Chapter 4 ATOMIC-MODELS: SIMPLE PROCESSOR EXAMPLE

We now begin to build models in DEVS-Scheme and organize them using the system entity structure. The model domain to be disccussed is that of simple computer architectures. We are interested in comparing the performance of single processor systems with multiprocessor systems including the multiserver, the pipeline, and the divide and conquer configurations. Performance evaluation is to be based on two fundamental measures, *turnaround time*, the average time taken by the system to process jobs (for example, solve a problem) and *throughput*, the rate at which completed jobs emerge from the system.

The multiprocessor configurations to be modelled each have a co-ordinator that sends problems (jobs) to some sub-ordinate processors and receives solutions from them. In the *multiserver* architecture, the co-ordinator re-routes incoming problems to whichever processor is free at the time. In the *pipeline* architecture, problems pass through the processors in a fixed sequence, each processor performing a part of the solution. In the *divide and conquer* architecture, problems are decomposed into subproblems that are worked on concurrently and independently by the processors before being be put together to form the final solution.

1 Performance of Simple Architectures

Table 4.1 shows typical performance characteristics of the architectures. To obtain these results, we assume that problems arrive to the system with a fixed interarrival time and all have the same difficulty. In other words, problems all require the same processing time, whose value we associate with the relevant processor. Thus, for a single processor with processing time p, the turnaround time for each problem is (by definition in this case) p. The maximum throughput occurs when the processor is always kept busy, i.e., it always has a next problem to work on as soon as it has finished the previous one. In this case, the processor can send out solved problems every p units of time, i.e., at the rate $\frac{1}{p}$.

The multiserver architecture in Table 4.1 is assumed to have three subordinate processors. If each processor has the same processing time p then the turnaround time is also p. The maximum throughput, again occurs when all of the processors are always kept busy, and with three processors the combined rate is 3 times that of the single processor. If there were n processors, the throughput could be increased by n (this ideal result neglects overhead due to co-ordination and communication time).

The pipeline architecture can also increase throughput without much effect on turnaround time. The turnaround time in this case is the sum of the times taken by each stage and the maximum throughput is that of the slowest stage (the bottleneck). Ideally, a problem can be divided into identically time consuming stages whose total time is less than the original processing time (any savings is due the fact that each stage can be optimized to perform only its specialized task).

Practically speaking, the only architecture that can both significantly reduce turnaround time and increase throughput is the divide and conquer configuration. For analysis purposes, this can

ARCHITECTURE	PARAMETERS	TURNAROUND TIME	MAXIMUM THROUGHPUT
simple processor	processing-time, p	p	1/p
multiserver	processing-times, 1) p1,p2,p3	3/throughput	1/p1 + 1/p2 + 1/p3
	2) p1=p2=p3=p	p	3/p
pipeline	processing-times, 1) p1,p2,p3	p1 + p2 +p3	1/max{pi}
	2) p1=p2=p3=p/3	p	3/p
divide & conquer	processing-times, 1) p1,p2,p3, cp, cm	cp + cm + max{pi}	1/max{pi,cp,cm}
	2) p1=p2=p3=p/3	cp + cm + p/3	1/max{p/3,cp,cm}

Figure 1: Performance characteristics of simple architectures.

be regarded as a pipeline consisting of the problem partitioner, the subproblem processors, and the compiler of partial results. For the subproblem processing stage to be finished, all of the subproblem processors must be finished, so the processsing time of this stage is the maximum of the individual processing times. Ideally, each of n subprocessors takes time p/n, where p is the original problem solution time, and the partitioner and compiler times are much smaller than this. In this case, both the time for an individual problem to be solved (turnaround time) and the time for a large group of problems to be solved (inverse of throughput) are reduced by a factor of n.

Insight into way these results arise can be gained by following the processing events as they happen in simulated models of these architectures. We shall show how to define simple, yet illuminating, versionsof these architectures in DEVS-Scheme.

