Chapter 5: DIGRAPH-MODELS AND EXPERIMENTAL FRAMES

Although a model, such as that of the processor, can be tested in a stand-alone fashion, it really does not "come to life" until it is coupled with a module capable of providing it input and observing its output. An experimental frame module is a coupled-model, which when coupled to a model, generates input external events, monitors its running, and processes its output (Figure 5.1). The design of an experimental frame reflects the objectives one has in experimenting with a model. Thus the same model might be coupled to different experimental frame modules which observe it under different conditions. (If desired, experiments under different experimental conditions can all be done in parallel by coupling a copy of the model to each frame.) Conversely, the same experimental frame module may be employed to experiment with different models under the same conditions.

We now show how to construct an experimental frame module to measure turn-around time and throughput in any of the simple computer architecture models.

1 Experimental Frame for Simple Computer Architectures

An example of such an experimental frame module, shown in Figure 5.1, is a digraph model consisting of a generator, GENR and transducer, TRANSD.

1.1 Generator

The generator (Figure 5.2) outputs a sequence of job identifiers spaced equally in time. The basic mechanism that produces this behavior is the "hold-in active inter-arrival-time" phrase in GENR's internal transition function. This phrase returns the model to the same phase, ACTIVE after each internal transition and schedules it to undergo a next transition in a time given by *inter-arrival-time* (which is constant in this case, but could vary in general). Just before the internal transition takes place, the output of a randomly determined *job-id* symbol is produced. Since GENR is itself a DEVS model, it can be tested in stand-alone fashion.

In principle, a generator is an autonomous model, (its behavior is self induced by recurring internal events) hence, it does not need an external transition function to dictate its response to external input events. However, we have added an input port 'stop which, when stimulated, stops the generator from producing any more outputs.

```
class genr:public atomic{

protected:

timetype int_arr_time;
int count;
```

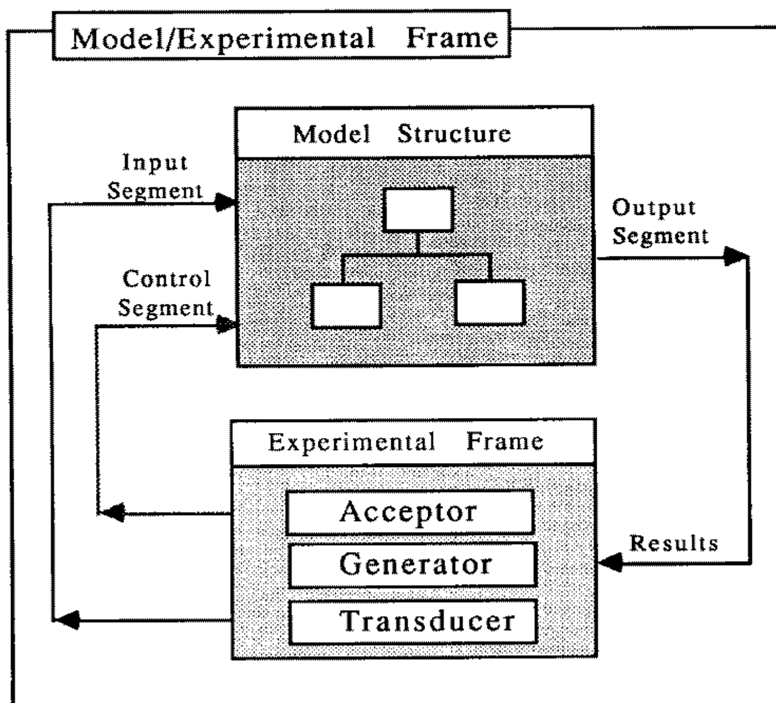
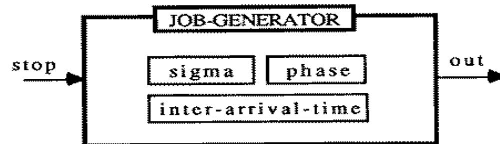


Figure 1: Structure of model/experimental frame pair

```
;-----------------------------------------------------------------
; This file contains the definition of the job generator
;-----------------------------------------------------------------
; It should perform following tasks:
; generates a job every inter-arrival-time
; Option: stops the generating sequence when receiving a stop
;         signal  (stop)
;-----------------------------------------------------------------

;; create a generator

(make-pair atomic-models 'genr)

; add another state variable: inter-arrival time

(send genr def-state '(inter-arrival-time))

; initialization

(send genr set-s (make-state 'sigma              0
                             'phase              'active
                             'inter-arrival-time 10
                 )
)

;; Add external transition function to terminate the generator
;; when the experiment is over instead of using keyboard interrupt

(define (ext-genr s e x)
  (case (content-port x)
        ('stop (passivate))  ;when receive stop signal passivate
        (else (continue))
))

;; definition of internal transition function

(define (int-genr s)
  (case (state-phase s)
        ('active
            (set! (state-sigma s) (state-inter-arrival-time s))
         )      ;;;reset sigma each time an internal transition occurs
))              ;;; Note: not really necessary for fixed inter-arrival
                ;;; time

;; definition of output function

; output the jobname (gensym) to port 'out

(define (out-genr s)
  (case (state-phase s)
        ('active
          (make-content 'port 'out 'value (gensym))
        )
        (else (make-content))
) )
```




```
ATOMIC-MODEL: GENR

    state variables: sigma = inf
                     phase = active

    parameters: inter-arrival-time = 10

    external transition function:
        case input-port
            stop: passive
            else: error

    internal transition function:
        case phase
            active: hold-in active inter-arrival-time
            passive: (does not arise)

    output function:
        case phase
            active: send random-job-name to port out
            passive: (does not arise)
```

