the internal transition function is to be applied first. Thus, the job in process completes and exists. Then the external function adds the third job to the end of the queue and starts working on the second job. Classic DEVS has a hard time doing all this as easily!

Exercise: Write a Parallel DEVS for the storage model with separate ports for store and query inputs. Also resolve the collision between external and internal transitions when a store input occurs.◆

Exercise: Write a Parallel DEVS for a generator that has ports for starting and stopping its generation. Explore various possibilities for using the elapsed time when an stop input occurs. How does each of these choice affect the behavior of the start input.◆

4.3.2 Parallel DEVS Coupled Models

Parallel DEVS coupled models are specified in the same way as in classic DEVS except that the Select function is omitted. While this is an innocent-looking change, its semantics are much different—they differ significantly in how

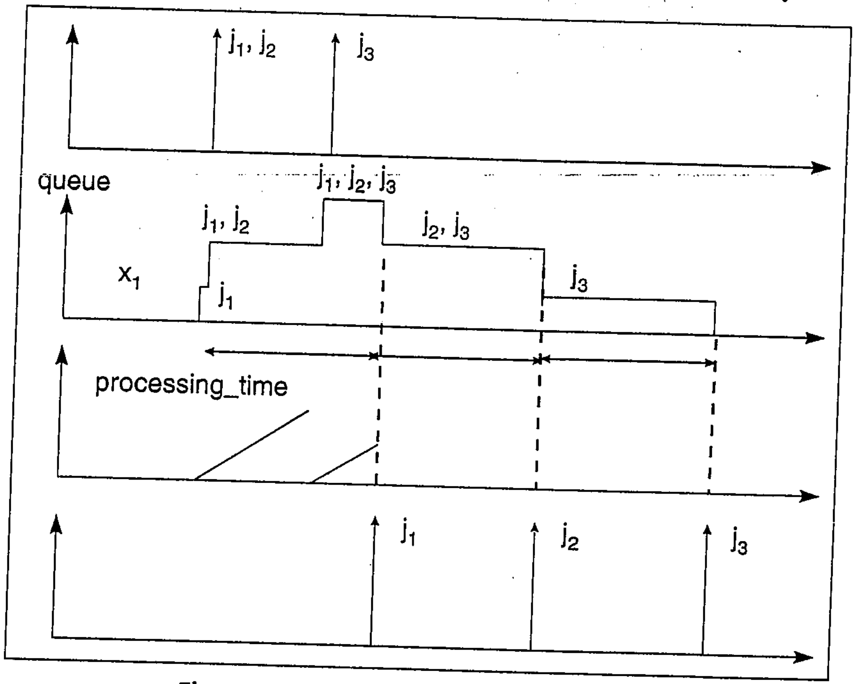


Figure 11 Parallel DEVS process with buffer.

imminent components are handled. In parallel DEVS there is no serialization of the imminent computations—all imminent components generate their outputs which are then distributed to their destinations using the coupling information. The detailed simulation algorithms for parallel DEVS will be discussed in later chapters. For now, let's revisit the simple pipeline in Fig. 8 to see how the interpretation underlying Parallel DEVS differs from that of Classic DEVS.

Simple Pipeline (Parallel DEVS)

Recall that the specification of the pipeline model is the same as in Classic DEVS except for omitting the Select function. In Fig. 12, processor0 and processor1 are imminent. In classic DEVS, only one would be chosen to execute. In contrast, in Parallel DEVS, they both generate their outputs. Processor0's output goes to processor1's input. Since now processor1 has both internal and external events, the confluent transition function is applied. As in Fig. 11, the confluent function is defined to first apply the internal transition function and then the external one. As we have seen, this causes processor1 to complete the finished job before accepting the incoming one.

Now consider a pipeline in which a downstream processor's output is fed back to an upstream processor. For example, let processor1's output be fed back to precessor1's input. In Classic DEVS, the Select function must always make the same choice among imminent components. Thus, either one or the other processor will lose its incoming job (assuming no buffers). However, in Parallel DEVS, both processors output their jobs and can handle the priorities between arriving jobs and just finished jobs in a manner specified by the confluent transition function.