2 A Simple Processor Model

We start with a rather simplistic model of a processor. Expressed in architecture, it takes the form of an atomic model called P. Basically, we represent only the time it takes to complete a job or solve a problem not the detailed manner in which such processing is done. Thus if the processor is idle, i.e., in *phase* 'passive, when a job arrives on the input port 'in, it stores the *job-id* (a distinct name for the job) and goes to work. This is achieved by the phrase "hold-in busy processing-time" , which sets the *phase* to 'busy and **sigma** (the time-left state variable) to *processing-time*.

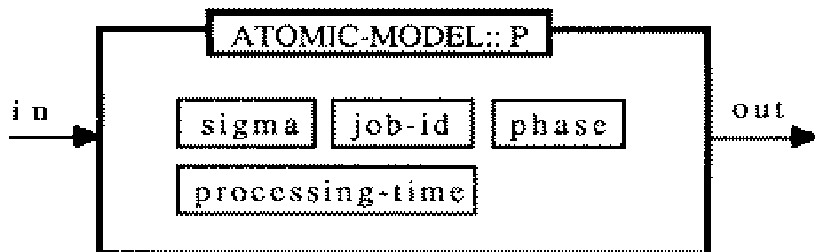
Such handling of incoming jobs is represented in the external transition function. Since this processor has no buffering capability, when a job arrives while the processor is busy it simply ignores it. This is achieved by the "continue" phrase which updates **sigma** to reflect the passage of elasped time, but otherwise leaves the state unchanged.

When the processor has finished processing, it places the job identity on port 'out and returns to the 'passive *phase*. Sending of the job is done by the output function which is called just before the state transition function. The latter contains the phrase "passivate" which returns the model to the idle state in which the *phase* is 'passive and **sigma** is 'inf.

Note that P has two state variables, *job-id* and *processing-time*, in addition to the standard ones **sigma** and **phase**. Since *processing-time*, once initialized, does not change during the run, it is actually a *parameter* (fixed characteristic) of the model.

Simple as this processor is, we can combine it with other components to create models of computer architectures that provide some insight into their performance. The basic model can also be refined to represent more complex aspects of computer operation as we shall see later.

It is important to note here that there is no way to generate an output directly from an external input event. An output can only occur just before an internal transition. To have an external event cause an output without delay, we have it "schedule" an internal state with a hold time of zero (see the multi-server model in Chapter 5, for example). The relationship between external transitions,




```
ATOMIC-MODEL: P

    state variables: sigma = inf
                     phase = passive
                     job-id = ()

    parameters: processing-time = 5

    external transition function:
       case input-port
            in: case phase
                     passive: store job-id
                              hold-in busy processing-time
                     busy: continue
            else: error

    internal transition function:
       case phase
            busy: passive
            passive: (does not arise)

    output function:
        send job-id to port out
```

Figure 2: Pseudo-code for a simple processor.

