

Chapter 6: A MODEL-BASE FOR SIMPLE MULTI-COMPUTER ARCHITECTURES

At this point, we have seen how to express atomic-models and digraph-models in DEVS-Scheme. Specifically, a simple processor model/experimental frame pair was constructed. We can now design other simple architectures to replace the simple processor in such a pair and study each one's performance under the same conditions.

Three basic multiprocessing configurations will be modeled, each having a co-ordinator which 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 our model, in contrast to typical hardware pipelines, the problems are routed by the co-ordinator from one processor to the next. In the *divide and conquer* architecture, problems are decomposed into subproblems that are worked on concurrently and independently by the processors. When all partial solutions are available, they are then sent to a compiling processor to be put together to form the final solution. In each of the architectures, problems arrive at the co-ordinator and emerge from it.

1 CO-ORDINATORS AND ARCHITECTURES

We want to study the throughput and turn-around time performance measures of the foregoing architectures and will do so by coupling the experimental frame in Chapter 5 to each of them. In this way, the performance relations presented in Table 4.1 can be tested. The architectures themselves are built up in the following manner:

1. define the co-ordinator— an atomic model— appropriate to each case,
2. create copies of the simple processor to serve as the subordinate processors, and
3. define each architecture as a digraph model coupling together the corresponding co-ordinator and the subordinate processors.

In what follows, for each architecture, we provide a pseudo-code description of the co-ordinator, its DEVS-Scheme implementation, and the DEVS-Scheme implementation of the architecture. Later we discuss testing and simulating the architectures. Although the models are highly simplified — problems are represented only by time of processing not by actual content—they illustrate significant aspects of model definition in DEVS-Scheme.

1.1 Multiserver Co-ordinator

As described in Figure 6.1, the co-ordinator, MUL-C keeps track of the status of its processors in corresponding state variables. A problem arrives at the input port 'in and is routed to the first

passive processor by being sent out on a corresponding to output port, 'x1, 'x2, or 'x3. If no processor is free, the problem is lost. When a solved problem returns on corresponding ports, 'y1, 'y2, or 'y3, MUL-C re-routes it to the output port 'out. For simplicity, we have MUL-C taking 0 time to do such work, as indicated by the “hold-in busy 0” phrase in the external transition function. This causes the output function to be invoked immediately after the external transition.

Note that in the case of a problem arriving when all processors are busy, the output function creates a null content structure (one with empty port and value slots). Since the output function must always produce a content object, such a null structure should be the default. Since its port slot is empty, a message containing this null content has nowhere to go.

C++:

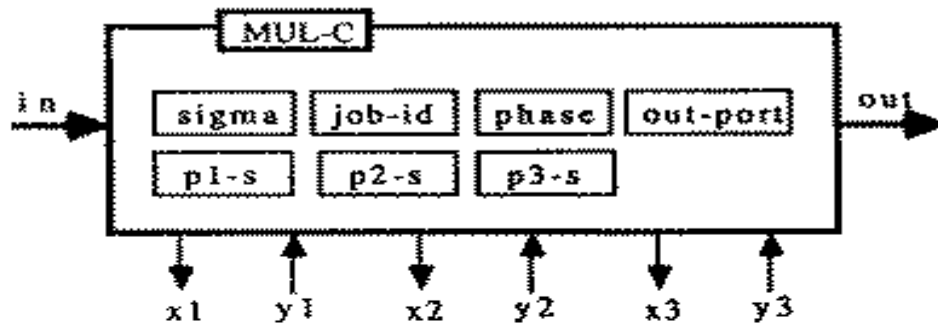
The three different coordinators are derived from a common coordinator class which is itself, derived from the simple processor. Class Stack is used to manage both jobs and processors.

```
class stack:public set{
public:
void remove(){remove_head();}
void clear()
{
while (!empty()) remove();
}
};

class coord:public proc{
public:
coord(char * name):proc(name,.01)
{
inports->add("x");
outports->add("y");
phases->add("send_out");
phases->add("send_y");
}
virtual void add_procs(devs * p){}
};

class mulch:public coord{

protected:
stack * procs;
stack * jobs;
proc * pcur;
```

**ATOMIC-MODEL: MUL-C**

state variables: sigma = inf
 phase = passive
 p1-s = passive
 p2-s = passive
 p3-s = passive
 out-port = ()
 job-id = ()

external transition function:

```

store job-id
case input-port
  in: sequentially select an idle subordinate and set the
      out-port (destination) to the corresponding value. i.e.
      if p1-s passive then set out-port = x1 and p1-s = busy
      if p2-s passive then set out-port = x2 and p2-s = busy
      if p3-s passive then set out-port = x3 and p3-s = busy
      when receive solution from a subordinate send to out:
      y1: reset p1-s to passive and set out-port = out
      y2: reset p2-s to passive and set out-port = out
      y3: reset p3-s to passive and set out-port = out
  hold-in busy 0

```

internal transition function:

```

case phase
  busy: passive
  passive: (does not arise)

```

output function:

```

case phase
  busy: case out-port
        x1, x2, x3, out: output job-id to out-port
        else make a null output