Exercise: Define the coupled model for a cyclic pipeline and draw a state trajectory similar to that in Fig. 12.◆

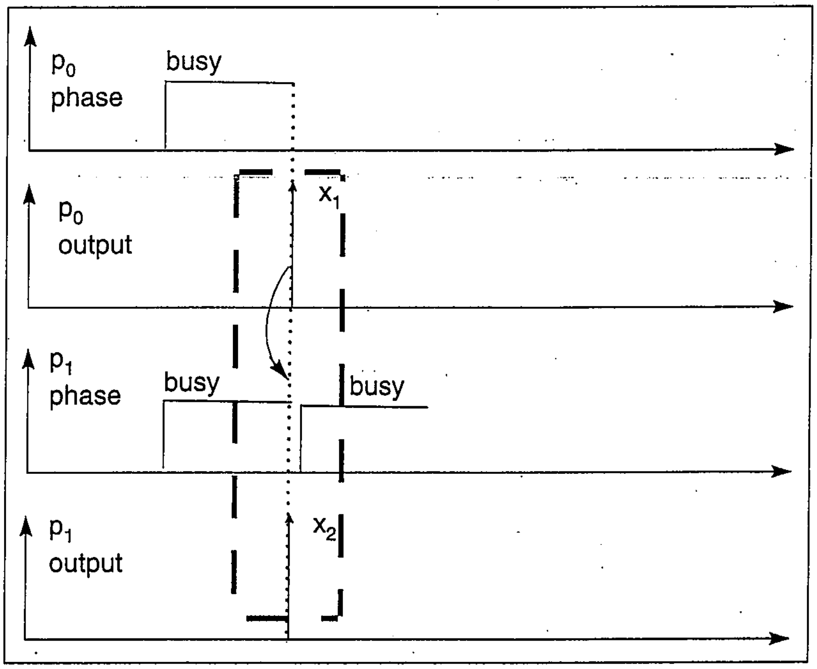


Figure 12 Illustrating handling of imminent components in Parallel DEVS.

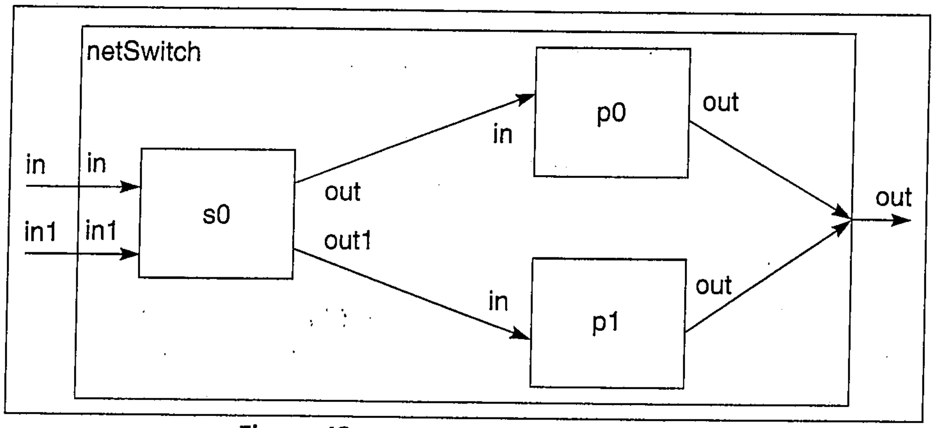


Figure 13 A switching net in Parallel DEVS.

Exercise: Write a Parallel DEVS basic model for a Switch that can accept simultaneous input on two ports as shown in Fig. 13. Write a Parallel DEVS coupled model as shown in the figure and draw its state trajectories.◆

4.4 Hierarchical Models

Hierarchical models are coupled models with components that may be atomic or coupled models. As Fig. 14 illustrates, we encapsulate a generator and a transducer into a coupled model called an experimental frame, *ef*. The experimental frame can then be coupled to a processor to provide it with a stream of inputs (from the generator) and observe the statistics of the processor's operation (the transducer). The processor can be an atomic model (as is the processor with buffer, for example) or can itself be a coupled model (the coupled model in Fig. 13).