Figure 2: Pseudo-code for job generator

```
;; connect the definitions of functions to generator

(send genr set-int-transfn int-genr)
(send genr set-ext-transfn ext-genr)
(send genr set-outputfn out-genr)
```

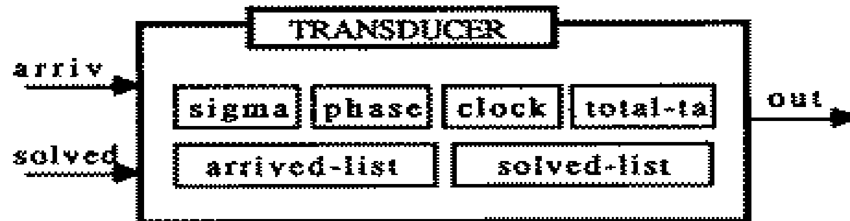
Figure 3: Atomic-model specification of job generator

```
public:

genr(char * name,timetype int_arr_time):atomic(name){
inports->add("null");
inports->add("stop");
inports->add("start");
outports->add("out");
phases->add("busy");
this->int_arr_time = int_arr_time ;
count = 0;
}


void initial_state(){
count = 0;
pasivate();
}

void  deltext(timetype e,message * x)
{
    Continue();

for (i=0; i< x->get_length();i++)
 if (message_on_port(x,"stop",i))
    passivate();

for ( i=0; i< x->get_length();i++)
 if (message_on_port(x,"start",i))
   {
   entity * val = x->get_val_on_port("start",i);
   hold_in("busy",int_arr_time);
   }
}

void  deltint( )
{
 count++;
 hold_in("busy",int_arr_time);
}

message *  out( )
{
message * m = new message();
content * con = make_content
```

```
     ("out",new entity(name_gen(name,count)));
m->add(con);
return m;
}


void show_state()
{
cout << "\nstate of  " << name << ": " ;
cout << "phase, sigma,count : "
         <<phase << " " << sigma << " " << count <<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;
  }
}
};
```

ENTITY<type> is a template for constructing a derived class of entity which holds an element of given <type>; the accessor methods are setv and getv; constructor is ENTITY<type>(val) (where ENTITY<type>(val).getv = val);

```
#define NUMent ENTITY<float>

class transd:public atomic{

protected:

function * arrived, * solved;
timetype clock,total_ta,observation_time;
```




```
ATOMIC-MODEL: TRANSD

    global variable: observation-interval

    state variables: sigma = observation-interval
                     phase = active
                     arrived-list = ()
                     solved-list = ()
                     clock = 0
                     total-ta = 0

    external transition function:
        advance local clock to agree with global clock
       case input-port
            ariv: append job to arrived-list
            solved: find job arrival time
                    total-ta = clock - arrival time
                    put the job to solved-list
        continue

    internal transition function:
       case phase
            active: passive
            ;; end of observation interval

    output function:
        case phase
             active: average-turnaround-time =
                          total-ta / solved-job-number
                     thruput = solved-job-number / clock
             else: no output
```

Figure 4: Pseudo-code for transducer

```
;----------------------------------------------------------------
; This file contains the definition of transducer
;----------------------------------------------------------------
;
; Transducer designed for the measurement of
;
;            1. average turnaround time of processed jobs
;            2. thoughput

; observation-interval must be set (default is 100)

(define observation-interval 100)

; now start model definition

(make-pair atomic-models 'transd)

(send transd def-state '(arrived-list  ;; all jobs which have arrived
                         solved-list   ;; all jobs which have been
                                       ;; processed
                         clock         ;; local clock
                         total-ta)     ;; total turnaround time of
                                       ;; processed jobs
)

(send transd set-s (make-state
                        'sigma         observation-interval
                        'phase         'active
                        'arrived-list  '()
                        'solved-list   '()
                        'clock         0
                        'total-ta      0
)                   )

;; external transition function takes care of recording arriving
;; and departing jobs and of accumulating total turnaround time

(define (ext-t s e x)
    (let (
         (problem-id (content-value x))
         )
     (set! (state-clock s) (+ (state-clock s) e))
     (case (content-port x)
      ('ariv (set! (state-arrived-list s)
                  (cons (list problem-id (state-clock s))
                        (state-arrived-list s)))
      )
      ('solved (let* (
                    (pair (assoc problem-id (state-arrived-list s)))
                    (prob-arrival-time (cadr pair))
                    (turn-around-time
                         (- (state-clock s) prob-arrival-time))
                    )
```

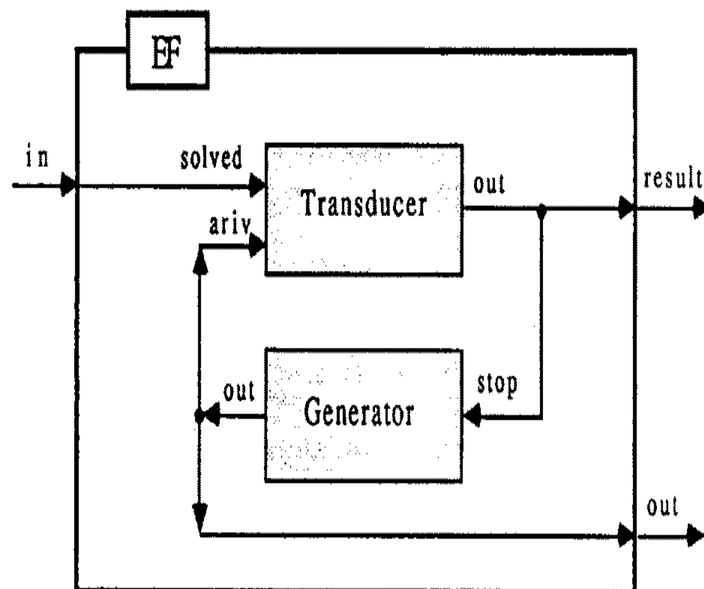
Figure 5: Atomic-model specification of transducer

```
          (newline log-file)
          (display "The arrived list: " log-file)
          (display (state-arrived-list s) log-file)
          (newline log-file)
          (display "The solved list:  " log-file)
          (display (state-solved-list s) log-file)
               (newline log-file)
          (display "Avg. turnaround time: " log-file)
              (display avg-ta-time log-file)
          (newline log-file)
          (display "ThruPut: " log-file)
              (display thruput log-file)
          (newline log-file)
          (close-output-port log-file)
          (make-content 'port 'out 'value (list avg-ta-time thruput))
        ) ;;let
       ) ;;active
       (else (make-content))
  ))

(send transd set-ext-transfn ext-t)
(send transd set-int-transfn int-t)
(send transd set-outputfn out-t)
```

