Diagram Structure

Package errors	errors.sun
Package event_types	eventtypes.sun
Package events	events.sun
Package signalinterface	signalinterface1.sun
System kernel	kernel.ssy
Block control_block	control_block.sbk
Process control	control.spr
Procedure checkinit	checkinit.spd
Block event_manager	event_manager.sbk
Process manager	manager.spr
Procedure dispatch	dispatch.spd
Procedure scheduletimed	scheduletimed.spd
Procedure scheduleuntimed	scheduleuntimed.spd
Procedure unschedule	unscheduletimed.spd

Package errors


```
/*
Enumeration type of possible gravities of errors
*/

NEWTYPE gravity
LITERALS
        light,
        fatal
ENDNEWTYPE;
```

Package event_types


```
/*
Enumeration type of possible eventtypes
*/

NEWTYPE eventtype
LITERALS
        initev, /* used in the kernel during initialization */
        simulateev,
        stopev,
        haltev,
        setev,
        getev,
        dumpev
ENDNEWTYPE;
```

```
USE event_types;
```

Package events 1(2)

```
/*
In this package events together with priorities on them are defined.
input priority: indicates how conflicts between different events
        from the interface have to be resolved. Only interesting
        for set-events.
eventpriority, priorityclass: used to sequentialize events scheduled
        for the same time. Only events with the same priority may be
        scheduled in parallel.
*/
```

```
/*
enumeration type of the priorityclasses,
together with a partial and linear order
*/
NEWTYPE priorityclass
LITERALS
        pinit, pstop, pset1, psimulate, pset2, pget1, pdump, pget2, phalt
OPERATORS
        higherclass : priorityclass, priorityclass -> boolean
AXIOMS
for all p1, p2, p3 in priorityclass (
        higherclass(pinit,pstop) == true;
        higherclass(pstop,pset1) == true;
        higherclass(pset1,psimulate) == true;
        higherclass(psimulate,pset2) == true;
        higherclass(pset2,pget1) == true;
        higherclass(pget1,pdump) == true;
        higherclass(pdump,pget2) == true;
        higherclass(pget2,phalt) == true;

        higherclass(p1,p1) == false;
        higherclass(p1,p2) and higherclass(p2,p3) ==> higherclass(p1,p3);
        higherclass(p1,p2) ==> not(higherclass(p2,p1));
)
ENDNEWTYPE;
```

```
/* the priority of events */

NEWTYPE eventpriority
STRUCT
        prioclass priorityclass;
        prio      natural;
ADDING
OPERATORS
        higher : eventpriority, eventpriority -> boolean;
AXIOMS
for all p1, p2 in eventpriority (
        higher(p1,p2) == higherclass(prioclassextract!(p1),
                                     prioclassextract!(p2))
                      or ( (prioclassextract!(p1) = prioclassextract!(p2))
                            and (prioextract!(p1) < prioextract!(p2))
                         );
)
ENDNEWTYPE;
```

```
USE event_types;
```

Package events 2(2)

```
/*
The inputpriorities
*/

SYNTYPE inputprio = natural
ENDSYNTYPE;
```

```
NEWTYPE event
STRUCT
        evtype eventtype;
        obj    pid;
        stamp  time;
        prio   eventpriority;
        frequency duration;
ADDING
OPERATORS
        collidingevents : event,event -> boolean;
        timedevent :      event       -> boolean;
        makehaltevent :   time, pid   -> event;
AXIOMS
for all e1, e2 in event(
for all p in pid(
for all t in time(

         collidingevents(e1,e2) ==
                 evtypeextract!(e1) = evtypeextract!(e2)
                 /* only setevents can collide */
             and evtypeextract!(e1) = setev
             and objextract(e1) = objextract!(e2)
             and stampextract!(e1) = stampextract!(e2)
             and prioextract!(e1) = prioextract!(e2);

         timedevent(e) == stampextract!(e) > 0;

         makehaltevent(t,p) ==
                 make!(haltev,p,t,phalt,0);

)))
ENDNEWTYPE;
```

```
USE errors;
USE event_types;
USE events;
```

Package signalinterface 1(2)

```
/*
static interface between kernel and interface ,
all signals ordered alphabetically */
```

```
SIGNAL ack;     /* schedule resp. unschedule of event was succesful */
                /* if_em */
SIGNAL binding_lost; /* sent when simulation and real-time are out of synchrony */
                /* cb_if */
SIGNAL endinit; /* end of initialization phase */
                /* if_cb */
SIGNAL errorfinished(charstring,gravity,natural); /* triggering an object lead
                   to an error */
                /* if_em */
SIGNAL errormsg(charstring,gravity);  /* errormessages of all kind */
                /* cb_if, em_if */
SIGNAL finished(integer); /* sent when objects are passive again after
                   having been triggered, argument is unique identifier for
                   parallel execution of events */
                /* if_em */
SIGNAL init;    /* start of initialization phase */
                /* if_cb */
SIGNAL initstart(time),initend(time),initslack(duration),
       initfactor(real),initlog(boolean);
SIGNALLIST initparam = initstart,initend,initslack,initfactor,initlog;
                /* if_cb */
SIGNAL logfin(INTEGER,TIME); /* an identifier and the time it has been received
                   by the manager */
                /* em_if */
SIGNAL logtrig(event,integer,time); /* an event which has been sent together
                   with the identifier and the realtime it has been sent */
                /* em_if */
SIGNAL ngets(integer); /* number of values to be received by interface */
                /* em_if */
SIGNAL ok;              /* response to endinit */
                /* cb_if, if_cb */
SIGNAL querytime;       /* request for the actual time */
                /* if_cb */
SIGNAL schedule(event,inputprio);  /* insert an event in the collection,
                   no distinction between timed, untimed and init events */
                /* if_em */
SIGNAL simulationended;          /* sent when simulation has come to an end */
                /* cb_if */
SIGNAL started; /* send when simulation is started */
                /* cb_if */
SIGNAL startsimulation;          /* signal to begin a simulation run */
                /* if_cb */
SIGNAL stopped; /* send when control has received a stop event */
                /* cb_if */
SIGNAL timeprio(time);  /* the actual simulation time */
                /* cb_if */
SIGNAL trigger(event,integer);   /* sends an event to an object to perform it. The
                   integer is an unique identifier to be sent back by a finished
                   signal. */
                /* em_if */
SIGNAL unschedule(event,inputprio); /* remove an event from the collection */
                /* if_em */
```

```
USE errors;
USE event_types;
USE events;
```

Package signalinterface


```
/* signals from the interface to the controlblock */

SIGNALLIST if_cb =
           init,endinit,(initparam),ok,querytime,startsimulation ;

/* signals from the control block to the interface */

SIGNALLIST cb_if =
           timeprio,ok,errormsg,simulationended,stopped,started,binding_lost;

/* signals from objects to event manager,
   the interface is seen as an object too */

SIGNALLIST if_em =
           schedule,unschedule,errorfinished,finished;

/* signals from the event manager to the objects */

SIGNALLIST em_if =
           ack,errormsg,trigger,logfin,logtrig,ngets;
```

```
USE errors;USE event_types;
USE events;USE signalinterface;
```

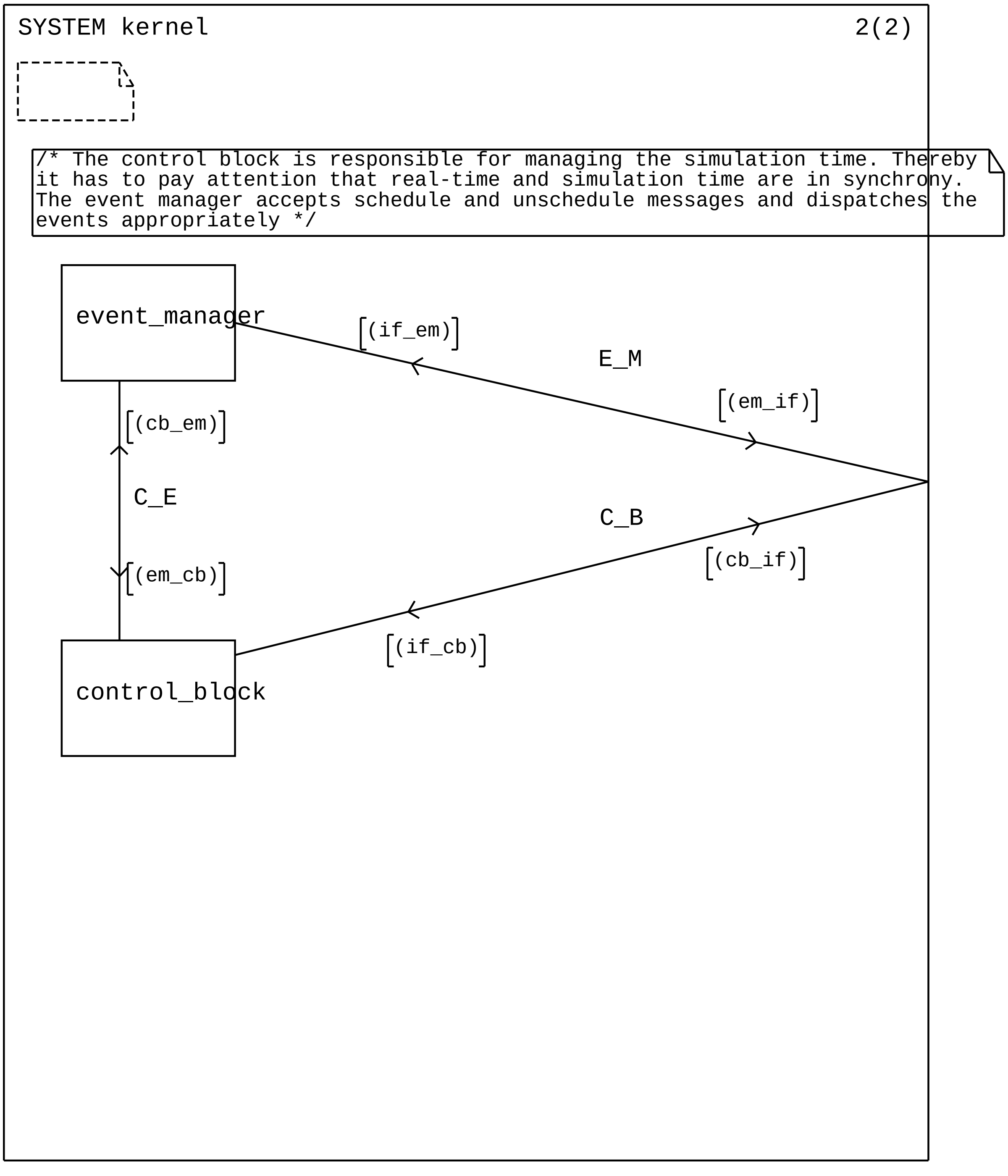
SYSTEM kernel 1(2)

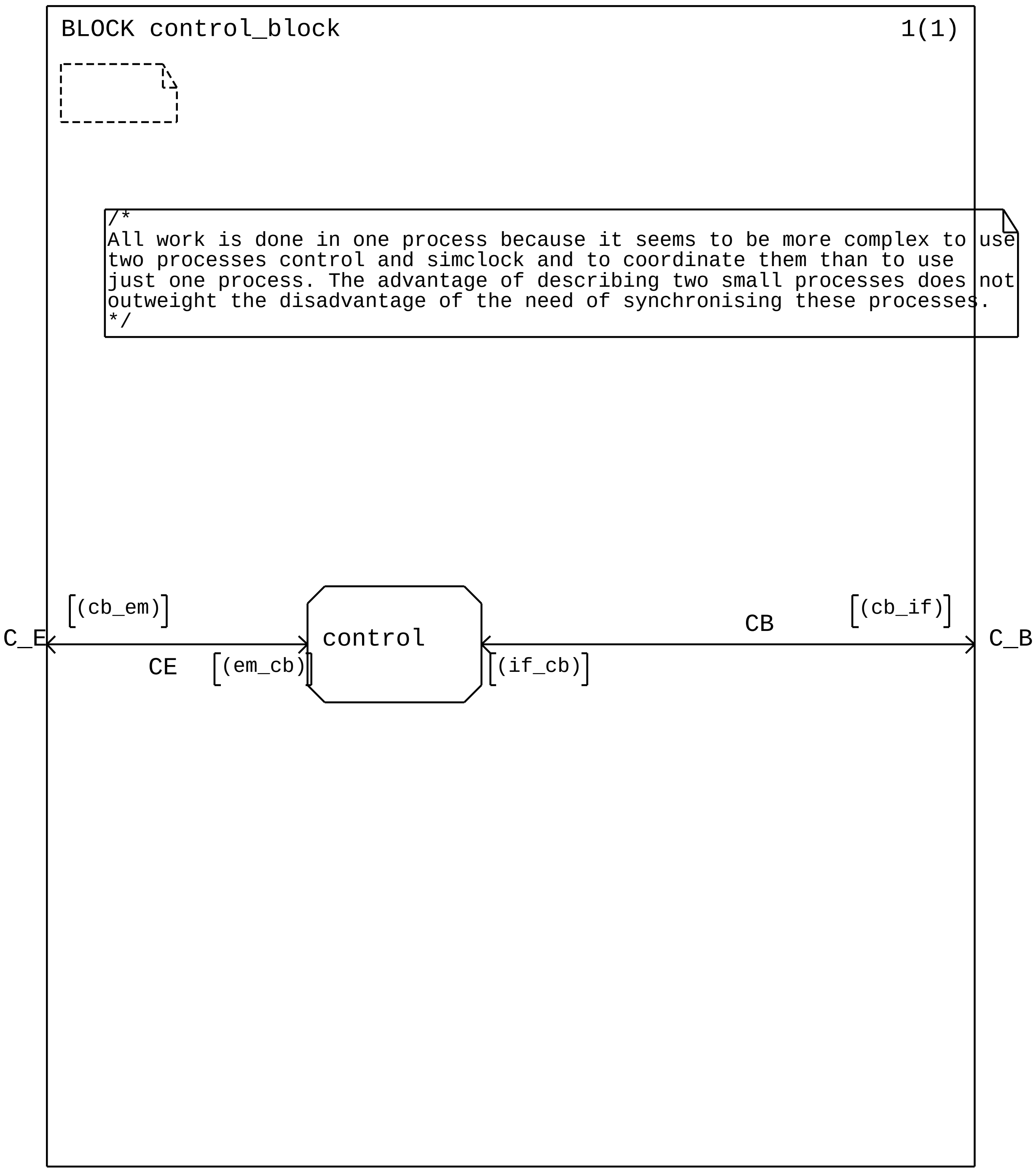
```
/*
Toplevel of the simulator kernel
similar to the diagram of the system structure in the  document
"The simulator kernel"
*/
```

```
/*
additional internal signals between the control block and the event collecion
*/

SIGNAL clear; /* initialize the kernel collection */
        /* cb_em */
SIGNAL dispatch(TIME); /* inform the manager about the actual simulation time */
        /* cb_em */
SIGNAL endtime(time); /* a priori known end of simulation time */
        /* cb _em */
SIGNAL fataldispatch; /* a fatal error has occured while dispatching
           events*/
        /* em_cb */
SIGNAL full; /* the kernel collection is full */
        /* em_cb */
SIGNAL initfinished; /* sent when all initial events have been processed */
        /* em_cb */
SIGNAL initlg(boolean); /* the logging flag */
        /* cb_em */
SIGNAL nexttime(time); /* after dispatching an actual set this is the time of
           the event which has to dispatched next */
        /* em_cb */
SIGNAL querystamp; /* request for a timestamp for an untimed event */
        /* em_cb */
SIGNAL startnow(time); /* from now on the eventmanager may receive schedule and
           unschedule messages, time is the actual simulation time */

/* signals from the control block to the eventmanager */

SIGNALLIST cb_em =
           clear,dispatch,endtime,finished,initlg,startnow;

/* signals from the eventmanager to the controlblock */

SIGNALLIST em_cb =
           fataldispatch,full,initfinished,nexttime,querystamp,trigger;
```

```
/* signals for dumping and retrieving are missing */
/* no dumping and retrieving is done at all */
```