4.5 Object-Oriented Implementations of DEVS: An Introduction

DEVS is most naturally implemented in computational form in an object-oriented framework. The basic class is *entity*, from which class *devs* is derived (Fig. 15). *Devs* in turn is specialized into classes *atomic* and *coupled*. *Atomic* enables models to be expressed directly in the basic DEVS formal-

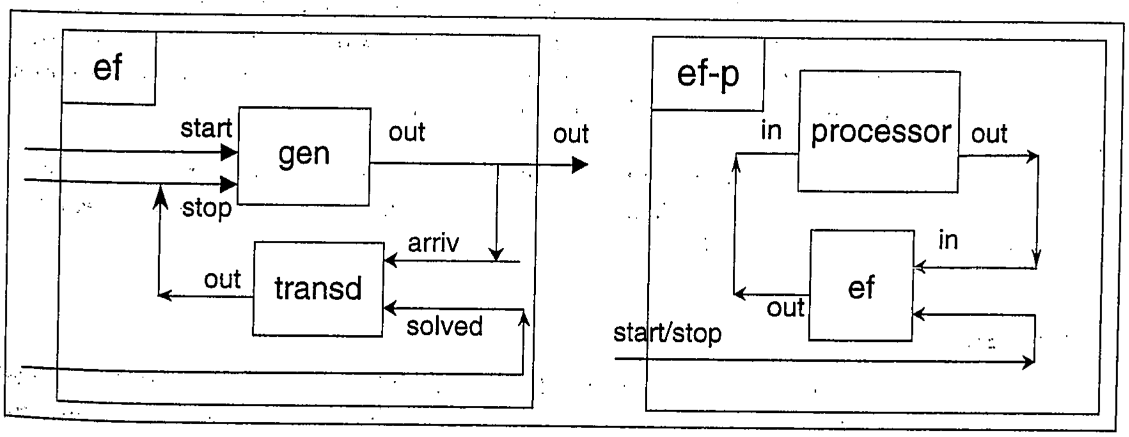


Figure 14 Hierarchical model.