Figure 6: Transducer Definition

```
                    (when (not (null? prob-arrival-time))
                       (set! (state-total-ta s)
                            (+ (state-total-ta s) turn-around-time))
                       (set! (state-solved-list s)
                            (cons problem-id (state-solved-list s)))
                     )
                  )
         )
        (else
          (bkpt "error: invalid input port name --> " (content-port x)))
         )
         (continue)
)


;; internal transition function is called only at end of run

(define (int-t s)
  (case (state-phase s)
     ('active (passivate))
))


;; output function serves to compute summary indexes: throughput and
;; average turnaround time

(define (out-t s)
   (case (state-phase s)
      ('active
        (let (

   ; a port log-file is opened to record transducer output in file "log"

             (log-file (open-output-file "log"))

   ; average turn-around time: total-ta divided by number of processed
   ;                                jobs

             (avg-ta-time
                (if (NULL? (state-solved-list s))
                  '()
                  (/ (state-total-ta s) (length (state-solved-list s)))
                )
             )

   ; thruput: number of processed jobs divided by observation interval

             (thruput
                (if (= (state-clock s) 0)
                   '()
                   (/ (length (state-solved-list s)) (state-clock s))
                )
              )
             )
```

```
(define-class digraph-models
  (instvars
          this-model       ;actual object this instance is
          (composition-tree (bi-tree))
          (children '())
          (influence-digraph (digraph))
          (selectfn (lambda (l) (car l)))
          (priority-list '())

  )
  (mixins coupled-models)
```



```
public:

transd(char * name,timetype observation_time):atomic(name){
inports->add("null");
inports->add("ariv");
inports->add("solved");
outports->add("out");
phases->add("active");
arrived = new function();
solved = new function();

this->observation_time = observation_time;
clock = 0.0;
total_ta = 0.0;
hold_in("active",observation_time);
}

void initial_state(){
 arrived = new function();
 solved = new function();

this->observation_time = observation_time;
clock = 0.0;
total_ta = 0.0;
hold_in("active",observation_time);
}

void deltext(timetype e,message * x)
{
clock = clock + e;
 Continue();
    entity * val;
 for (int i=0; i< x->get_length();i++)
   {
    if (message_on_port(x,"ariv",i))
       {
       val = x->get_val_on_port("ariv",i);
           arrived->add(val,new NUMent(clock));
       }
    if (message_on_port(x,"solved",i))
      {
       val = x->get_val_on_port("solved",i);
       if (arrived->key_name_is_in(val->get_name()))
        {
        entity * ent = arrived->assoc(val);
```

```
        NUMent * num = (NUMent *)ent;
        timetype arrival_time = num->getv();
        timetype turn_around_time = clock - arrival_time;
        total_ta = total_ta + turn_around_time;
         solved->add(val, new NUMent(clock));
        }
       else {
             cerr << "arriving job " ; val->print();cout << " did not depart!";
             exit(1);
             }
     }
   }
}
void  deltint(){
clock = clock + sigma;
passivate();
}

message *  out( )
{
timetype avg_ta_time = 0;
if (!solved->empty()) avg_ta_time = total_ta/solved->get_length();
float thruput;
if (clock > 0)
  thruput = solved->get_length()/clock;

cout << "jobs arrived :" << arrived->get_length()<<endl;
cout << "jobs solved :" << solved->get_length()<<endl;
cout << "\nAVG TA = " << avg_ta_time <<endl;
cout << "\nTHRUPUT = " << thruput <<endl;

message * m = new message();
content * con = make_content("out",new entity("finished"));
m->add(con);
return m;
}

void show_state()
{
cout << "\nstate of  " << name << ": " ;
cout << "phase, sigma,arrived,solved: " <<endl
         <<phase << " " << sigma << " ";
            arrived->domain_objects()->print(); solved->domain_objects()->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->print();
      }
  cout << ")" << endl;
  }
}
};
```

1.2 Transducer

The transducer (Figure 5.3) is designed to measure two performance indexes of interest for computer processors: the *throughput* and average *turnaround time* of jobs in a simulation run. Recall that *throughput* is the average rate of job departures from the architecture, estimated by the ENTITY¡float¿ of jobs processed during the observation interval, divided by the length of the interval. A job's *turnaround* time is the length of time between its arrival to the processor and its departure from it as a completed job (i.e., solved problem). Note that for the simple processor P, the turnaround time is the same as the processing time. However, for more complex architectures, this relationship is not necessarily true as we shall see.