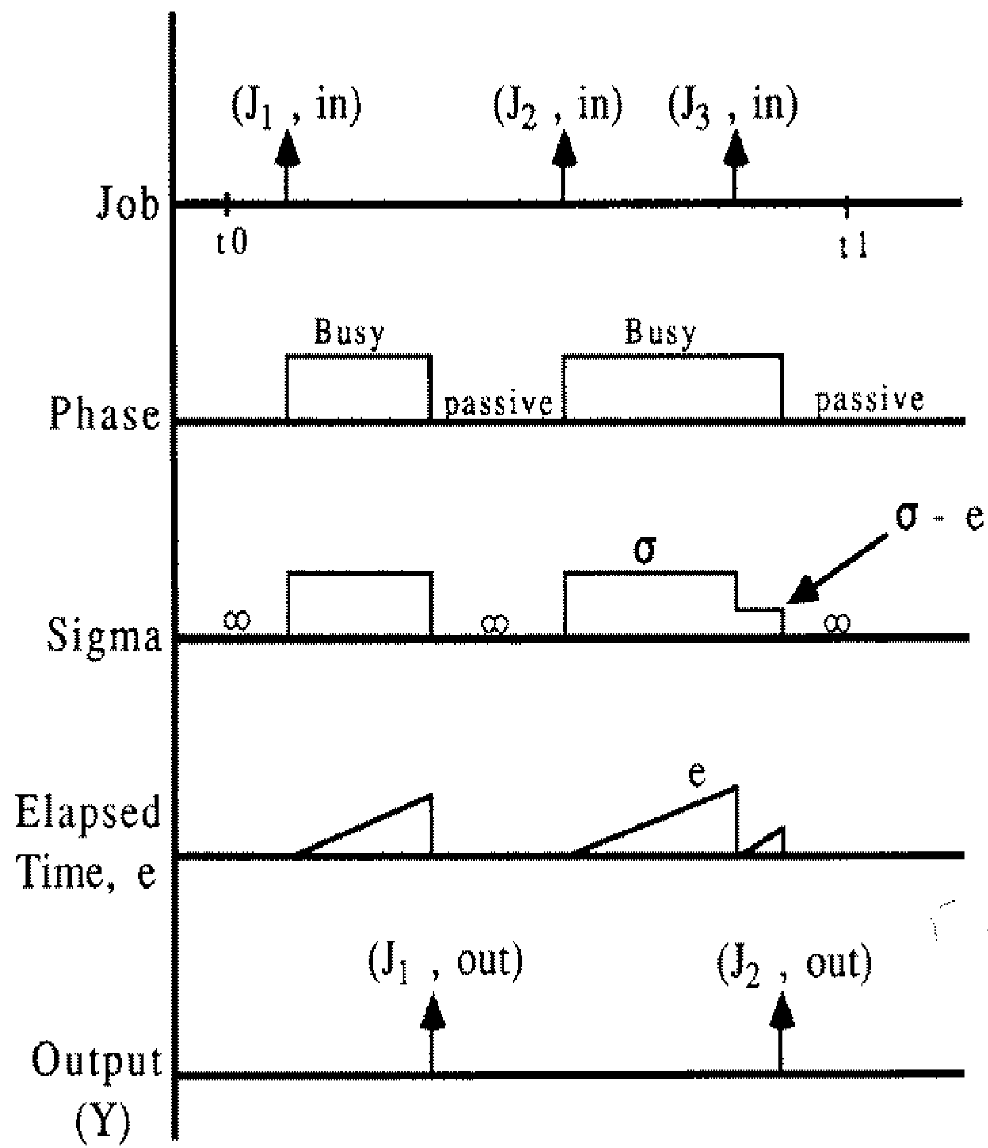


Figure 3: Figure 4.2. Trajectory for simple processor.

internal transitions, and outputs are as shown in Figure 4.2.

3 Normal Form Atomic-model Specification

To explain how the simple processor model is coded in DEVS-Scheme, we consider the class *atomic-models* in more depth. From its class definition (Figure 4.3), we see that *atomic-models* has instance variables *x*, *s*, *y*,and *e*, corresponding to the sets of the DEVS formalism. It also has instance variables *int-transfn*, *ext-transfn*, *outputfn*, and *time-advancefn* to hold the corresponding functions in the formalism.

Usually, we employ the normal form of *atomic-models*. This sets up the state as a Scheme structure by calling:

(define-structure state sigma phase ...)

where the ...denotes other state variables supplied by a method called *def-state* . *Sigma* holds the time remaining to the next internal event. This is precisely the time-advance value to be produced by the time-advance function. Thus, as in Figure 4.3, every atomic model is created with its *time-advancefn* slot set to the access procedure supplied by Scheme, *state-sigma*. The normal form can be over-ridden simply by redefining the values of instance variables *s* and *time-advancefn*.

instance variables are directly employed as state variables. *phase* and **sigma** are such instance variables. Additional state variables are obtained by adding them to a derived class of atomic

```
class atomic:public devs{

protected:

phasetype phase;  //phasetype is string
timetype sigma;   .
```

* (hold-in '⟨phase⟩ ⟨time⟩)

sets state-variable *phase* to '⟨phase⟩ and **sigma** to ⟨time⟩. This causes the model to stay in the given phase for the specified time.