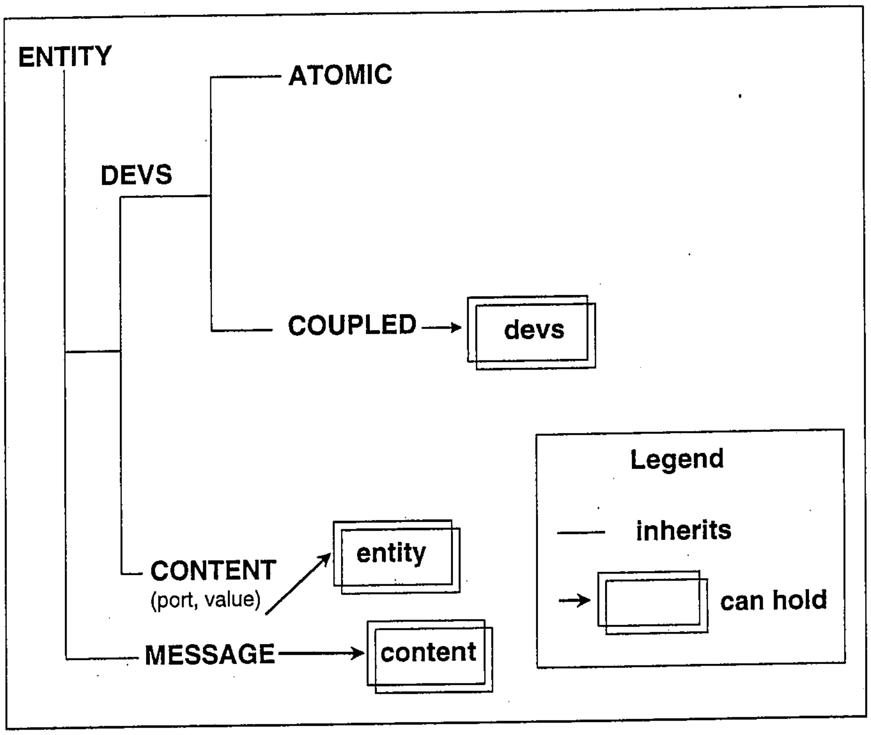


Figure 15 Basic classes in object-oriented DEVS.

ism. *Coupled* supports construction of models by coupling together components. Class *content* is derived from *entity* to carry *ports* and their *values*, where the latter can be any instance of *entity* or its derived classes. Typically a modeler defines such derived classes to structure the information being exchanged among models. In the implementation of Parallel DEVS, the outputs of component models are instances of class *message*, which are containers of *content* instances.

Figure 16 illustrates the implementation of a simple processor as a derived class of *atomic* in DEVSJAVA, a realization of Parallel DEVS. Notice the one-to-one correspondence between the formal DEVS elements, e.g., external transition function, and their direct expression as methods in DEVSJAVA.

Figure 17 suggests how an experimental frame class can be defined as a derived class of coupled. Notice the correspondence to the coupled model formalism of DEVS, viz., creation of components as instances of existing classes, declaration of ports and couplings.

Exercise: Use the approach of Fig. 17 to define the hierarchical model in Fig. 14 in DEVSJAVA. (Hint: Add instances of classes *ef* and *processor* as components.)◆

4.5.1 Structural Inheritance

Inheritance, as provided by object-oriented languages such as JAVA, enables construction of chains of derivations forming class inheritance hierarchies. This means that DEVSJAVA supports not just immediate derivation from ɜnd

ɜrits

can hold

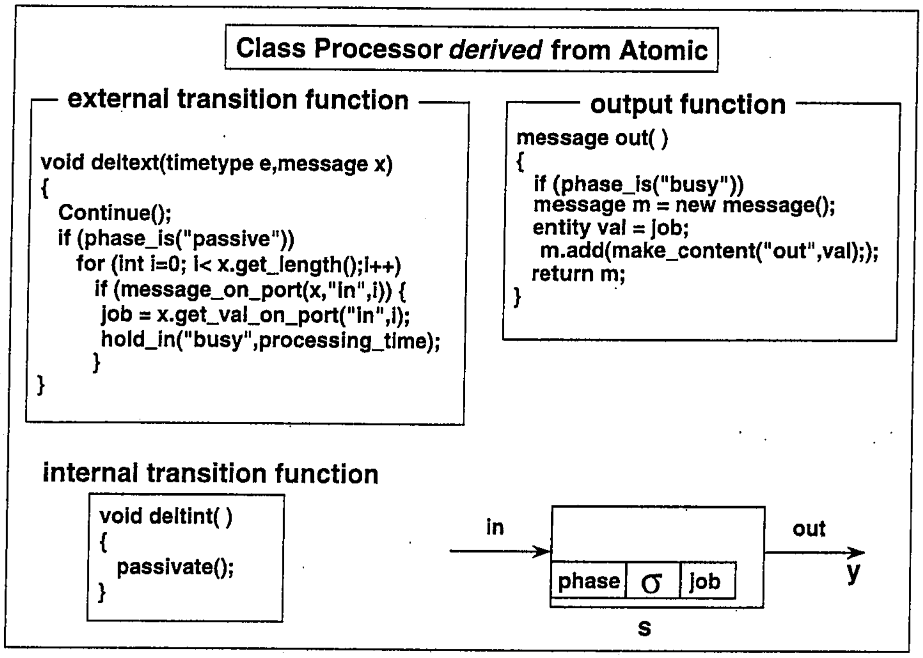
ᐟS.