USE errors;USE event_types;
USE events;USE signalinterface;

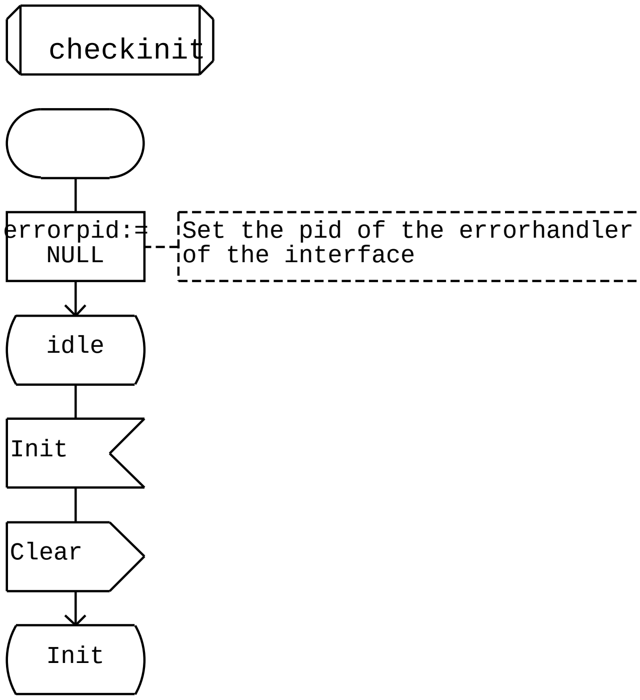


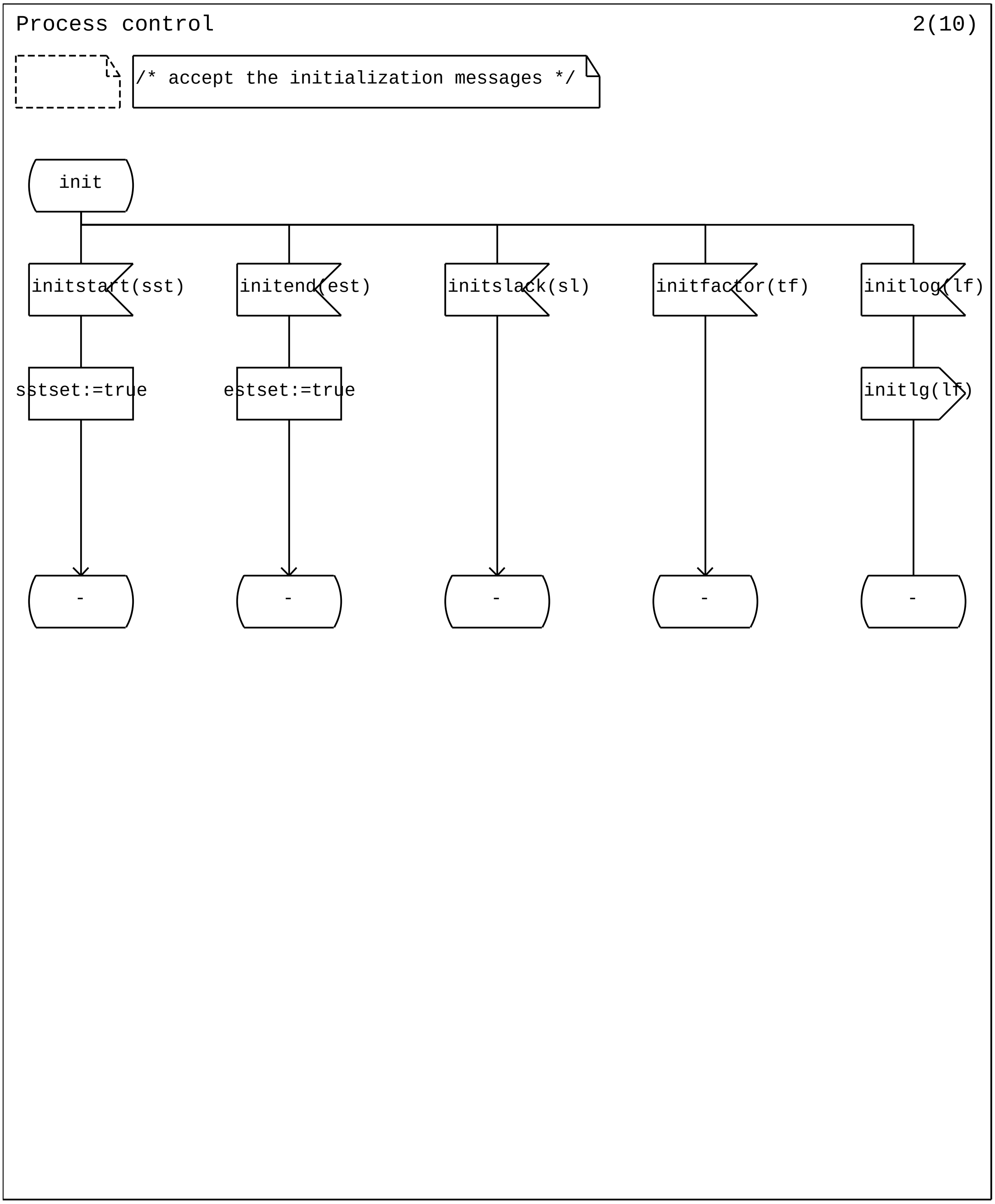


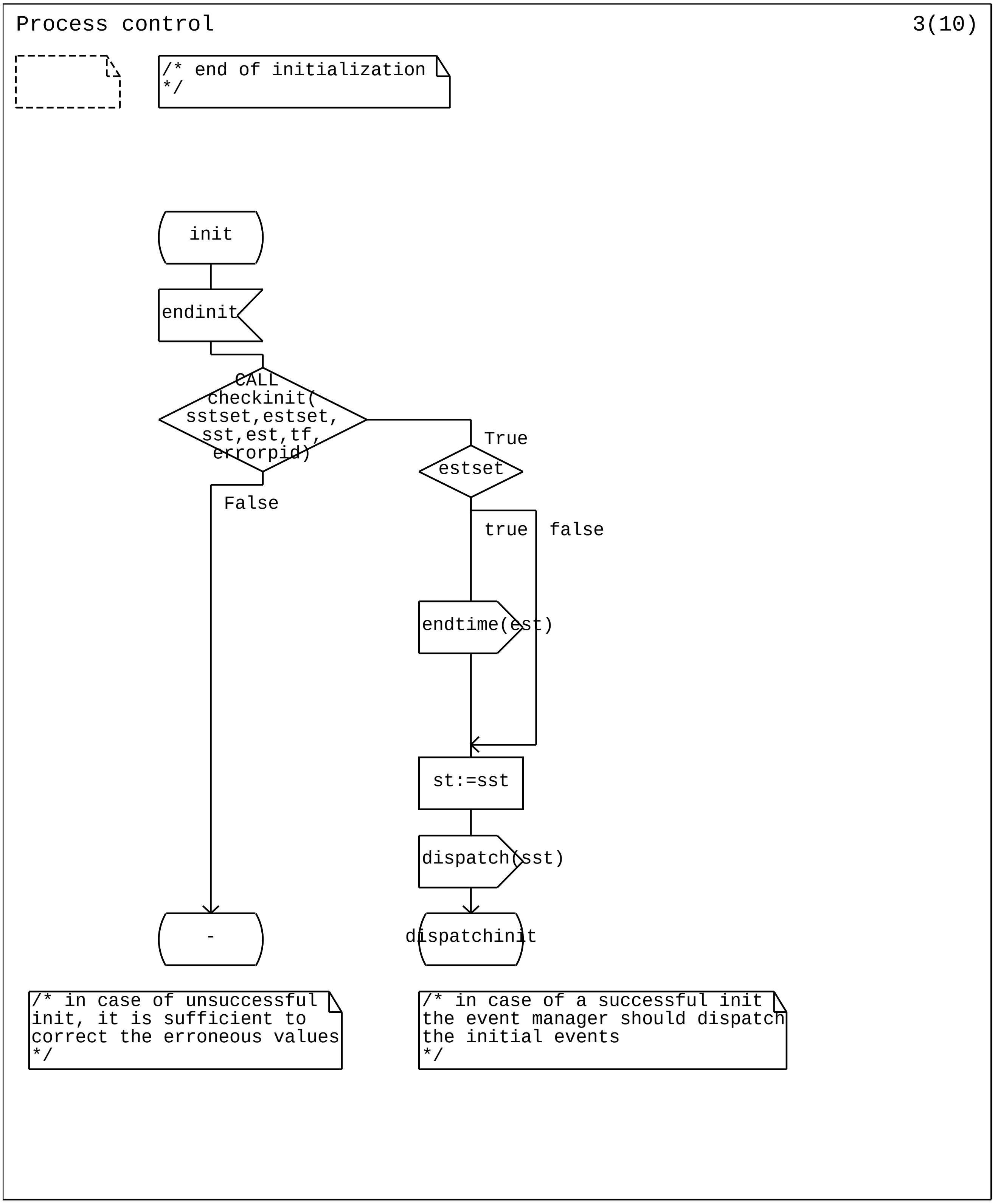
BLOCK control_block
1(1)
/*
All work is done in one process because it seems to be more complex to use
two processes control and simclock and to coordinate them than to use
just one process. The advantage of describing two small processes does not
outweight the disadvantage of the need of synchronising these processes.
*/
(cb_em)
C_E
CE
(em_cb)
control
(if_cb)
CB
(cb_if)
C_B

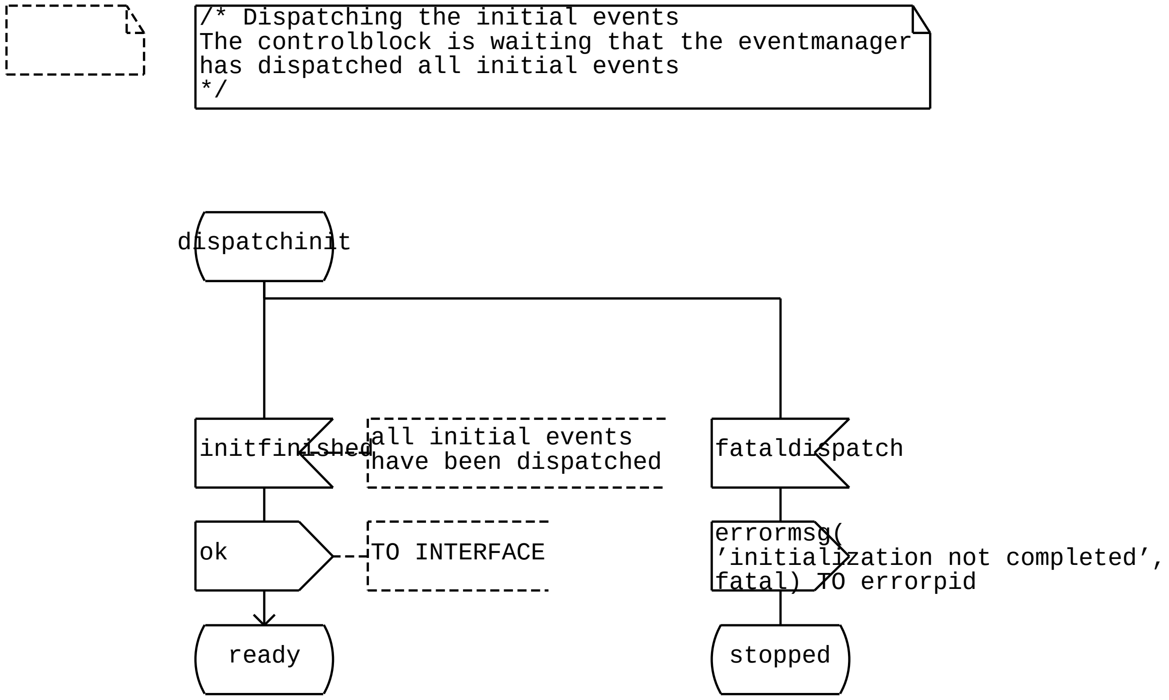
```
/*
This process describes the control part of the simulator kernel including
the simulation clock.
*/
```

```
TIMER wakeup; /* wait for sending the next dispatch */

DCL sst TIME,       /* start simulation time */
    est TIME,       /* end of simulation time */
    st  TIME,       /* actual simulation time */
    rst TIME,       /* start real time */
    wt  TIME,       /* next time to send dispatch */
    tf  REAL,       /* timing factor */
    sl  DURATION := 1,       /* slack */
    lf  BOOLEAN,    /* logging flag */
    sstset BOOLEAN := false, /* flag to indicate whether the starting has been set */
    estset BOOLEAN := false, /* flag to indicate whether the endtime has been set */
    e event,        /* triggering event */
    errorpid Pid,   /* process of the interface */
    i NATURAL;      /* unique identifier of triggering events */
```