```

Figure 1: Pseudo-code for co-ordinator of multi-server architecture

```

: Multi-server Co-ordinator :
:-----:
: This file contains the definition of the co-ordinator in
: Multi-server architecture.
:-----:
: It should perform following tasks:
: 1) Gets a job from input and sends the job to any of the three
: processors that is not busy. If all processors are busy,
: job is lost.
: 2) When finished job is returned from one of the processors,
: sends it to output.
:-----:

: make a pair for the co-ordinator in multi-server module
(make-pair atomic-models 'mul-c)

(send mul-c def-state '(
    p1-s      ;;phase of p1
    p2-s      ;;phase of p2
    p3-s      ;;phase of p3
    out-port  ;;holds next destination port
    job-id    ;;holds job id
})

)
;; initialize the state

(send mul-c set-s      (make-state      'sigma 'inf
                                'phase 'passive
                                'p1-s 'passive
                                'p2-s 'passive
                                'p3-s 'passive
                                'out-port '()
                                'job-id  '()
                                )

)

;; external transition function

(define (ext-mc s e x)
  (set! (state-out-port s) '()) ; default, no port to be sent to
  (set! (state-job-id s) (content-value x))
  (case (content-port x)
    ; case 1. input from outside of the world
    ('in
     ; find a processor not busy and send the job to it

     (cond
      ( (equal? (state-p1-s s) 'passive)
        (set! (state-out-port s) 'x1)
        (set! (state-p1-s s) 'busy))
      )
    )
  )

```

Figure 2: Atomic-model specification of co-ordinator of multi-server architecture

```

public:

mulco(char * name):coord(name){
jobs = new stack();
procs = new stack();
}

void add_procs(proc * p){
procs->add((proc *)p);
pcur = p;
}

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);
if (!procs->empty())
{
element * e = procs->get_head();
pcur = (proc *) (e->get_ent());
//only one job on input can be accepted
procs->remove();
hold_in("send_y",0.1);
}
}

// (proc,job) pairs returned on port x

for (/* int */ i=0; i< x->get_length();i++)
if (message_on_port(x,"x",i))
{
entity * val = x->get_val_on_port("x",i);
procs->add(((pair *)val)->get_ent());
entity * jb = ((pair *)val)->get_value();
jobs->add(jb);
}
if (!jobs->empty())
hold_in("send_out",0.1);
}

```

```

    }
}

void deltint( )
{
if (phase_is("send_out")) jobs->clear();
passivate();
}

message * out( )
{
message * m = new message();
if (phase_is("send_out"))
for (element * pt = jobs->get_head();pt != NULL;pt = pt->get_right())
{
entity * job = pt->get_ent();
m->add( make_content("out",job));
}
else
if (phase_is("send_y"))
m->add(make_content("y",new pair(pcur,job)));
return m;
}

void show_state()
{
cout << "\nstate of " << name << ": " ;

cout << "phase, sigma,pcur,jobs,procs : "
<<phase << " " << sigma << " " ;
pcur->print();

jobs->print(); procs->print();
cout<<endl;

}

};

```

1.1.1 Multiserver Architecture Specified Directly as a Digraph Model

Figures 2a,b describe the digraph model, MUL-ARCH that implements the multiserver architecture. Note how the external-input coupling connects the 'in port of MUL-ARCH to the 'in port

of MUL-C, while the external-output coupling similarly connects the 'out ports together. The internal coupling connects the sending and receiving ports of MUL-C to corresponding ports of the subordinate processors.

1.1.2 State Trajectories and Performance Indexes

Let us trace a typical state trajectory to illustrate the operation of the coupled-models in general, and the multiserver architecture in particular. We start in an initial state in which the multiserver co-ordinator and all subordinate processors are idle. Imagine that the experimental frame, *ef*, is coupled to the architecture in the same manner as it was for the simple processor architecture. This will result in jobs arriving on port 'in of MUL-ARCH from GENR and leaving on port 'out for TRANSD. Figure 6.2c shows how we can represent the time behavior for a coupled-model. The incoming job stream is represented by the segment of external input events shown on the top horizontal axis. The co-ordinator and each of the three processors is assigned its own axis.

Following the course of the first job arrival, J1, on port 'in of MUL-ARCH, the external input coupling scheme will send J1 to port 'in of the co-ordinator, MUL-C.¹ Having received J1 and being passive, MUL-C goes into state BUSY (dictated by its external transition function). After waiting there for a very short time (actually zero), the co-ordinator puts J1 on port 'x1 (as dictated by the output function) and immediately returns to the passive phase (due to the internal transition function).

The internal coupling of MUL-ARCH then causes J1 to be appear on port 'in of processor P1. Since the latter is idle, it accepts the job and enters the BUSY phase for a time given by its *processing-time* parameter (recall the description of the simple processor P in Section 4.2, of which P1 is an isomorphic copy). Let p represent the value of the processing time. For simplicity in the sequel, we shall assume that p is a constant and the same for all processors. After time p has elapsed, P1 will place J1 on port 'out. The external output coupling now determines that J1 appears on port 'out of MUL-ARCH and leaves the architecture as a processed job as illustrated in Figure 6.2c.

Now let a second job, J2, arrive T time units after J1's arrival. If T is bigger than p , then P1 will be passive by the time J2 arrives and will start processing it. However, if T is smaller than p , then P1 will be busy when J2 arrives. Rather than losing J2 as was the case for the simple processor, here the multi-server co-ordinator comes into play. Knowing that P1 is busy, MUL-C sends J2 to the next free processor, which happens to be P2. More truthfully, MUL-C places J2 on its output port 'x2, which is coupled by the internal coupling of MUL-ARCH to P2's input port 'in. J2 will be sent out of MUL-ARCH p units later in a manner similar to J1's processing.

A third job, J3, arriving while both P1 and P2 are busy, will be sent to P3. However, a fourth job that arrives while all processors are busy will be lost. As illustrated in Figure 6.2c, if the job

¹Recall that in DEVS-Scheme this transmission is realized by having a co-ordinator C:MUL-ARCH call on its devs-component MUL-ARCH to use its *get-receivers* and *translate* methods to return the receivers (here only MUL-C) of the external event, J1 and the port on which it will be sent. Please do not confuse the co-ordinator C:MUL-ARCH used in the simulation engine with the co-ordinator MUL-C, a model component.

inter-arrival-time, T , is a constant, equal to $p/3$, then the fourth and subsequent jobs arrive just after a (exactly one) processor has finished its work. The figure makes clear that this is an arrival pattern in which processors are always kept busy. The emerging jobs are separated in time by $p/3$ so that the throughput is $3/p$. Since the processors are always kept busy, there is no way to produce a higher rate of job completions. Thus we can justify the entry for the multi-server architecture with constant processing time in Table 4.1. Clearly, each job still takes time p units to be processed, so that the average turnaround time is p as in the Table.

