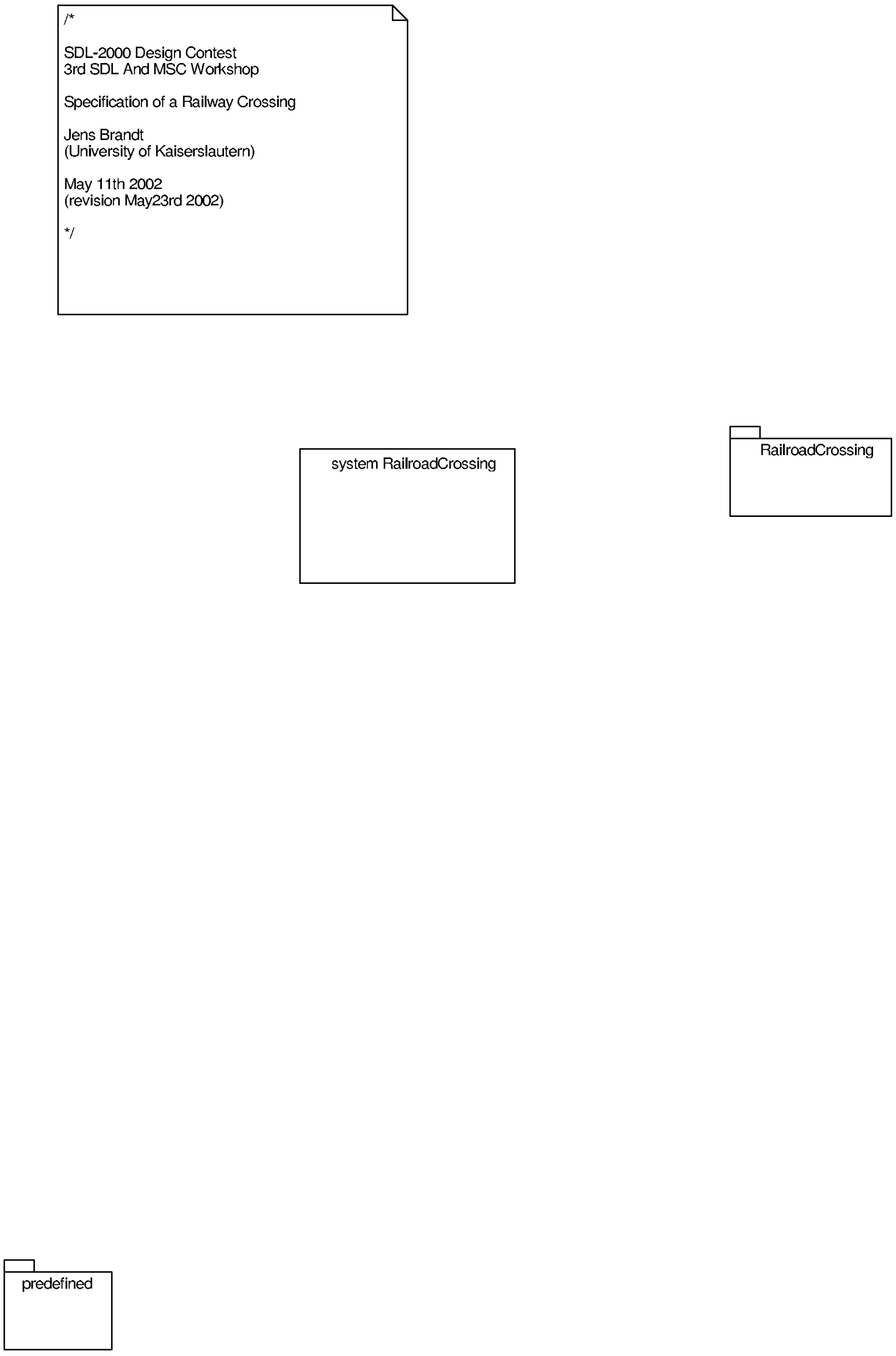


/*

SDL-2000 Design Contest
3rd SDL And MSC Workshop

Specification of a Railway Crossing

Jens Brandt
(University of Kaiserslautern)

May 11th 2002
(revision May23rd 2002)

*/

Package predefined

```
NEWTYPE Boolean
 LITERALS
  true,false;
 OPERATORS
  "not": Boolean         -> Boolean;
  "and": Boolean, Boolean -> Boolean;
  "or" : Boolean, Boolean -> Boolean;
  "xor": Boolean, Boolean -> Boolean;
  "=>" : Boolean, Boolean -> Boolean;
ENDNEWTYPE Boolean;

NEWTYPE Integer
 LITERALS
  NAMECLASS ('0':'9')+;
 OPERATORS
  "-"  : Integer         -> Integer;
  "+"  : Integer, Integer -> Integer;
  "-"  : Integer, Integer -> Integer;
  "*"  : Integer, Integer -> Integer;
  "/"  : Integer, Integer -> Integer;
  "mod": Integer, Integer -> Integer;
  "rem": Integer, Integer -> Integer;
  "<"  : Integer, Integer -> Boolean;
  ">"  : Integer, Integer -> Boolean;
  "<=" : Integer, Integer -> Boolean;
  ">=" : Integer, Integer -> Boolean;
  float: Integer         -> Real;
  fix  : Real            -> Integer;
ENDNEWTYPE Integer;

SYNTYPE Natural = Integer
 CONSTANTS >= 0
ENDSYNTYPE Natural;


NEWTYPE Real
 LITERALS
  NAMECLASS (('0':'9')+) OR (('0':'9')*'.'('0':'9')+);
 OPERATORS
  "-"  : Real          -> Real;
  "+"  : Real,Real     -> Real;
  "-"  : Real,Real     -> Real;
  "*"  : Real,Real     -> Real;
  "/"  : Real,Real     -> Real;
  "<"  : Real,Real     -> Boolean;
  ">"  : Real,Real     -> Boolean;
  "<=" : Real,Real     -> Boolean;
  ">=" : Real,Real     -> Boolean;
/* ASN.1 operator: */
  power: Integer, Integer -> Real;
ENDNEWTYPE Real;

NEWTYPE PId
 LITERALS
  null;
 OPERATORS
  unique! : PId -> PId;
ENDNEWTYPE PId;
```

```
NEWTYPE Character
 LITERALS
  NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
  BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
  DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
  CAN, EM,  SUB, ESC, FS,  GS,  RS,  US,
  ' ', '!', '"', '#', '$', '%', '&', '''',
  '(', ')', '*', '+', ',', '-', '.', '/',
  '0', '1', '2', '3', '4', '5', '6', '7',
  '8', '9', ':', ';', '<', '=', '>', '?',
  '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
  'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
  'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
  'X', 'Y', 'Z', '[', '\', ']', '^', '_',
  '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
  'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
  'x', 'y', 'z', '{', '|', '}', '~', DEL;
    □/* '''' is an apostrophe, ' ' is a space, '~' is a tilde */
 OPERATORS
  chr  : Integer -> Character;
  num  : Character -> Integer;
  "<"  : Character, Character -> Boolean;
  "<=" : Character, Character -> Boolean;
  ">"  : Character, Character -> Boolean;
  ">=" : Character, Character -> Boolean;
ENDNEWTYPE Character;

NEWTYPE Charstring String (Character,'')
 ADDING LITERALS
    NAMECLASS '''' ((' ':'&') OR '''''' OR ('(':'~'))+ '''';
ENDNEWTYPE Charstring;

NEWTYPE Duration
 LITERALS
  NAMECLASS (('0':'9')+) OR (('0':'9')*'.'('0':'9')+);
 OPERATORS
  duration!: Real          -> Duration;
  "+"  : Duration, Duration -> Duration;
  "-"  : Duration          -> Duration;
  "-"  : Duration, Duration -> Duration;
  "*"  : Real,     Duration -> Duration;
  "*"  : Duration, Real     -> Duration;
  "/"  : Duration, Real     -> Duration;
  "<"  : Duration, Duration -> Boolean;
  ">"  : Duration, Duration -> Boolean;
  "<=" : Duration, Duration -> Boolean;
  ">=" : Duration, Duration -> Boolean;
ENDNEWTYPE Duration;

NEWTYPE Time
 LITERALS
  NAMECLASS (('0':'9')+) OR (('0':'9')*'.'('0':'9')+);
 OPERATORS
  time!: Duration      -> Time;
  "<"  : Time, Time    -> Boolean;
  "<=" : Time, Time    -> Boolean;
  ">"  : Time, Time    -> Boolean;
  ">=" : Time, Time    -> Boolean;
  "+"  : Duration, Time -> Time;
  "+"  : Time, Duration -> Time;
  "-"  : Time, Duration -> Time;
  "-"  : Time, Time    -> Duration;
ENDNEWTYPE Time;
```

```
GENERATOR equality(TYPE item)
  OPERATORS
  "=" : equality, equality -> Boolean;
  "/=": equality, equality -> Boolean;
/*!Z105*/
 encode: equality -> Bitstring;
 encode: equality, Encoding -> Bitstring;
 decode: Bitstring -> equality;
 decode: Bitstring, Encoding -> equality;
/*!Z105END*/
ENDGENERATOR;


GENERATOR ordered(TYPE item)
  OPERATORS
    "<"  : ordered, ordered -> Boolean;
    ">"  : ordered, ordered -> Boolean;
    "<=" : ordered, ordered -> Boolean;
    ">=" : ordered, ordered -> Boolean;
ENDGENERATOR;

GENERATOR String(TYPE Itemsort LITERAL emptystring)
  /* Strings are "indexed" from one */
  LITERALS
    emptystring;
  OPERATORS
    mkstring : Itemsort                -> String;
    length   : String                  -> Integer;
    first    : String                  -> Itemsort;
    last     : String                  -> Itemsort;
    "//"     : String, String          -> String;
    extract! : String, Integer         -> Itemsort;
    modify!  : String, Integer, Itemsort -> String;
    substring: String, Integer, Integer  -> String;
ENDGENERATOR String;

GENERATOR Powerset(TYPE Itemsort)
  LITERALS
    empty;
  OPERATORS
    "in"    : Itemsort, Powerset -> Boolean;
    incl    : Itemsort, Powerset -> Powerset;
    del     : Itemsort, Powerset -> Powerset;
    "<"     : Powerset, Powerset -> Boolean;
    ">"     : Powerset, Powerset -> Boolean;
    "<="    : Powerset, Powerset -> Boolean;
    ">="    : Powerset, Powerset -> Boolean;
    "and"   : Powerset, Powerset -> Powerset;
    "or"    : Powerset, Powerset -> Powerset;
ENDGENERATOR Powerset;



GENERATOR Array(TYPE Index, TYPE Itemsort)
  OPERATORS
    make!   : Itemsort            -> Array;
    modify! : Array, Index, Itemsort -> Array;
    extract!: Array, Index        -> Itemsort;
ENDGENERATOR Array;
```

Package predefined

```
/*!Z105*/ /* Don't change this line */
/* ASN.1 types */
SYNTYPE
  IA5String = Charstring
ENDSYNTYPE;

SYNTYPE
  NumericString = Charstring (from ("0".."9"))
ENDSYNTYPE;

SYNTYPE
  Printablestring = Visiblestring
ENDSYNTYPE;

SYNTYPE
  Visiblestring = Charstring (from
("A".."Z"|"a".."z"|"0".."9"|"''",'(',')','+',',','-','.','/',':','=','?'))
ENDSYNTYPE;

NEWTYPE Graphicstring
  inherits Charstring
  operators all;
ENDNEWTYPE Graphicstring;

NEWTYPE Universalstring
  inherits Charstring
  operators all;
ENDNEWTYPE Universalstring;

NEWTYPE Enumeration
  operators
    pred  : Enumeration -> Enumeration;
    succ  : Enumeration -> Enumeration;
    first : Enumeration -> Enumeration;
    last  : Enumeration -> Enumeration;
    num   : Enumeration -> Integer;
    "<"   : Enumeration, Enumeration -> Boolean;
    "<="  : Enumeration, Enumeration -> Boolean;
    ">"   : Enumeration, Enumeration -> Boolean;
    ">="  : Enumeration, Enumeration -> Boolean;
ENDNEWTYPE Enumeration;

SYNONYM PLUS_INFINITY  Real = external;
SYNONYM MINUS_INFINITY Real = external;
```

```
/*!Z105*/ /* Don't change this line */
NEWTYPE Bit
  inherits Boolean
  literals 0 = false, 1 = true;
  operators all;
ENDNEWTYPE Bit;

Encoding ::=  ENUMERATED{BER,CER,DER,PER};

NEWTYPE Bitstring String0(Bit,''B);
  adding
    literals nameclass('0' or '1')*'B',
             nameclass(('0':'9') or ('A':'F'))*'H';
  operators
    "not": Bitstring           -> Bitstring;
    "and": Bitstring, Bitstring -> Bitstring;
    "or" : Bitstring, Bitstring -> Bitstring;
    "xor": Bitstring, Bitstring -> Bitstring;
    "=>" : Bitstring, Bitstring -> Bitstring;
ENDNEWTYPE Bitstring;

SYNTYPE Octet = Bitstring constants size (8)
ENDSYNTYPE Octet;

NEWTYPE Octetstring String(Octet,''B)
    literals nameclass(('0' or '1')8)+'B',
             nameclass((('0':'9') or ('A':'F'))2)+'H';
    operators
      bitstring   : Octetstring -> Bitstring;
      octetstring : Bitstring -> Octetstring;
      Bit_String   : Octetstring -> Bitstring; /* SDL 96 version */
      Octet_String : Bitstring -> Octetstring; /* SDL 96 version */
ENDNEWTYPE Octetstring;

syntype Octet_String = Octetstring endsyntype;
syntype Bit_String = Bitstring endsyntype;

NEWTYPE NULL
  literals null;
ENDNEWTYPE NULL;

NEWTYPE Object_element
  literals nameclass ('0':'9')+;
ENDNEWTYPE Object_element;

NEWTYPE Object_identifier String(Object_element,emptystring)
ENDNEWTYPE Object_identifier;

NEWTYPE Any_type
ENDNEWTYPE Any_type;

GeneralizedTime      ::=      Visiblestring;
ATCTime ::= Visiblestring;
UTCTime ::= Visiblestring;
EXTERNAL_Type ::= sequence
        { direct_reference      Object_identifier optional,
          indirect_reference    Integer           optional,
          data_value_descriptor ObjectDescriptor  optional,
          encoding choice { single_ASN1_type Any_type,
                            octet_aligned    Octetstring,
                            arbitrary        Bitstring
                          }
        };
ObjectDescriptor ::= Graphicstring;
```

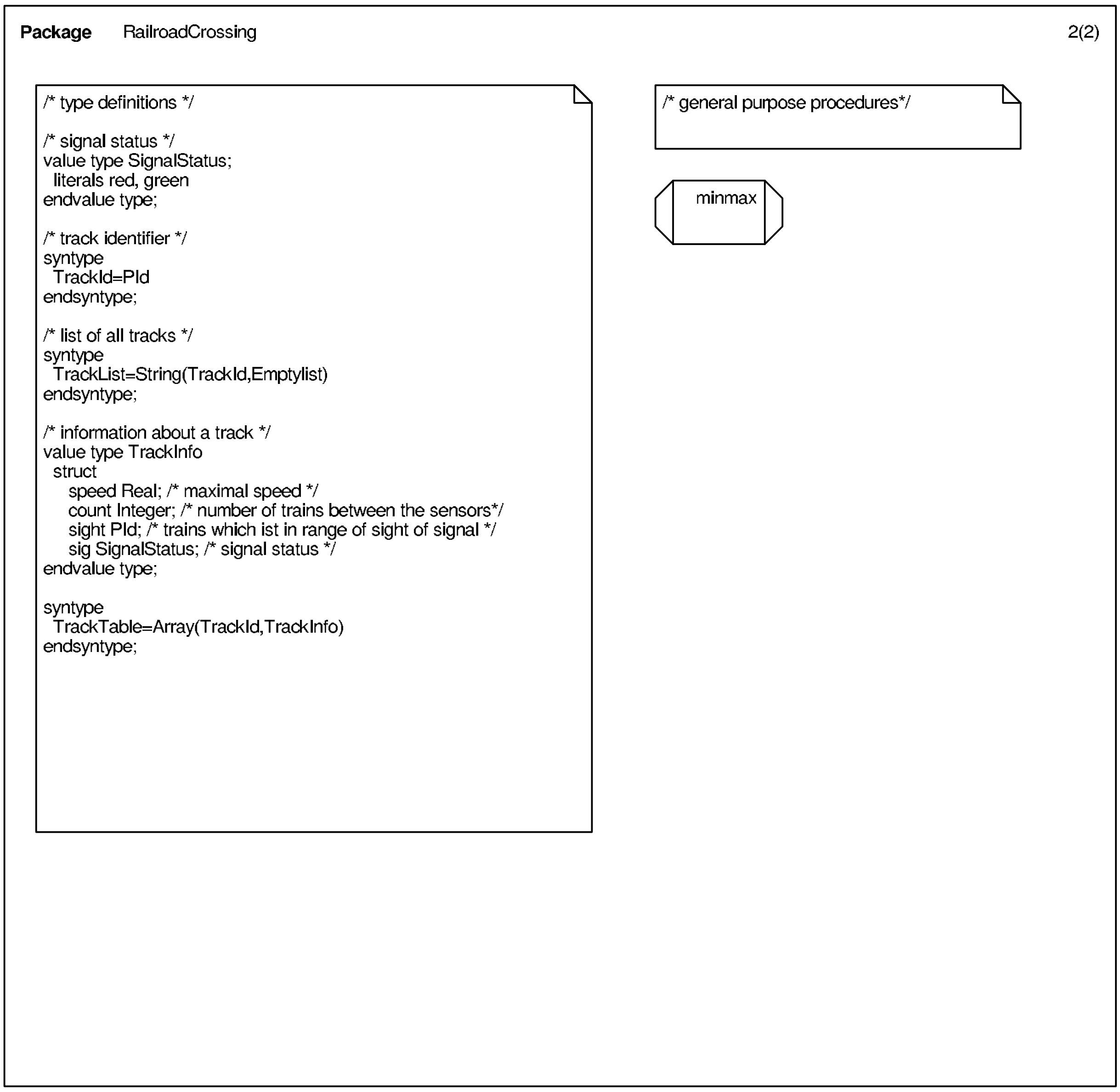
```
/***** ASN.1 GENERATORS *****/
GENERATOR String0(TYPE Itemsort, LITERAL Emptystring)
    String(Itemsort,Emptystring)
ENDGENERATOR;

GENERATOR Bag(type Itemsort)
  literals Empty;
  operators
    incl   : Itemsort, Bag -> Bag;
    del    : Itemsort, Bag -> Bag;
    length : Bag           -> Integer;
    take   : Bag           -> Itemsort;
    makebag: Itemsort      -> Bag;
    "in"   : Itemsort, Bag -> Boolean;
    "<"    : Bag, Bag      -> Boolean;
    ">"    : Bag, Bag      -> Boolean;
    "<="   : Bag, Bag      -> Boolean;
    ">="   : Bag, Bag      -> Boolean;
    "and"  : Bag, Bag      -> Bag;
    "or"   : Bag, Bag      -> Bag;
ENDGENERATOR;

/*!SDL2000*/ /* Don't change this line */
exception
        OutOfRange,          /* A range check has failed. */
        InvalidReference,    /* Null was used incorrectly. Wrong Pid for this signal. */
        NoMatchingAnswer,    /* No answer matched in a decision without else part. */
        UndefinedVariable,       /* A variable was used that is "undefined". */
        UndefinedField,      /* An undefined field of a choice or struct was accessed. */
        InvalidIndex,        /* A String or Array was accessed with an incorrect index. */
        DivisionByZero,      /* An Integer or Real division by zero was attempted. */
        Empty;               /* No element could be returned. */
```