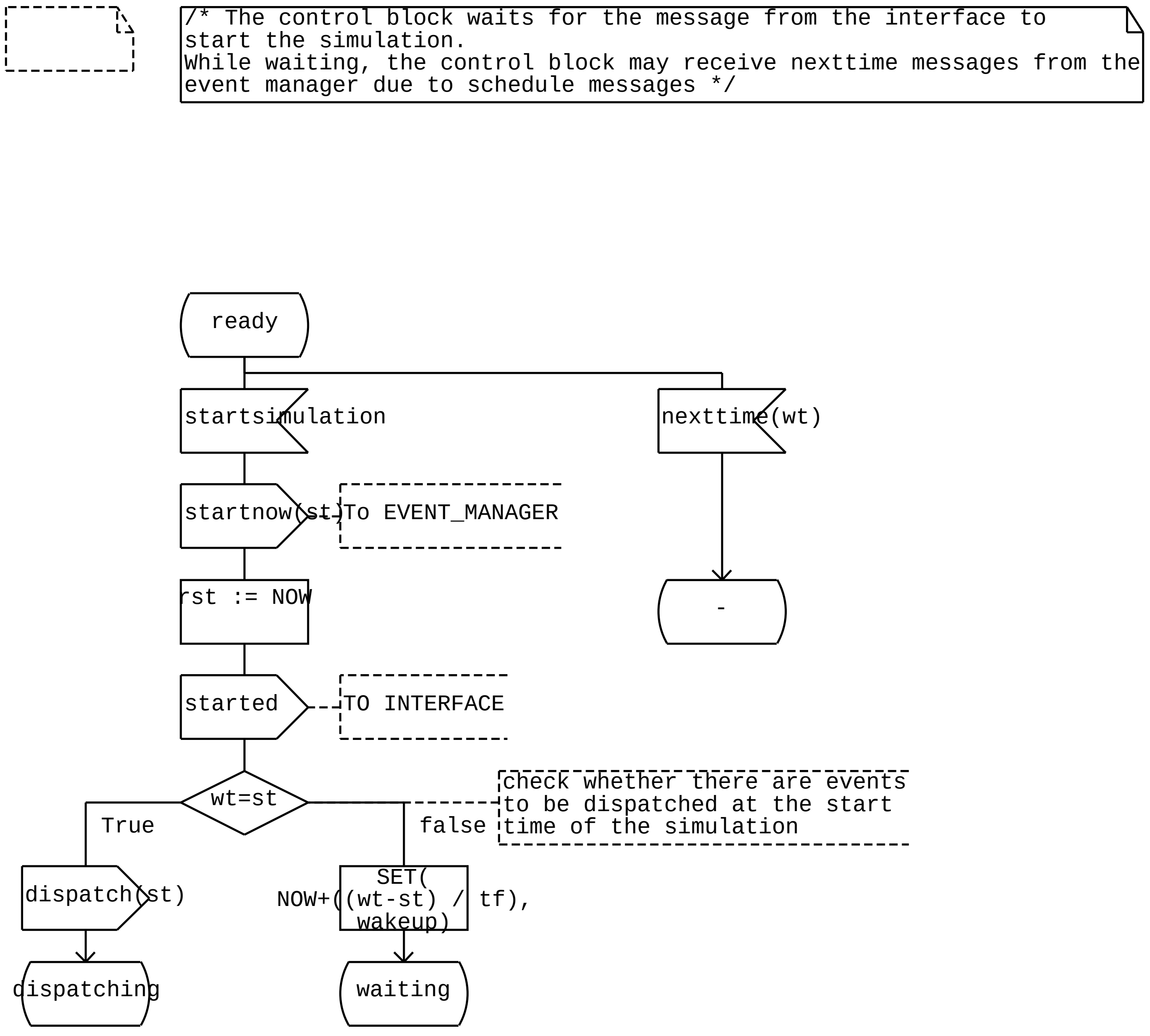



Process control
2(10)
/* accept the initialization messages */
init
initstart(sst)
initend(est)
initslack(sl)
initfactor(tf)
initlog(lf)
sstset:=true
estset:=true
initlg(lf)
-
-
-
-
-

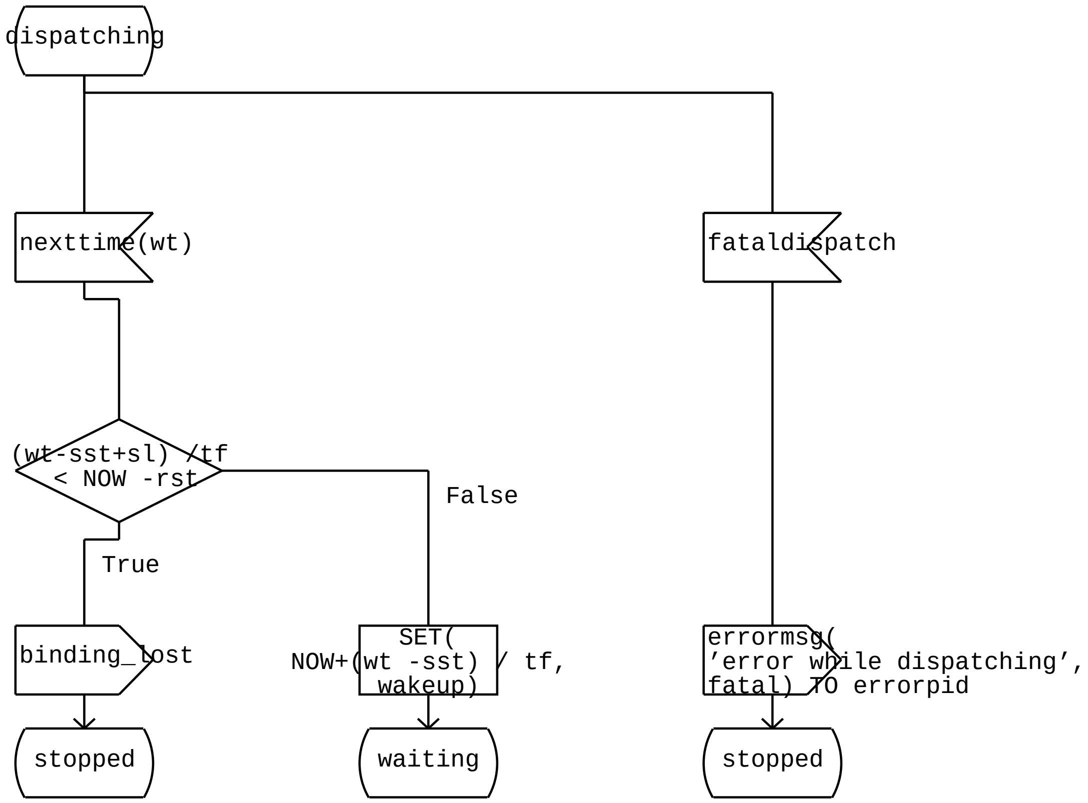

Process control
3(10)
/* end of initialization
*/
init
endinit
CALL
checkinit(
sstset,estset,
sst,est,tf,
errorpid)
True
estset
False
true
false
endtime(est)
st:=sst
dispatch(sst)
-
dispatchinit
/* in case of unsuccessful
init, it is sufficient to
correct the erroneous values
*/
/* in case of a successful init
the event manager should dispatch
the initial events
*/

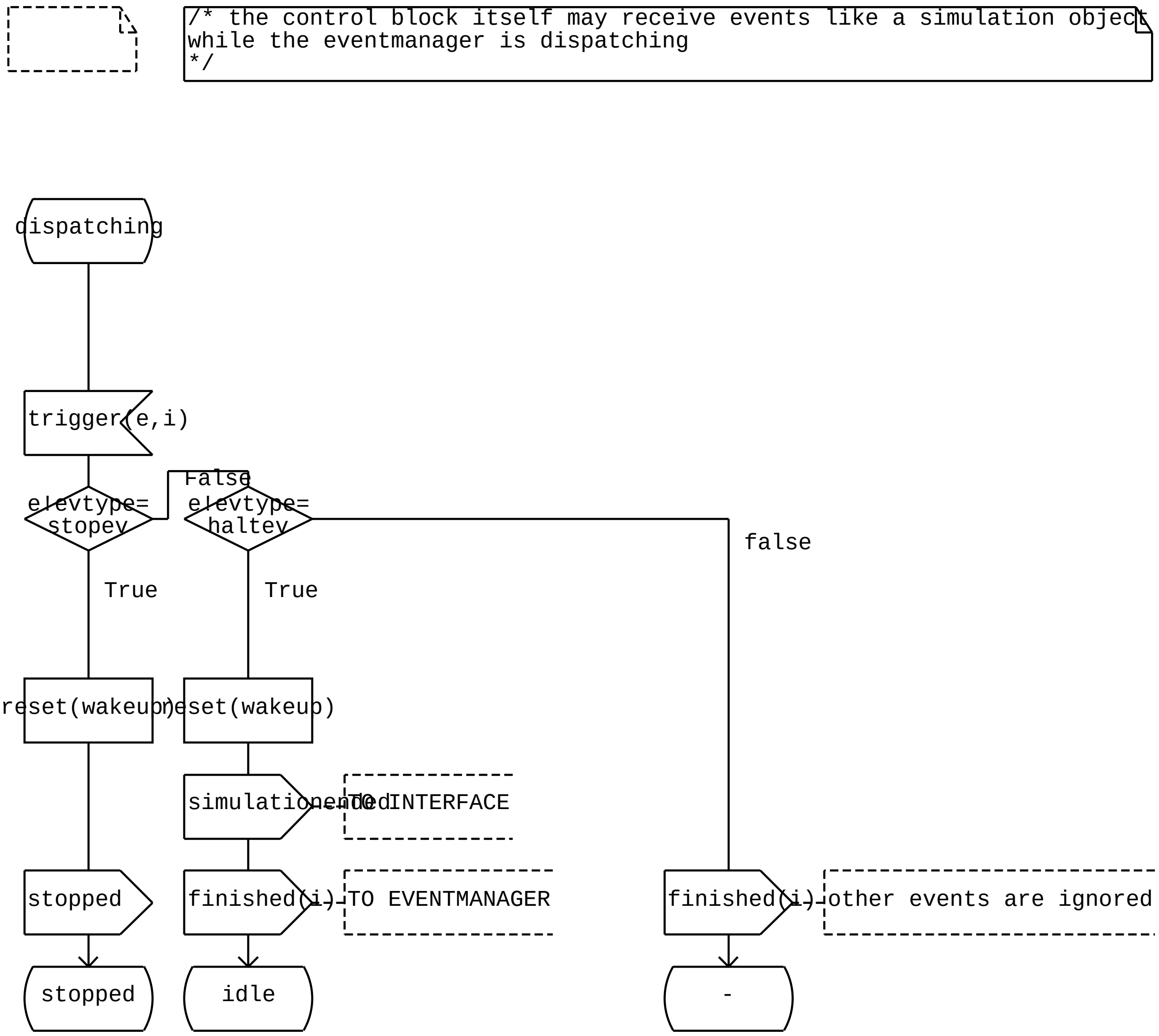

/* Dispatching the initial events
The controlblock is waiting that the eventmanager
has dispatched all initial events
*/
dispatchinit
initfinished
all initial events
have been dispatched
ok
TO INTERFACE
ready
fataldispatch
errormsg(
'initialization not completed',
fatal) TO errorpid
stopped