In the case of heterogeneous processing times $\{p_i\}$, the fastest that each processor, P_i , can emit jobs is at rate $1/p_i$. The maximum throughput is the sum of these rates. The average turnaround time associated with this departure rate can be derived from Little's relation (Sauer and Chandy, 1980). It is the number of jobs in the system (i.e., 3) divided by the departure rate, accounting for the corresponding entry in Table 4.1. Note that Little's formula relates the two performance indexes, throughput and turnaround time, as inverses of each other, just as intuition would have us imagine. However, it holds only for certain kinds of systems in which jobs are distributed uniformly around the processors at all times.

C++:

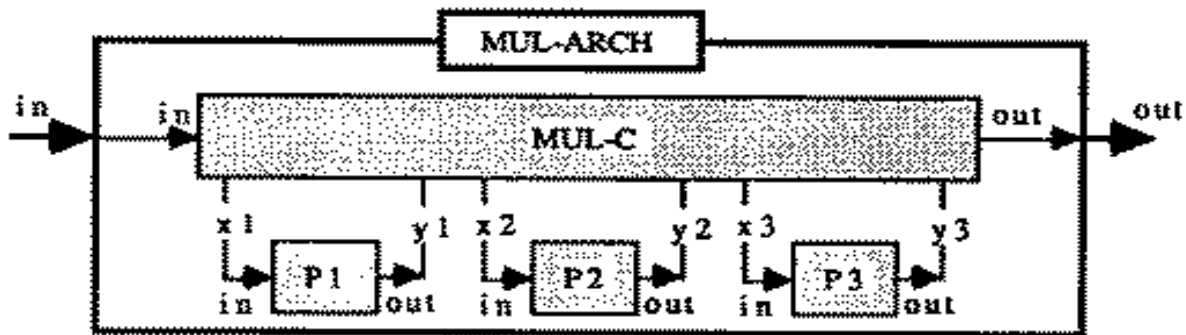
```
class mul:public digraph{
public:
mul(char * name,timetype proc_time,int size):digraph(name)
{
    inports->add("in");
    outports->add("out");

    mulco * co = new mulco("co");
    add(co);

    add_coupling(this, "in", co, "in");
    co->add_coupling(co,"out",this,"out");

    for (int i = 1; i <= size; i++){
        proc * p = new proc(name_gen("p",i),
                            proc_time);
        add(p);
        co->add_procs(p);
    }

    for (element *p = components->get_head();p != NULL;p = p->get_right()){
        entity * ent = p->get_ent();
        devs * comp = (devs *)ent;
        if (!ent->equal(co))
        {
```

**DIGRAPH-MODEL: MUL-ARCH****COMPOSITION TREE:**

root: MUL-ARCH
 leaves: MUL-C, P1, P2, P3

EXTERNAL-INPUT COUPLING:

MUL-ARCH.in → MUL-C.in

EXTERNAL-OUTPUT COUPLING:

MUL-C.out → MUL-ARCH.out

INFLUENCE DIGRAPH:

MUL-C → P1, P2, P3
 P1 → MUL-C
 P2 → MUL-C
 P3 → MUL-C

INTERNAL COUPLING:

MUL-C.x1 → P1.in	MUL-C.x2 → P2.in
MUL-C.x3 → P3.in	P1.out → MUL-C.y1
P2.out → MUL-C.y2	P3.out → MUL-C.y3

PRIORITY LIST:

P1 P2 P3 MUL-C

Define the select function in order to avoid the loss of jobs which arrive at the same time the processors finish their jobs. Thus, the processors all have higher priority than the co-ordinator.

Figure 3: Pseudo-code for the multi-server architecture

5.0in

```

:;;;;;;;;;;;;; multi-server architecture ;;;;;;;;;;;;;;
;-----
; This file contains the construction of the multi-server
; architecture using a digraph-model. The components
; are retrieved from prototypes in the model-base.
; Components: Three sub-processors are copies of p in file "p.m"
;             One co-ordinator : mul-c defined in "mul-c.m"
;-----

(load-from model-base_directory p.m)
(load-from model-base_directory mul-c.m)

;; make three copies from original p processor and copy its
;; initial state

(send p make-new 'p1)
(send p make-new 'p2)
(send p make-new 'p3)

;;now couple them to the multi-server

(make-pair digraph-models 'mul-arch)

;;build composition tree and influence digraph

(send mul-arch specify-children (list mul-c p1 p2 p3))

;; external-input coupling

(send mul-arch add-couple mul-arch mul-c 'in 'in)

;; external-output coupling

(send mul-arch add-couple mul-c mul-arch 'out 'out)

;;internal coupling between processors and co-ordinator

(send mul-arch add-couple mul-c p1 'x1 'in)
(send mul-arch add-couple p1 mul-c 'out 'y1)
(send mul-arch add-couple mul-c p2 'x2 'in)
(send mul-arch add-couple p2 mul-c 'out 'y2)
(send mul-arch add-couple mul-c p3 'x3 'in)
(send mul-arch add-couple p3 mul-c 'out 'y3)

;; define the select function to avoid job loss when it
;; is sent from co-ordinator to a processor just as the
;; process is finishing its current job: processors first,
;; then co-ordinator

(send mul-arch set-priority (list p1 p2 p3 mul-c))

```

Figure 4: Digraph model specification of the multi-server architecture


```
'in → 'x1
'y1 → 'x2
'y2 → 'x3
'y3 → 'out
```

1.2.1 Pipeline Architecture

The coupling of the pipeline architecture follows exactly the form of the multiserver architecture with PIP-C replacing MUL-C as the co-ordinator of the processors. Specifying the digraph-model, PIP-ARCH is therefore a straightforward revision of the specification of MUL-ARCH. Later we shall show how the system entity structure formulation enables us to take advantage of such isomorphisms to reduce the amount of specification needed.