Package RailroadCrossing 1(2)

```
/* signal definitions */

signal openGate;
signal closeGate;
signal gateOpen;
signal gateClosed;

signal trainApproaching( TrackId );
signal trainLeaving( TrackId );
signal detectLeaving( TrackId );
signal detectApproaching( TrackId );

signal trainSignal( SignalStatus );
signal setSignals( TrackList, SignalStatus );
signal settingDone( TrackList, SignalStatus );
signal leaving;

signal carsWaiting;
signal manyCarsWaiting;

signal trackAnnounce( TrackId,Real );
signal inSight( TrackId,PId );
signal position( Real,Real );

/* signallist definitions */

signallist trainSensor=detectApproaching, detectLeaving;
signallist trainDetection=trainApproaching, trainLeaving;
signallist carSensor=carsWaiting, manyCarsWaiting;
signallist gateControl=closeGate,openGate;
signallist gateStatus=gateClosed, gateOpen;
```

```
/* track layout */

synonym posSensor1 Real=500;
        /* position of the "approaching sensor" */
synonym posSignal Real=1500;
        /* position of the "signal" */
synonym posSensor2 Real=2000;
        /* position of the "leaving sensor" */
synonym posEnd Real=2500;
        /* end of the track*/

/* track parameters */

synonym fastSpeed Real=80;
        /* maximal speed of fast trains*/
synonym regularSpeed Real=50;
        /* maximal speed of regular trains*/
```

```
signal nextTrain(PId);
```

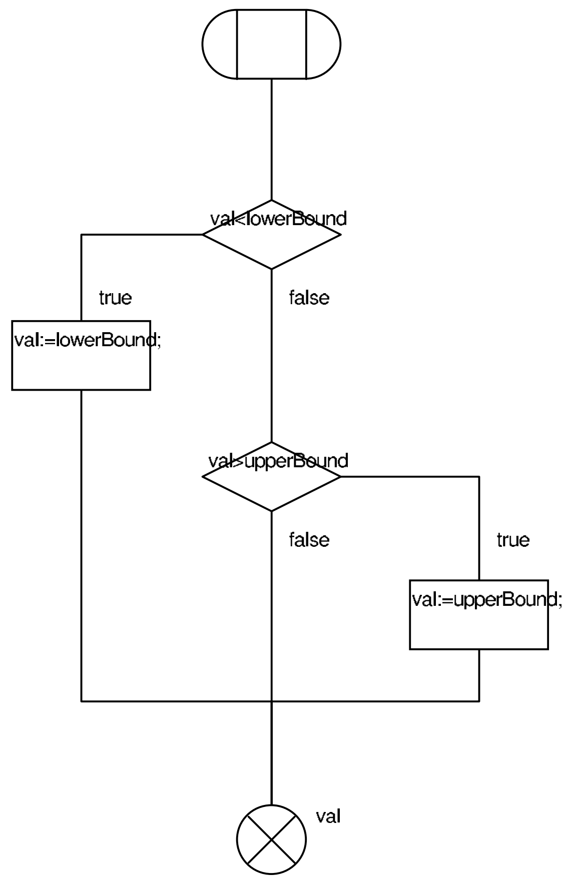


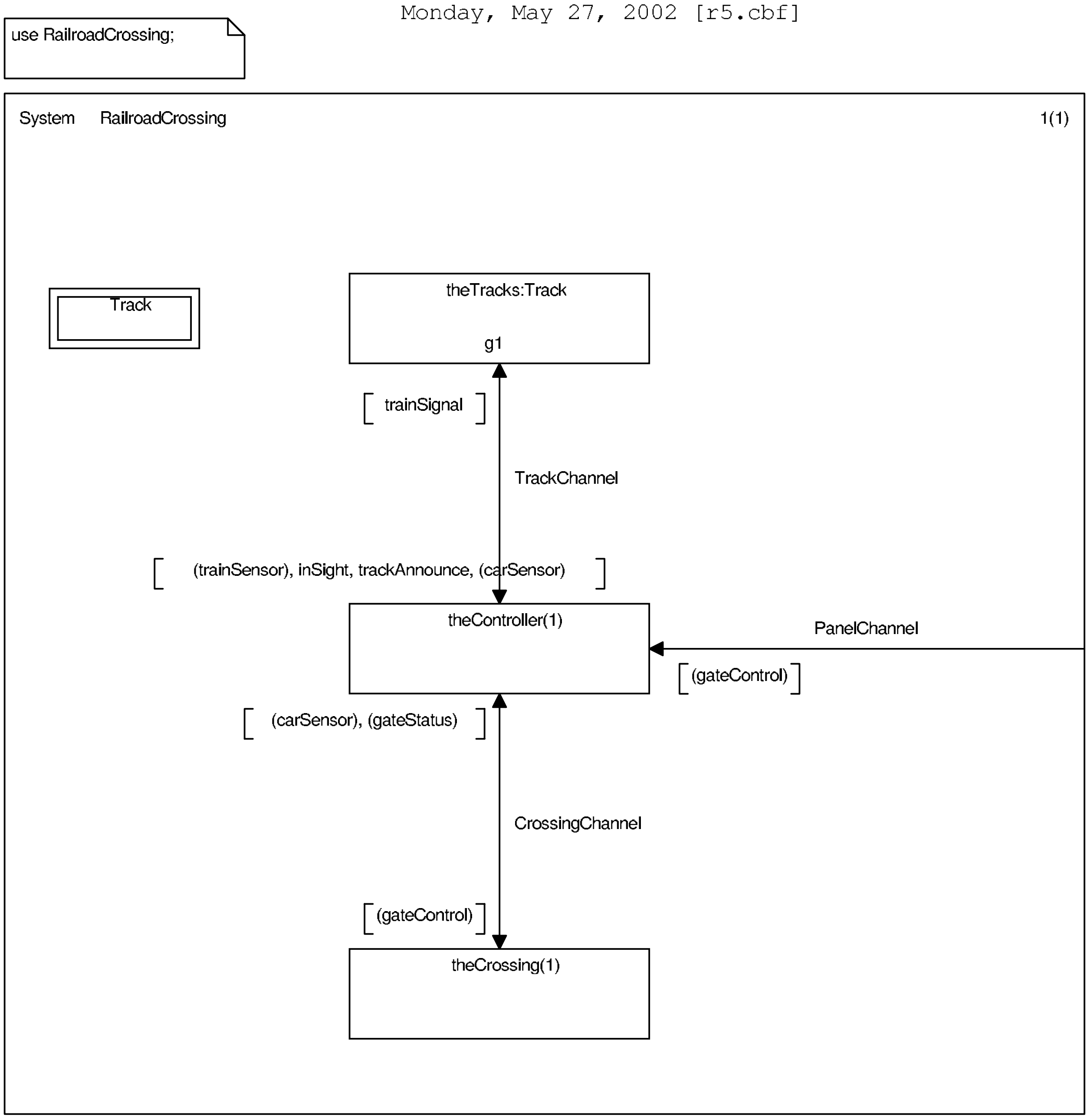
```
/* type definitions */

/* signal status */
value type SignalStatus;
  literals red, green
endvalue type;

/* track identifier */
syntype
  TrackId=PId
endsyntype;

/* list of all tracks */
syntype
  TrackList=String(TrackId,Emptylist)
endsyntype;

/* information about a track */
value type TrackInfo
  struct
    speed Real; /* maximal speed */
    count Integer; /* number of trains between the sensors*/
    sight PId; /* trains which ist in range of sight of signal */
    sig SignalStatus; /* signal status */
endvalue type;

syntype
  TrackTable=Array(TrackId,TrackInfo)
endsyntype;
```

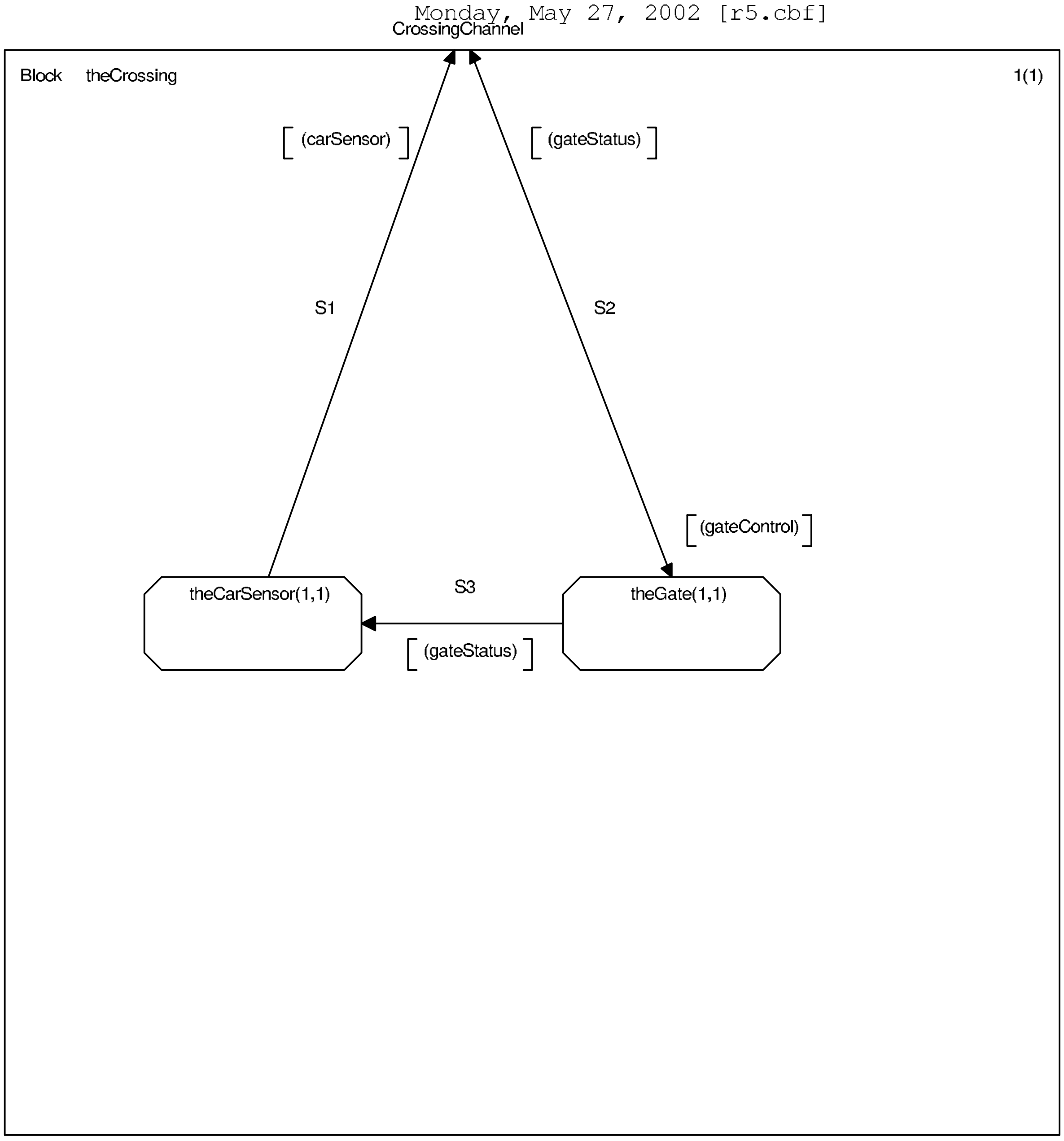
```
/* general purpose procedures*/
```

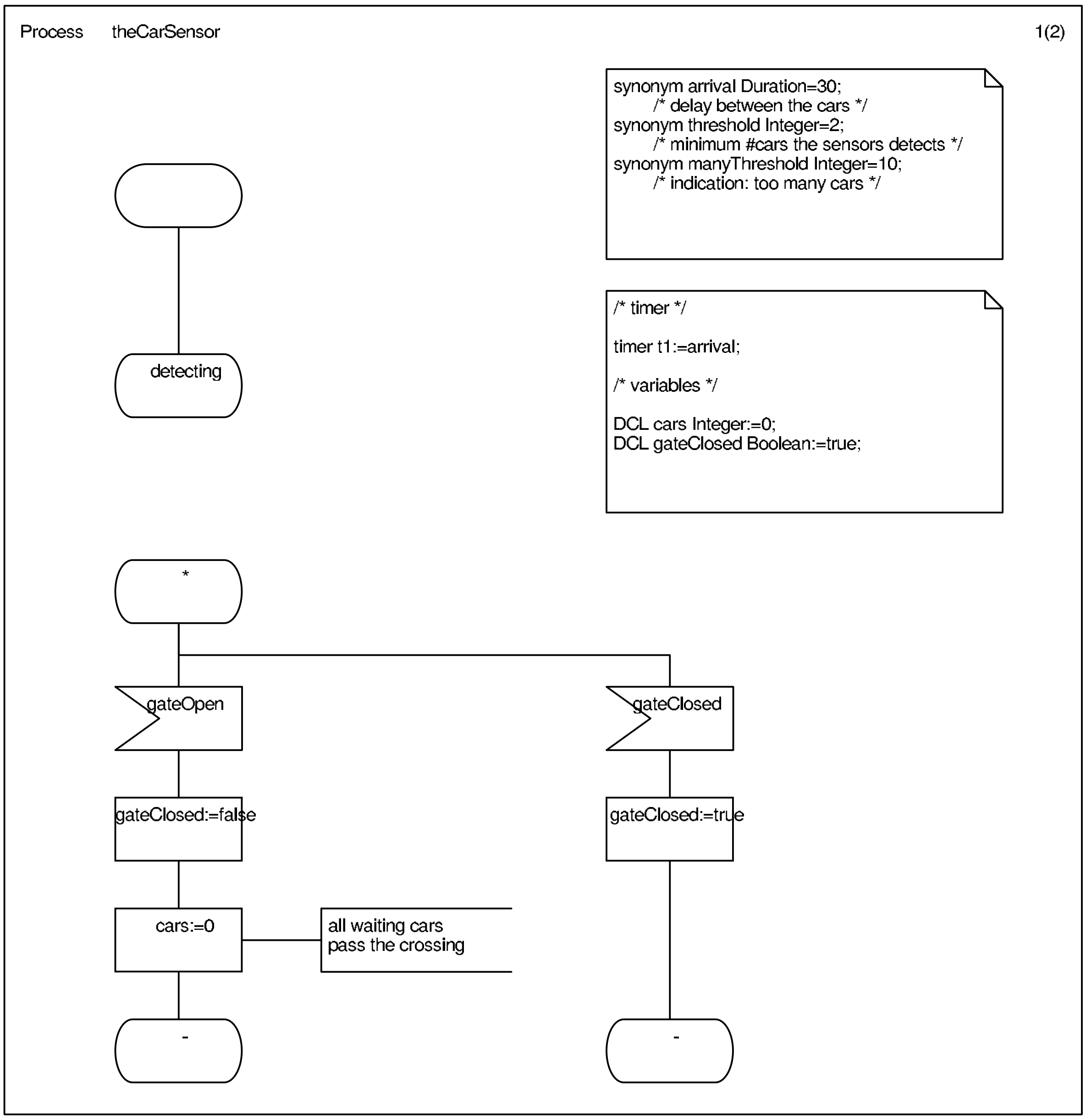
Procedure minmax 1(1)

fpar lowerBound Real, upperBound Real, val Real; returns Real



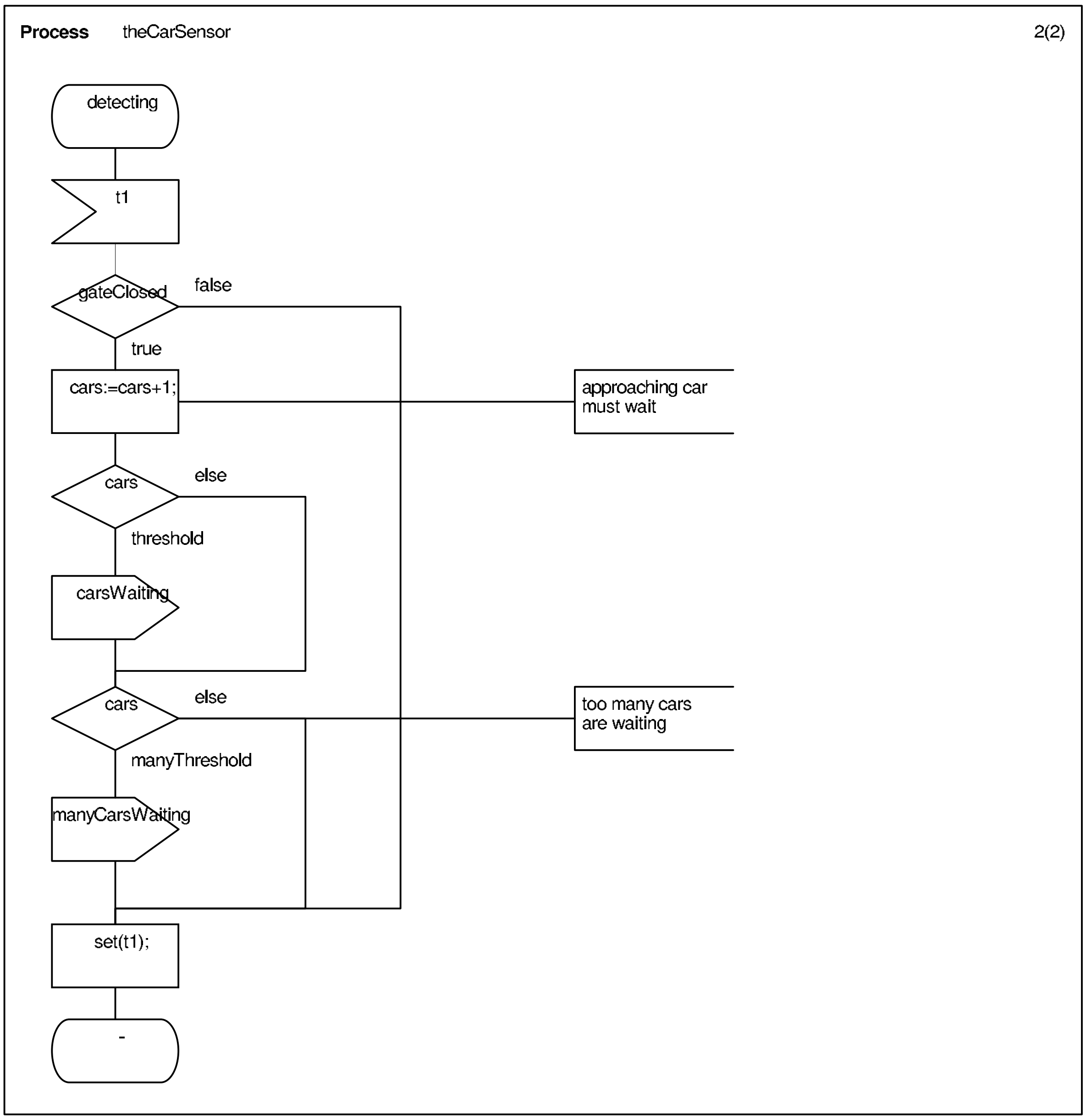

use RailroadCrossing;
System RailroadCrossing
1(1)
Track
theTracks:Track
g1
trainSignal
TrackChannel
(trainSensor), inSight, trackAnnounce, (carSensor)
theController(1)
PanelChannel
(gateControl)
(carSensor), (gateStatus)
CrossingChannel
(gateControl)
theCrossing(1)