* (passivate-in '⟨phase⟩)

is the equivalent of (hold-in '⟨phase⟩ 'inf), where 'inf plays the role of infinity in DEVS-Scheme. This means that the arithmetic properties of infinity are extended to 'inf. For example, subtracting any finite number *r* from 'inf returns 'inf. Also the minimum of any finite number *r* and 'inf is *r*; thus a passive component (which has time-advance equal 'inf) is never selected as the one with

smallest next event time.

```
INFINITY is a global constant with a large value
```

* (passivate)

sets *phase* to 'passive and **sigma** to 'inf, it is thus the equivalent of (passivate-in 'passive).

* (continue)

reduces **sigma** by the elapsed time, e. This allows the next internal event to occur at the same time it was scheduled for despite an external event interruption.

```
class atomic:public devs{

public:

void passivate(){
phase ="passive";
sigma = INFINITY;
}

Bool phase_is(phasetype  phase);

void hold_in(phasetype  phase, timetype sigma);

void passivate_in(phasetype   phase){hold_in(phase,INFINITY);}

void Continue(){  //capital to avoid class w/ keyword
if (sigma < INFINITY)
  sigma = sigma - e;
}

timetype  ta(){return sigma;}
};
```

4 DEVS-Scheme Atomic-model implementation of simple processor

Using normal form tools, the pseudo-coded description of the simple processor in Figure 4.1 is readily translated into the atomic-model specification of Figure 4.5.

```
(macro hold-in (lambda (expr)              ;;(hold-in '<phase> <time>)
   (let (
        (phase (cadr expr))
        (time (caddr expr))
        )
`(begin
 (set! (state-phase s) ,phase)
 (set! (state-sigma s) ,time)
))))


(macro passivate-in (lambda (expr)         ;;(passivate-in '<phase>)
   (let (
        (phase (cadr expr))
        )
`(begin
 (set! (state-phase s) ,phase)
 (set! (state-sigma s) 'inf)
))))


(macro passivate (lambda(expr)              ;;(passivate)
   `(begin
        (set! (state-phase s) 'passive)
        (set! (state-sigma s) 'inf)
)))


(macro continue (lambda(expr)              ;;(continue)
   `(when (not (equal? (state-sigma s) 'inf))
             (set! (state-sigma s) (- (state-sigma s) e))
))
```

Figure 4: Figure 4.4. Macros for **sigma** and **phase**.

Let us examine the role of each of the parts of this definition. The constructor

(make-pair atomic-models 'p)

creates both the atomic-model P as well as the simulator S:P assigned to it. The use of *make-pair* obviates the need to separately create (using *mk-ent*), and link together (using a function called *attach*), models and processors.

```
class proc:public atomic //declares  proc as a derived class of atomic
```

The message

(send p def-state ...)

is used to set up the additional state variables required for the model. Recall that all atomic models have **sigma** and **phase** upon creation by default. In this case, additional state variables are defined for the name of the job to be processed (*job-id*) and the time required to process a job (*processing-time*).

```
protected:
timetype processing_time;
entity * job;
```

The message

(send p set-s (make-state ...))

creates an instance of the state structure and sets the instance variable s of the model to this structure. In this way the state variables are initialized. Note that in particular **sigma** and **phase** must be initialized. When *sigma* has the value 'inf, this indicates that the model will not have an internal transition unless an external transition occurs.

```
//constructor performs initialization of parameters and state

proc(char * name,timetype processing_time):atomic(name){
inports->add("null");
inports->add("in");
outports->add("out");
phases->add("busy");
this->processing_time = processing_time ;
job = NULL:
}
```

```
//inital state is called by initialize for subsequent state resetting

void initial_state(){
job = NULL;
pasivate();
}
```

The definition

(define (ext-f s e x) . . .)

specifies the external transition function. Every external transition function must be defined using s, e, x as arguments (actually, any names can be used so long as the types are correct). The external transition begins by looking at the *content-port* of input x. If this input port equals 'in, the *phase* of the processor must be checked. If the *phase* equals 'passive the following actions are taken: First, the state variable *job-id* is set to the value of the message content appearing on port 'in; second the *phase* is set to 'busy with a **sigma** equal to the state variable processing time. For the case of *phase* equals 'busy the processor will ignore the input and continue in its current state.

```
void  deltext(timetype e,message * x) {

Continue(); //update sigma as a default

if (phase_is("passive"))
 for (int i=0; i< x->get_length();i++) //scan through the message
 if (message_on_port(x,"in",i)) //check each content for port ‘‘in’’
   {
   job = x->get_val_on_port("in",i); //set job to the value of current content
  hold_in("busy",processing_time);
   }
}
//if there are are more than  one values on port "in" only the last scanned is preserved in thi

//devs::message_on_port(message * x,char * p, int i) checks whether p is a valid input port for
```