To compute the performance measures, the transducer, TRANSD places job-ids that arrive at its 'ariv input port on its *arrived-list* together paired with their arrival times. When, and if, the *job-id* also appears at the 'solved input port, TRANSD places it on the *solved-list* on the *solved-list* and also computes its turnaround time. TRANSD maintains its own local clock to measure arrival and turnaround times. The DEVS formalism does not make available the simulation clock time to model components. Thus models have to maintain their own clocks if timing is needed. They can easily do so by accumulating elapsed time information which is available in the form of *sigma* and *e*.

Note that, in contrast to a generator, a transducer is essentially driven by its external transition function. In TRANSD, an internal transition is used only to cause an output at the end of the observation interval. In a more general experimental frame, the role of terminating the simulation run would be handled by a component called an acceptor.

As illustrated in the atomic-models specification of TRANSD, any atomic model, can write

directly into DOS files to maintain a log of events over time.

2 Development of Digraph-models

Recall that *digraph-models* is a specialized subclass of *coupled-models*. A digraph model is a coupled model which is composed of a set of explicitly given components with explicitly specified coupling. As Figure 5.4, a digraph model has instance variables corresponding to the parts of the DEVS mult-component formalism. Methods are used to specify the components of the digraph-model (*build-composition-tree*) and to specify the coupling of the components (*set-inf-dig, set-int-coup, set-ext-out-coup, set-ext-inp-coup*).

To show how to specify a digraph model we will construct an experimental frame module by coupling the generator and transducer components just discussed.

2.1 Experimental Frame Digraph Model

GENR and TRANSD are coupled together to form the experimental frame EF, a digraph-model shown in Figure 5.5. The input port 'in of EF is for receiving solved jobs which are sent to the 'solved input port of TRANSD via the external input coupling. There are two output ports: 'out, which transmits job identifiers sent to it by GENR, and 'result which transmits the performance measures computed by TRANSD. Both these transmissions are brought about by external output couplings. Finally, there are two internal couplings: the output port 'out of GENR sends job identifiers to the 'ariv port of TRANSD and the output port 'out of TRANSD which couples to the 'stop input port of GENR.

It should be noted that output lines may diverge to indicate the occurrence of simultaneous events. Thus for example, when GENR sends out a job identifier on port 'out, it goes at the same clock time, both to the 'ariv port of TRANSD and port 'out of EF, hence eventually to some processor model. Also, convergence of input lines, i.e., two or more source ports connected to the same destination port, can occur. Convergence does not pose a problem since at most one component is active and can be sending an output at any given moment.

The composition tree (Figure 5.6) depicts the digraph model (EF) as the root of a tree with its leaves being thecomponent models (GENR, TRANSD) comprising the digraph-model. In the general case the leaves can be either coupled-models or atomic-models. As shown, port pairs in the external-input coupling specification are assigned to appriopriate arcs from the root to the leaves; port pairs in the external-output coupling are associated with arcs in the other direction.

The influence digraph (Figure 5. 6) depicts how the components influence each other. In EF, GENR influences TRANSD and TRANSD influences GENR. In the general case there does not have to be a bidirectional influence. The influencees of a component are those components whose input ports are coupled to it's output ports. Port pairs in the internal coupling specification are

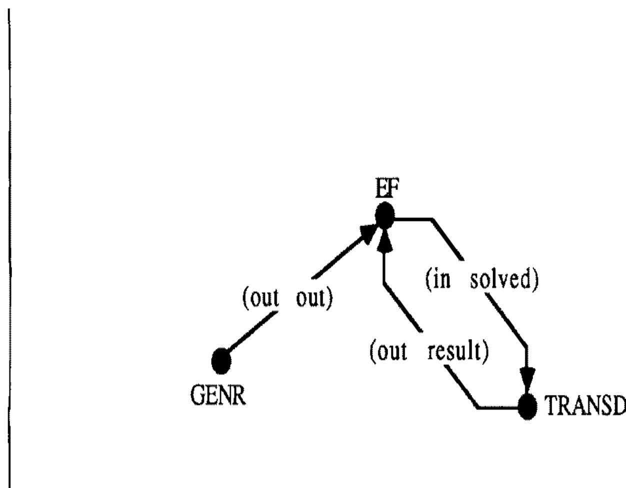


Figure 7: Composition tree and influence digraph for experimental frame

assigned to apropriate arcs in the influence digraph.

The DEVS-Scheme implementation of EF (Figure 5.7a) illustrates the general approach to defining *digraph-models*. Recall that *composition-tree* and *influence-digraph* are instance variables of a digraph model which are assigned directed graph objects. The methods *build-composition-tree* and *set-inf-digraph* construct the respective objects and the other methods add port pairs to designated arcs in the directed graphs. The input required from the modeller can be reduced to a minimum using more "intelligent" methods (Figure 5.7b). Since, as will be shown, the system entity structure provides a yet more user-friendly approach to specifying digraph-models, we do not delve into greater detail here.

```
class exframe:public digraph{
public:
exframe(char * name,timetype int_arr_time,
               timetype,observe_time):digraph(name)
{

 genr * g = new genr("g",int_arr_time);

 transd * t  = new transd("t",observe_time);

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

```
;----------------------------------------------------------------
; This file contains the construction of the experiment frame
; by using digraph-models. The components
; are retrieved from the models.m defined in model-base.
;; components: One generator (genr.m)  -- genr
;               one transducer (transd.m) -- transd
;----------------------------------------------------------------

;; load components of experimental frame, generator and transducer

(load "/scheme/devs/simparc/mbase/genr.m")
(load "/scheme/devs/simparc/mbase/transd.m")

;; couple them in a digraph-model

(make-pair digraph-models 'ef)

;; specify the root and leaves of the composition-tree

(send ef build-composition-tree ef (list genr transd))