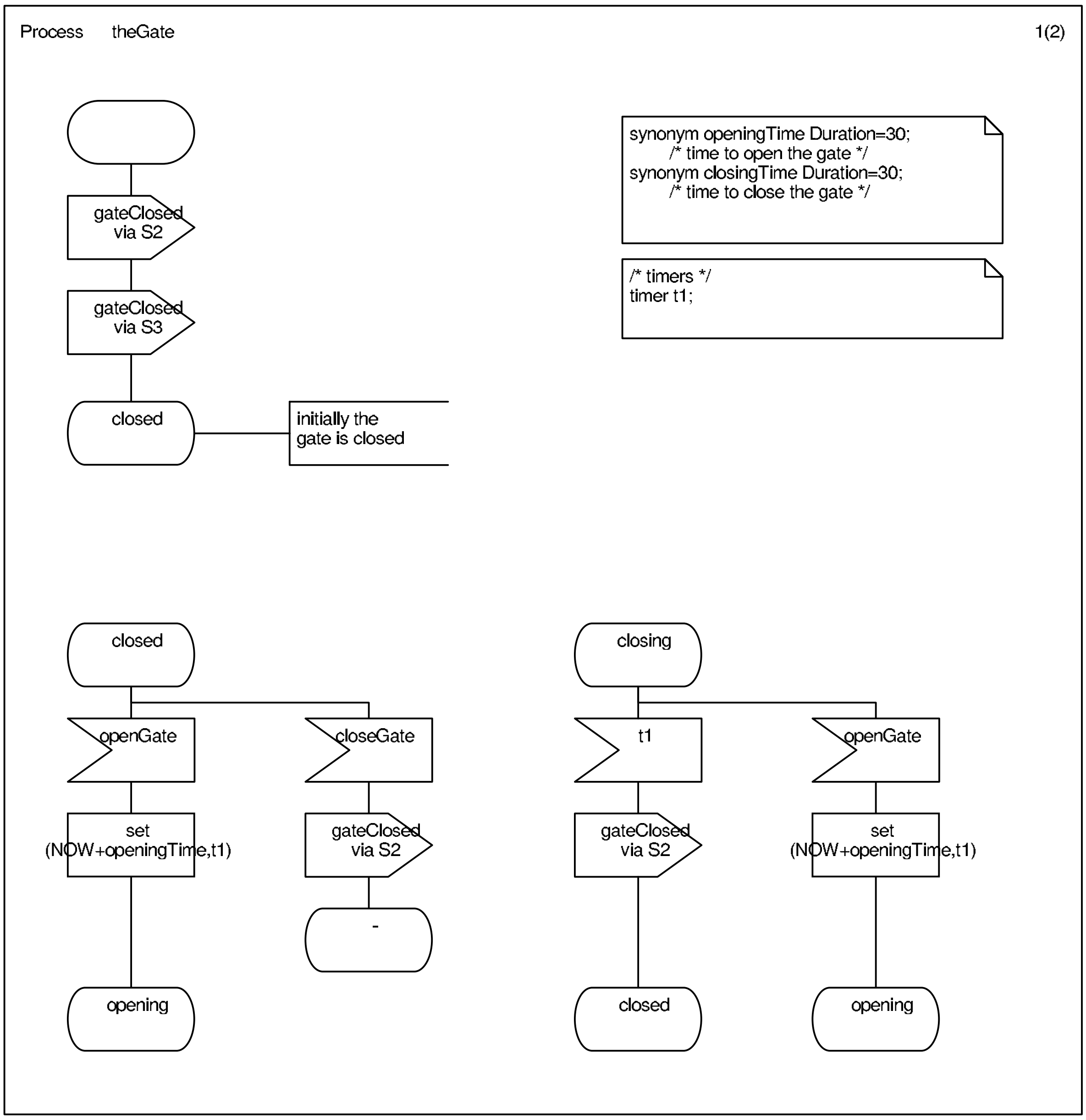

CrossingChannel
Block theCrossing
1(1)
(carSensor)
(gateStatus)
S1
S2
(gateControl)
theCarSensor(1,1)
S3
theGate(1,1)
(gateStatus)

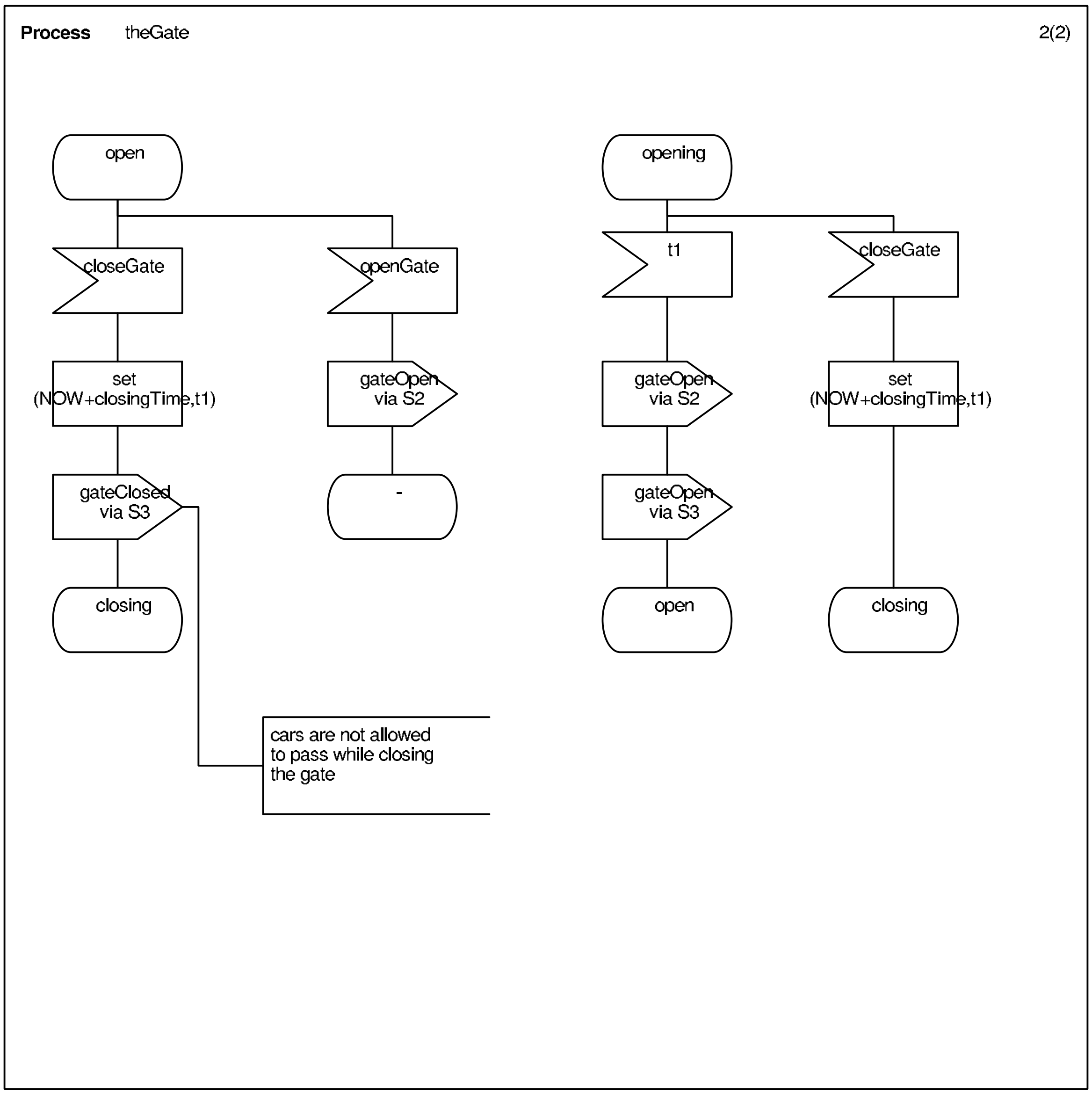


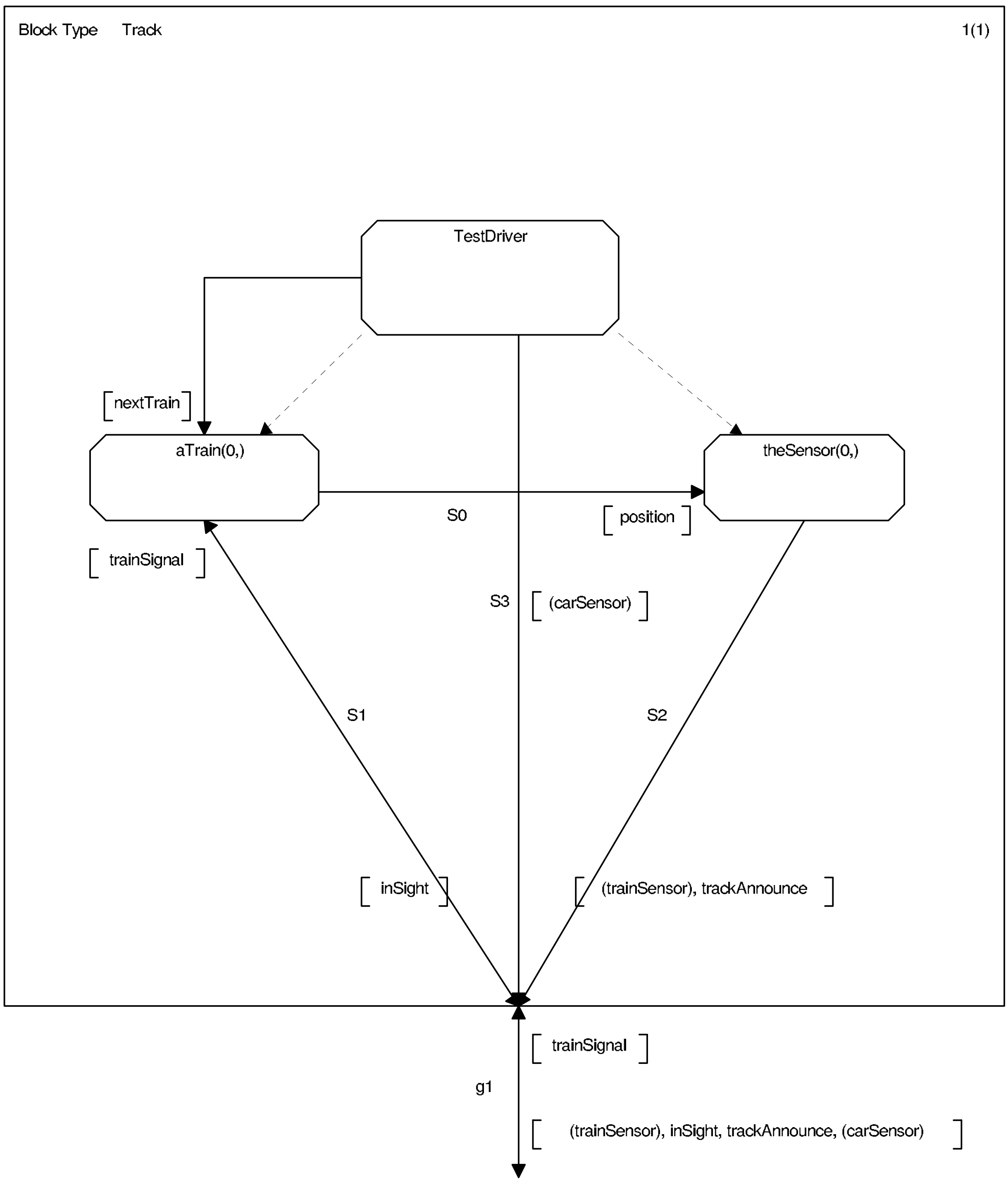
```
synonym arrival Duration=30;
        /* delay between the cars */
synonym threshold Integer=2;
        /* minimum #cars the sensors detects */
synonym manyThreshold Integer=10;
        /* indication: too many cars */
```

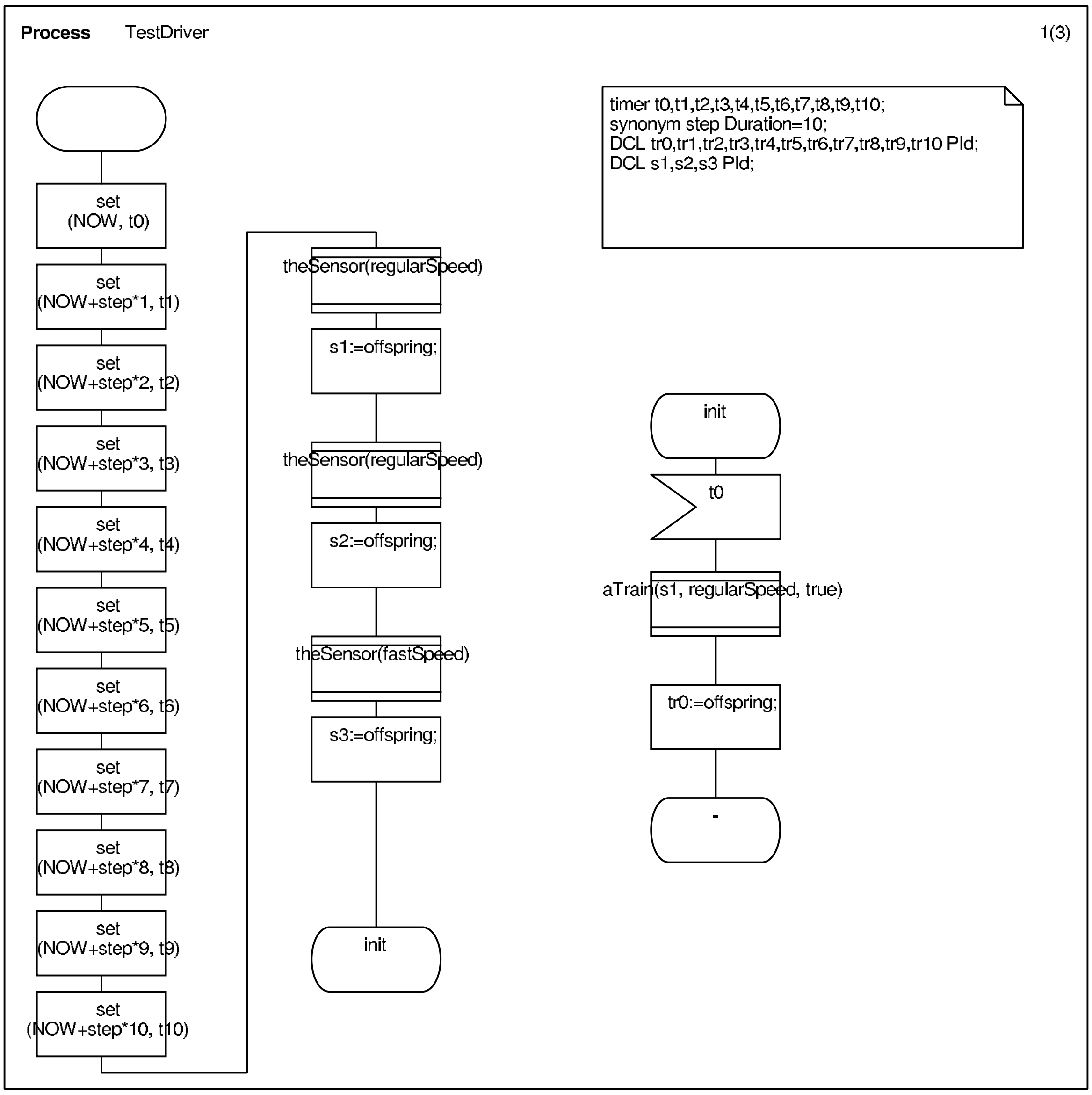
```
/* timer */

timer t1:=arrival;

/* variables */

DCL cars Integer:=0;
DCL gateClosed Boolean:=true;
```

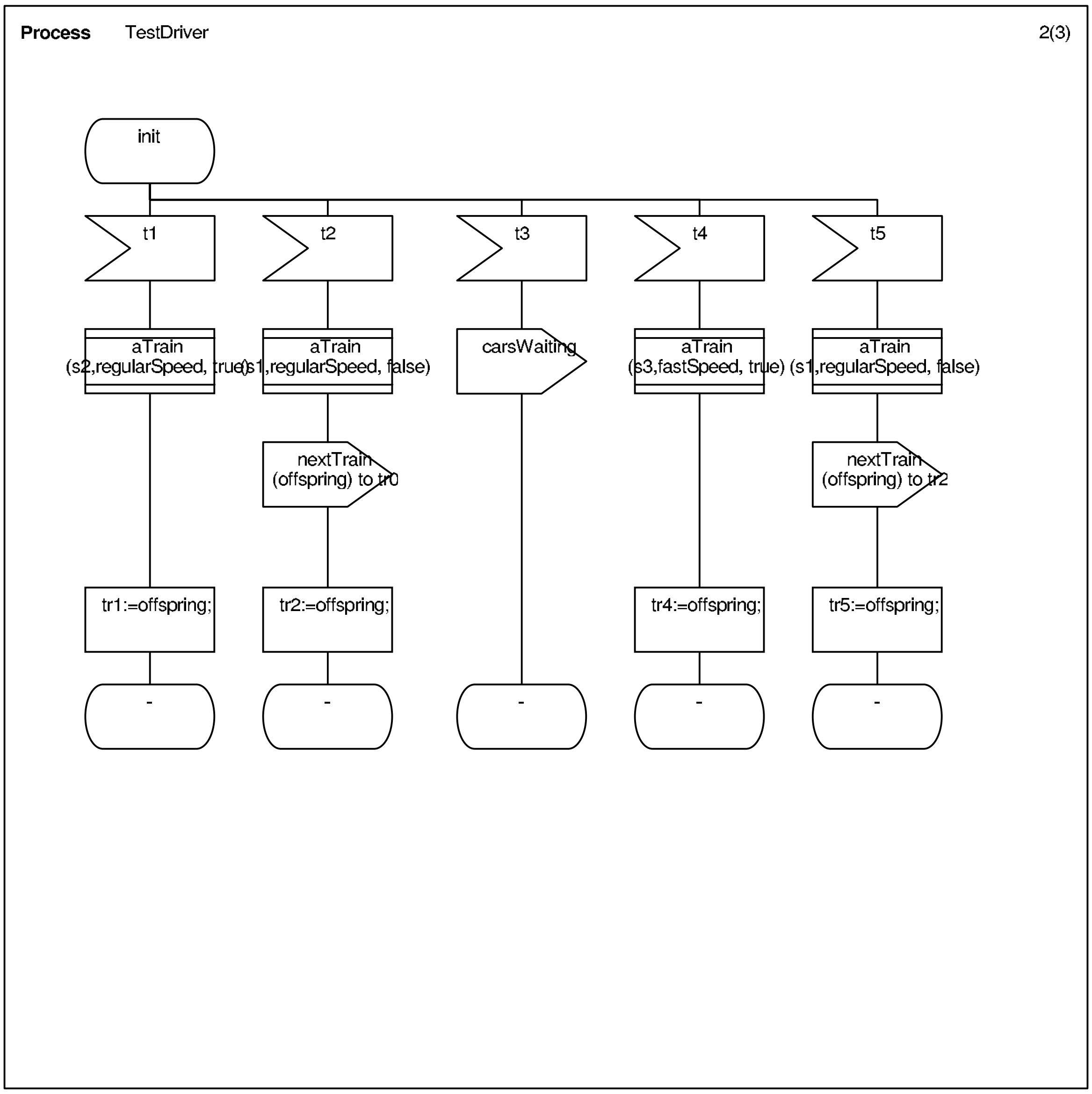

Process theCarSensor
2(2)
detecting
t1
gateClosed
false
true
cars:=cars+1;
approaching car must wait
cars
else
threshold
carsWaiting
cars
else
too many cars are waiting
manyThreshold
manyCarsWaiting
set(t1);
-