Figure 6.3c displays a typical state trajectory for the pipeline architecture. Note the progress of jobs through the successive stages of the pipeline. Clearly, the turnaround time is sum of the processing times a job encounters. How soon can job J2 arrive after J1 and not be lost? Let T be the time separating their arrivals. So long as J2 encounters only idle processors, this time difference is preserved as J2 follows J1 through the system. However, if T is smaller than some processing time, p_i , then P_i will be busy with J1 when J2 arrives. Said another way, the maximum throughput is the rate at which jobs can emerge from the slowest, or *bottleneck* processor. Jobs emerging from a faster processor upstream of the bottleneck will eventually encounter the bottleneck; a faster processor downstream can only get its input at the rate emerging from the bottleneck. Since $\max p_i$ is the largest processing time, its inverse is the maximum throughput as in Table 4.1.

Consider the problem: minimize $\max p_i$ subject to $\sum p_i = p$. The answer is $\frac{p}{n}$ as can be shown by induction on n . This means that the best partitioning of a problem for pipeline processing with n stages occurs when each stage takes the same time, $\frac{p}{n}$.

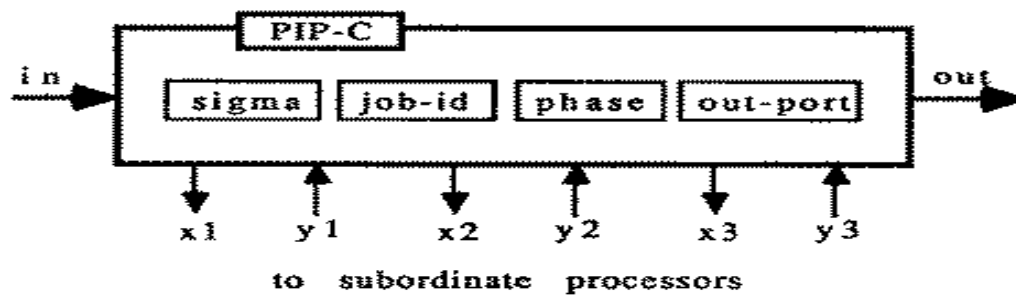
C++:

```
class pipco:public coord{

protected:
function * next;
proc * pcur,* pfirst;
stack * jobs;

public:

pipco(char * name):coord(name){
next = new function();
jobs = new stack();
phases->add("send_first");
}
```

**ATOMIC-MODEL: PIP-C**

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

external transition function:

```
store job-id
case input-port
  in: set out-port to x1
  y1: set out-port to x2
  y2: set out-port to x3
  y3: set out-port to out
hold-in busy 0
```

internal transition function:

```
case phase
  busy: passive
```

output function:

```
case phase
  busy: case out-port
    x1, x2,x3,out: output job-id to out-port
    else make a null output
```

Figure 6: Pseudo-code for co-ordinator of pipeline architecture

3.0in

```

: Pipeline Co-ordinator :
-----
: This file contains the definition of the co-ordinator in
: Pipe-line architecture.
-----
: It should perform following tasks:
: 1) Gets a job from input and sends the job to the first
: processor.
: 2) If a finished job is returned from one of the processors,
: sends it to next processor in the pipeline.
: If the returning processor is the last processor, sends the
: job to output.
-----

: make a pair for the co-ordinator in pipeline module

(make-pair atomic-models 'pip-c)

(send pip-c def-state '(
                        out-port ;;holds next destination port
                        job-id
                        )
)

;; initialize the state

(send pip-c set-s (make-state 'sigma 'inf
                              'phase 'passive
                              'out-port '()
                              'job-id '()
)

)

;; external transition function

(define (ext-ppc s e x)
  (set! (state-out-port s) '())
  (set! (state-job-id s) (content-value x))
  (case (content-port x)
    ; case 1. input from outside world
    ('in
     ; always send to first processor

     (set! (state-out-port s) 'x1)
     )
    ;case 2. input from the subordinate processors

    ('y1 (set! (state-out-port s) 'x2 ))
    ('y2 (set! (state-out-port s) 'x3 ))
    ('y3 (set! (state-out-port s) 'out))
    ;send last result out
  ) ; end of case
  (hold-in 'busy 0)
)

```

```
void add_procs(devs * p)
{
    if (next->empty())
        pfirst = (proc *)p;
    else next->replace(pcur,p);

    next->add(p,new entity("null"));
    pcur = (proc *)p;
}

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);
                pcur = pfirst;
                hold_in("send_first",0.1); //only one input job
            }
        for (/* int */ i=0; i< x->get_length();i++){
            if (message_on_port(x,"x",i))
            {
                entity * val = x->get_val_on_port("x",i);
                jobs->add(val);
            }
            if (!jobs->empty())
                hold_in("send_y",0.1);
        }
    }
}

void deltint( )
{
    if (phase_is("send_y"))
        jobs->clear();
    passivate();
}
```

```

}

message * out( )
{
message * m = new message();
  if (phase_is("send_first"))
    m->add( make_content("y",new pair(pcur,job)));
else if (phase_is("send_y"))
{
for (element *p = jobs->get_head(); p != NULL;p = p->get_right()){
  entity * ent = p->get_ent();
  entity * p_return = ((pair *)ent)->get_ent();
  pcur = ((proc *)next->assoc(p_return));
  job = ((pair *)ent)->get_value();
  if (!pcur->eq("null"))
    m->add( make_content("y",new pair(pcur,job)));
  else
    m->add( make_content("out",job));
}
}

return m;
}

void show_state()
{
cout << "\nstate of " << name << ": " ;
cout << "phase, sigma, jobs, pcur : "
    << phase << " " << sigma << " " ;
    jobs->print(); pcur->print();

    cout<<endl;
}

};