;; add the external input port pairs to the arcs of the
;; composition-tree

(send ef set-ext-inp-coup transd (list (cons 'in 'solved)))

;; add the external output port pairs to the arcs of the
;; composition-tree

(send ef set-ext-out-coup genr (list (cons 'out 'out)))
(send ef set-ext-out-coup transd (list (cons 'out 'result)))

;; specify the influencees of each component

(send ef set-inf-dig (list (list genr transd) (list transd genr)))

;; add the internal port-pairs to the arcs of the influence-digraph

(send ef set-int-coup transd genr (list (cons 'out 'stop)))

(send ef set-int-coup genr transd (list (cons 'out 'ariv)))
```

Figure 8: Digraph-model specification of EF showing how the composition and influence digraph are constructed

```
;; load components of experimental frame, generator and transducer

(load "/scheme/devs/simparc/mbase/genr.m")
(load "/scheme/devs/simparc/mbase/transd.m")

;; couple them in a digraph-model

(make-pair digraph-models 'ef)

;; method specify-children calls build-composition-tree and
;; set-inf-dig

(send ef specify-children (list genr transd))

;; method add-couple figures out where to place the specified
;; port pairs from the source and destination given in each case

(send ef add-couple ef transd 'in 'solved)
(send ef add-couple genr ef  'out 'out)
(send ef add-couple transd ef  'out 'result)
(send ef add-couple transd genr 'out 'stop)
(send ef add-couple genr transd 'out 'ariv)
```

Figure 9: Digraph-model specification of EF using more user-friendly methods specify-children and add-couple

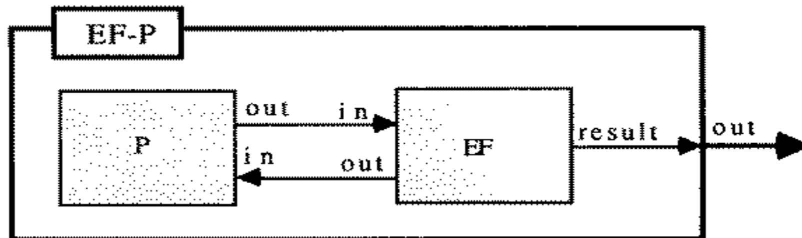
```
     add_component(g);
     add_component(t);

     add_coupling(this, "in", t, "solved");
     add_coupling(this, "start", g, "start");

//    get_coupling()->print();

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

//    t->get_coupling()->print();

    g->add_coupling(g, "out", this, "out");
    g->add_coupling(g, "out", t, "ariv");

//    g->get_coupling()->print();
}

};
```

2.2 Model/Experimental Frame Pairs

To experiment with the simple processor model P we couple it together with the experimental frame component EF to form the digraph model EF-P(Figure 5.8). Note that coupling together an experimental frame and a model forms a closed, input-free, system. Thus there is no external-input coupling in EF-P. The external-output coupling makes the transducer output available at the port 'result of EF-P. The internal coupling is rather simple: the 'out ports of each of the components are coupled to the 'in ports of the other. The experimental frame EF can be coupled to any other model in this way. Of course, to make sense, the model has to use its input port for receiving job-identifiers and its produce such job-identifiers at its output port. This will be true for a selection of simple architecture models we shall discuss. Used in such a way, the experimental frame will generate jobs for the architecture and measure its throughput and turnaround time.

Figure 5.9 contains the specification for digraph model EF-P and also enables this model-frame pair to be simulated using a root-co-ordinator called R.

```
//the header files of the experimental frame and processor must be included

class ef_p:public digraph
{
public:
ef_p(char * name,timetype int_arr_time,timetype observe_time,
```



DIGRAPH-MODEL: EF-P

<u>COMPOSITION TREE:</u>

root: EF-P
leaves: EF, P

<u>EXTERNAL INPUT COUPLING:</u> none

<u>EXTERNAL OUTPUT COUPLING:</u>

EF.result -> EF-P.out

<u>INFLUENCE DIGRAPH:</u>

P -> EF
EF -> P

<u>INTERNAL COUPLING:</u>

P.out -> EF.in
EF.out -> P.in

<u>PRIORITY LIST:</u> (P EF)

Figure 10: Composition tree and influence digraph for model/frame pair

```
;----------------------------------------------------------------
; This file uses a digraph model to couple the simple processor
; with the experimental frame.
;----------------------------------------------------------------

;; load the components

(load "/scheme/devs/simparc/mbase/p.m")
(load "/scheme/devs/simparc/coupbase/ef.m")

;; couple the experimental frame with the processor by p-ef

(make-pair digraph-models 'ef-p)

;; build the composition tree and influence digraph

(send ef-p specify-children (list p ef))

;; internal coupling

(send ef-p add-couple p ef 'out 'in)
(send ef-p add-couple ef p 'out 'in)

;; external coupling

(send ef-p add-couple ef ef-p 'result 'out)

;; define the select function to avoid losing a job when it
;; arrives at the time the processor is finishing: processor first
;; then generator

(define (sel-p slst)
   (cond ((member p slst) p)
         ((member ef slst) ef)
)  )

(send ef-p set-selectfn sel-p)

;; equivalently

 (send ef-p set-priority (list p ef))

;; is shorter and preferable when using flat-devs and deep-devs

;; the final touch, attach a root co-ordinator

(mk-ent root-co-ordinators r)

;; initialize it with the co-ordinator for ef-p

(initialize r c:ef-p)