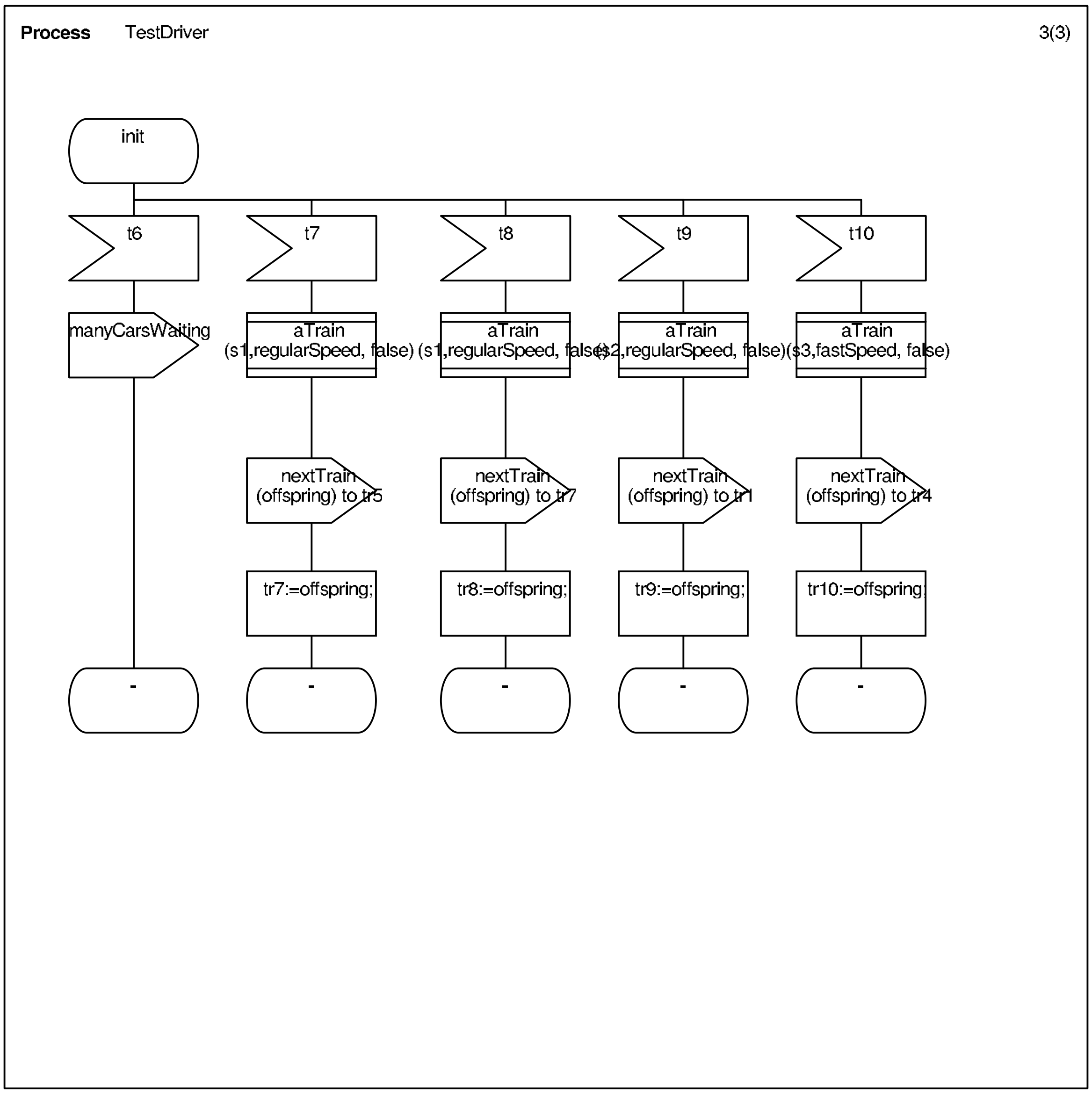

Process theGate
1(2)
synonym openingTime Duration=30;
/* time to open the gate */
synonym closingTime Duration=30;
/* time to close the gate */
/* timers */
timer t1;
gateClosed via S2
gateClosed via S3
closed
initially the gate is closed
closed
openGate
closeGate
set (NOW+openingTime,t1)
gateClosed via S2
-
opening
closing
t1
openGate
gateClosed via S2
set (NOW+openingTime,t1)
closed
opening

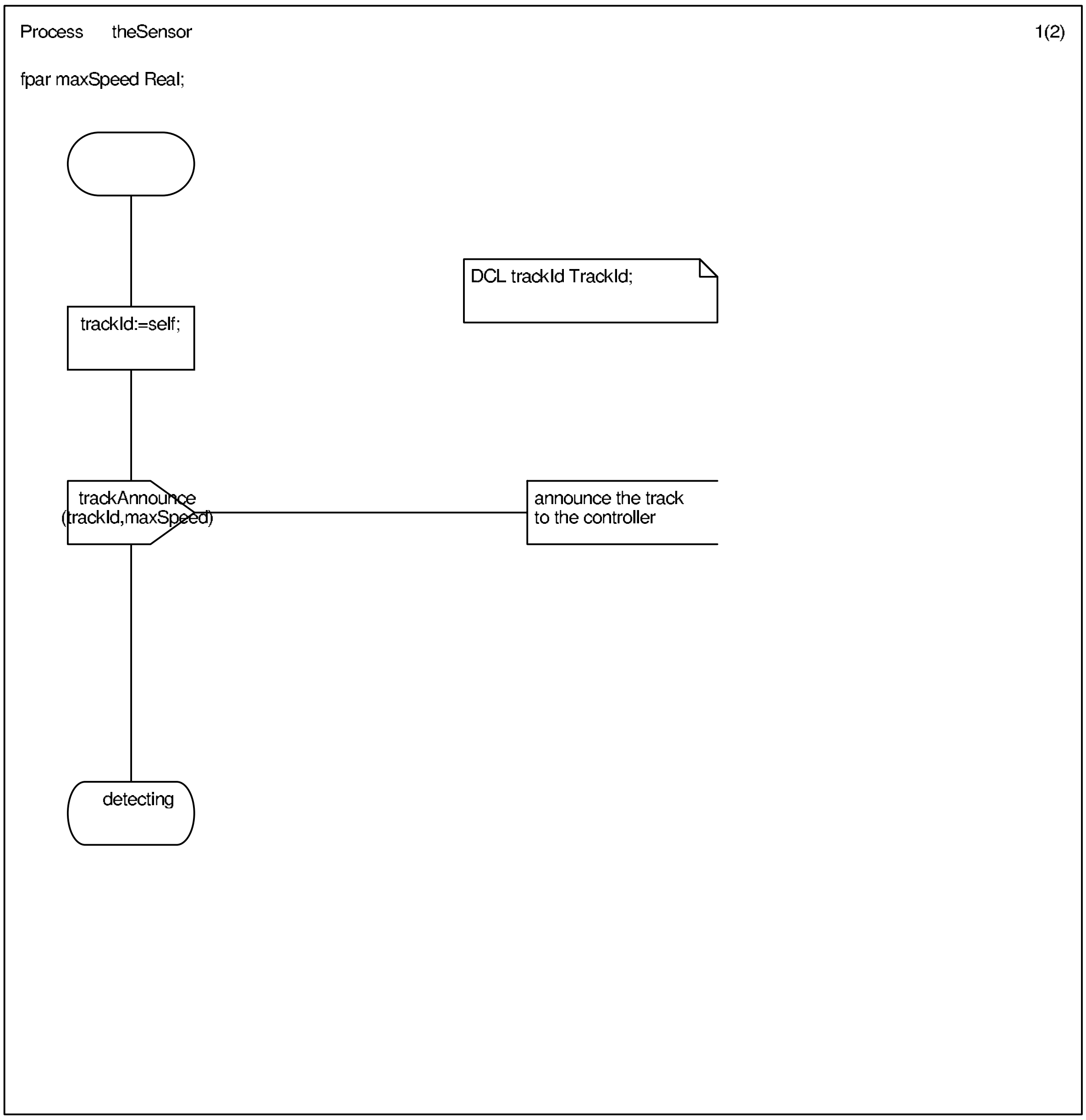

Process theGate
2(2)
open
closeGate
set (NOW+closingTime,t1)
gateClosed via S3
closing
openGate
gateOpen via S2
-
cars are not allowed to pass while closing the gate
opening
t1
gateOpen via S2
gateOpen via S3
open
closeGate
set (NOW+closingTime,t1)
closing

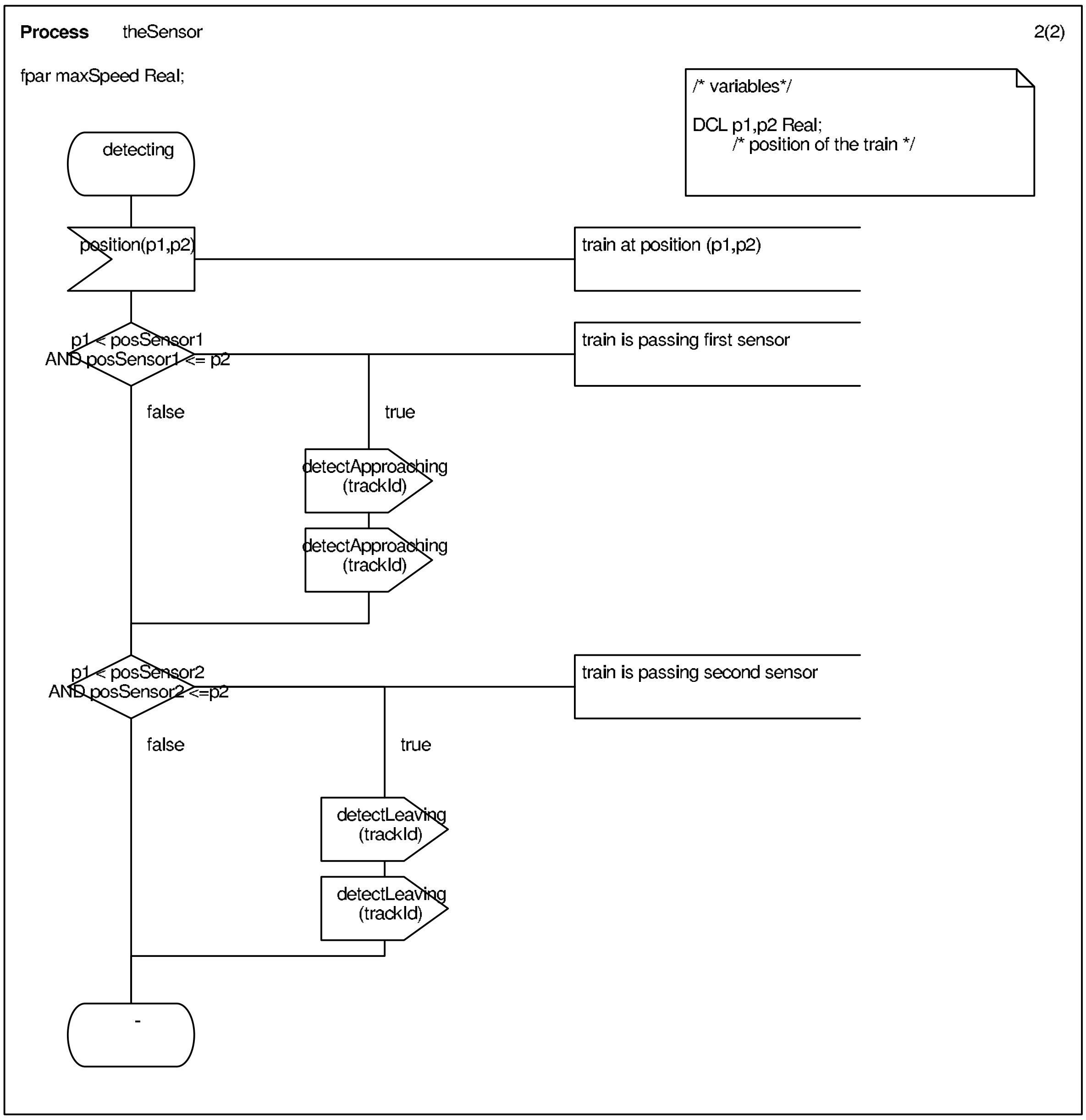

Block Type Track
1(1)
TestDriver
nextTrain
aTrain(0,)
theSensor(0,)
S0
position
trainSignal
S3
(carSensor)
S1
S2
inSight
(trainSensor), trackAnnounce
trainSignal
g1
(trainSensor), inSight, trackAnnounce, (carSensor)


Process TestDriver
1(3)
timer t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10;
synonym step Duration=10;
DCL tr0,tr1,tr2,tr3,tr4,tr5,tr6,tr7,tr8,tr9,tr10 PId;
DCL s1,s2,s3 PId;
set (NOW, t0)
set (NOW+step*1, t1)
set (NOW+step*2, t2)
set (NOW+step*3, t3)
set (NOW+step*4, t4)
set (NOW+step*5, t5)
set (NOW+step*6, t6)
set (NOW+step*7, t7)
set (NOW+step*8, t8)
set (NOW+step*9, t9)
set (NOW+step*10, t10)
theSensor(regularSpeed)
s1:=offspring;
theSensor(regularSpeed)
s2:=offspring;
theSensor(fastSpeed)
s3:=offspring;
init
init
t0
aTrain(s1, regularSpeed, true)
tr0:=offspring;
-


Process TestDriver
2(3)
init
t1
t2
t3
t4
t5
aTrain (s2,regularSpeed, true)
aTrain (s1,regularSpeed, false)
carsWaiting
aTrain (s3,fastSpeed, true)
aTrain (s1,regularSpeed, false)
nextTrain (offspring) to tr0
nextTrain (offspring) to tr2
tr1:=offspring;
tr2:=offspring;
tr4:=offspring;
tr5:=offspring;
-
-
-
-
-

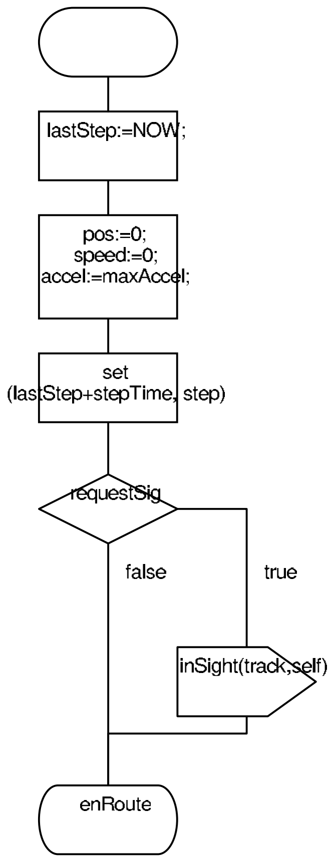

Process TestDriver
3(3)
init
t6
t7
t8
t9
t10
manyCarsWaiting
aTrain (s1,regularSpeed, false)
aTrain (s1,regularSpeed, false)
aTrain (s2,regularSpeed, false)
aTrain (s3,fastSpeed, false)
nextTrain (offspring) to tr5
nextTrain (offspring) to tr7
nextTrain (offspring) to tr1
nextTrain (offspring) to tr4
tr7:=offspring;
tr8:=offspring;
tr9:=offspring;
tr10:=offspring;
-
-
-
-
-


Process theSensor
1(2)
fpar maxSpeed Real;
DCL trackId TrackId;
trackId:=self;
trackAnnounce (trackId,maxSpeed)
announce the track to the controller
detecting


Process theSensor
2(2)
fpar maxSpeed Real;
/* variables*/
DCL p1,p2 Real;
/* position of the train */
detecting
position(p1,p2)
train at position (p1,p2)
p1 < posSensor1 AND posSensor1 <= p2
train is passing first sensor
false
true
detectApproaching (trackId)
detectApproaching (trackId)
p1 < posSensor2 AND posSensor2 <=p2
train is passing second sensor
false
true
detectLeaving (trackId)
detectLeaving (trackId)
-

Process aTrain 1(4)

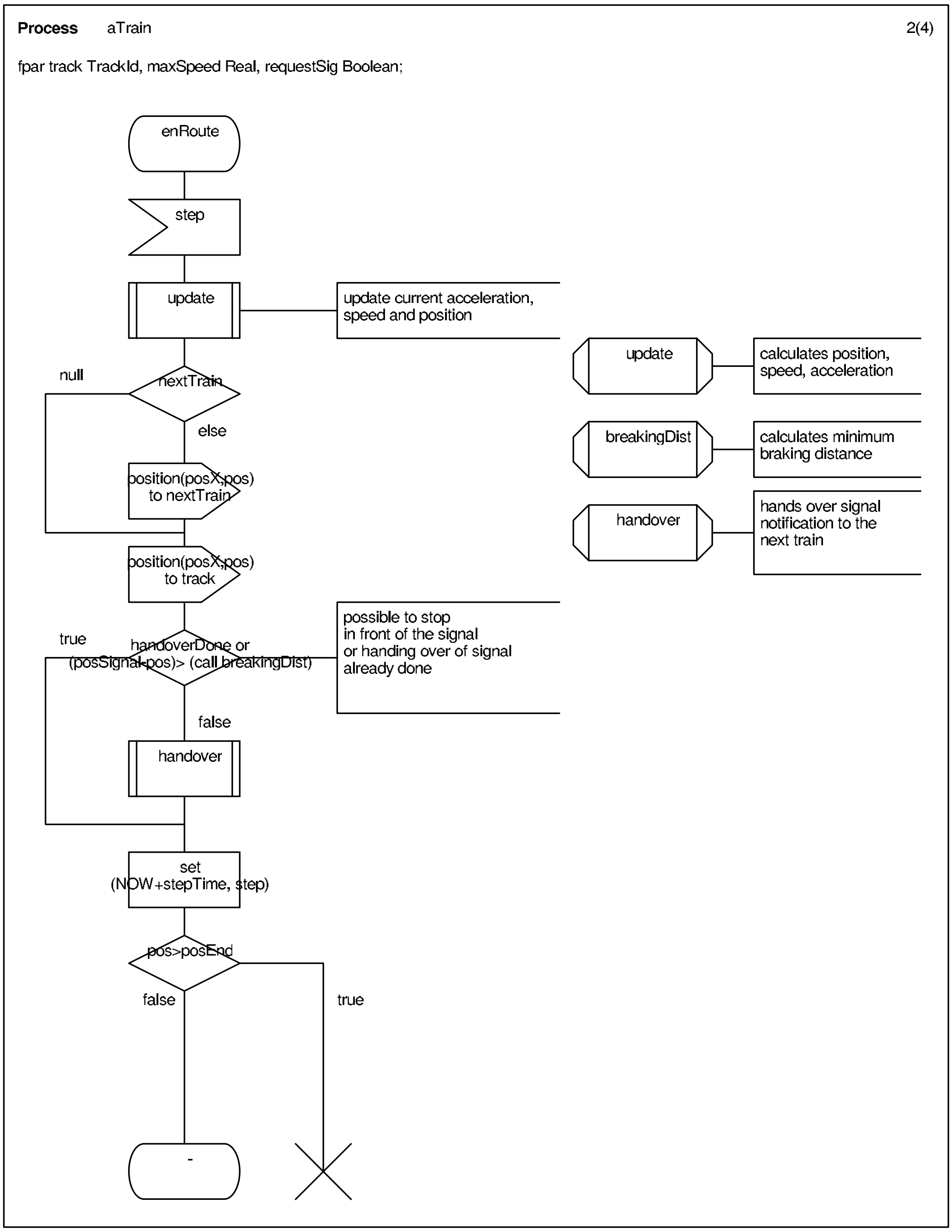
fpar track TrackId, maxSpeed Real, requestSig Boolean;