The definition

(define (int-f s). . .)

specifies the internal transition function. The internal transition function first checks the *phase* of the processor. If *phase* equals 'busy the macro *passivate*, is used to set the *phase* to 'passive and **sigma** to 'inf.

```
void  deltint( )
{
if (phase_is("busy"))
passivate();
}
```

The definition

(define (out-f s)...)

specifies the output function This function first checks the *phase* of the processor. If the *phase* equals 'busy, an output will be generated. The output must always return a content structure. Here it is created with

(make-content 'port 'out 'value (state-job-id s))

which generates an output on the port called 'out with value *job-id*. Again note that an output is produced just before the internal transition occurs. Here this transition is that from *phase* 'busy to 'passive.

```
message *  out( )
{
message * m = new message();//make a new message
content * con = make_content("out",job); //make a new content
m->add(con); //insert the content into the message
return m; //message contains  only one content in this model
}

//devs:: make_content(char * port,entity * value) calls the constructor for content
```

The three forms

(send p set-ext-transfn ext-f)
(send p set-int-transfn int-f)
(send p set-outputfn out-f)

statements are used to assign the procedures defined for external transition, internal transition, and output functions to the corresponding slots of atomic-model P. The atomic model is now ready for testing and simulation.

In DEVS-C++ the above assignments steps do not happen. This is because, each new model is formulated as a new class. Thus the basic functions are defined by redefining those inherited from the atomic class.

5 Simulation of Atomic-models

Recall that an atomic-model P is handled by its attached simulator S:P. We can understand the workings of a DEVS-Scheme atomic model by examining the simulation process performed by its simulator.

Recall that there are two message types arriving to a simulator. A **-message* indicates that the next internal event is that of its model. An *x-message* is an arrival of an external event to the model.

- A simulator S:M of an atomic model M processes an **-message* as follows:
  - checks if the time carried by the **-message* agrees with the *time-of-next-event*; this should agree because a **-message* should only arrive to S:M if its model is imminent, i.e., has the smallest *time-of-next-event*.
  - emits:

    (send M output?)

    thereby causing M to apply its output function; the reslult is a content structure containing a *port* and a *value*. This is where the output is generated prior to the internal transition.
  - embeds the above content in a *y-message* and sends it out to the next higher level, i.e., the parent co-ordinator if M is a component in a coupled model or to the root-co-ordinator otherwise.
  - emits: (send M int-transition)

    thus causing M to apply its internal transition function (*int-transfns*) to update its state, s.
  - updates the *time-of-last-event* and *time-of-next-event* slots:
    time-of-last-event = *time-of-next-event*
    time-of-next-event = *time-of-next-event* +
    (send m time-advance?)
  - creates a *done-message* which indicates that the state transition has been carried out and which reports the new *time-of-next-event* to the next higher level, i.e., the parent co-ordinator if M is a component in a coupled model or to the root-co-ordinator otherwise.
- When a simulator S:M receives an *x-message* it:
  - checks if the time carried by the *x-message* lies between the times in its time-of-last event and *time-of-next-event* slots; this should be so because an *x-message* should only arrive to S:M after the last event and before its next internal event.

- computes the elapsed time,
  e = time carried by *x-message* - *time-of-last-event* and sends it to M:
  (send M set-e e)
- sends the content of the *x-message*, to M
  (send M set-x (message-content *x-message*))
- emits:
  (send M ext-transition)

  thus causing it to apply its external transition function *(ext-transfn s e x)* where it has just received the updated values of e and x.
- updates the *time-of-last-event* and *time-of-next-event* slots:

  time-of-last-event = *time-of-next-event*
  time-of-next-event = *time-of-next-event* +
  (send m time-advance?)
- creates a done message which indicates that the state transition has been carried out and which reports the new *time-of-next-event* to the next higher level, i.e., the parent co-ordinator if M is a component in a coupled model or to the root-co-ordinator otherwise.