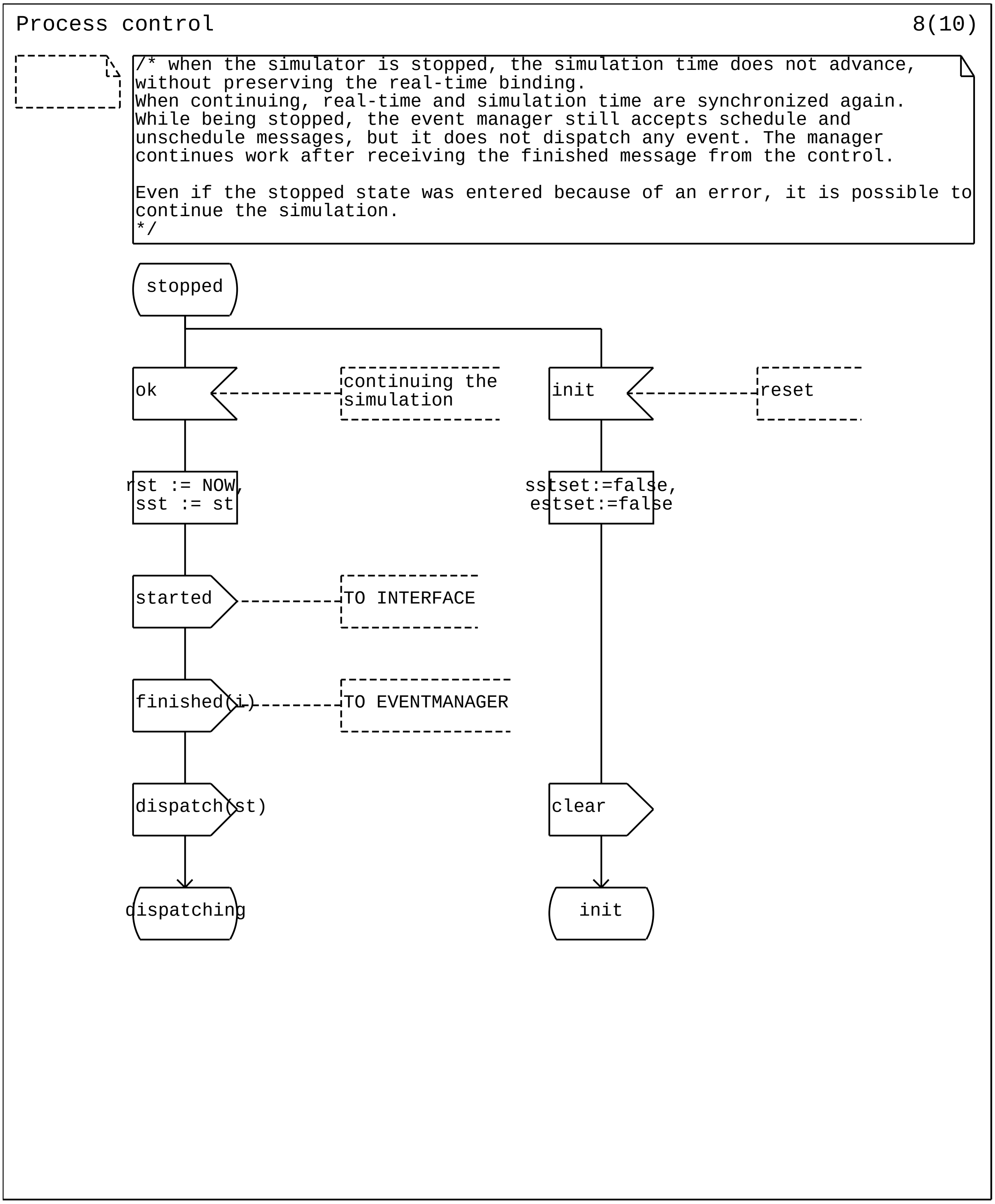
Process control



/* Now the event manager is dispatching events.
If it has finished dispatching all events with the same timestamp,
it sends a nexttime message to the controlblock, which checks whether
the real-time binding is lost or not
*/

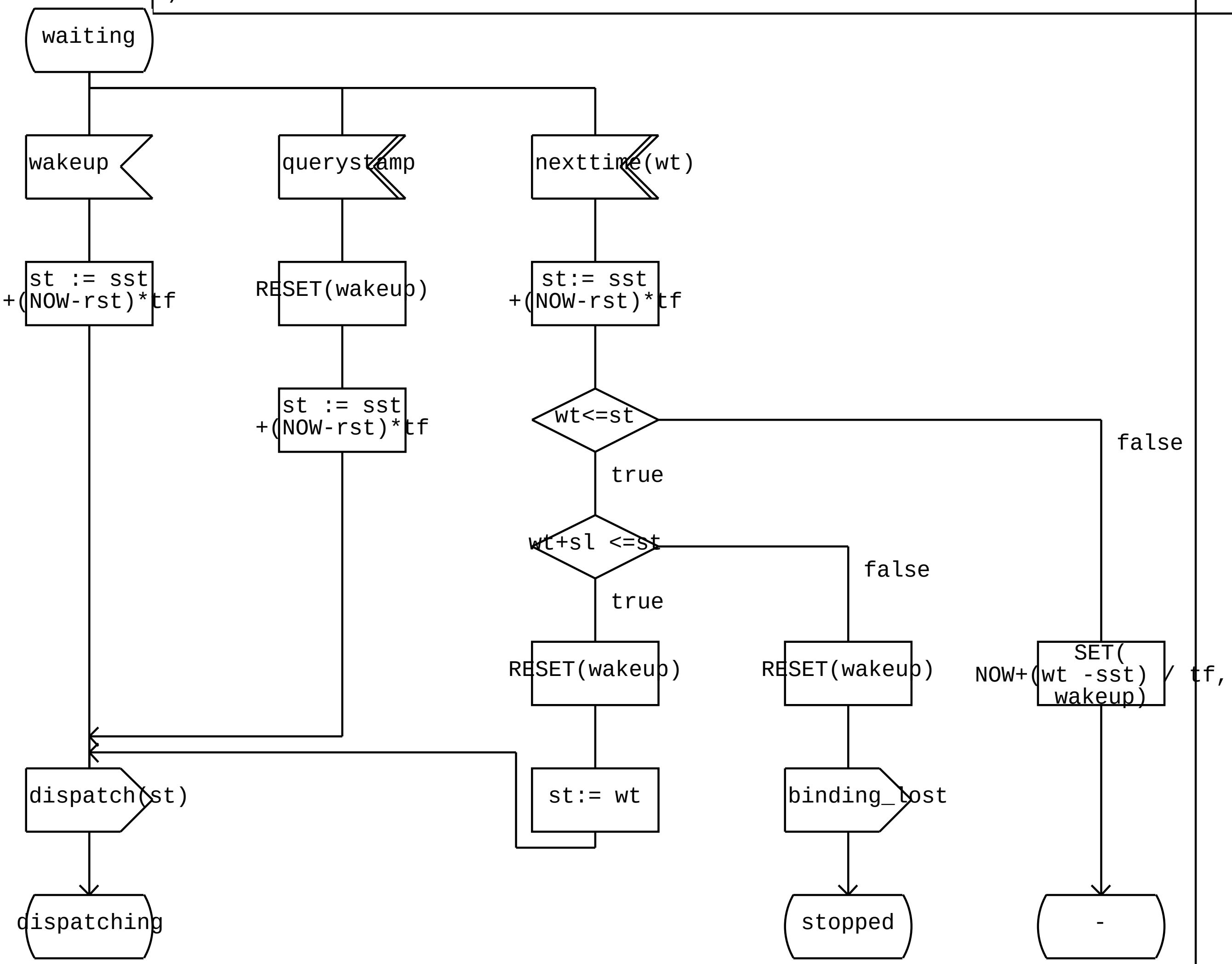


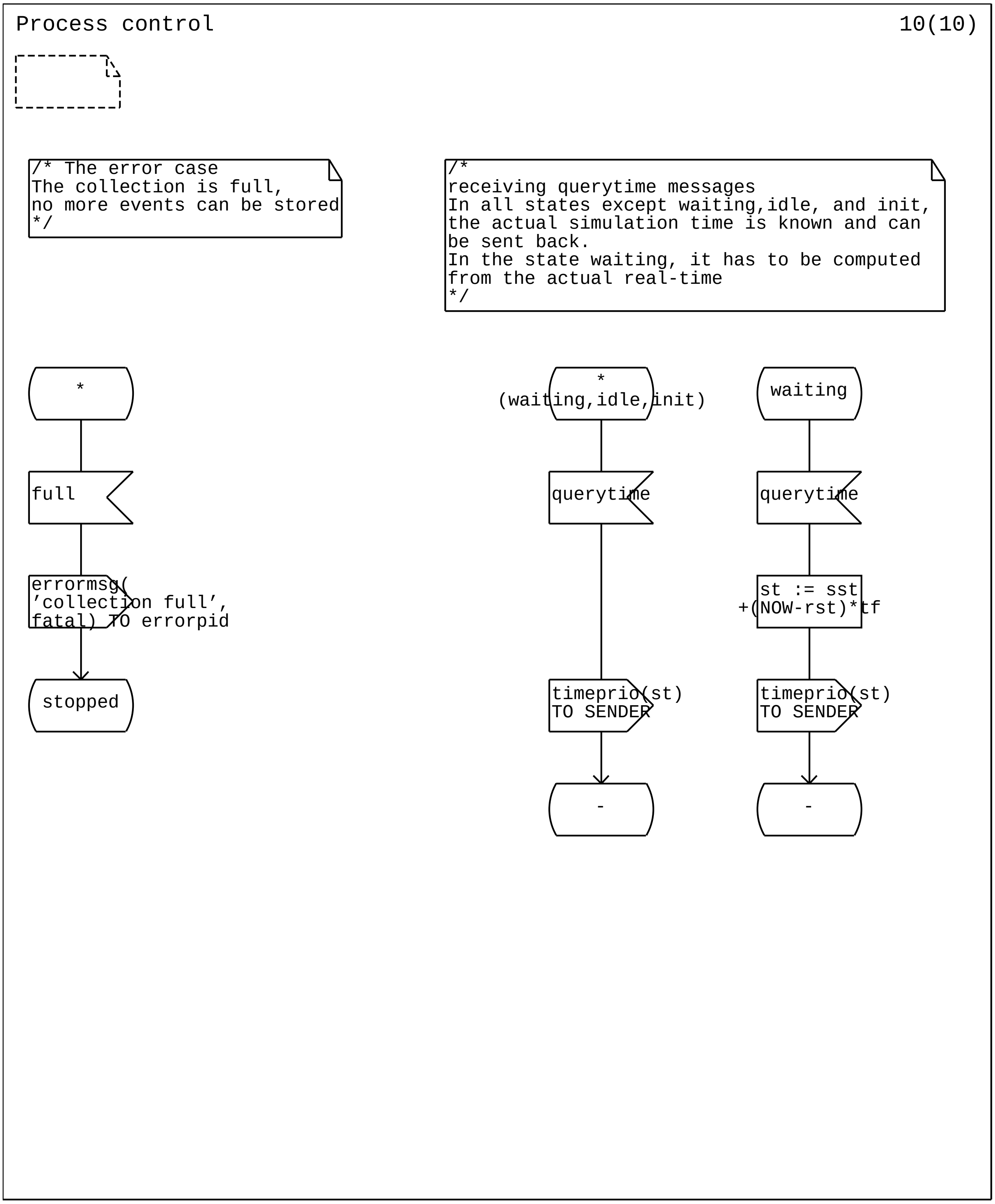

/* the control block itself may receive events like a simulation object
while the eventmanager is dispatching
*/
dispatching
trigger(e,i)
e!evtype=
stopev
False
e!evtype=
haltev
false
True
True
reset(wakeup)
reset(wakeup)
simulationended
TO INTERFACE
stopped
finished(i)
TO EVENTMANAGER
finished(i)
other events are ignored
stopped
idle
-



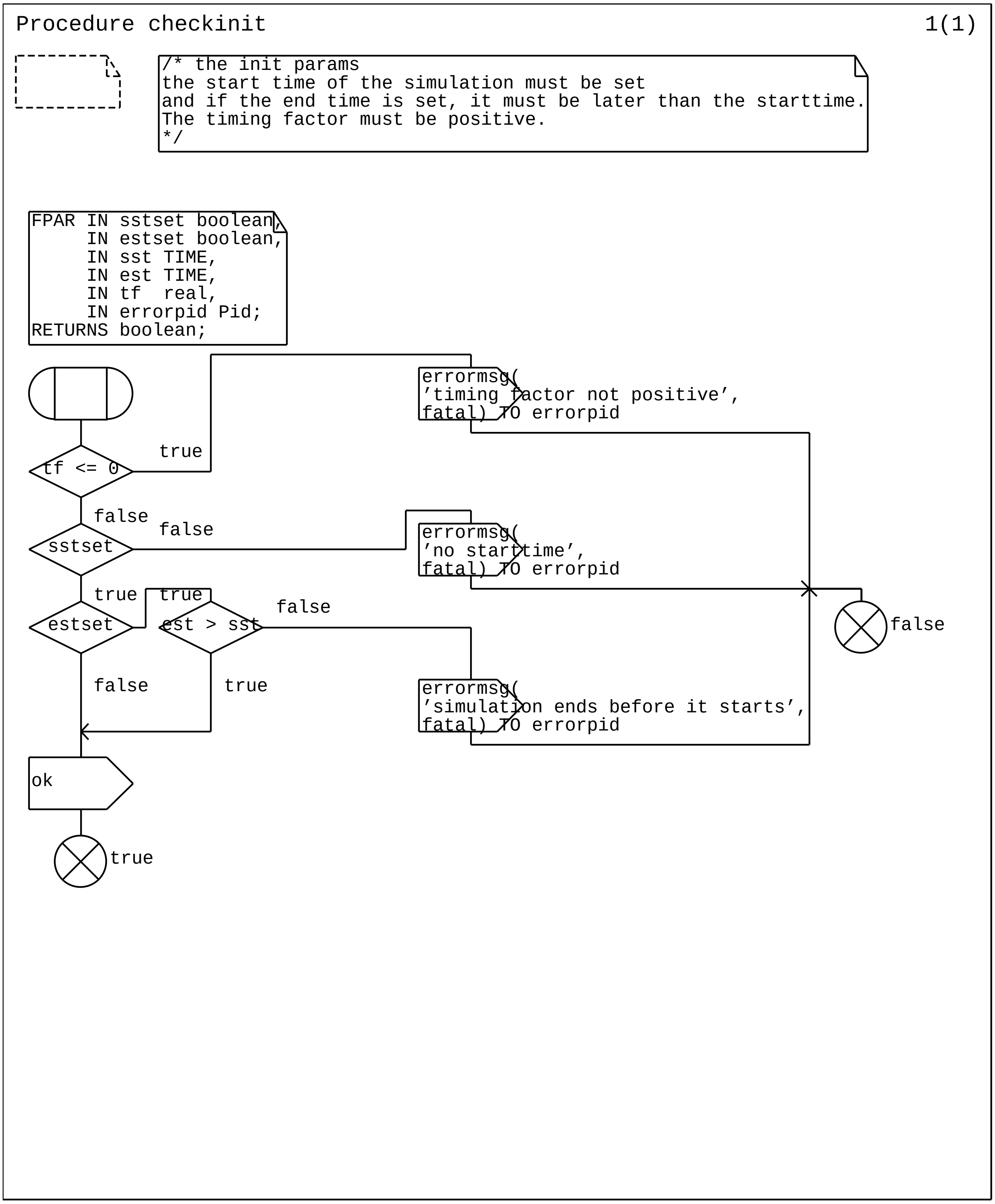
Process control
8(10)
/* when the simulator is stopped, the simulation time does not advance,
without preserving the real-time binding.
When continuing, real-time and simulation time are synchronized again.
While being stopped, the event manager still accepts schedule and
unschedule messages, but it does not dispatch any event. The manager
continues work after receiving the finished message from the control.
Even if the stopped state was entered because of an error, it is possible to
continue the simulation.
*/
stopped
ok
continuing the simulation
init
reset
rst := NOW,
sst := st
sstset:=false,
estset:=false
started
TO INTERFACE
finished(i)
TO EVENTMANAGER
dispatch(st)
clear
dispatching
init

```
/* The kernel is waiting for the time, when the next events
have to be dispatched, to arrive,
The time to dispatch new events may arrive in two ways:
- Some amount of real-time has passed without the manager
  receiving untimed events. Then the timer wakeup sends a signal.
- A new untimed event is accepted by the manager
In both cases simulation time is computed from real-time and
appropriate dispatch messages are sent to  the manager.
If another nexttime message arrives, It has to be checked whether
this new time is still within the real-time binding.
If it is too late, an error message is sent,
If it is only too late by the slack, then immediately the
dispatch message is sent,
otherwise the timer is reset with the new value.
nexttime must have priority over wakeup, because otherwise events which are too late
would be dispatched, before recognizing, that they are too late.
*/
```