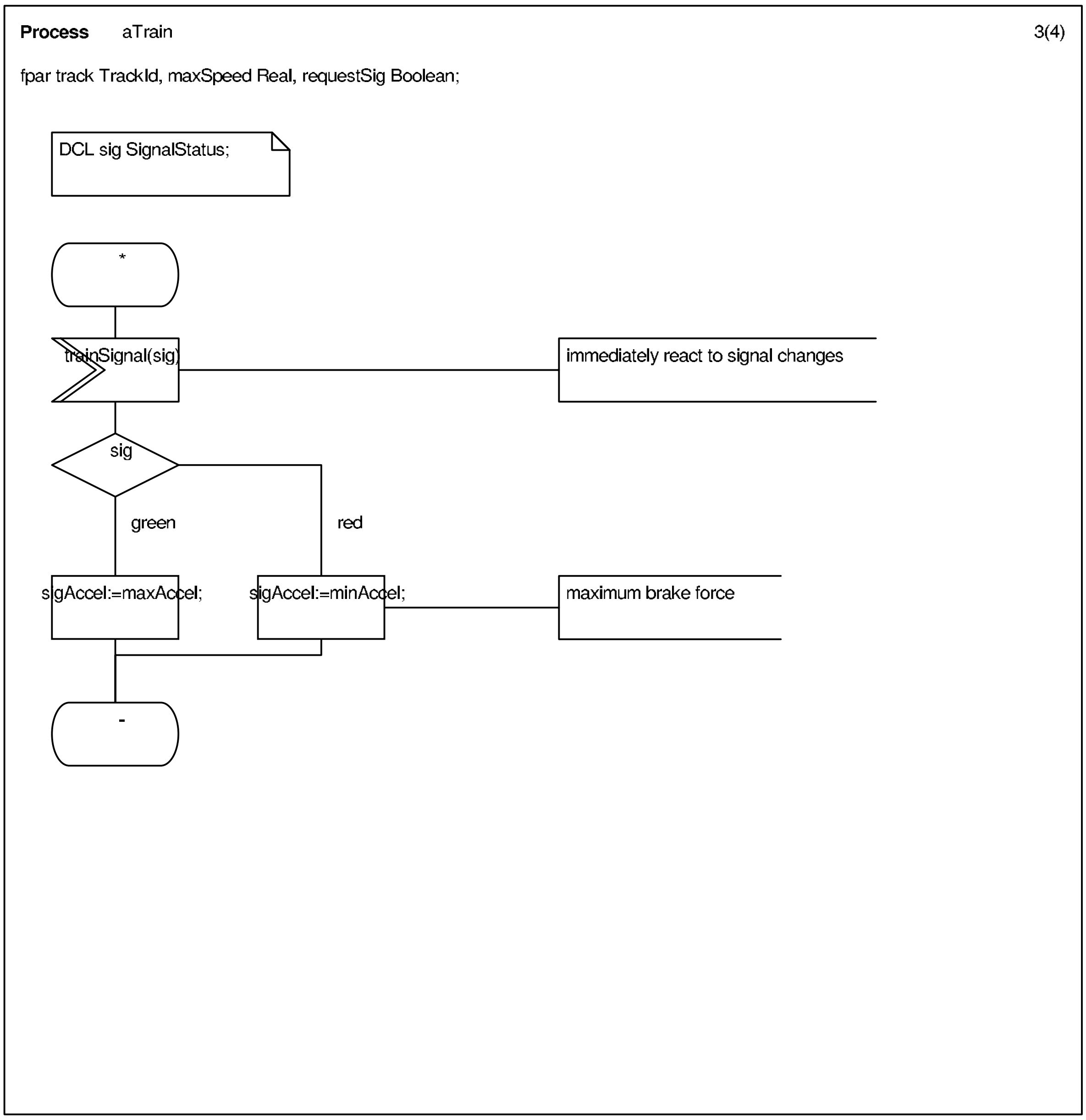


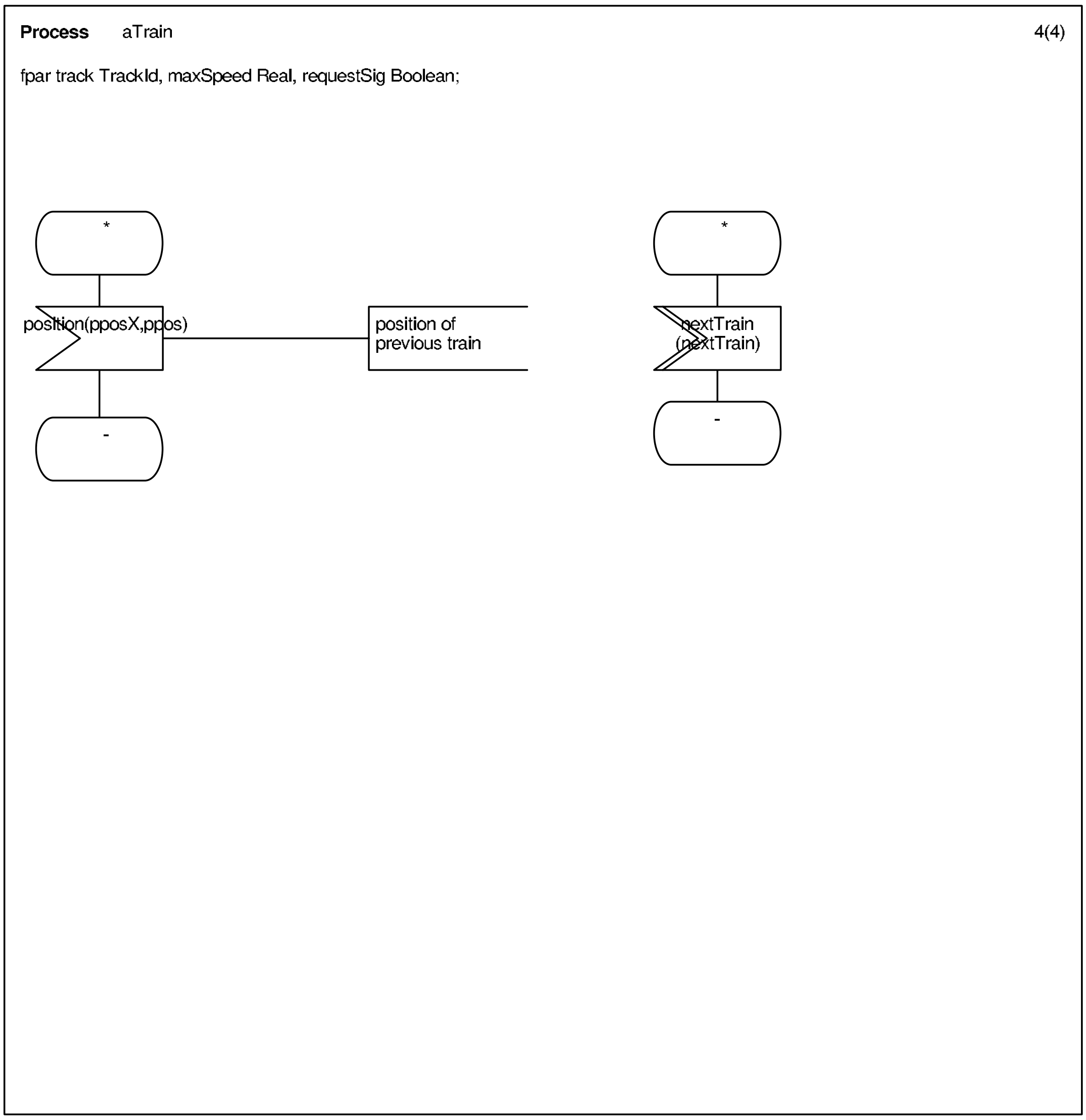
```
synonym maxAccel Real=10;
        /* maximum acceleration */
synonym minAccel Real=-20;
        /* maximum brake acceleration */
synonym minSpeed Real=0;
        /* minimum speed */
synonym minDist Real=50;
        /* minimum distance between the trains */
```

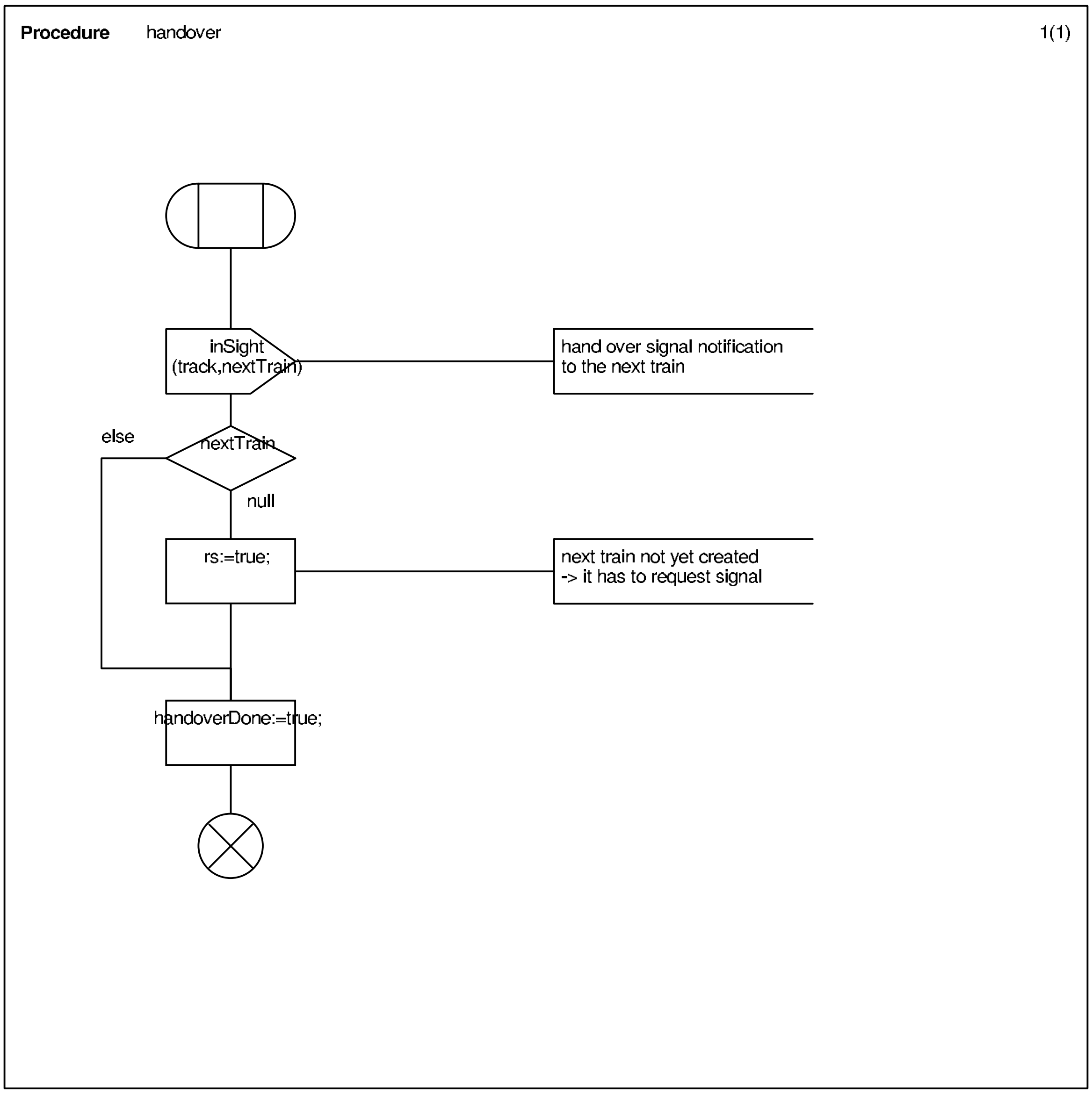
```
/* extend input alphabet */

signalset position;
        /* inter train communication */
```

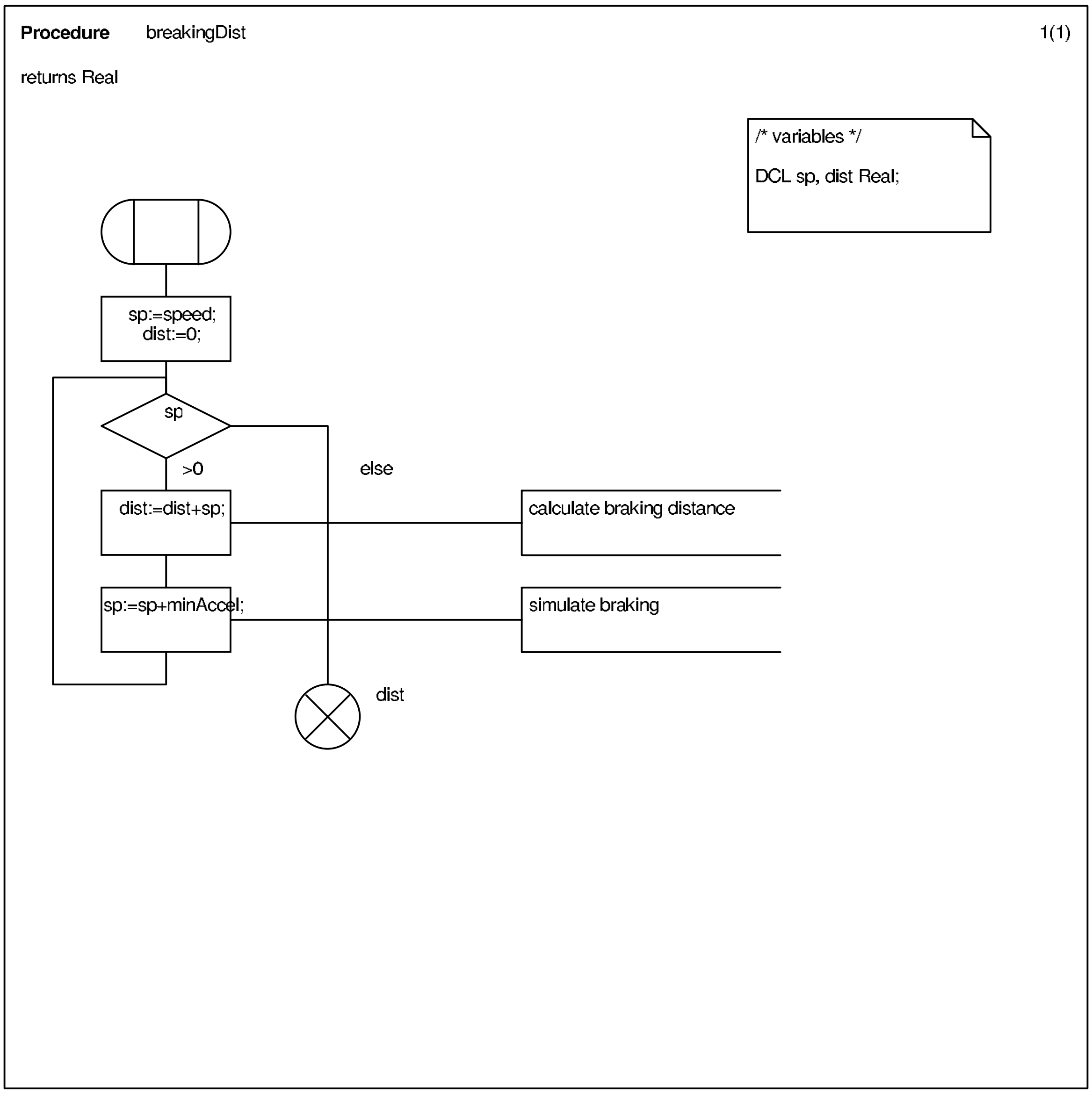
```
/* timer */

timer step;
DCL lastStep Time;
synonym stepTime Duration=10;

/* variables */

DCL pos, posX Real;
        /* current position */
DCL pposX, ppos Real :=-1;
        /* current position of previous train */
DCL speed Real;
        /* current speed */
DCL accel Real;
        /* current acceleration */
DCL sigAccel Real:=maxAccel;
        /* maximum acceleration permitted by the signal */
DCL nextTrain PId:=null;
        /* next train */
DCL rs Boolean:=false;
        /* next train has to request signal */
DCL handoverDone Boolean:=false;
        /* handing over of signal notification done */
```

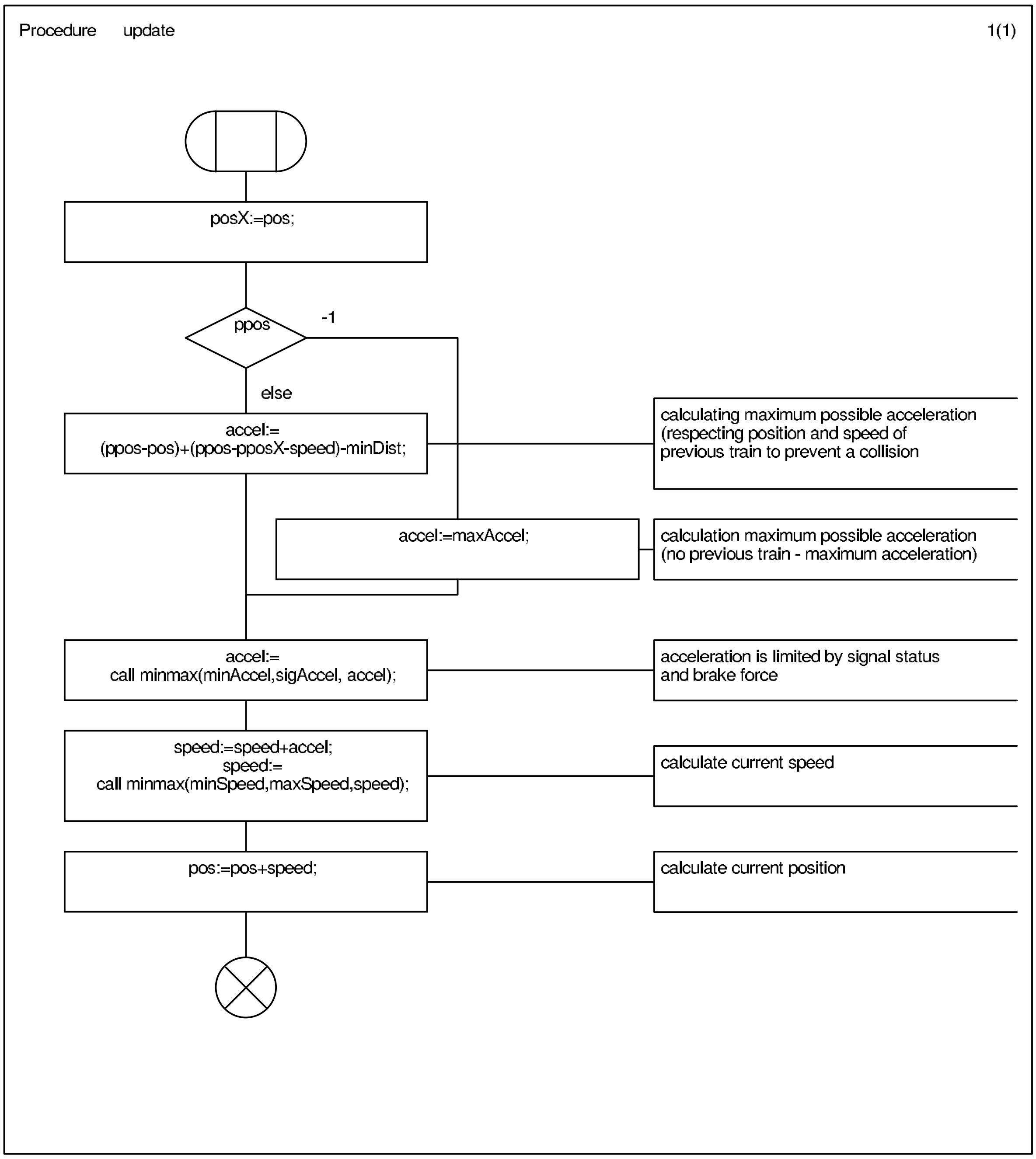

Process aTrain
2(4)
fpar track TrackId, maxSpeed Real, requestSig Boolean;
enRoute
step
update
update current acceleration, speed and position
null
nextTrain
else
position(posX,pos) to nextTrain
position(posX,pos) to track
true
handoverDone or (posSignal-pos)> (call breakingDist)
possible to stop in front of the signal or handing over of signal already done
false
handover
set (NOW+stepTime, step)
pos>posEnd
false
true
-
update
calculates position, speed, acceleration
breakingDist
calculates minimum braking distance
handover
hands over signal notification to the next train