;; start a simulation run

(restart r)
```

Figure 11: Digraph-model specification of EF-P

```
        timetype proc_time,int) :digraph(name)
 {
   exframe * e  = new exframe("e",int_arr_time,observe_time);
   proc* p  = new proc("p",proc_time);

    inports->add("start");
    outports->add("result");

    add_component(e);
    add_component(d);

    add_coupling(this, "start", e, "start");

   e->add_coupling(e, "out", d, "in");
   e->add_coupling(e, "result", this, "out");

   d->add_coupling(d, "out", e, "in");

 }


};
```

2.3 The Select Function: breaking schedule ties

Recall that in the DEVS coupled-model formalism, the select function contains the the rules for breaking scheduling ties. Such ties are relatively rare, but nevertheless can cause unpleasant behavior distortions. For example, in EF-P, if the generator and processor have inter-arrival-times and processing-times which are equal (or more generally, exact multiples of each other), they will have equal next-event-times when both are ready to send out a job simultaneously. In particular, if the generator inter-arrival-time and the processing-time are equal, every second job is lost if the generator carries out its next-event first: the generator output encounters a BUSY processor and is ignored. To prevent this obvious distortion from occuring, the *select-fn* instance variable of EF-P can be defined so that the processor is always chosen as the imminent component when there is tie.

As shown in Figure 5.9, there are two ways of specifying the select-fn. It can be defined directly or the *set-priority* method can be used. In the latter case, a particular kind of select function is implemented, namely that basedon a priority scheme. In the EF-P example, the direct definition (*sel-p*) is actually equivalent to the select-fn defined by the *set-priority* method. The priority-based approach is usually adequate and is properly handled by the *flattening* and *deepening* methods to be discussed later. Direct definition can be used when the linear ordering of the priority scheme is not sufficient.

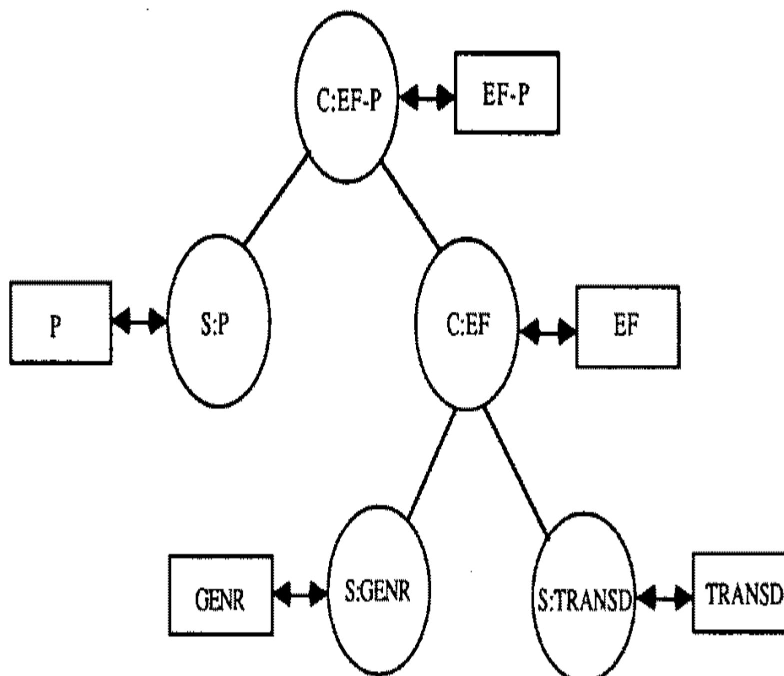


Figure 12: DEVS-Scheme processor hierarchy for simulation of EF-P

3 Co-ordinator of Coupled-Models

`There is select function in DEVS-C++`

Figure 5.10 shows the hierarchical structure of the processor configuration employed in DEVS-Scheme to simulate the model EF-P. We shall discuss the simulation process with reference to this example. Since we have already considered the simulation of atomic-models, we can assume the handling of the leaf components, GENR, TRANSD and P by their respective simulators is understood.

Recall that a simulation cycle starts with a **-message* sent by the root-co-ordinator to the co-ordinator of the outermost model, namely C:EF-P in Figure 5.10.

Consider then how a co-ordinator C:M of a coupled model M works. For example, M could be the model/frame pair EF-P or the frame EF. Arrival of a **-message* at a co-ordinator indicates that the next internal eventis is to be carried out within its scope. Thus C:M responds to a **-message* by transmiting it to its imminent child, the child with minimum *time-of-next-event* (or selected by *selectfn*), if more than one has the minimum time of next event). C:M places the imminent child in the *wait-list*. For C:EF-P, the imminent child may be either EF (when it is time to generate a new job) or P (when a job has been processed). For C:EF, GENR is repeatedly the imminent child

until TRANSD becomes imminent to stop the simulation.

We have seen that when S:GENR receives a **-message* it sends a *y-message* (applying its output function to produce a content with *port* 'out and *value* a job-id) to its parent C:EF. S:GENR also forces the execution of GENR's internal transition function, and sends a *done-message* with GENR's *time-of-next-event*.

When a C:M receives a *y-message* from its imminent child, it consults its internal coupling scheme to obtain the children and their respective input ports to which the message should be sent. C:M employs the methods *get-influencees* and *translate* to do this. Recall that in the case of digraph-models, the influencees of a component are explicitly specified in the influence-digraph. Thus the *get-influencees* method for digraph-models looks under the entry for the imminent child in the *influence-digraph* to obtain its influencees. The internal coupling port-pairs are also kept in the*influence-digraph*.

For each influencee, *translate* gets the input port corresponding to the output content-port in the *y-message*. Each influencee is sent an *x-message* which is contains the input-port just found in the content-port slot and is otherwise identical to the *y-message*. For C:EF, when GENR is imminent, the *y-message* it puts out on port 'out is translated to an *x-message* on port 'ariv and sent to its influencee, TRANSD. C:M then places each of the influencees in its *wait-list*: used to keep track of the processors of its components which have not yet completed their transitions.

Similarly, C:M consults the external output coupling table to see whether the *y-message* should be transmitted to its parent. The method translate is used again. For digraph-models, it looks up the port-pairs associated with the edge I-M (where I is the imminent child) in the *composition-tree*, to get the output port associated with content-port in the *y-message*, if there is one. If so the appropriately ammended *y-message* is transmitted to the parent of C:M or to the root-co-ordinator. If there is no output port then nothing is sent since there is no external output to be derived from that generated by the imminent-child in this transition.

For C:EF, when GENR is imminent, the *y-message* its puts out on port 'out appears as a *y-message* appearing at port 'out of parent EF. Since EF is the imminent child of EF-P, the latter *y-message* is sent to C:EF-P, where it will be sent as an *x-message* (as just discussed) to the simulator of P, the influencee of EF. Handling of such an *x-message* by a simulator has already been discussed. In this case, assuming P is idle, P is set into phase 'BUSY. When the processing-time has elapsed, S:P generates a *y-message* (a job-id on port 'out) which goes to its parent C:EF-P and then as an *x-message* to C:EF, the influencee of P. Let us see how this is sent as an *x-message* to S:TRANSD.

An *x-message* to C:M represents the arrival of an external event to M; it bears the global model time and comes from C:M's parent if M is itself a component in a more encompassing model. To carry out the external input coupling scheme, C:M transmits this message to the processors of M's receivers, using its *get-receivers* and *translate* methods. Every sub-class of coupled models can have its own *get-receivers* and *translate* methods. In the case of digraph-models, we have seen that the external input coupling isspecified by attaching port-pairs to the composition tree.

For digraph-models, the method *get-receivers* returns the subset of children of M that are coupled to the content-port in the *x-message*. For each receiver, R, the method *translate* looks up the set of external-input port-pairs associated with the edge M-R, to find the appropriate input port. For C:EF, the only receiver for an *x-message* on external input port 'in is TRANSD and translate maps port 'in to port 'solved. Thus the *x-message* that C:M sends on to S:TRANSD is the same as it receives except that the content-port is changed to 'solved. After retransmitting the *x-message* to all the receivers, C:M puts them in the*wait-list*.