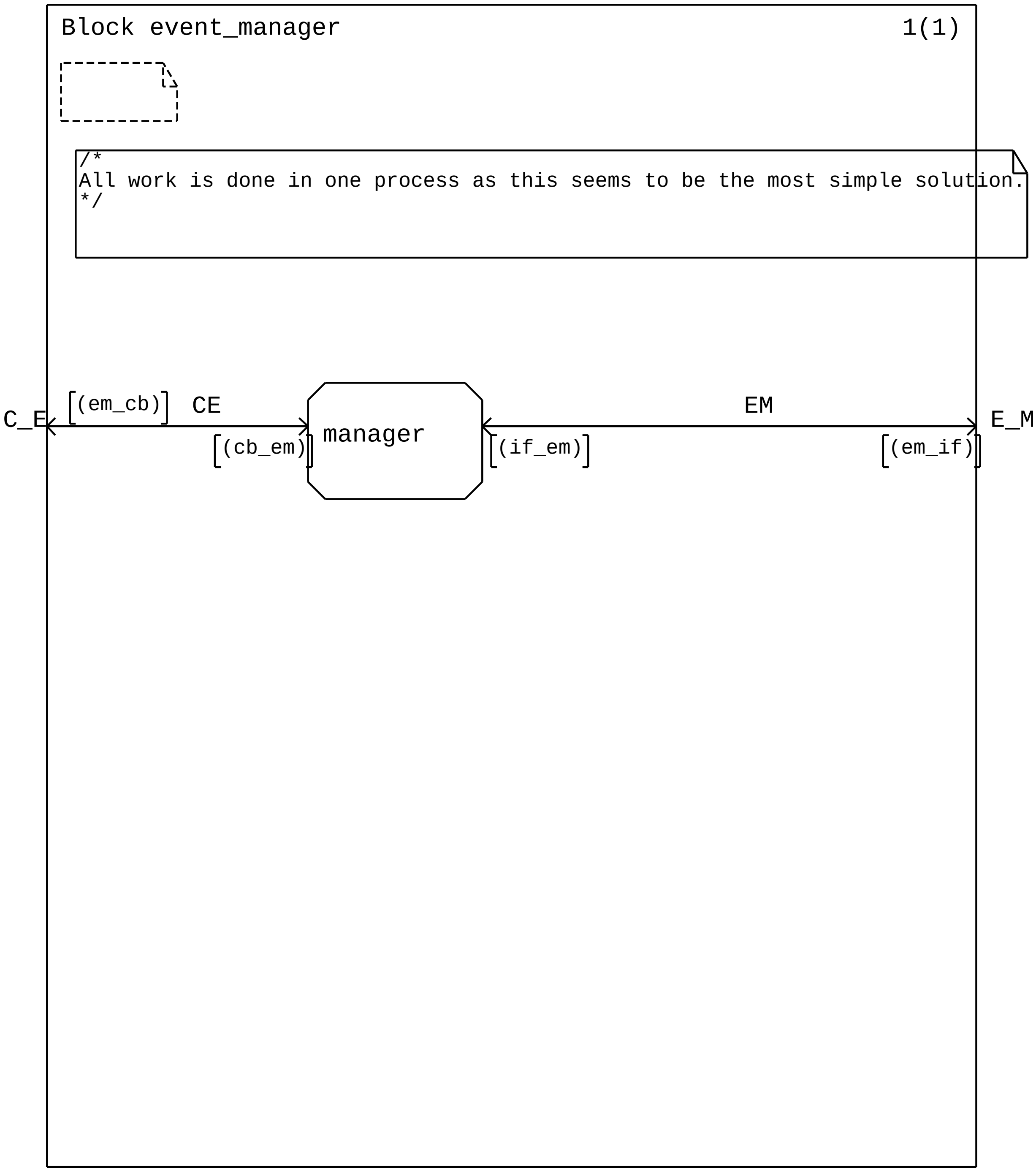




Process control
10(10)
/* The error case
The collection is full,
no more events can be stored
*/
/*
receiving querytime messages
In all states except waiting,idle, and init,
the actual simulation time is known and can
be sent back.
In the state waiting, it has to be computed
from the actual real-time
*/
*
full
errormsg(
'collection full',
fatal) TO errorpid
stopped
*
(waiting,idle,init)
querytime
timeprio(st)
TO SENDER
-
waiting
querytime
st := sst
+(NOW-rst)*tf
timeprio(st)
TO SENDER
-



Procedure checkinit
1(1)
/* the init params
the start time of the simulation must be set
and if the end time is set, it must be later than the starttime.
The timing factor must be positive.
*/
FPAR IN sstset boolean,
IN estset boolean,
IN sst TIME,
IN est TIME,
IN tf real,
IN errorpid Pid;
RETURNS boolean;
tf <= 0
true
errormsg(
'timing factor not positive',
fatal) TO errorpid
false
sstset
false
errormsg(
'no starttime',
fatal) TO errorpid
true
estset
true
est > sst
false
false
errormsg(
'simulation ends before it starts',
fatal) TO errorpid
false
true
ok
true


Block event_manager
1(1)
/*
All work is done in one process as this seems to be the most simple solution.
*/
C_E
(em_cb)
CE
(cb_em)
manager
(if_em)
EM
(em_if)
E_M

```
/* The event manager is written as one process, with the following tasks:
- managing the event collection
- accepting new events
- dispatching actual events*/
```

```
/*
 an internalevent is a struct consistig of an event and its input priority
*/

NEWTYPE internalevent
STRUCT
        ev event;
        iprio inputprio;
ADDING
OPERATORS
        makeie : event,inputprio -> internalevent;
AXIOMS
for all e in event(
for all p in inputprio(

evextract!(makeie(e,p)) == e;
iprioextract!(makeie(e,p)) == p;
))
ENDNEWTYPE;
```

```
/*
Set of identifiers to distinguish events dispatched in parallel
*/
NEWTYPE idset
POWERSET(natural)
ENDNEWTYPE;
```

```
/*
Set of events (without inputpriority)
*/
NEWTYPE actualset
POWERSET(event)
ADDING
OPERATORS
        /* priorityclass of one event in the (nonemtpy) set */
        prclass : actualset -> priorityclass;
        /* one event of a nonempty set */
        someact : actualset -> event;
        /* size of the set */
        size : actualset -> natural;
AXIOMS
for all e in event(
for all a in actualset(

        prclass(incl(e,a)) == prioclassextract!(prioextract!(e));

        someact(incl(e,a)) == e;

        size(empty) == 0;
        size(incl(e,a)) == 1 + size(a);
))
ENDNEWTYPE;
```

```
/*
a struct consisting of untimed events together with the sender of the corresponding
schedule message
*/
NEWTYPE ut
STRUCT
        ie internalevent;
        snd PID;
ADDING
OPERATORS
        makeut : internalevent,pid -> ut;
AXIOMS
for all ie in internalevent(
for all p in pid(

ieextract!(makeut(ie,p)) == ie;
sndextract!(makeut(ie,p)) == p;
))
ENDNEWTYPE;
```

```
/*
Sets of untimed events
*/
NEWTYPE utset
POWERSET(ut);
ADDING
OPERATORS
        someut : utset -> ut;
AXIOMS
for all uts in utset(
for all u in ut(

        someut(incl(u,uts)) == u;
))
ENDNEWTYPE;
```

```
/* Up to now an eventcollection is an infinite datatype, in an implementation this
must be a finite one, generating a signal full to the controlblock if it is not possible
to insert further events.
*/
```

```
/* the central datatype, a set of internal events */
NEWTYPE eventcollection
POWERSET(internalevent)
ADDING
OPERATORS
            /* is an event colliding with events in the collection */
        colliding : internalevent, eventcollection -> boolean;
            /* insert a noncolliding event */
        ins       : internalevent, eventcollection -> eventcollection;
            /* give the earliest timestamp in the collection */
        first     : eventcollection -> time;
        first1    : time, eventcollection -> time;
            /* give all events with the earliest time stamp */
        actual    : eventcollection -> actualset;
        actual1   : eventcollection -> eventcollection;
        actual2   : eventcollection -> actualset;
            /* gives the eventcollection without the events with the earliest timestamp */
        notactual : eventcollection -> eventcollection;
            /* reinserts periodic actual events in the collection */
        periodic : eventcollection,eventcollection -> eventcollection;
AXIOMS
for all c,c1 in eventcollection(
for all e,e1 in internalevent(
for all t in time(

colliding(e,empty) == false;
colliding(e,incl(e1,c)) ==
        if collidingevents(evextract!(e1),evextract!(e))
           and iprioextract!(e1) > iprioextract!(e2)
        then true
        else colliding(e,c)
        fi;

ins(e,empty) == incl(e,empty);
ins(e,incl(e1,c)) ==
        if collidingevents(evextract!(e1),evextract!(e))
        then incl(e,c)
        else incl(e1,ins(e,c))
        fi;

first(incl(e,c)) == first1(stampextract!(evextract!(e)),c);
first1(t,empty) == t;
first1(t,incl(e,c)) ==
        if t < stampextract!(evextract!(e))
        then first1(t,c)
        else first1(stampextract!(evextract!(e)),c)
        fi;
```

```
actual(empty) == empty;
actual(incl(e,c)) == actual2(actual1(incl(e,c),empty));

actual1(empty,e1) == e1;
actual1(incl(e,c),empty) == actual1(c,incl(e,empty));
actual1(incl(e1,c1),incl(e,c)) ==
        if stampextract!(evextract!(e1)) < stampextract!(evextract!(e))
        then /* smaller timestamp */
             actual1(c1,incl(e1,empty))
        else if stampextract!(evextract!(e1)) = stampextract!(evextract!(e))
             then /* same timestamp */
                  if higher(priorityextract!(evextract!(e1)),
                            priorityextract!(evextract!(e)))
                  then /* new higher priority */
                       actual1(c1,incl(e1,empty))
                  else if higher(priorityextract!(evextract!(e)),
                                 priorityextract!(evextract!(e1)))
                       then /* lower priority */
                            actual1(c1,incl(e,c))
                       else actual(c1,incl(e1,incl(e,c)))
                       fi
                  fi
             else /* later timestamp */
                  actual(c1,incl(e,c))
             fi
        fi;

actual2(empty) == empty;
actual2(incl(e,c)) == incl(evextract!(e),c);

notactual(c) == periodic(c,actual1(c,empty));

periodic(c,empty) == c;
periodic(empty,c) == c;
periodic(c,incl(e1,c1)) ==
        if frequencyextract!(evextract!(e1)) = 0
        then /* nonperiodic event */
             periodic(del(e1,c),c1)
        else /* periodic event*/
             if colliding(stampmodify!(evextract!(e1),
                                       stampextract!(evextract!(e1)) +
                                       frequencyextract!(evextract!(e1))),
                          c)
             then periodic(del(e1,c),c1)
             else periodic(ins(stampmodify!(evextract!(e1),
                                            stampextract!(evextract!(e1)) +
                                            frequencyextract!(evextract!(e1))),
                               del(e1,c)),
                           c1)
             fi
         fi;
)))
ENDNEWTYPE;
```

```
/*
This process manages the eventcollection and dispatches the actual events.
It works in two phases: initialization and normal
During the initialization phase it is assumed that only the interface sends schedule
signals to the kernel. The events sent are assumed to be untimed ones of the eventtype
set. When receiving the signal stamp_and_dispatch from the controlblock the start
simulation time is known to the manager and it starts to dispatch the initial events.
At the same time the manager behaves as in the normal phase, accepting all possible
events. The normal phase is entered, when the signal startnow is received.

The manager is initialized by the signal clear, and the parameters end of simulation
time and logging flag can be changed in every state, but not while dispatching events.

The manager maintains an eventcollection, a set of timed events together with their
input priority and a set of untimed events, also together with their input priority.
When it is time to dispatch events, both sets are merged and the actual sets (those
with the earliest time and priority) are dispatched.
*/
```