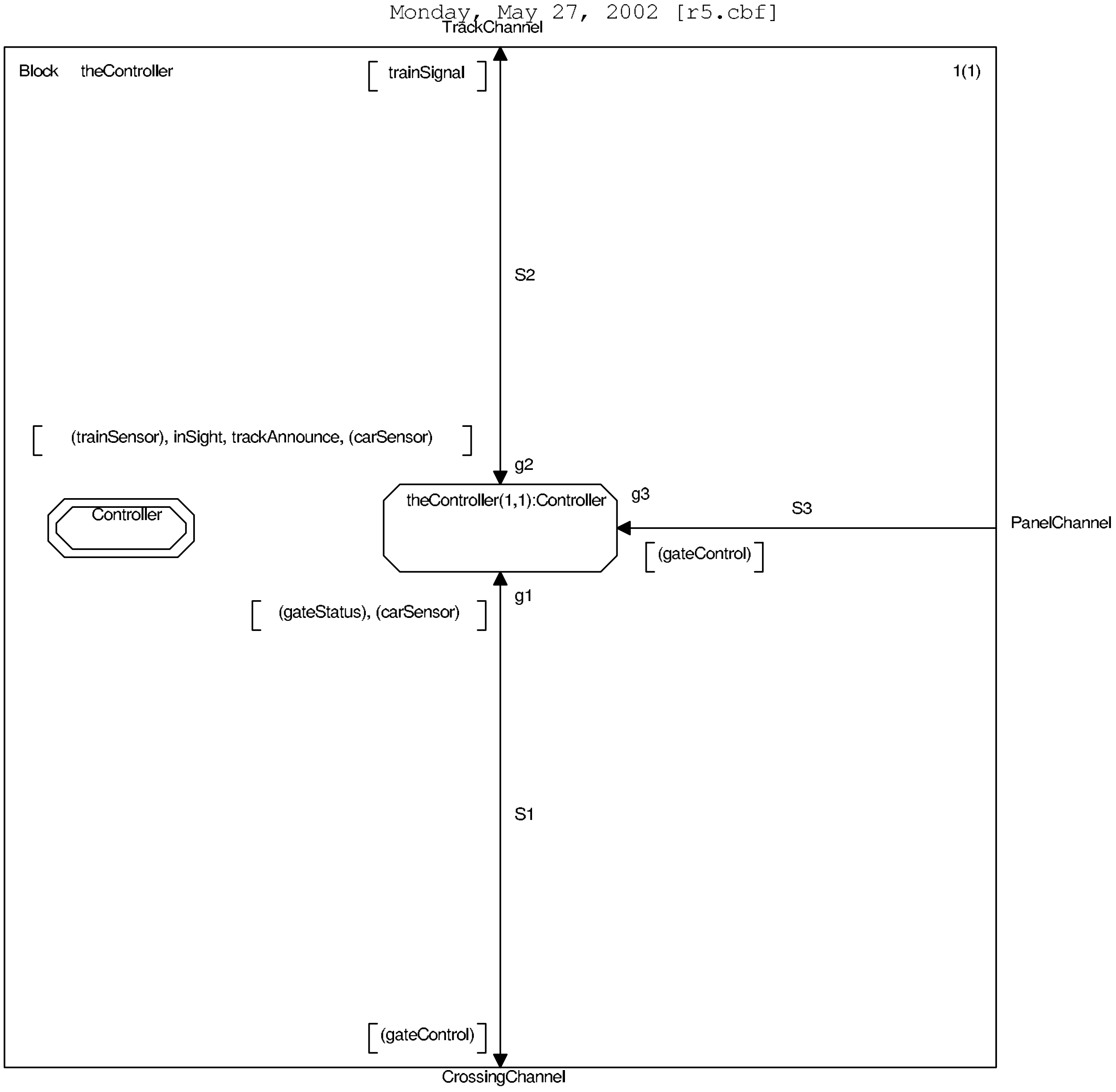

Process aTrain
3(4)
fpar track TrackId, maxSpeed Real, requestSig Boolean;
DCL sig SignalStatus;
*
trainSignal(sig)
immediately react to signal changes
sig
green
red
sigAccel:=maxAccel;
sigAccel:=minAccel;
maximum brake force
-

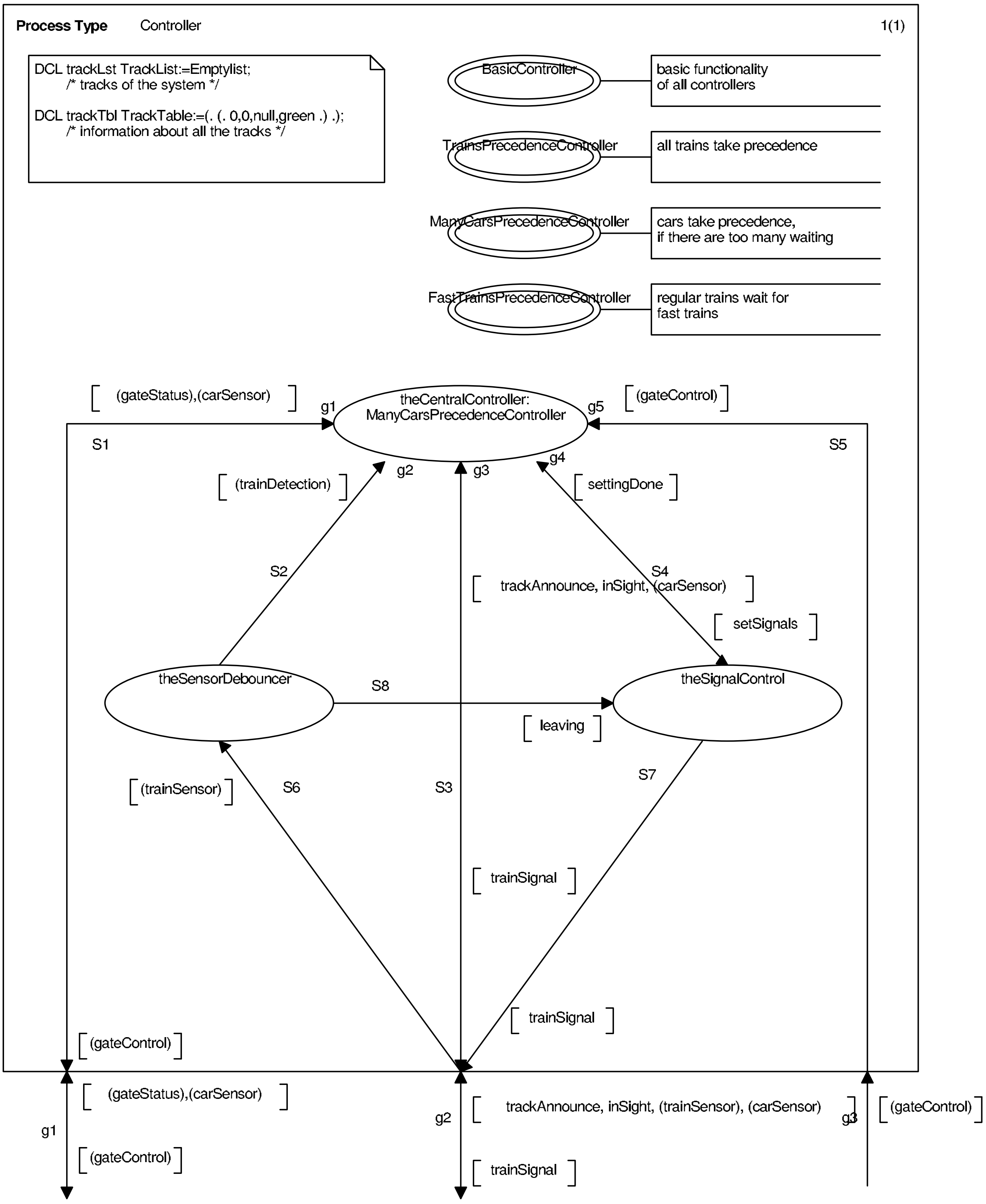

Process aTrain
4(4)
fpar track TrackId, maxSpeed Real, requestSig Boolean;
*
position(pposX,ppos)
position of previous train
-
*
nextTrain (nextTrain)
-


Procedure handover
1(1)
inSight (track,nextTrain)
hand over signal notification to the next train
else
nextTrain
null
rs:=true;
next train not yet created -> it has to request signal
handoverDone:=true;

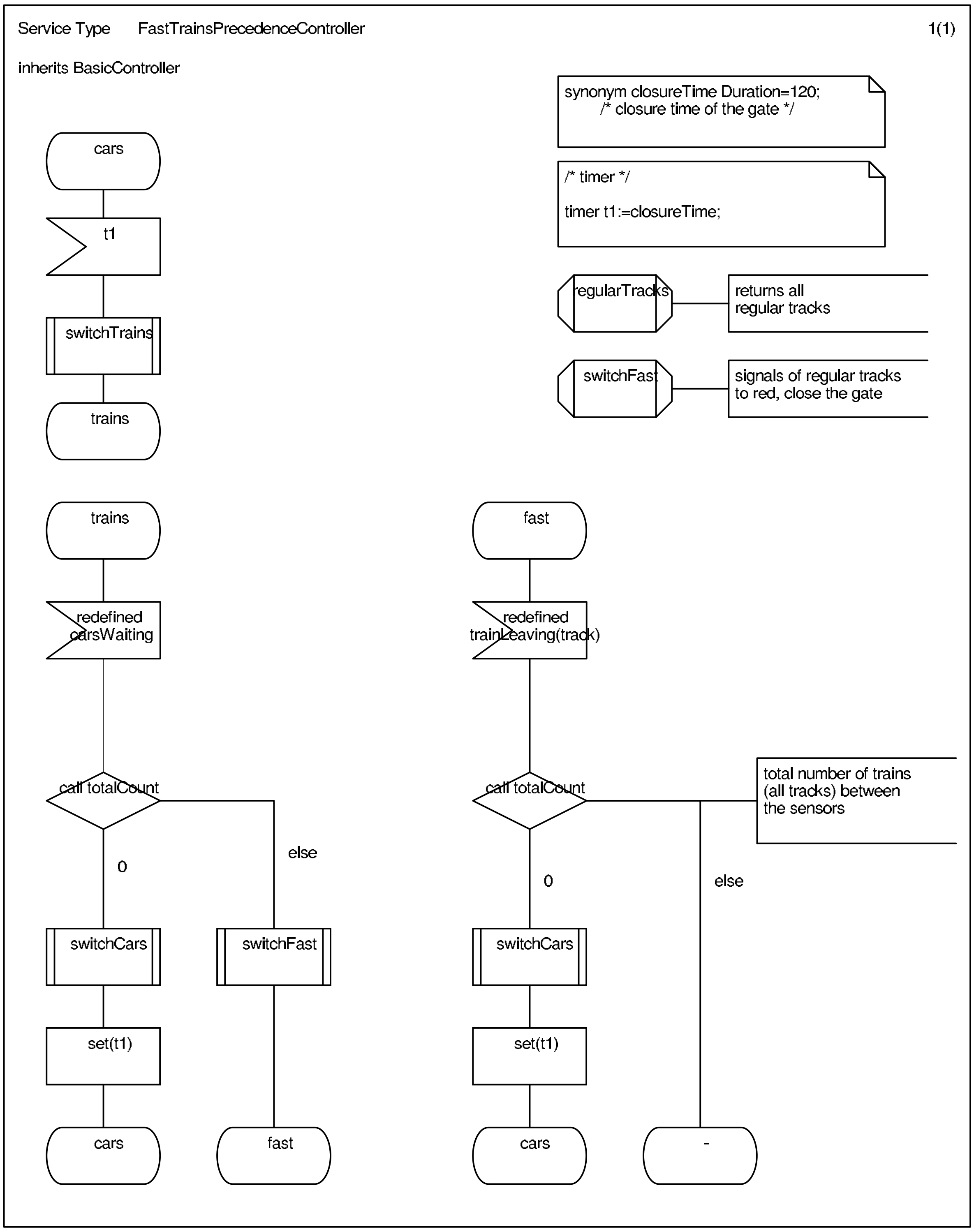

Procedure breakingDist
1(1)
returns Real
/* variables */
DCL sp, dist Real;
sp:=speed;
dist:=0;
sp
>0
else
dist:=dist+sp;
calculate braking distance
sp:=sp+minAccel;
simulate braking
dist

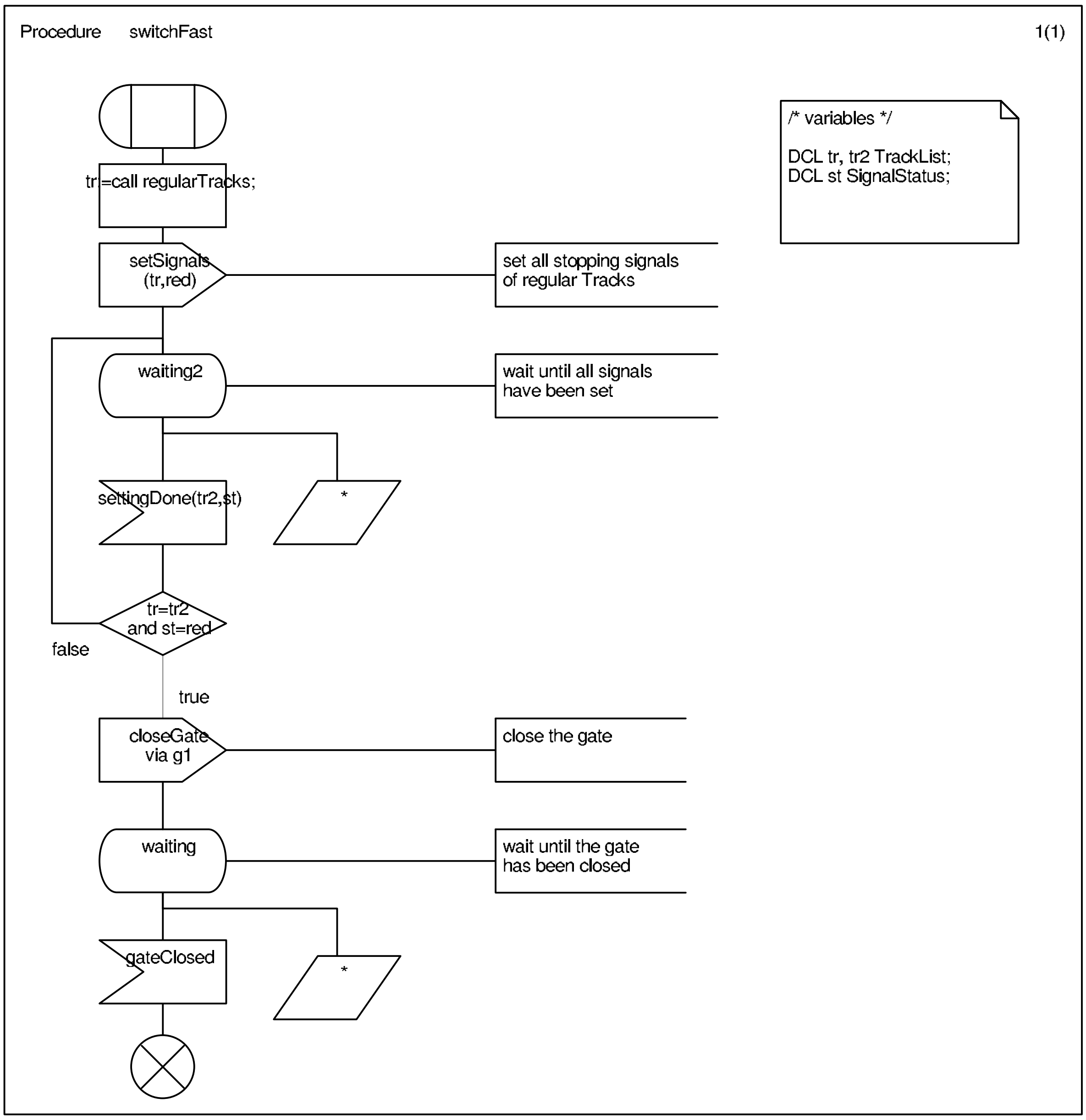

Procedure update
1(1)
posX:=pos;
ppos
-1
else
accel:= (ppos-pos)+(ppos-pposX-speed)-minDist;
calculating maximum possible acceleration (respecting position and speed of previous train to prevent a collision
accel:=maxAccel;
calculation maximum possible acceleration (no previous train - maximum acceleration)
accel:= call minmax(minAccel,sigAccel, accel);
acceleration is limited by signal status and brake force
speed:=speed+accel; speed:= call minmax(minSpeed,maxSpeed,speed);
calculate current speed
pos:=pos+speed;
calculate current position