```

1.3 Divide and Conquer Co-ordinator

As described in Figure 6.6.4, the divide and conquer co-ordinator, DC-C is somewhat more complex than the preceding co-ordinators. Part of its operation is like that of the pipeline. It routes an incoming problem first to the partitioner, then to the processors, to the compiler, and then out. Conceptually, the partitioner divides the problem into sub-problems to be sent to the individual

```

;;;;;;; internal transition function
(define (int-ppc s)
  (case (state-phase s)
    ('busy
     (passivate)
    ) ) )

;;;;;;; output function
;;;;;;; output the value to corresponding port
(define (out-ppc s)
  (case (state-phase s)
    ('busy
     (case (state-out-port s)
       ((x1 x2 x3 out)
        (make-content 'port (state-out-port s)
                      'value (state-job-id s))
        )
       (else (make-content))
     )
    ) ) ) )

;;;;;;; assignment to the model
(send pip-c set-ext-transfn ext-ppc)
(send pip-c set-int-transfn int-ppc)
(send pip-c set-outputfn out-ppc)

```

Figure 8: Internal Transition Function

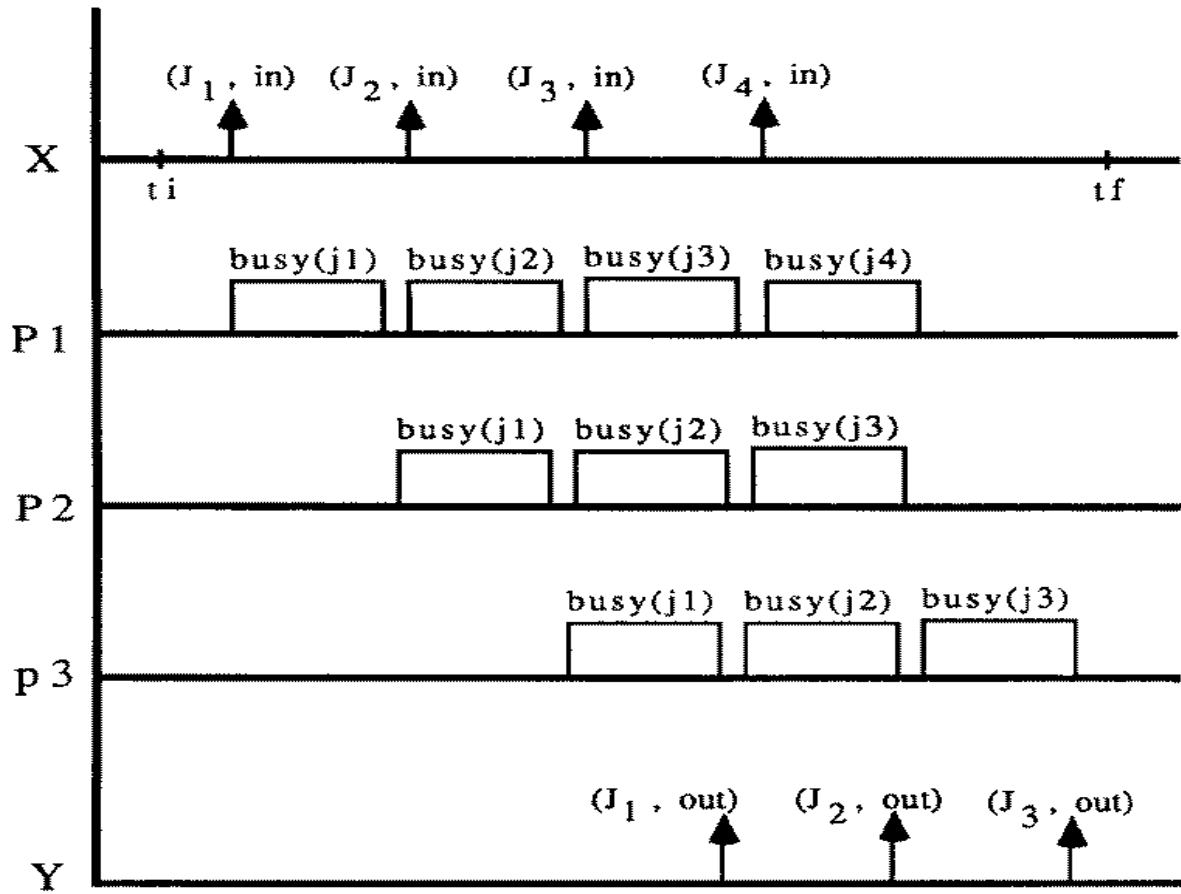


Figure 9: Sketch trajectory for pipeline architecture

processors. In our simplified model, the partitioning time is accounted for but the problem is not actually partitioned. Instead the job identifier is sent to each of the processors simultaneously. This is done in the output function *out-dc* using a list of content structures:

```
(list
  (make-content 'port 'x1 'value (state-job-id s))
  (make-content 'port 'x2 'value (state-job-id s))
  (make-content 'port 'x3 'value (state-job-id s))
).
```

In general, the simulation process in DEVS-Scheme will output the successive content structures in any normal list one after another without advancing the simulation clock, thus effectively outputting them simultaneously. Before sending out the “partial problems” to the processor, DC-C checks that they are all free. This keeps the processors all working on the same problem.