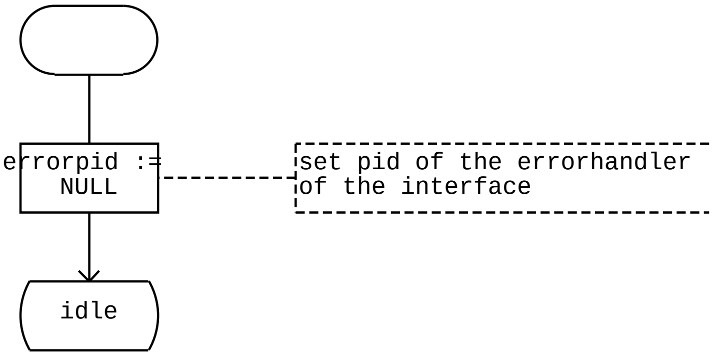
```
DCL e event,      /* event to insert or remove from collection */
    ip inputprio, /* inputpriority of such an event */
    pr priorityclass,    /* the eventpriority of actual dispatched events */
    lf BOOLEAN := false, /* flag whether to log the dispatched events */
    est TIME,     /* endtime of simulation */
    estold TIME,  /* copy of est */
    estset BOOLEAN := false, /* flag to indicate whether estset is set */
    c eventcollection,   /* the eventcollection */
    uts utset,    /* a set of untimed events */
    errorpid Pid, /* Pid of the errorhandler */
    st TIME;      /* simulation time */
```

scheduletimed

scheduleuntimed

unschedule

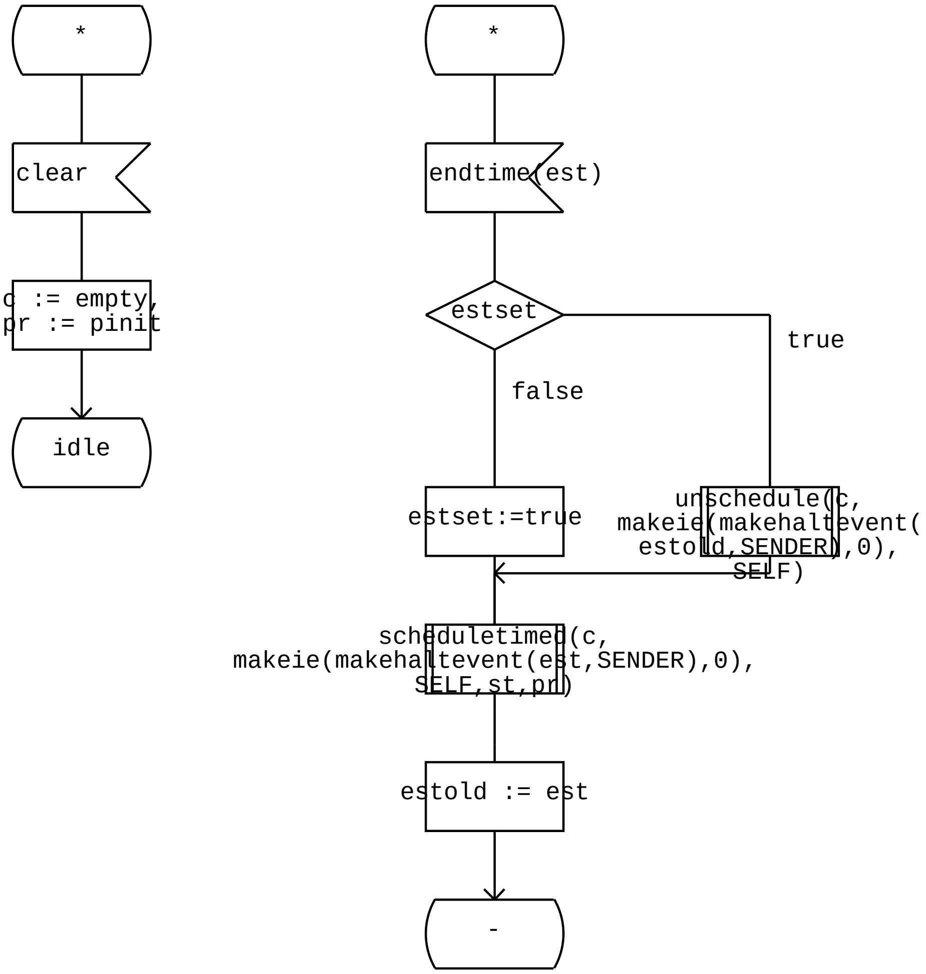
dispatch

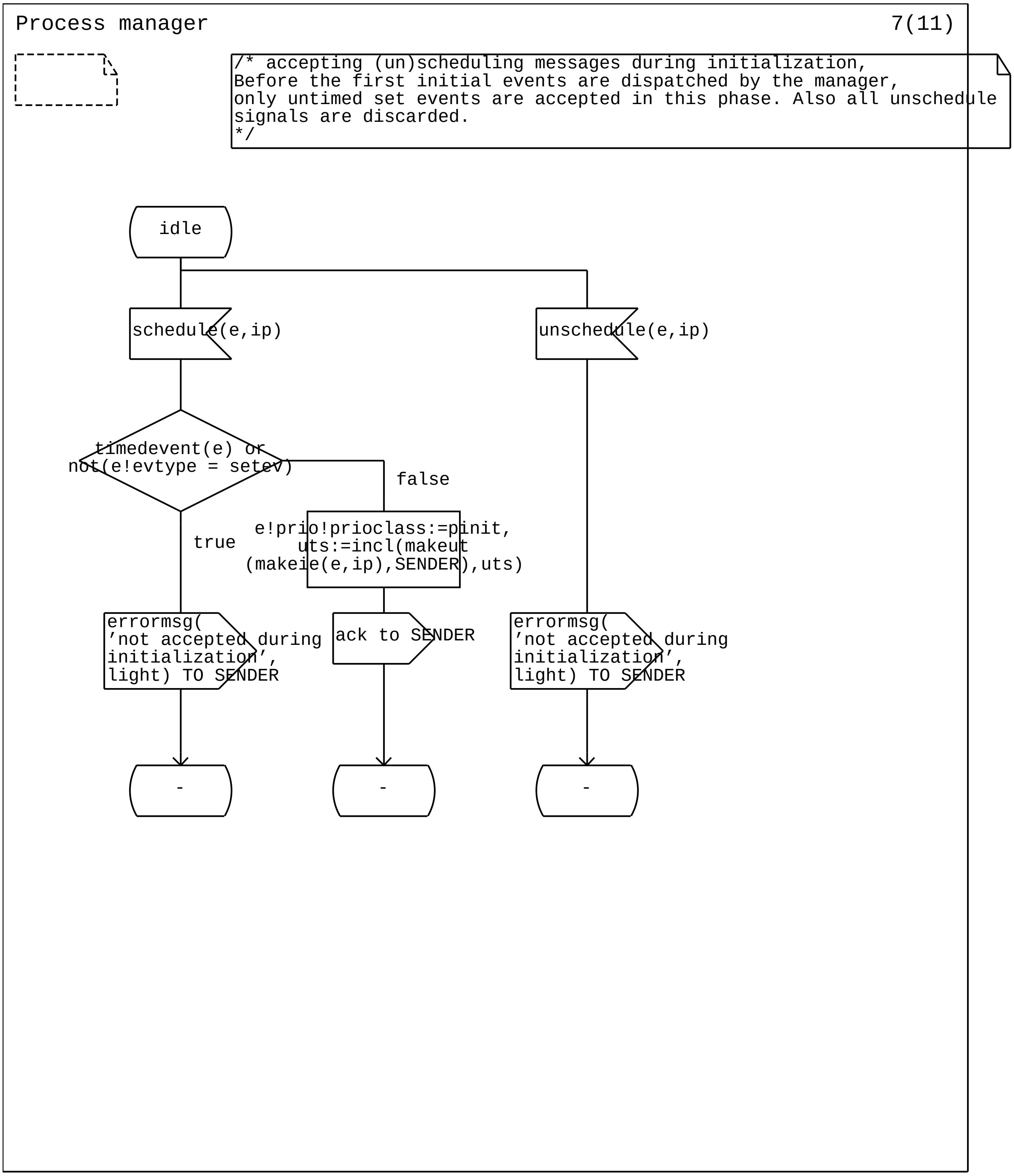


Process manager 6(11)

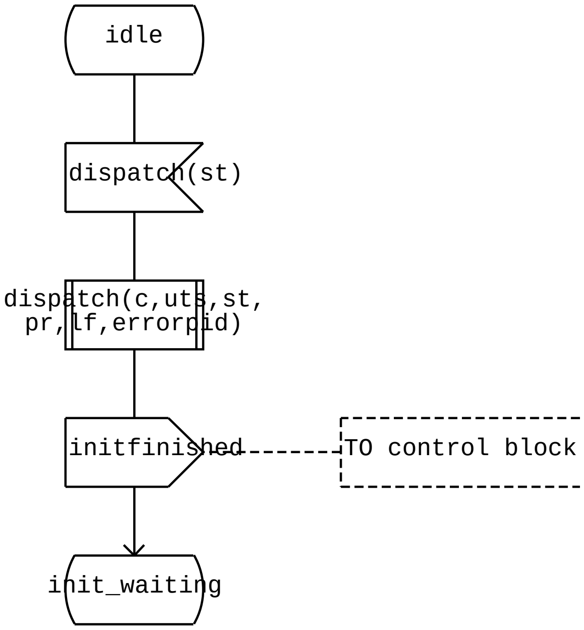
/* accepting initialization messages
It is possible to accept initialization messages in each step.(except the procdure dispatch).

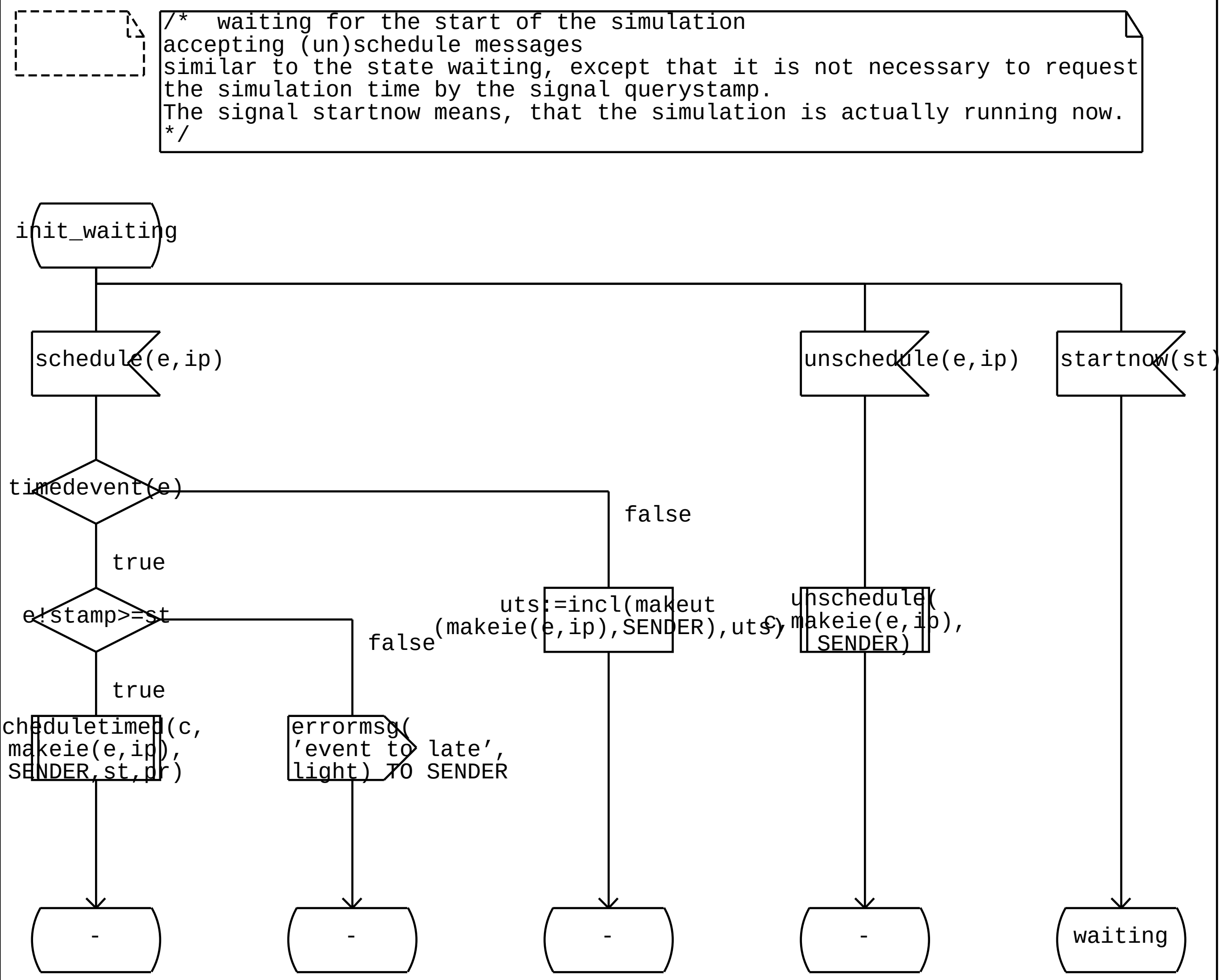
The signal clear means to initialize the kernel and thereby to abort the current simulation.
a new enttime removes an old halt event in the collection and inserts a new one.
a initlg signal just changes the logging flag.
In the last two cases computation is continued in the previous state.
*/