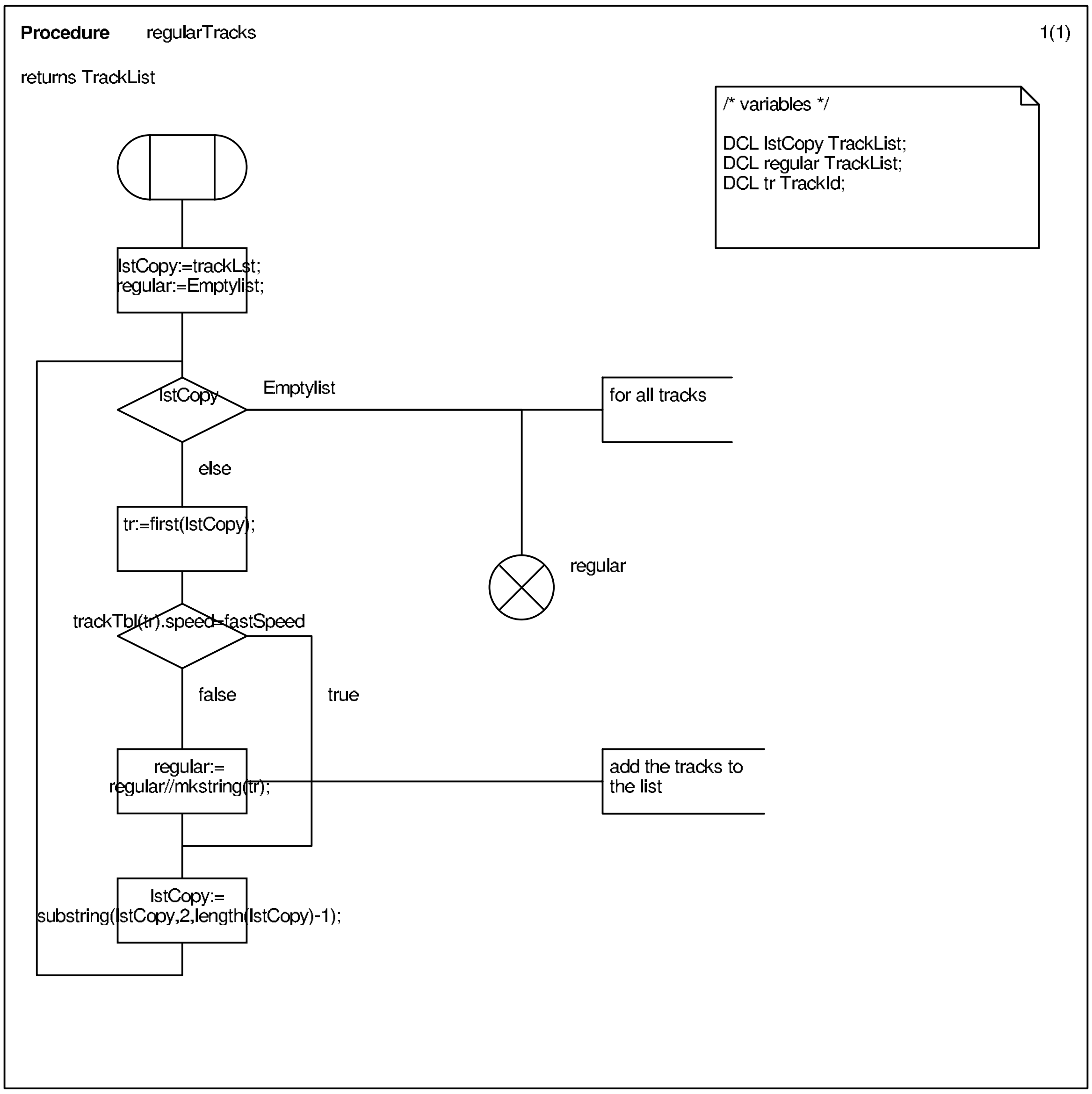

TrackChannel
Block theController
[trainSignal]
1(1)
S2
[(trainSensor), inSight, trackAnnounce, (carSensor)]
g2
Controller
theController(1,1):Controller
g3
S3
PanelChannel
[(gateControl)]
g1
[(gateStatus), (carSensor)]
S1
[(gateControl)]
CrossingChannel

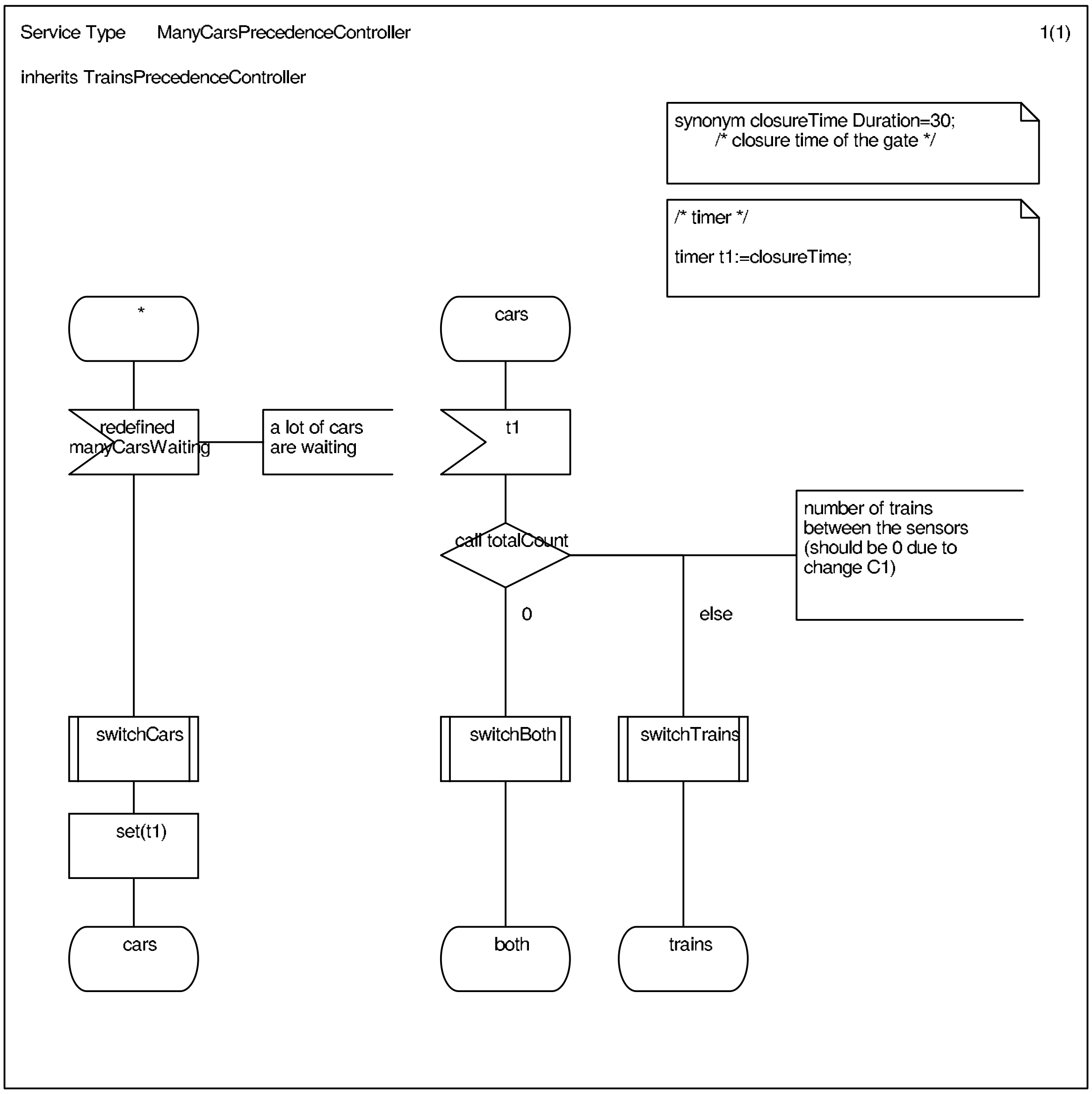


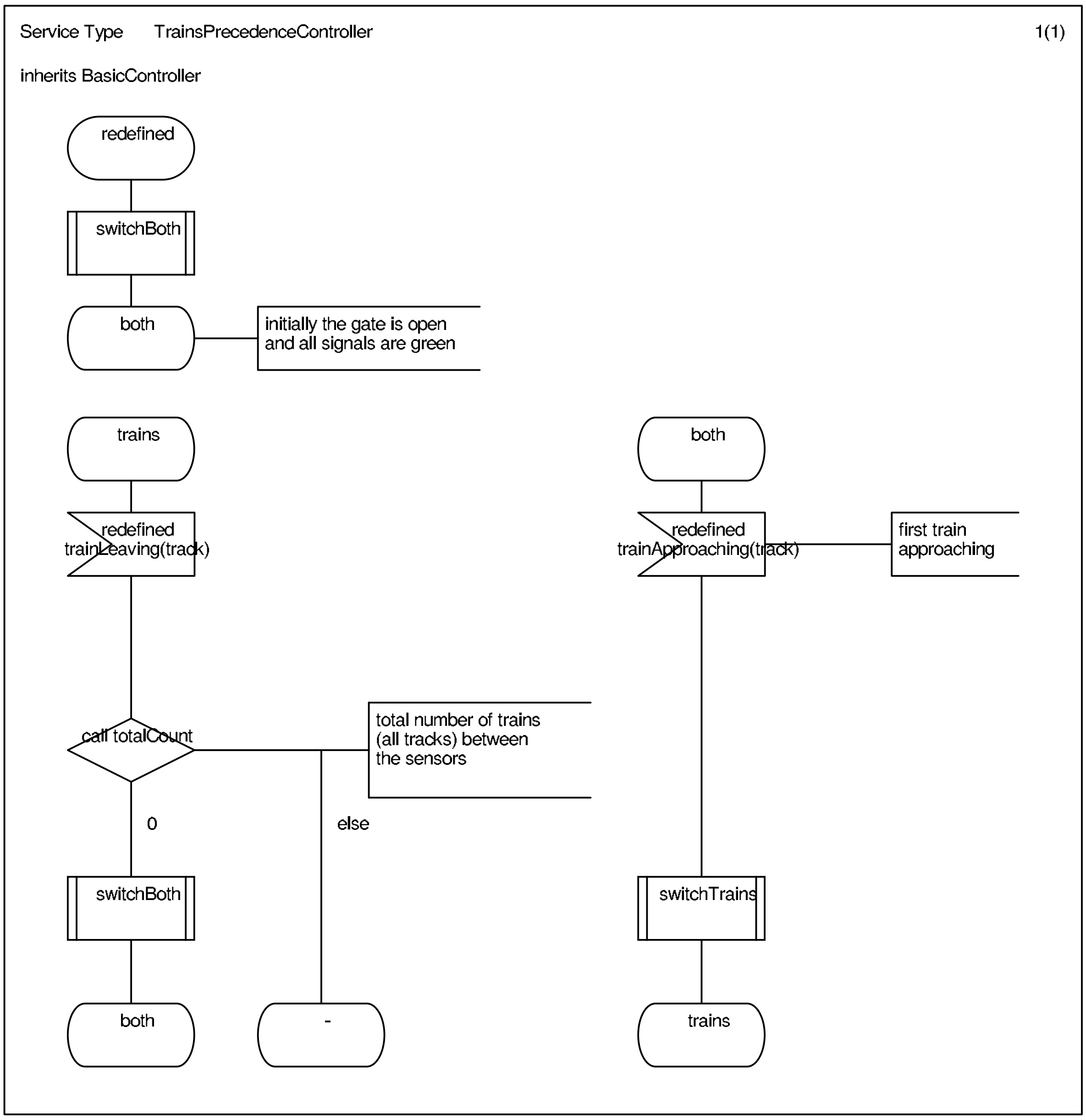
Process Type Controller
1(1)
DCL trackLst TrackList:=Emptylist;
/* tracks of the system */
DCL trackTbl TrackTable:=(. (. 0,0,null,green .) .);
/* information about all the tracks */
BasicController
basic functionality of all controllers
TrainsPrecedenceController
all trains take precedence
ManyCarsPrecedenceController
cars take precedence, if there are too many waiting
FastTrainsPrecedenceController
regular trains wait for fast trains
theCentralController: ManyCarsPrecedenceController
[(gateStatus),(carSensor)]
[(gateControl)]
[(trainDetection)]
[settingDone]
[trackAnnounce, inSight, (carSensor)]
[setSignals]
theSensorDebouncer
theSignalControl
[leaving]
[(trainSensor)]
[trainSignal]
[trainSignal]
[(gateControl)]
S1 S2 S3 S4 S5 S6 S7 S8
g1 g2 g3 g4 g5
[(gateStatus),(carSensor)]
[(gateControl)]
[trackAnnounce, inSight, (trainSensor), (carSensor)]
[trainSignal]
[(gateControl)]
g1 g2 g3

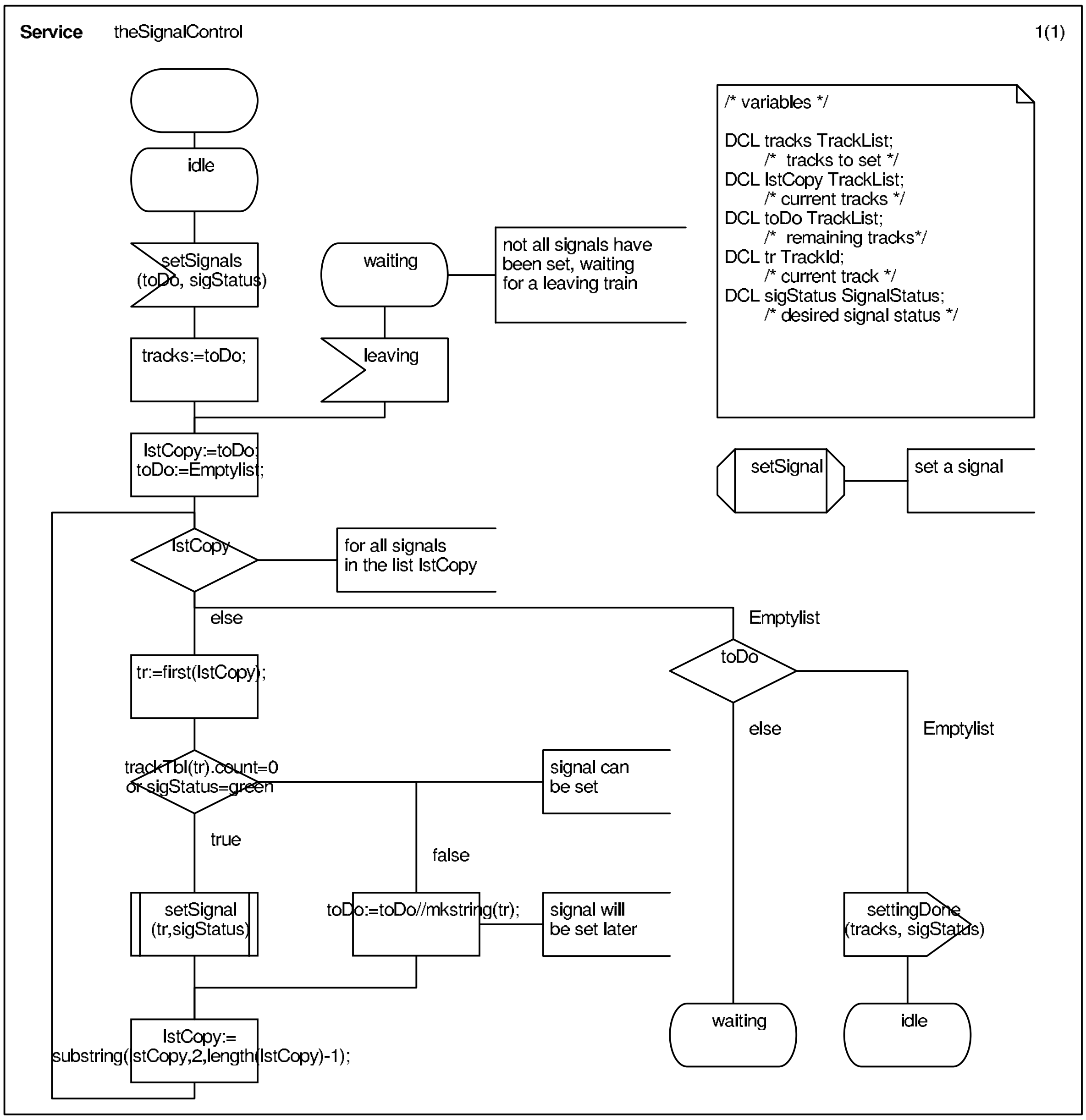

Service Type FastTrainsPrecedenceController
1(1)
inherits BasicController
synonym closureTime Duration=120;
/* closure time of the gate */
/* timer */
timer t1:=closureTime;
regularTracks
returns all regular tracks
switchFast
signals of regular tracks to red, close the gate
cars
t1
switchTrains
trains
trains
redefined carsWaiting
call totalCount
0
else
switchCars
set(t1)
cars
switchFast
fast
fast
redefined trainLeaving(track)
call totalCount
total number of trains (all tracks) between the sensors
0
else
switchCars
set(t1)
cars
-


Procedure switchFast
1(1)
tr:=call regularTracks;
setSignals (tr,red)
set all stopping signals of regular Tracks
waiting2
wait until all signals have been set
settingDone(tr2,st)
*
tr=tr2 and st=red
false
true
closeGate via g1
close the gate
waiting
wait until the gate has been closed
gateClosed
*
/* variables */
DCL tr, tr2 TrackList;
DCL st SignalStatus;


Procedure regularTracks
1(1)
returns TrackList
/* variables */
DCL lstCopy TrackList;
DCL regular TrackList;
DCL tr TrackId;
lstCopy:=trackList;
regular:=Emptylist;
lstCopy
Emptylist
for all tracks
else
tr:=first(lstCopy);
regular
trackTbl(tr).speed=fastSpeed
false
true
regular:=
regular//mkstring(tr);
add the tracks to
the list
lstCopy:=
substring(lstCopy,2,length(lstCopy)-1);

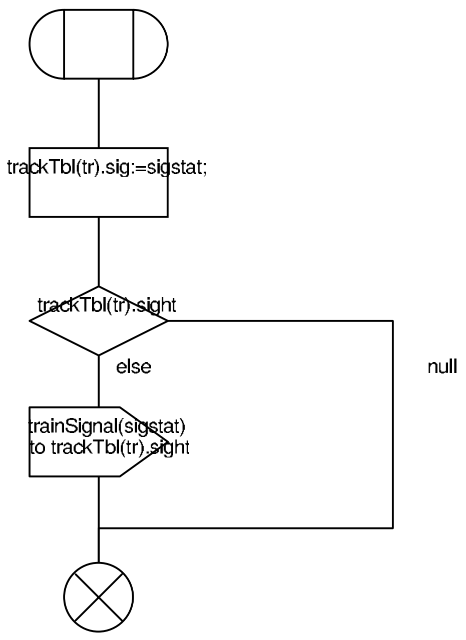

Service Type ManyCarsPrecedenceController
1(1)
inherits TrainsPrecedenceController
synonym closureTime Duration=30;
/* closure time of the gate */
/* timer */
timer t1:=closureTime;
*
redefined manyCarsWaiting
a lot of cars are waiting
switchCars
set(t1)
cars
cars
t1
call totalCount
number of trains between the sensors (should be 0 due to change C1)
0
else
switchBoth
switchTrains
both
trains