ɪg together com-
and their *values,*
classes. Typically
ıformation being
el DEVS, the out-
ɜ, which are con-

e processor as a
DEVS. Notice the
nents, e.g., exter-
ɔds in DEVSJAVA.
ın be defined as a
ıe coupled model
ances of existing

ırchical model in
and *processor* as

ı as JAVA, enables
tance hierarchies.
ɜ derivation from


Class Processor *derived* from Atomic
external transition function
output function
internal transition function
in
out
y
phase
σ
job
s

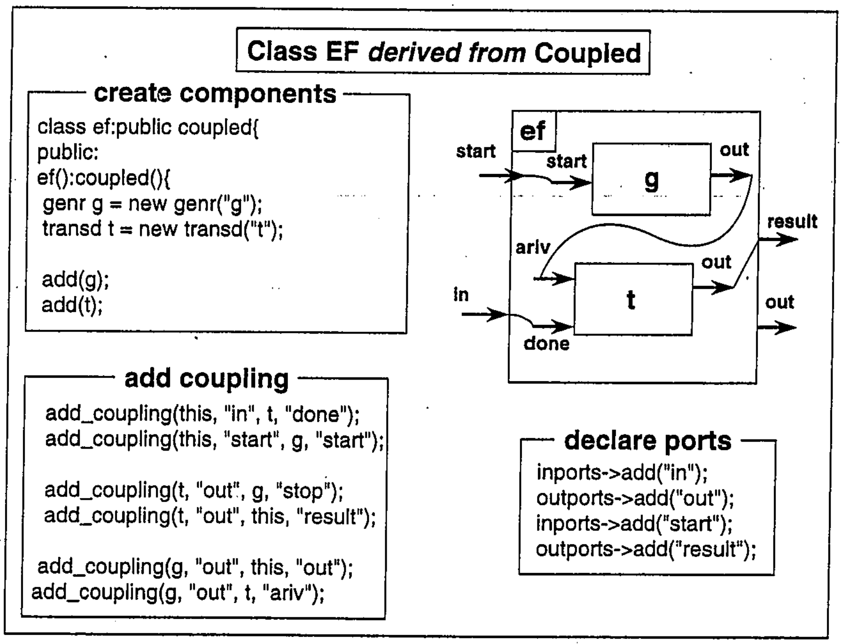
```
void deltext(timetype e,message x)
{
  Continue();
  if (phase_is("passive"))
    for (int i=0; i< x.get_length();i++)
      if (message_on_port(x,"in",i)) {
       job = x.get_val_on_port("in",i);
       hold_in("busy",processing_time);
      }
}
```

```
message out( )
{
  if (phase_is("busy"))
  message m = new message();
  entity val = job;
   m.add(make_content("out",val););
  return m;
}
```

```
void deltint( )
{
   passivate();
}
```

Figure 16 Processor as an atomic model in DEVSJAVA.

classes *atomic* and *coupled,* but also derivation of new model classes from those **already defined.** This is called *structural* inheritance because structure definitions, such as ports in atomic models and couplings in coupled models can be reused, and even modified, thus exploiting the polymorphism of object-orientation.



```
class ef:public coupled{
public:
ef():coupled(){
 genr g = new genr("g");
 transd t = new transd("t");

 add(g);
 add(t);
```

```
 add_coupling(this, "in", t, "done");
 add_coupling(this, "start", g, "start");

 add_coupling(t, "out", g, "stop");
 add_coupling(t, "out", this, "result");

add_coupling(g, "out", this, "out");
add_coupling(g, "out", t, "ariv");
```

```
inports->add("in");
outports->add("out");
inports->add("start");
outports->add("result");
```

Figure 17 Experimental frame as a coupled model in DEVSJAVA.