C++:

```
class dco:public coord{

protected:
  int num_procs;
  int num_results;

public:

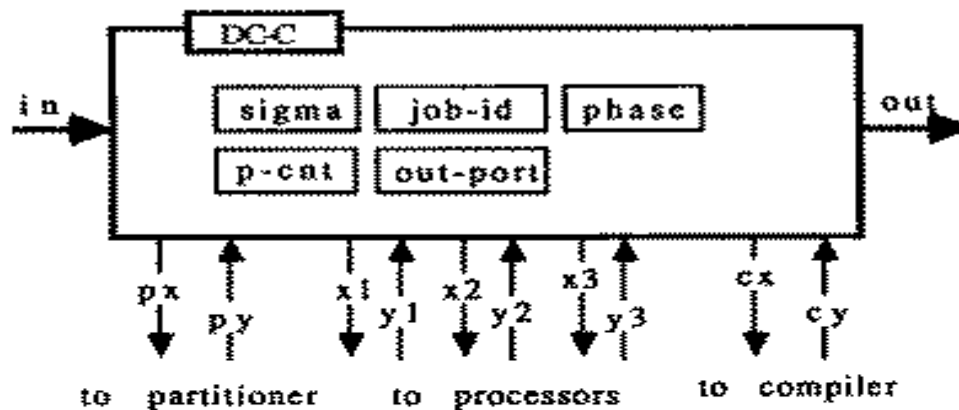
  dco(char * name):coord(name){
    num_procs = 0;
    num_results = 0;
  }

  virtual void add_procs(devs * p)
  {
    num_procs++;
    num_results++;
  }

  void deltext(timetype e,message * x)
  {

  Continue();

  if (phase_is("passive")){
    for (int i=0; i< x->get_length();i++)
```



ATOMIC-MODEL: DC-C ;; divide & conquer co-ordinator

state variables: sigma = inf phase = passive
 out-port = () job-id = ()
 p-cnt = 3 ;; count of free processors

external transition function:

store job-id
 case input-port

in: always send to problem partitioner by setting out-port to 'px
 py: if three sub-processors are free then set out-port to 'xin
 indicating to output function that a job is available in job-id
 (problem is not partitioned in this simple model). Otherwise
 job is lost.

y1, y2, y3: partial result returned by one of the processors
 increment p-cnt by one.
 if p-cnt = 3 then send job-id to the port cx
 set out-port to cx
 clear p-cnt

hold-in busy 0

internal transition function:

case phase
 busy: passive

output function:

case phase
 busy: case out-port
 xin: send job-id to each processor
 px,cx,out: output to given port
 else make a null output

Figure 10: Pseudo-code for co-ordinator of divide-and-conquer architecture

```

;::::::::::::: Divide and conquer co-ordinator :::::::::::::::
;-----
; This file contains the definition of the co-ordinator in
; divide and conquer architecture.
;-----
; It should perform following tasks:
; 1) Gets a problem from input and sends the problem to
;    problem-partitioner
; 2) When the divided problem is sent back, decides whether the
;    sub-processors are ALL available. If they are, the sub-problems
;    will be send to all sub-processors. If not, problem is lost.
; 3) After collecting all the returned results from sub-processors,
;    sends the returned partial results to post-compiler.
; 4) Gets the final result from compiler and sends it to output.
;-----

;; make a pair for the co-ordinator in divide and conquer module
(make-pair atomic-models 'dc-c)

(send dc-c def-state '(
    p-cnt      ;; number of partial solutions
                ;; received
    out-port   ;; destination for next output
    job-id
))

;; initialize the states of this module
(send dc-c set-s (make-state 'sigma      'inf
                             'phase     'passive
                             'p-cnt     3
                             'out-port  '()
                             'job-id   '()
))

;::::::::::: Definition of divide and conquer co-ordinator
;; external transition function
(define (ext-dc s e x)
  (set! (state-out-port s) '())
  (set! (state-job-id s) (content-value x))
  (case (content-port x)

    ; case 1. arrival of a problem
    ('in   ;; Always send to partition processor
     (set! (state-out-port s) 'px)
    )

    ; case 2. input from partition processor
    ('py   ;; check whether the processors are all free
     (if (= (state-p-cnt s) 3)

```

Figure 11: Atomic-model specification of co-ordinator of divide-and-conquer architecture

```

    if (message_on_port(x,"in",i))
    {
        job = x->get_val_on_port("in",i);
        num_results = num_procs;
        hold_in("send_y",0.1);
    }
} //needed
else if (phase_is("busy")){
    for (int i=0; i< x->get_length();i++)
    if (message_on_port(x,"x",i))
    {
        num_results--;
        if (num_results == 0)
            hold_in("send_out",0.1);
    }
}
else {
    cerr << "phase error: " << name ;
    exit(0);
}
}

void delrint( )
{
    if (phase_is("send_y"))
        passivate_in("busy");
    else
        passivate();
}

message * out( )
{
    message * m = new message();
    if (phase_is("send_out"))
        m->add( make_content("out",job));
    else if (phase_is("send_y"))
        m->add(make_content("y",job));
    return m;
}

void show_state()
{
    cout << "\nstate of " << name << ": " ;
    cout << "phase, sigma,num_procs,num_results : "

```

```

    <<phase << " " << sigma <<
        " " <<num_procs <<
        " " <<num_results<<endl;
}

};

```

1.3.1 Divide and Conquer Architecture

As illustrated in Figure 6.5a, the coupling required to form the divide and conquer architecture is very similar to the previous cases. The only difference is that five, rather than just three processors are connected to the co-ordinator. Indeed, if we group the partitioner and the compiler together with the co-ordinator, to form a new co-ordinator (now itself a digraph model), we once again have an architecture isomorphic to the multiserver. We shall take this approach when creating a compact system entity structure for this model family (Section 7.3.3).