Consider now that C:M is waiting to receive *done-message*s from the children on its *wait-list* (i.e., its imminent child as well as from all its inflencees in the ascending *y-message* case or the receivers in the descending *x-message* case). With the arrival of each *done-message*, C:M updates its *tN-children*, associating the *time-of-next-event* carried by the *done-message* with the child that sent it. When all*done-message*s have arrived, C:M computes the minimum of the *tN-children* and determines its new imminent ermines its new imminent child (using the *selectfn*) to use when receiving the next **-message*. Also it sends this new minimum as the time of its own next internal event in a *done-message* to its parent.

Children on its *wait-list* (i.e., its imminent child as well as from all its inflencees in the ascending *y-message* case or the receivers in the descending *x-message* case). With the arrival of each *done-message*, C:M updates its *tN-children*, associating the *time-of-next-event* carried by the *done-message* with the child that sent it. When all *done-message*s have arrived, C:M computes the minimum of the *tN-children* and determines its new imminent child (using the *selectfn*) to use when receiving the next **-message*. Also it sends this new inimum as the time of its own next internal event in a *done-message* to its parent. For example, if C:EF's *tN-children* is ((GENR 10)(TRANSD 1000)), this indicates that GENR is imminent and C:EF's own *time-of-next-event* is also 10.

Then C:EF send a *done-message* with time set to 10 to its parent, C:EF-P. The latter having received all its *done-message*s, sends a *done-message* to the root-co-ordinator. Assuming its *tN-children* is ((EF 10)(P 15)), C:EF-P's *done-message* contains the *time-of-next-event* 10. The *done-message* is sent to the root-co-ordinator and turned around to appear as a **-message* to C:EF-P with time 10. This starts a new simulation cycle, in this case with a GENR generating a second job.

In pause mode, a DEVS-Scheme simulation displays all the messages as they are processed. This aids understanding of the simulation process. With regard to model verification, it permits step-by-step tracking of the events, and hence makes for quicker discovery of sources of error.

3.1 Sample Simulation Results

Table 1 illustrates results obtained by simulating the single processor/frame pair. The processing time required by each job is fixed at 15. Each row represents a simulation run with a different job inter-arrival time. There are three cases:

1. When the inter-arrival time is greater than the processing time, the processor finishes a job before the next one arrives. Thus, departures are determined solely by arrivals and the throughput should be the same as the inter-arrival rate (1/inter-arrival time).

2. When the inter-arrival time is the same as the processing time, the processor is always kept busy, and the maximum throughput is attained. Note that in this case, both the generator and the processor are imminent at the same time. Moreover, the processor has no queueing capability so that it loses an incoming job if it has not released the current one. Thus, when both generator and processor are imminent, the selectfn must choose the processor to enable it to finish first.

3. When the inter-arrival time is less than the processing time, if the processor had unlimited queueing capability, maximum throughput would be maintained. However, the current processor does not have such ability, and will lose jobs if they come in while it is busy. Thus, maximum throughput will be attained only when the inter-arrival time exactly divides into the processing time.

The actual results obtained with simulation agree with the above expectations. Differences are attributable to well known end-effect inaccuracies due to the relatively short observation interval (100) (see Delany and Vacari (1989) for discussion of simulation output analysis).

As expected, the average turnaround time of jobs is identical to the processing time.

Similar considerations to those expressed in cases 1), 2) and 3) above can be applied to the basic multi-computer architectures to derive the results in Table 4.1. We next consider implementations of these architectures in DEVS-Scheme.

4 APPLICABILITY OF FRAMES TO MODELS: MODEL INSTRUMENTATION

An experimental frame E is *applicable* to a model M if when E is coupled to M, the experimentation objectives that gave rise to E can be achieved. A rigorous definition of the general concept of applicability requires considerable work as in Chapter 13 of Zeigler (1984). Here we are concerned with a more limited issue in applicability: whether the output variables specified in a frame E can be observed in a model M. The practical significance of this issue is the following:

M has output ports that enable it to send external events to other model components. If information desired by a frame E is available through such a port, we may couple this port to an input port of E. For example, the departing job-id that a simple processor places on port 'out is available to the 'solved port of TRANSD (via the 'in port of EF). However, what happens if information desired by a frame is not available through an existing output port of a model M?

	Throughput		Average turnaround time	
Inter-arrival time:	theoretical	experimental	theoretical	experimental
25	0.0400	0.0444	15	15.0
20	0.0500	0.0526	15	15.0
15	0.0667	0.0666	15	15.0
10	0.0500	0.0525	15	15.0
5	0.0667	0.0630	15	15.0

Figure 13: Performance of simple processor

One approach is to modify M so that it computes the desired values and sends them out on a newly created output port. There are three problems with this approach: 1) we must manually modify the code of M thereby possibly introducing error, 2) there are now two kinds of output ports, those that are necessary for M to interact with other components (presumably matching corresponding "ports" in reality) and those invented to couple M to an observation process (which may not exist in reality), and 3) due to the modifications and additional ports, the code of M has been rendered more difficult to understand and modify later.

A second approach which is supported by DEVS-Scheme obviates these problems at the cost of somewhat reduced simulation speed. We can "instrument" an atomic-model so that the value of any expression computable in its environment can be output prior to external or internal events. Such values appear on two special ports: '%ext-event% and '%int-event%, for external, and internal, events, respectively. The code of M, as it appears in its file specification, is not modified in this instrumentation. It thereforeremains uncompromised, and can be just as easily be instrumented to satisfy the needs of other experimental frames one may desire to apply to the model.

The methods, *add-int-event-observation* and *add-ext-event-observation*, which perform the foregoing instrumentation, are shown in Figure 5.11. For example, we can cause a model M to output the value of an expression, *exp*, in its environment on port '%int-event% just before every internal transition by appending the command to the end of its file:

(send M add-int-event-observation exp)

This modifies the object M (not the file code for M) so that its output function now sends out the value of *exp* to port '%int-event% in addition to its original output. One kind of transition is excepted: no ouput on port