5.1 Significance of Model/Simulator Separation

The foregoing simulator is *generic* for the *atomic-models* class. This means it works with any model in this class or any of its subclasses. The interface between a model and its attached simulator is defined by the methods, such as *int-transition*, *ext-transition*, etc. representing queries and operations that the simulator requests of the model. The details of the model's internal structure are hidden behind this interface.

There are important consequences of DEVS-Scheme's object-oriented approach to behavior generation. First any model may be simulated by a simulator if it has the methods required by the interface. This underlies the genericity just mentioned and makes it easy to add sub-classes to the *atomic-models* class. Second, the removal of all simulation-related information from model specification enables us to treat models as knowledge, i.e., to be placed in a model base. Later we shall see how a decision-making device, other than a simulator, can interrogate and exercise a model (Section 12.3).

6 Stand-alone Testing of an Atomic Model

As indicated before, one of the most important features of DEVS-Scheme is that models at any level of complexity can be independently tested. Let us see how this applies to the ultra-important atomic-model level. To test the model, one sends it message sequences that would be sent by its

simulator to have it execute internal transitions and respond to external events.

Consider for example, a test message sequence that sends a job to the atomic-model P, forces the associated external transition (in which the *job-id* should be recorded) and then the resulting internal transition (in which the job is completed). We assume that P starts in an idle state, i.e., any state with *phase* = 'passive and **sigma** = 'inf.

a. (send p set-x (make-content 'port 'in 'value 'x1))

The effect should be that *x* is assigned a content (IN X1) (the order is: *port, value*).

b. (send p set-e 0)

c. (send p ext-transition)

The effect should be that *s* becomes the state (5 BUSY X1 5) (the order is: *sigma, phase, job-id, processing-time*).

d. (send p output?)

The effect should be that *y* is assigned the content (OUT X1).

e. (send p int-transition)

The effect should be that *s* becomes (INF PASSIVE X1 5).

Message a) causes P to set its external input event (*x*) slot to a content structure consisting of a port 'in and 'value x 1. Message b. tells P to set the elapsed time (*e*) slot to 0. These messages mimic those generated by S:P when it receives an *x-message* on port 'in with value X1 and determines the elapsed time of P to be 0.

Message c) causes P to execute its external transition function. Again this is what the simulator would send to P in order to have it respond to the current *x-message*.

Messages d) and e) cause P to execute its output and internal transition functions respectively. Again, note the output occurs prior to the internal transition. S:P would request this to be done when it receives a **-message* indicating that the time has arrived for its model to execute its internal event. In this case, the stored *job-id* is output on port 'out and the model returns to an idle state.

Messages a), b), and c), may be replaced by a single message which performs the same functions. This message uses a method called *inject*. The *inject* method creates a content-structure with the given *port* and *value*, sets the elapsed time, *e*, to the value supplied (optional), and calls for an *ext-transition*. The format f the message using *inject* is:

(send ⟨atomic-model⟩ inject ⟨'port⟩ ⟨'value⟩ {⟨elapsed-time⟩}).

With *inject*, the message sequence testing for P's handling of a job arrival while it is idle is reduced to the following:

f. (send p inject 'in 'x1 0)

The result is state s = (5 BUSY X1 5)

g. (send p output?)

The result is output y = (OUT X1)

h. (send p int-transition)

The result is state s = (INF PASSIVE X1 5)

To test the "continue" alternative of the external transition function a job must be sent into the processor when it is busy. The following sequence of messages is used to test that condition:

i. (send p inject 'in 'x1 0)

The result is state s = (5 BUSY X1 5)

j. (send p inject 'in 'x2 3)

The result is state s = (2 BUSY X1 5)

Notice: The job being processed is still X1. Thus job X2 was ignored as it should be. *Sigma* is now 2 as it should be since, the continue statement reduced **sigma** by e to get the time remaining, i.e., 5-3 = 2.

k. (send p output?)