Process manager
7(11)
/* accepting (un)scheduling messages during initialization,
Before the first initial events are dispatched by the manager,
only untimed set events are accepted in this phase. Also all unschedule
signals are discarded.
*/
idle
schedule(e,ip)
unschedule(e,ip)
timedevent(e) or
not(e!evtype = setev)
false
true
e!prio!prioclass:=pinit,
uts:=incl(makeut
(makeie(e,ip),SENDER),uts)
errormsg(
'not accepted during
initialization',
light) TO SENDER
ack to SENDER
errormsg(
'not accepted during
initialization',
light) TO SENDER
-
-
-

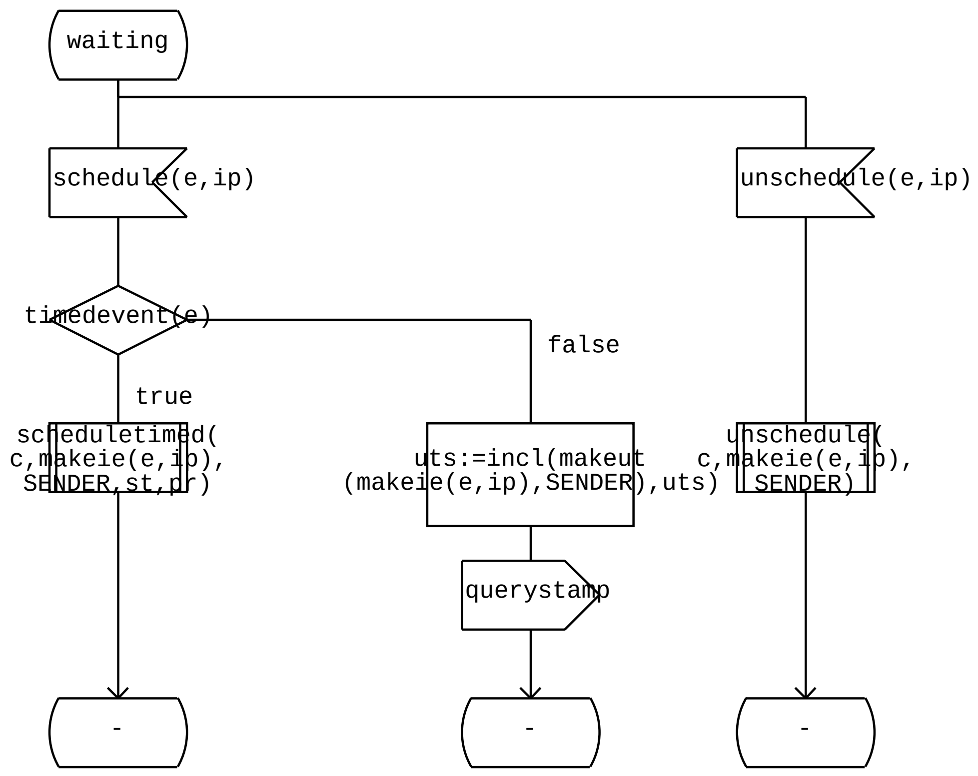
```
/* dispatching initial events.
The initial events are dispatched when receiving a stamp_and_dispatch message.
It is not possible to start the simulation before receiving this signal!

As soon as this signal has been received, all schedule and unschedule messages
are accepted (as if the simulation would be running)

Then the init events are dispatched and when this is done an
initfinished message is sent to the control block.
*/
```




/* waiting for the start of the simulation
accepting (un)schedule messages
similar to the state waiting, except that it is not necessary to request
the simulation time by the signal querystamp.
The signal startnow means, that the simulation is actually running now.
*/
init_waiting
schedule(e,ip)
unschedule(e,ip)
startnow(st)
timedevent(e)
false
true
e!stamp>=st
false
true
uts:=incl(makeut
(makeie(e,ip),SENDER),uts)
unschedule(
c,makeie(e,ip),
SENDER)
scheduletimed(c,
makeie(e,ip),
SENDER,st,pr)
errormsg(
'event to late',
light) TO SENDER
-
-
-
-
waiting

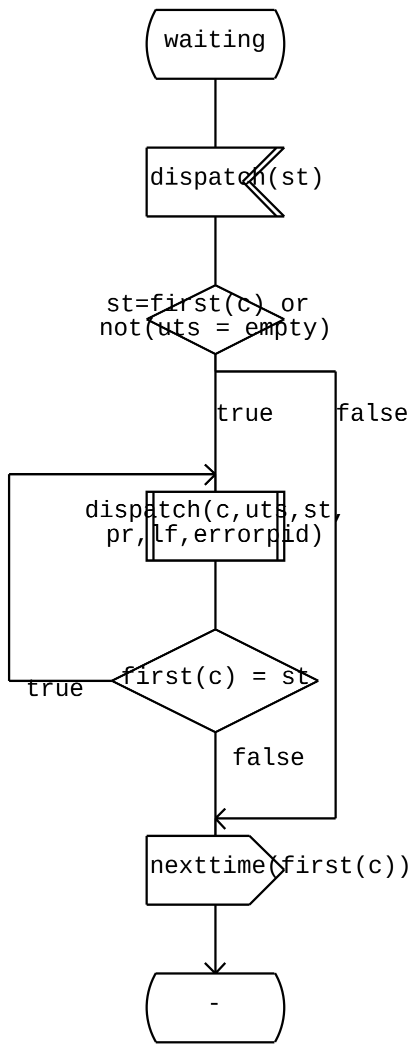
Process manager 10(11)

/* the simulation is running, but no events are dispatched currently.
If an untimed event is received via a schedule message, than the event is inserted
into the set uts.
If a timed event is received it is inserted in the eventcollection (resp. rejected
if it is too late. If its timestamp is earlier than that of the earliest event in
the collection, than a new nexttime message to the controlblock is sent.
*/



/* As long as the manager is not dispatching events, there are two possibilities to start dispatching. The manager is invoked because it is time to dispatch timed events or because there are newly arrived untimed ones. In both cases, the manager receives a signal dispatch from the control block with the actual simulation time. If there are events to be dispatched, (there are timed events with this time stamp or there are untimed events, then these will be dispatched, otherwise, a new nexttime message is sent to the control block.

The actual simulation time is set to the new value and the untimed events are stamped and inserted into the event collection to detect collisions. Then the actual events are dispatched. If there are further events with the same timestamp after dispatching, then these will be dispatched, until there are no events with the same timestamp. Then a nexttime message is sent to the control block and the manager waits until it is invoked again.
*/



Procedure dispatch

```
/* This procedure gets the current eventcollection and the current set of
untimed events. These are stamped and inserted in the eventcollection to
do the collision detection. The the actual set is computed and its events
are dispatched.

dispatches the events of the first class to be dispatched.
If this is done it returns.

Reinitialization of the manager is not possible while dispatching events.
The corresponding signals are saved.
*/
```

```
FPAR
     IN/OUT c eventcollection,
     IN/OUT uts utset,
     IN/OUT st TIME,
     IN/OUT pr priorityclass,
     IN     lf BOOLEAN,
     IN     errorpid Pid;
```

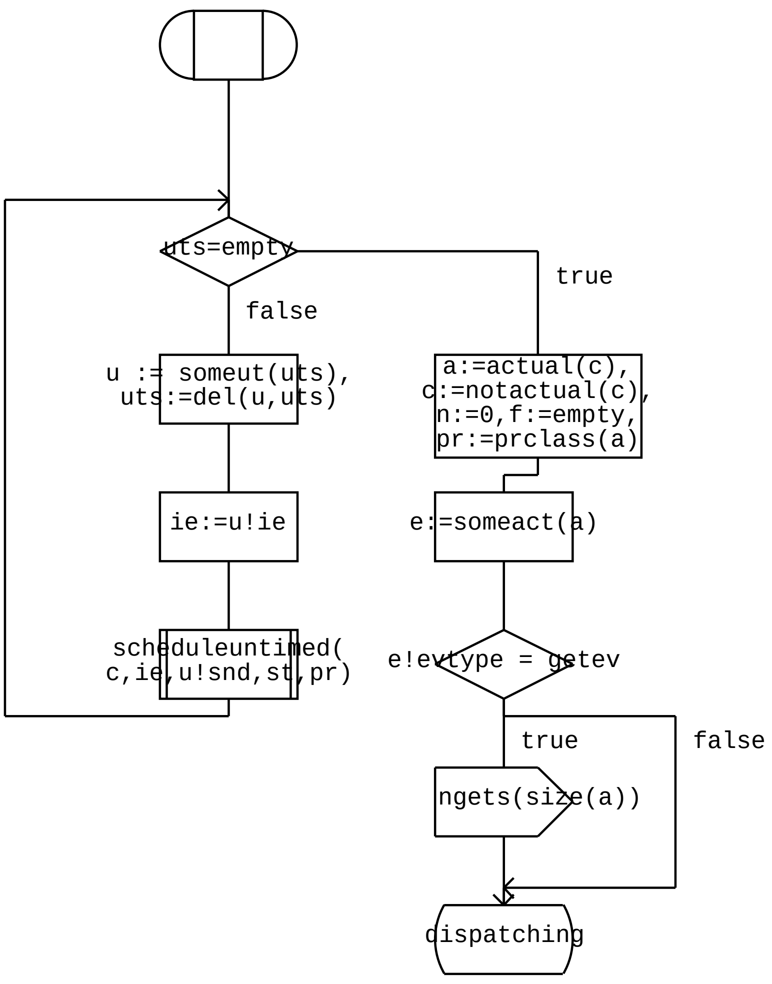
```
DCL    a actualset, /* the actual set */
       u ut,        /* an untimed event */
       ie internalevent,
       n integer, /* used to generate unique identifiers */
       id integer, /* received identifier */
       f idset, /* set of identifiers */
       e event, /* event to insert or remove from collection */
       ip inputprio, /* inputpriority of such an event*/
       str charstring, /* an error message */
       gr gravity;     /* gravity of an error */
```