A typical state trajectory for the divide and conquer architecture is shown in Figure 6.5c. You will see that the three processors, act in effect, as one stage in the sequence from input to output. This is so since to accept a new problem the co-ordinator DC-C requires that all processors have finished the subproblems of the current one. Since they are processed concurrently, the time to solve all subproblems is the time taken to finish the longest one, i.e., $\max p_i$. Considering DC-ARCH as a pipeline, and using the results just obtained for the latter, verifies the results in Table 4.1.

C++:

```

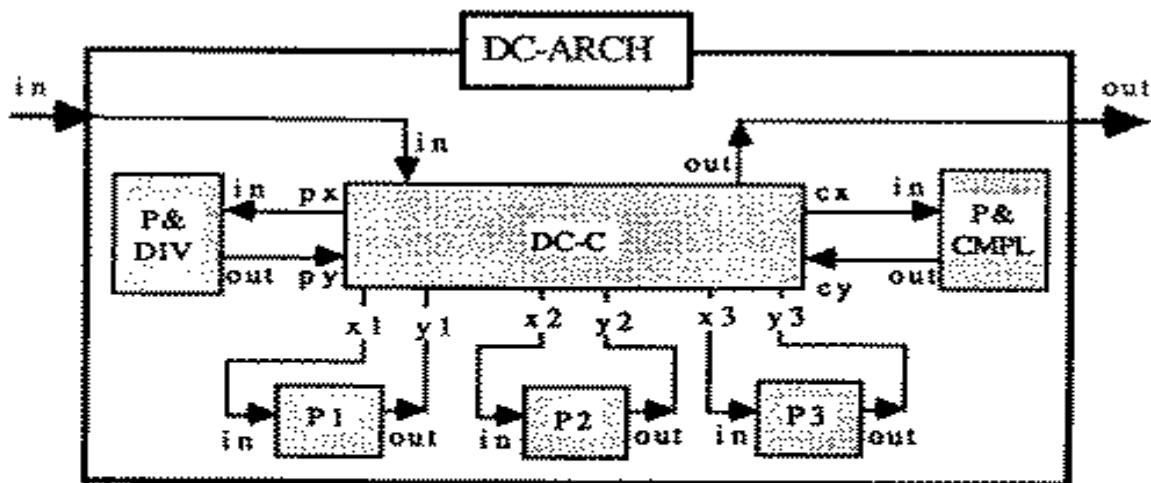
class dc:public digraph{
public:
dc(char * name,timetype proc_time,int size):digraph(name)
{
    inports->add("in");
    outports->add("out");

    coord * co = new dco("co");
    add(co);

    add_coupling(this, "in", co, "in");
    co->add_coupling(co,"out",this,"out");

    for (int i = 1; i <= size; i++){
        proc * p = new proc(name_gen("p",size),
                            proc_time/size);
        add(p);
    }
}

```

**DIGRAPH-MODEL: DC-ARCH****COMPOSITION TREE:**

root: MUL-ARCH

leaves: DC-C, P&DIV, P1, P2, P3, P&CMPL

EXTERNAL-INPUT COUPLING: DC-ARCH.in → DC-C.in**EXTERNAL-OUTPUT COUPLING:** DC-C.out → DC-ARCH.out**INFLUENCE DIGRAPH:**

DC-C → P1, P2, P3, P&DIV, P&CMPL

P1 → DC-C P2 → DC-C P3 → DC-C

P&DIV → DC-C P&CMPL → DC-C

INTERNAL COUPLING:

DC-C.x1 → P1.in

P&DIV.out → DC-C.py

DC-C.x2 → P2.in

P1.out → DC-C.y1

DC-C.x3 → P3.in

P2.out → DC-C.y2

DC-C.px → P&DIV.in

P3.out → DC-C.y3

DC-C.cx → P&CMPL.in

P&CMPL.out → DC-C.cy

define the selection function to avoid collision when a job arrives at the same time the processor finishes:

PRIORITY LIST: P&CMPL P1 P2 P3 P&DIV DC-C

Figure 12: Pseudo-code for divide-and-conquer architecture

5.0in

```

;;;;;;;;;;;;; Divide and Conquer Architecture ;;;;;;;;;;;;;;
-----
; This file contains the construction of the divide and conquer
; architecture by using digraph-model. The components
; are retrieved from the models.m defined in model-base.
;; components: One job partition process (p.m) -- p&div.
;               Three sub-processors (p.m) -- p1, p2, p3.
;               One post-compiler (p.m) -- p&cmpl.
;               One co-ordinator (dc-c.m) --- dc-c.
-----

;; get processor and co-ordinator

(load-from model-base_directory p.m)
(load-from model-base_directory dc-c.m)

;; make five copies from the original p processor
; first the pre-job-partition processor

(send p make-new 'p&div)

; and the post-compiler

(send p make-new 'p&cmpl)

; three sub-processors

(send p make-new 'p1)
(send p make-new 'p2)
(send p make-new 'p3)

;; now couple them together

(make-pair digraph-models 'dc-arch)

(send dc-arch specify-children (list dc-c p&div p1 p2 p3 p&cmpl))

;; external-input coupling

(send dc-arch add-couple dc-arch dc-c 'in 'in)

;; external-output coupling

(send dc-arch add-couple dc-c dc-arch 'out 'out)

;; internal coupling

(send dc-arch add-couple dc-c p&div 'px 'in)
(send dc-arch add-couple p&div dc-c 'out 'py)

```

Figure 13: Digraph-model specification of divide-and-conquer architecture

```

    co->add_procs(p); //for dc only
  }
for (element *p = components->get_head();p != NULL;p = p->get_right()){
  entity * ent = p->get_ent();
  devs * comp = (devs *)ent;
  if (!ent->equal(co))
  {
    co->add_coupling(co, "y", comp,"in");
    comp->add_coupling(comp,"out",co,"x");
  }
}
};

```

2 TESTING THE ARCHITECTURES

Testing of digraph models for correctness can be done conveniently using the simulation process. After loading the file containing the multi-server architecture, for example, we can start simulation with the command (*restart r*) where *r* is the root-co-ordinator created and initialized to C:MUL-ARCH. However, since all atomic-model components are in their passive phases, the simulation will immediately terminate. In general, the state of a coupled model is determined by the states of its atomic models. These must be set as desired to put the coupled model in a non-passive state.