The result is output y = OUT X1

l. (send p int-transition)

The result is state s = (INF PASSIVE X1 5)

In case a test fails, the modeller must locate and correct the error. Usually, the requires resetting the model's state. There are several ways to do this, appropriate to different circumstances. To inspect and alter a single state variable, we can use the methods, *get-sv* and *set-sv* as in:

(send p get-sv 'phase)

which returns the value of the *phase* state variable, and

(send p set-sv 'phase 'busy)

which sets the *phase* state variable to 'busy.

To reset the state completely, we can use the *make-state* function as in Figure 4.5.

Sequences of messages may be put into a test file (e.g., p.tst for testing the simple processor P, shown in Figure 4.6). The test file can be archived in a subdirectory of the appropriate modelling domain (for example, test is a subdirectory of simparc). Test sequences should cover the different combinations of inputs and states in which the model must respond correctly. The expected responses are shown as comments and may be compared with the actual ones. If all comparisons succeed then we may gain full confidence in the model and place it into the model base as a verified model. The stored test file can be reused when modifications are made in the model.

If an error occurs in a coupled model, and all components have been verified in the foregoing way, we can look for the source to be in the coupling or tie-breaking specifications of the coupled model. This bottom-up testing makes it possible to confidently build up successively higher level subcomponents until the final model is achieved. We return to consider testing of hierarchical models later.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   Content of the simple processor definition file p.m      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;---------------------------------------------------------------
; This file contains the definition of a simple processor
; without buffering capability
;---------------------------------------------------------------
; It performs following tasks:
; Gets a job from input and sends the job to the output after
; its processing time.
;---------------------------------------------------------------

;;;;;;;; make a pair for the processor and its simulator

(make-pair atomic-models 'p)

;;;;;;;; set up additional variables job-id and processing-time

(send p def-state
            '(
             ;;;state-variables:

                job-id           ;name of the processed job

             ;;;parameters

               processing-time  ; processing time of this processor
             )
)

;;;;;;;; initialize variables

(send p set-s
            (make-state 'sigma      'inf
                        'phase      'passive
                        'job-id     '()
                        'processing-time 5
            )
)

;;;;;;;; define the external transition function

(define (ext-f s e x)
        (case (content-port  x)
              ('in (case (state-phase s)
                     ('passive
                          (set! (state-job-id s) (content-value x))
                          (hold-in 'busy (state-processing-time s))
                     )
                     ('busy (continue))
                     )
              )
        )
)
```

Figure 5: Atomic-model specification of simple processor.

```
;;;;;;;; define the external transition function

(define (ext-f s e x)
        (case (content-port  x)
              ('in (case (state-phase s)
                    ('passive
                         (set! (state-job-id s) (content-value x))
                         (hold-in 'busy (state-processing-time s))
                    )
                    ('busy (continue))
                    )
              )
        )
)

;;;;;;;; define the internal transition function

(define (int-f s)
  (case (state-phase s)
    ('busy (passivate))
) )


;;;;;;;; define the output function

(define (out-f s)
  (case (state-phase s)
    ('busy
        (make-content 'port 'out 'value (state-job-id s))
    )
    (else (make-content))
) )


;;;;;;;; assignment to the model

(send p set-ext-transfn ext-f)
(send p set-int-transfn int-f)
(send p set-outputfn out-f)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;          Content of the test file p.tst                 ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;-------------------------------------------------------------
; This file contains the test sequence for the simple processor
;;;-------------------------------------------------------------

; in case the model-base_directory is not already defined:

(define model-base_directory "/scheme/devs/simparc/mbase/")

; load the file containing the processor

(load-from model-base_directory p.m)

; arrival of external event job_in_1 on  port 'in at elapsed time 0

(send p inject 'in 'job_in_1 0)

; STATE S = (5 BUSY JOB_IN_1 5) processor is now processing job_in_1

; after 5 time units have elapsed the following should happen

(send p output?)          ;;; execute the output function

; OUTPUT Y = OUT JOB_IN_1             ;;;; output the job

(send p int-transition)  ;; execute the internal transition function

; STATE S = (INF PASSIVE JOB_IN_1 5) ;;;; back to passive state

; repeat external event arrival with job_in_2

(send p inject 'in 'job_in_2)

; STATE S = (5 BUSY JOB_IN_2 5) ;;;; sigma is 5 and phase is BUSY

; input when the processor is busy  ---> job_in_3 at elapsed time 2

(send p inject 'in 'job_in_3 2)

; state s = (3 BUSY JOB_IN_2 5) ;;;;  job_in_3 is lost,
                                ;;;;  job_in_2 has 3 time units left
                                ;;;;  due to the continue statement

; finally after 3 units the following should happen

(send p output?)