Exercise: Define a class of "balking" processors in which a job is sent out immediately on a "balk" port if it cannot be processed upon arrival. Rewrite the this class as a subclass of class processor and assess the extent to which code is reusable.◆

Exercise: Define a class pipeSimple for a simple pipeline of processors (Fig. 8). Derive a class from pipeSimple in which the components are "balking" processors and whose "balk" ports are coupled to a "balk" external output port. Assess the reusability of code and the ease or difficulty in achieving such reuse.◆

4.6 Summary

The form of DEVS (discrete event system specification) discussed in this chapter provides a hierarchical, modular approach to constructing discrete event simulation models. In doing so, the DEVS formalism embodies the concepts of systems theory and modeling introduced in Chapters 1. We will see later that DEVS is important not only for discrete event models, but also because it affords a computational basis for implementing behaviors that are expressed in DESS and DTSS, the other basic systems formalisms. This chapter introduced both Classic DEVS and the revision oriented to parallel and distributed simulation called Parallel DEVS. We also saw the essentials of how DEVS is implemented in an object-oriented framework.

Having a good understanding of how DEVS is used to model real systems should motivate us to ask deeper questions, such as whether DEVS always specifies meaningful systems, how simulation of a DEVS model works, and how other formalisms can be embedded in DEVS. To answer such questions requires that we dig deeper in the underlying system theory, and this is the focus of the next part of the book.

4.7 Sources

Information on DEVSJAVA and a downloadable version of the software are available on the Web site: www-ais.ece.arizona.edu. DEVS-based, object-oriented modeling and simulation has been applied in many fields. Example applications are listed below.

1. Barros, F. J., "Structural Inheritance in the Delta Environment." *Proceedings of the Sixth Annual Conference on AI, Simulation and Planning in High Autonomy Systems*, La Jolla, CA, 1996, pp. 141–147.
2. Concepcion, A. I., and B. P. Zeigler, "DEVS Formalism: A Framework for Hierarchical Model Development." *IEEE Trans. Software Engineering*, **14**(2), 228–241 (1988).
3. Moon, Y., B. P. Zingler, V. L. Lopes, J. Kim. "DEVS Approximation of Infiltration Using Genetic Algorithm Optimization of a Fuzzy System." *J. Math. Comp. Model.* **23**, 215–228 (1996).

h a job is sent out
on arrival. Rewrite
he extent to which

line of processors
ponents are "balk-
balk" external out-
lifficulty in achiev-

ı) discussed in this
›nstructing discrete
ılism embodies the
. Chapters 1. We will
ent models, but also
ıting behaviors that
ms formalisms. This
ı oriented to parallel
so saw the essentials
nework.
o model real systems
hether DEVS always
/S model works, and
ıswer such questions
ıeory, and this is the

›n of the software are
VS-based, object-ori-
nany fields. Example

ıvironment." *Proceed-*
ıtion and Planning in
l–147.
lism: A Framework for
Software Engineering,

VS Approximation of
›n of a Fuzzy System."

4. Nidumolu, R. N. Menon and B. P. Zeigler, "Object-Oriented Business Process Modeling and Simulation: a Discrete-Event System Specification (DEVS) Framework. *Sim. Pract. and Theory,* **6**, pp. 531–571 1998.
5. Ninios, P., *An Object Oriented DEVS Framework for Strategic Modelling and Industry Simulation.* London Business School, London, 1994.
6. Sato, R., and H. Praehofer, "A Discrete Event System Model of Business Systems." *IEEE Trans. System, Man and Cybernetics,* **27**(1), 1–22.(1997).
7. Vahie, S., "Dynamic Neuronal Ensembles: Issues in Representing Structure Change in Object-Oriented, Biologically-based Brain Models," Presented at Intl. Conf. on Semiotics and Intelligent Systems, Washington, DC, 1996.
8. Wainer, G., and N. Giambiasi, "Cell-DEVS with Transport and Inertial Delay." Presented at SCS European MultiConference, Passau, Germany, 1998.
9. Zeigler, B. P., H. Sarjoughian, and W. Au., "Object-Oriented DEVS." Presented at Enabling Technology for Simulation Science, SPIE AeoroSense 97, Orlando, FL, 1997.