There are two ways to start atomic-model components in different states: 1) internally, by entering the desired state variable values when queried by the restart command or 2) externally, by sending external events to the components.

2.1 Internal Initialization

In internal initialization we must know what states we wish to establish. For example, we could set the co-ordinator, MUL-C into the state that it would be in after receiving the first job:

```
STATE S = (0 BUSY BUSY PASSIVE PASSIVE X1 JOB_IN1)
```

This state can be established by interactively responding to the prompt of the restart command for entering state variable values for MUL-C. Alternatively, we can put the following in a file such as test.s:

```
(send mul-c set-s (make-state 'sigma 0
                          'phase 'busy
                          'p1-s 'busy
                          'p2-s 'passive
```

```

        'p3-s 'passive
        'out-port 'x1
        'job-id 'job_in1
    )
)

```

This message will automatically be loaded if we issue the *restart* command with the file as an argument: (*restart r "test.s"*).

2.2 External Initialization

External initialization requires that we send an appropriate external event to the component we wish to initialize. For example, by entering:

```
(send mul-c inject 'in 'job_in1)
```

with MUL-C in its idle state, we send MUL-C into the same state as given above. Here we rely on the fact that we have already tested the component for its response to external inputs, so that we are confident that it will enter the correct state.

External initialization can also be done at any level in a hierarchical model since the method *inject* is also defined for coupled-models. For example,

```
(send mul-arch inject 'in 'job_in1)
```

will produce the same effect as above. The method, *inject*, for coupled-models behaves similarly to a co-ordinator when receiving an external event. It sends the translated content structure to the receivers determined by the external-input coupling. Since the receivers may also be coupled-models, the method is recursive. Figure 6.6 displays the code for the *inject* method for both coupled and atomic models. This illustrates how recursive calling of methods works. Such recursion is often needed in DEVS-Scheme methods due to the hierarchical structure.

Having one or more components set to desired initial states, we can run the simulation in pause mode. This will enable viewing the messages as they are created and routed through the components. If a message does not get created as it should, or go to the right destination, then we can terminate the simulation and view the current state of the model as a start toward analyzing the source of the error.

The *inject* method is also very useful in testing a model for response to external events. Often, we run a simulation until it reaches a particular state. If the time advance for this state is infinity, (the model passivate in this state), then the simulation stops naturally. Otherwise, we can terminate the simulation before the next transition takes place. Now we can inject an external event into the model to test its response in this state both statically, i.e., by examining the new state after an external event, and dynamically, i.e., by restarting the simulation from the new state.

C++:

```

(define-method (coupled-models inject) (port value . elapsed-time)
  ;; the " . " makes the arguments following it
  ;; into a list; a null list is also accepted
  ;; effectively yielding optional arguments
  (let (
    (e (when (number? (car elapsed-time)) (car elapsed-time)))
    (destinations (get-receivers))
  )
    (for-each (lambda(child)
      (let (
        (tr-port (translate this-model child port))
      )
        (when tr-port
          (send child inject tr-port value e)
          ;; note no check of the child's class is needed
        )
      ))
      destinations)
  ))

(define-method (atomic-models inject)(port value . elapsed-time)
  (set-x (make-content 'port port 'value value))
  (when (number? (car elapsed-time)) (set-e (car elapsed-time)))
  (ext-transition)
  )

```

Figure 14: Definition of the inject method for class atomic-models and coupled-models

```
// Testing the multiserver coordinator
// include the multiserver header file

int main(int argc, char ** argv)
{

mulco * mc = new mulco("mc");
proc * pn = new proc("pn",10);
proc * pn1 = new proc("pn1",10);

mc->initialize();
mc->add_procs(pn);
mc->add_procs(pn1);
mc->show_state();
mc ->inject("in",new entity("job1"));
mc->show_output();
mc->show_state();
mc->simulate(1);
mc ->inject("in",new entity("job2"));
mc->show_output();
mc->show_state();
mc->simulate(1);

mc->show_state();

mc->inject("x", new pair(pn1,new entity("result1")),10);
mc->show_output();

mc->simulate(1);
mc->show_state();
mc->inject("x", new pair(pn,new entity("result")),11);

mc->start_sim(argc,argv);

return 0;
}

//testing the divide and conquer architecture
//include the proper header file

int main(int argc, char ** argv)
```

```

{

dc * d = new dc("d",9,3);

d ->initialize();

d ->inject("in",new entity("job1"));
d ->show_output();
d ->simulate(3);
d ->show_state();
d ->inject("in",new entity("job1"),10);
d ->start_sim(argc,argv);

return 0;
}

\\the divide and conquer coupled to the experimental frame

class efdc:public digraph
{
public:
efdc(char * name,timetype int_arr_time,timetype proc_time,
      timetype observ_time,int size)
      :digraph(name)
{
ef * e = new ef("e",int_arr_time,observ_time);
dc * d = new dc("d",proc_time,size);

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

add(e);
add(d);

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

// get_coupling()->print();

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

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

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

```

```
//    d->get_coupling()->print();  
  
}  
};
```