; OUTPUT Y = OUT JOB_IN_2  ;; still output only the previous job_in_2

(send p int-transition)

; STATE S = (INF PASSIVE JOB_IN_2 5)  ;; back to passive state
```

Figure 6: Test sequence for the simple processor.

```
class proc:public atomic{

protected:
timetype processing_time;
entity * job;

public:

proc(char * name,timetype processing_time):atomic(name){
inports->add("null");
inports->add("in");
outports->add("out");
phases->add("busy");
this->processing_time = processing_time ;
job = NULL;
}

void initial_state(){
job = NULL;
pasivate();
}

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);
   }

}

void  deltint( )
{
if (phase_is("busy"))
passivate();
}

message *  out( )
```

```
{
message * m = new message();
content * con = make_content("out",job);
m->add(con);
return m;
}


void show_state()
{
cout << "\nstate of  " << name << ": " ;
cout << "phase, sigma,job_id : "
         <<phase << " " << sigma << " ";
            job->print();
            cout <<endl;
}


void show_output()
{
 if (!output->empty())
  {
        cout << "\noutput of: " << name << ":  (" ;
  for (int i=0; i< output->get_length();i++)
      {
      content *   con = output->read(i);
      cout << "(" << con->p->get_name()
            << " " << con->devs->get_name()
             << " " << con->val->get_name() <<")";
      }
  cout << ")" << endl;
  }

}
};


int main(int argc,char ** argv){
proc * p  =  new proc("p",10);
 p ->initialize();
  p ->show_state();
 p ->inject("in",new entity("job1"),0);
 p->show_output();
 p ->show_state();
 p ->deltint();
```

```
 p ->show_state();

 p ->inject("in",new entity("job2"));
 p ->show_state();
 p ->inject("in",new entity("job3"),2);
 p ->show_state();
 p->show_output();
 p ->deltint();
 p ->show_state();

return 0;
}
```

7 Simple Processor with Buffering and Random Processing Times

The simple processor model introduced in this chapter can be made more realistic in a variety of ways. Often the processing time in such a model is not constant but is sampled from a probability distribution. This is easy to arrange in DEVS-Scheme by modifying the external transition function in Figure 4.5 by adding a statement such as:

(set! (state-processing-time s) (random 100))

before the hold-in statement. This samples the processing time from a uniform distribution in the range [0,99]. Other distributions such as the exponential or normal may be used as explained in many books on discrete-event simulation.

```
hold_in("busy", exponential(100,2);
```

Buffering such as described in the model of Figure 4.4 in Chapter 3 is also readily specified. In Figure 4.5, we need only treat the *job-id* state variable as able to hold a list of jobs rather than a single one. When a job arrives, we add it to this list (for first-in-first-out queueing, we add it at the end). A statement to this effect can be placed right at the beginning of the external transition function. The rest of the function need not be altered! The internal transition function however, is modified so that after removing the first job in the queue, if the queue is not empty, we start processing the next job. The output function is modified so that the first job in the queue is sent out.

Discrete-event simulation is often associated with simulation of queueing models and one might imagine that queueing is an inevitable factor in any such model. However, as new approaches to manufacturing such as just-in-time production (Manivannan, 1989) have shown, queues are evidence of inadequate process co-ordination and impose a costly overhead that often can be avoided. In the models to be discussed in this book, we intentionally do not incorporate queues, in favor of more sophisticated co-ordination schemes. The reader may wish to compare performance of the models in the ensuing chapters with, and without, queues. Modularity, and model base concepts, facilitate exploration of such alternatives.