```
(define-method (atomic-models add-ext-event-observation)(exp)
(def-state '(%ext-phase% %ext-sigma% %ext-result%))
(let (
     (old-int int-transfn)
     (old-ext ext-transfn)
     (old-out outputfn)
     (model (eval name user-initial-environment))
     )
(set! int-transfn (lambda(s)
   (if (equal? (state-phase s) '%ext-report%)
    (hold-in (state-%ext-phase% s) (state-%ext-sigma% s))
    (old-int s)
   )))
(set! ext-transfn (lambda (s e x)
   (let* (
          (result (eval exp model))
          )
   (old-ext s e x)
    (eval `(begin
      (set-sv '%ext-result%  ',result)
      (set-sv '%ext-sigma%  (get-sv 'sigma))
      (set-sv '%ext-phase% (get-sv 'phase))
      (hold-in '%ext-report% 0)
      ) model)
  )
))
(set! outputfn (lambda(s)
(if (equal? (state-phase s) '%ext-report%)
    (make-content 'port '%ext-event%
                  'value (state-%ext-result% s))
    (old-out s)
    )))
))
```

Figure 14: The add-ext-event-observation method

As an example of external event instrumentation, consider the following:

```
(send P add-ext-event-observation
        '(when (equal? (state-phase s) 'busy)
               (list (content-value x) 'lost))).
```

This will cause P to put out the list '(j lost) on port '%ext-event% whenever a job, j arrives while P is busy; the empty list is output on port '%ext-event% if a job arrives when P is free. Notice, in Figure 5.11b, that the expression *exp* is evaluated before the external transition takes place. Thus, if P is free when a job arrives, it will still be in phase PASSIVE when the (equal? (state-phase s) 'busy) test is performed. This results in an empty *content-value* for port '%ext-event% as desired.

Outfitting P with capability to report ignored jobs by direct means is discussed in Chapter 8. If the real world counter-part of P does not have such a capability, it is best if the model, P, does not have it either. Note however, that an experimental frame for P may legitimately wish to get such information. The *add-ext-event-observation* method permits us to satisfy the frame requirements without corrupting the file specification of P.

A model can be given both internal and external event instrumentation. For example, we can add the command

```
(send P add-int-event-observation
              '(get-vals '(job-id processing-time))).
```

Processor model P will now output information about lost jobs on port '%ext-event% and about accepted jobs on port '%int-event% as shown inFigure 5.12. Note that we can omit all instrumentation ports from graphical displays, thus showing only intra-model related couplings. Likewise, we can omit all non-special ports from a display to emphasize model/frame couplings.

In summary, we have demonstrated a partial implementation of the applicability relation proposed in Chapter 13 of (Zeigler, 1984). In this approach, a frame E is applicable to a model M if the variables required by E can be evaluated in the environment of M. If E is applicable to M, we can outfit M with the %int-event% and %ext-event% output ports and couple them to appropriate input ports of E. This avoids corruption of of M's specification. Once more, this separation of model and frame properties fosters reusability and better comprehension.