*

clear,
endtime,
initlg

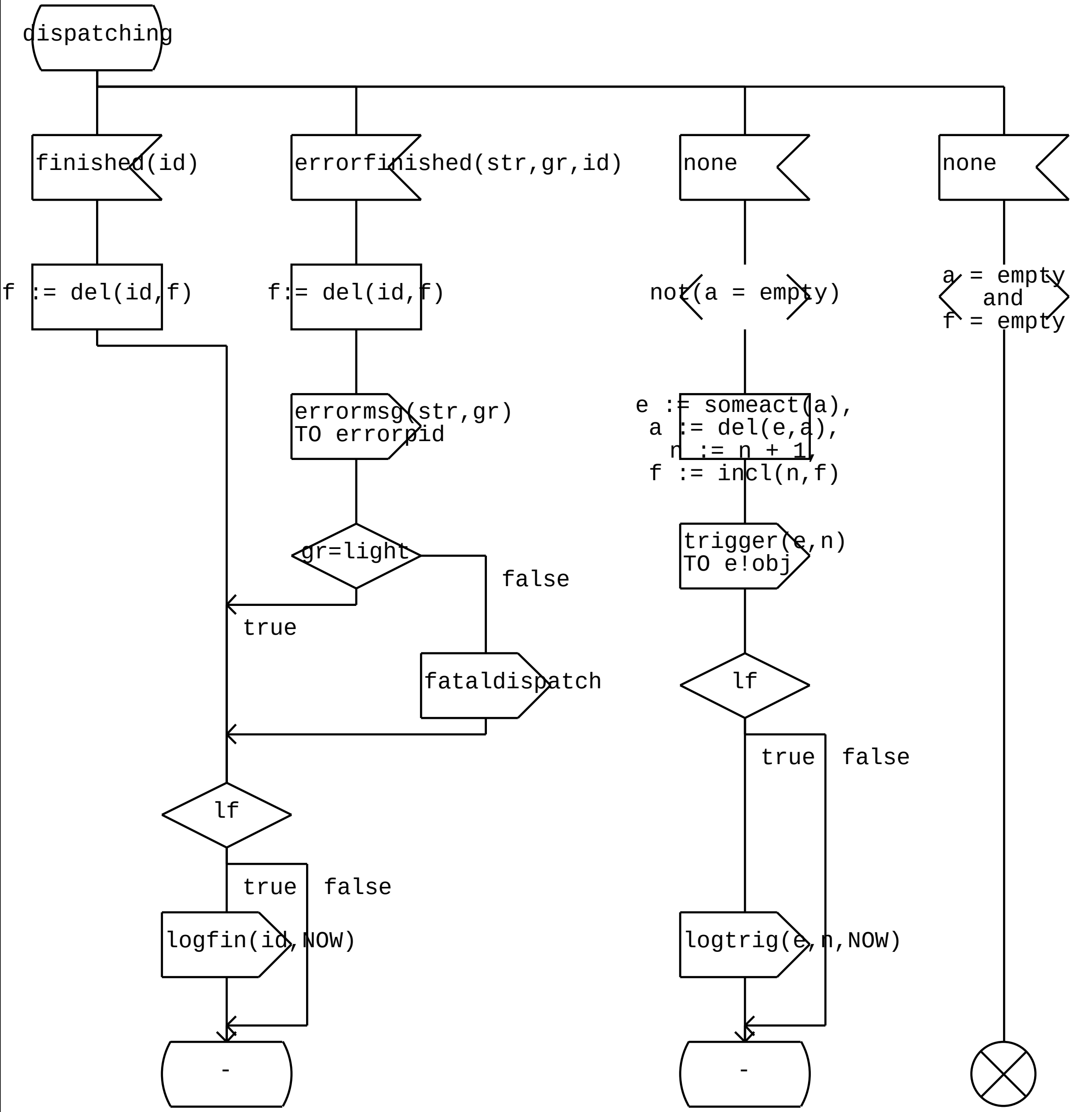
Procedure dispatch

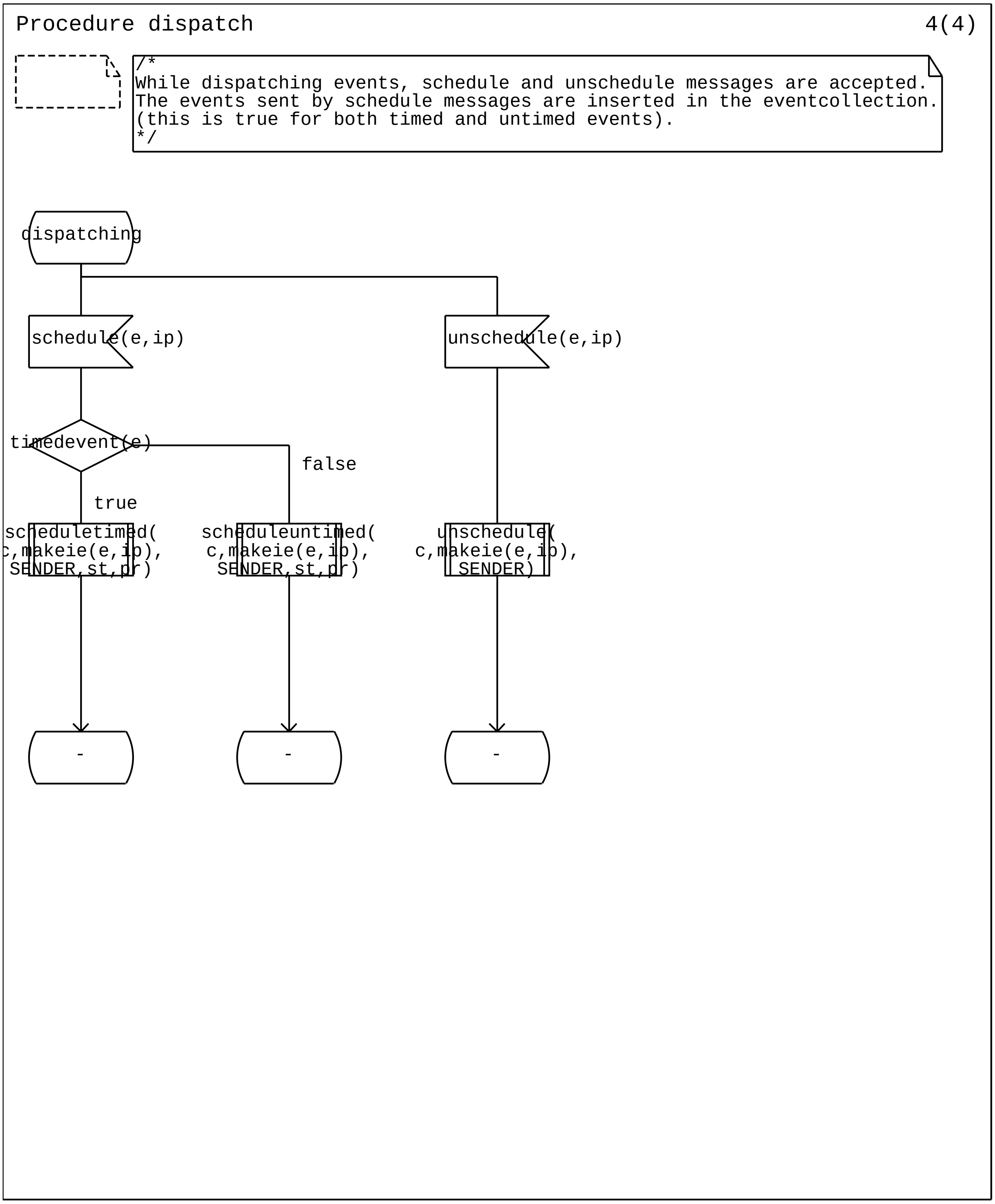
/*
The events in the set uts are stamped with the actual simulation time,
then they are inserted in the eventcollection, whereby possible collisions
can be detected.
If all these events have been processed, then the set of actual events is computed,
and dispatching is started. If the actual events are get events, then the size of
the actual set is propagated to the interface.
*/



Procedure dispatch

/*
As long as there are events in the actual set, these are dispatched.
The returned finished messages are deleted from the set f.
Dispatching ends, if there are no further events in the actual set and all finished messages are received.
In case of an error, the errormessages are propagated to the interface and in case of a fatal error, the control block is informed.
*/

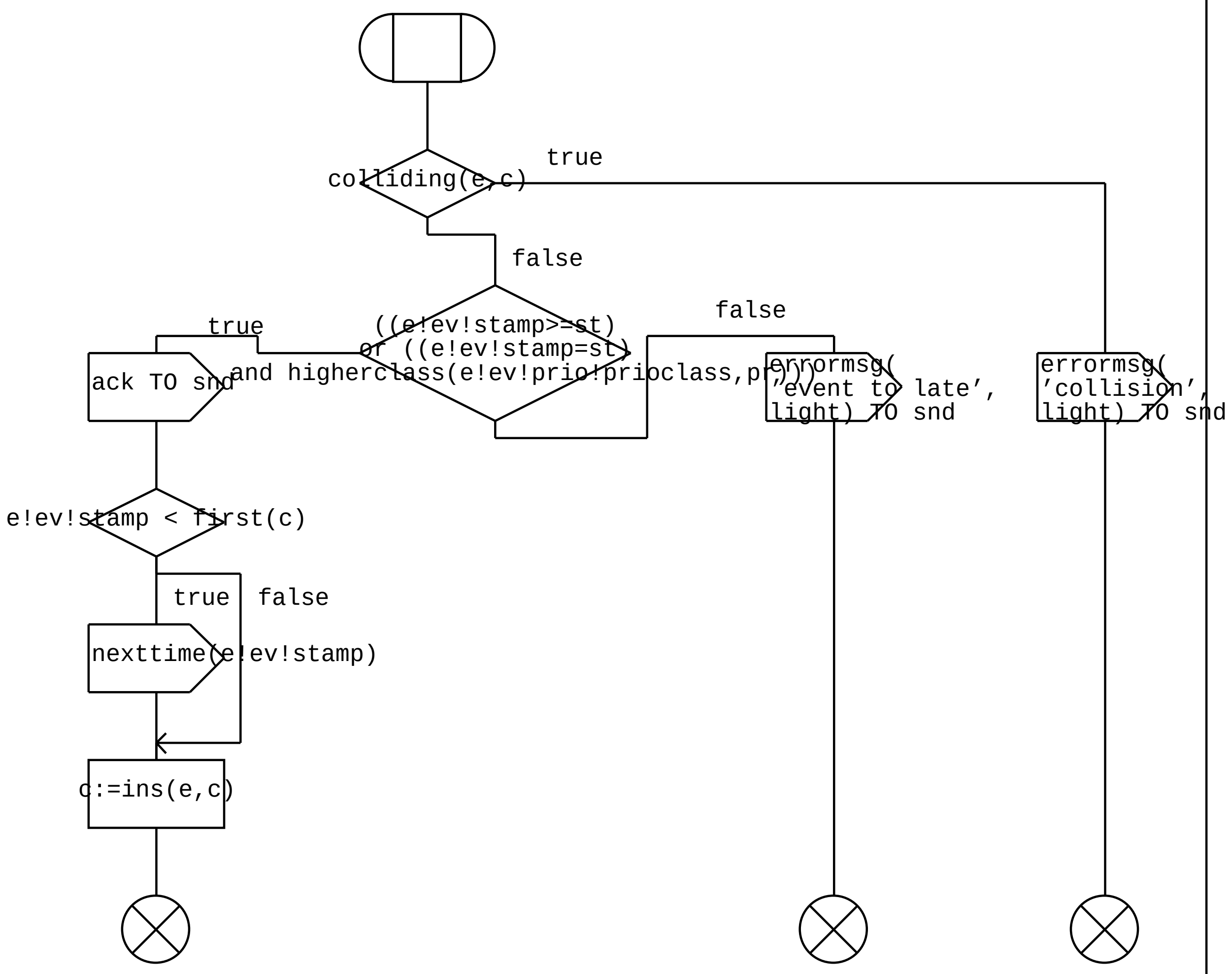


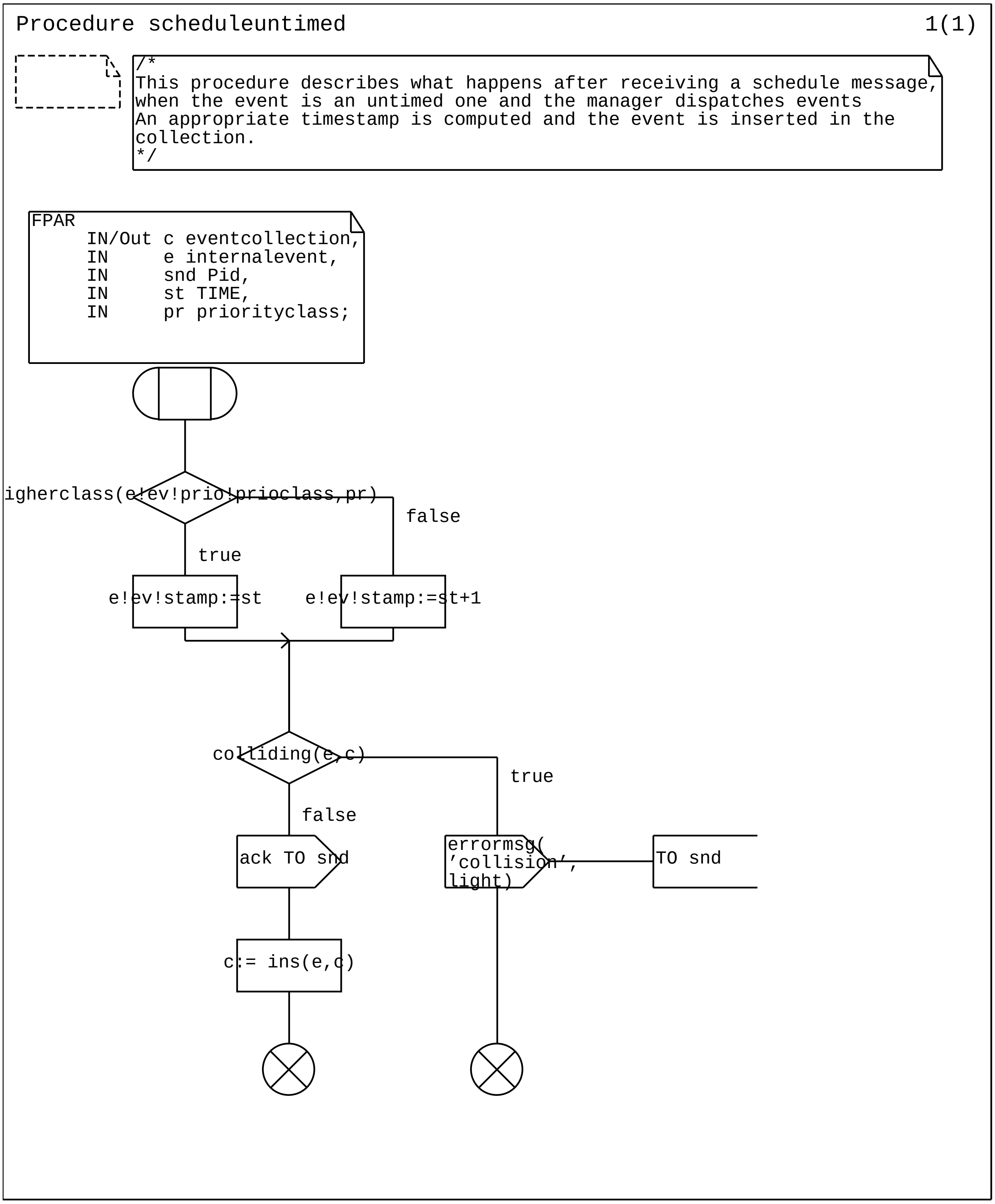

Procedure dispatch
4(4)
/*
While dispatching events, schedule and unschedule messages are accepted.
The events sent by schedule messages are inserted in the eventcollection.
(this is true for both timed and untimed events).
*/
dispatching
schedule(e,ip)
unschedule(e,ip)
timedevent(e)
false
true
scheduletimed(
c,makeie(e,ip),
SENDER,st,pr)
scheduleuntimed(
c,makeie(e,ip),
SENDER,st,pr)
unschedule(
c,makeie(e,ip),
SENDER)
-
-
-

Procedure scheduletimed 1(1)

```
/*
This procedure describes what happens after receiving a schedule message,
when the event is a timed one
It is checked, whether there is a colliding event in the collection,
if not it is inserted,
If its timestamp is prior to the first one of the collection, a nexttime
signal is sent to the controlblock.
*/
```

```
FPAR
     IN/Out c eventcollection,
     IN     e internalevent,
     IN     snd Pid,
     IN     st TIME,
     IN     pr priorityclass;
```





Procedure scheduleuntimed
1(1)
/*
This procedure describes what happens after receiving a schedule message,
when the event is an untimed one and the manager dispatches events
An appropriate timestamp is computed and the event is inserted in the
collection.
*/
FPAR
IN/Out c eventcollection,
IN e internalevent,
IN snd Pid,
IN st TIME,
IN pr priorityclass;
igherclass(e!ev!prio!prioclass,pr)
false
true
e!ev!stamp:=st
e!ev!stamp:=st+1
colliding(e,c)
true
false
ack TO snd
errormsg(
'collision',
light)
TO snd
c:= ins(e,c)

Procedure unschedule 1(1)

```
/*
This procedure describes what happens after receiving an unschedule message,
when the event is a timed one
It is checked, whether this event is in the collection
if not an errormessage is produced,
otherwise it is removed,
*/
```

```
FPAR
     IN/Out c eventcollection,
     IN     e internalevent,
     IN     snd Pid;
```