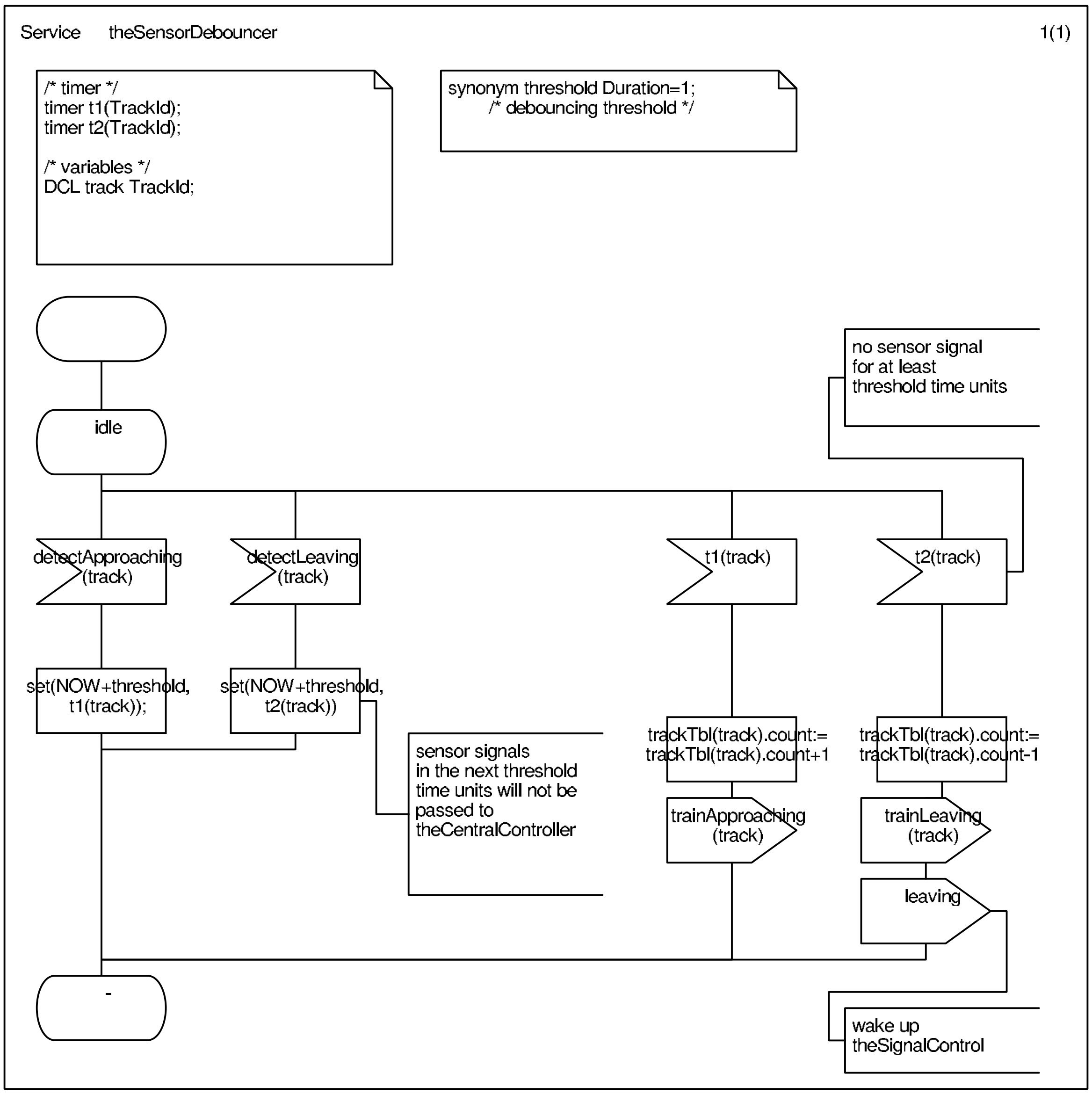

Service Type TrainsPrecedenceController
1(1)
inherits BasicController
redefined
switchBoth
both
initially the gate is open and all signals are green
trains
redefined trainLeaving(track)
call totalCount
total number of trains (all tracks) between the sensors
0
else
switchBoth
both
-
both
redefined trainApproaching(track)
first train approaching
switchTrains
trains

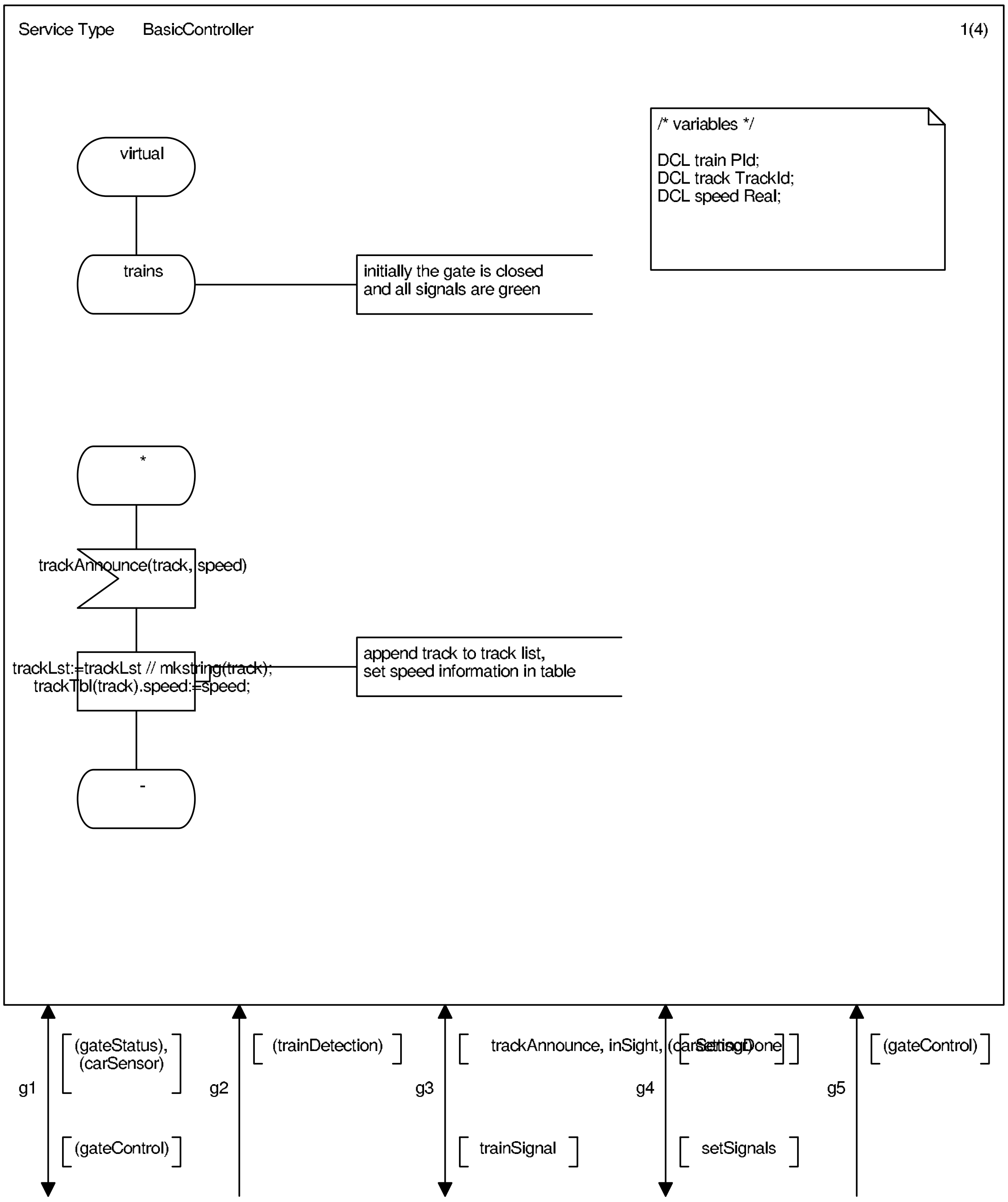

Service theSignalControl
1(1)
idle
setSignals
(toDo, sigStatus)
waiting
not all signals have
been set, waiting
for a leaving train
tracks:=toDo;
leaving
/* variables */
DCL tracks TrackList;
/* tracks to set */
DCL lstCopy TrackList;
/* current tracks */
DCL toDo TrackList;
/* remaining tracks*/
DCL tr TrackId;
/* current track */
DCL sigStatus SignalStatus;
/* desired signal status */
lstCopy:=toDo;
toDo:=Emptylist;
setSignal
set a signal
lstCopy
for all signals
in the list lstCopy
else
Emptylist
toDo
tr:=first(lstCopy);
else
Emptylist
trackTbl(tr).count=0
or sigStatus=green
signal can
be set
true
false
setSignal
(tr,sigStatus)
toDo:=toDo//mkstring(tr);
signal will
be set later
settingDone
(tracks, sigStatus)
lstCopy:=
substring(lstCopy,2,length(lstCopy)-1);
waiting
idle

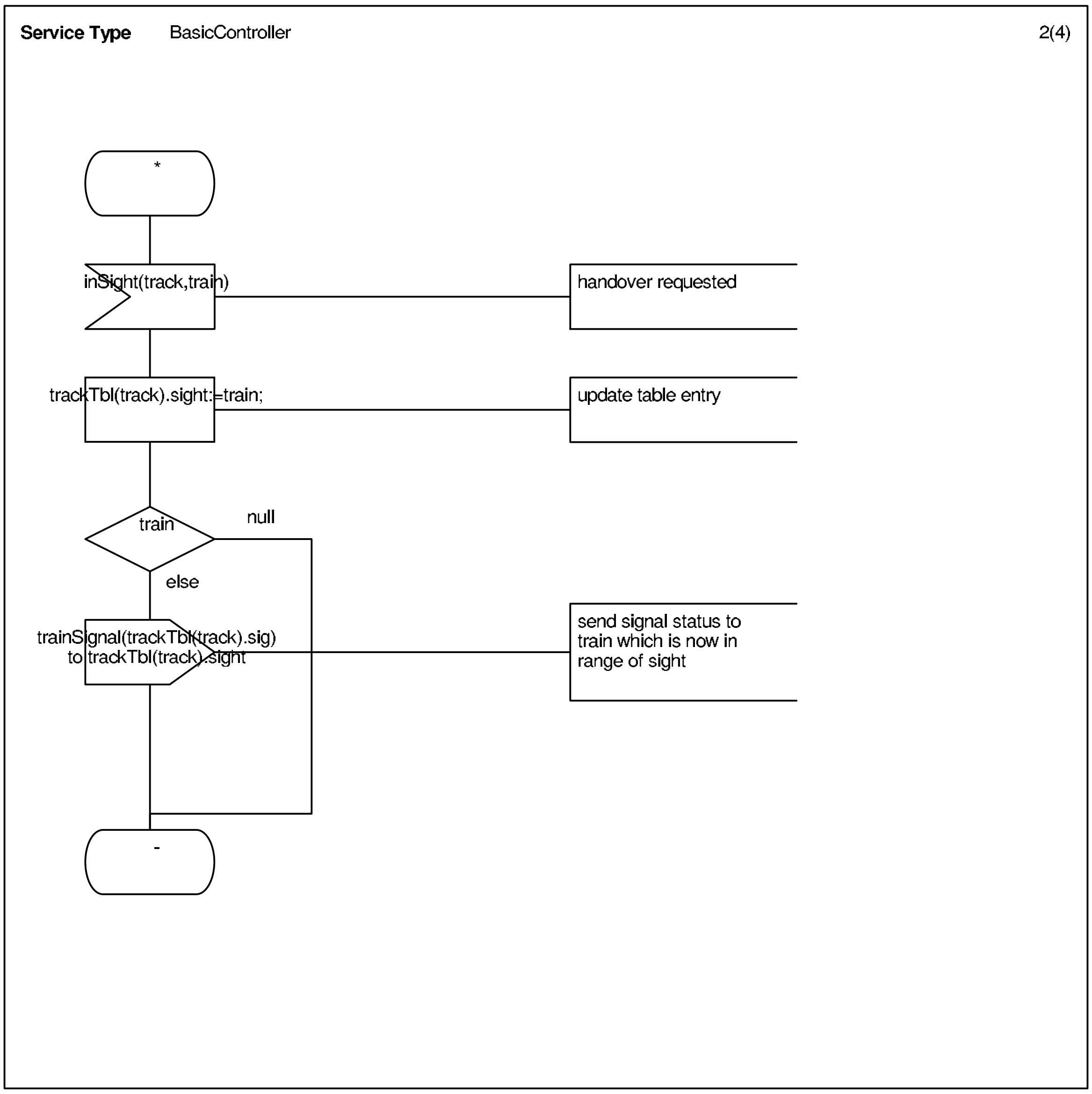
Procedure setSignal 1(1)

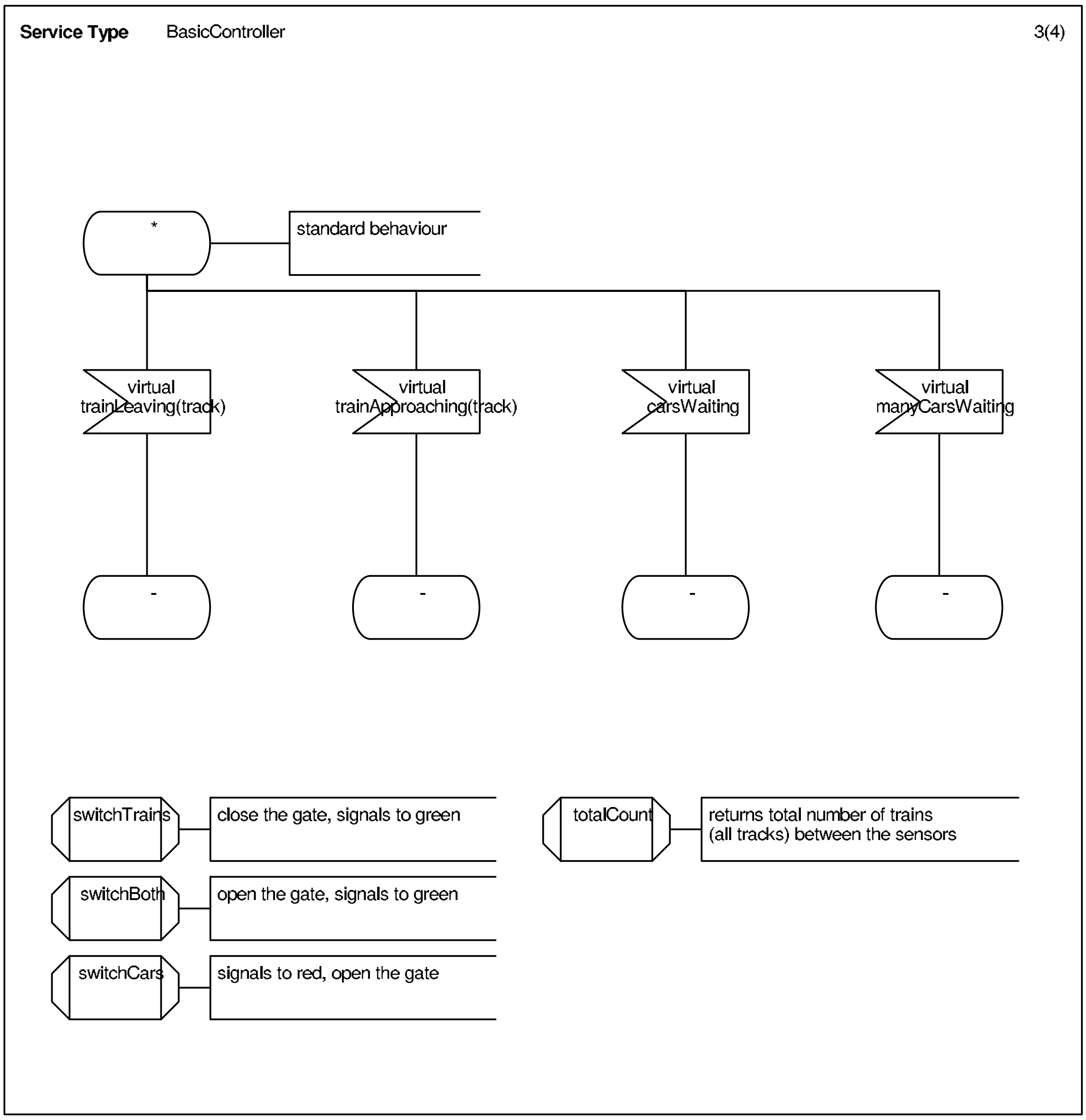
fpar tr TrackId, sigstat SignalStatus;

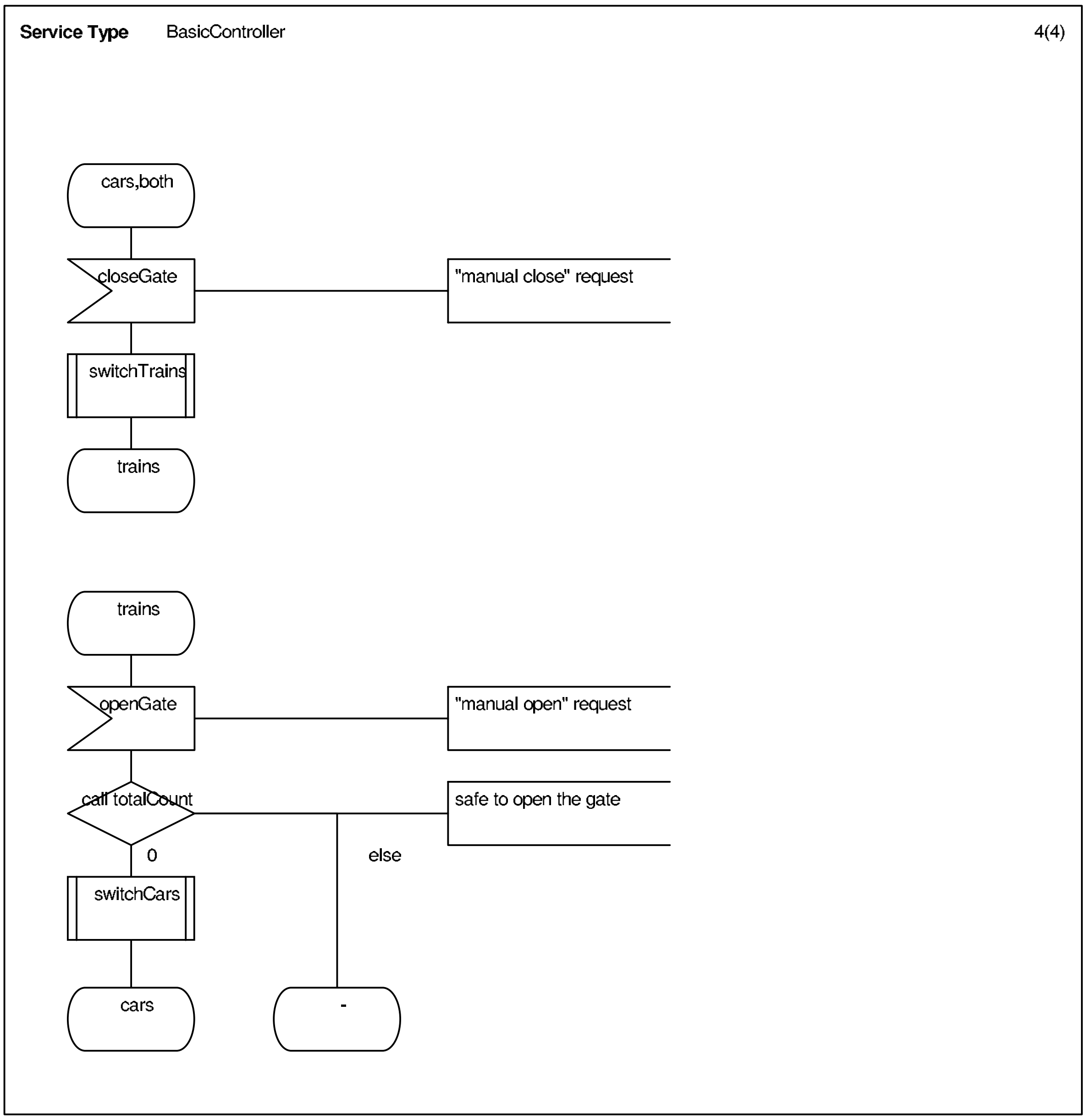


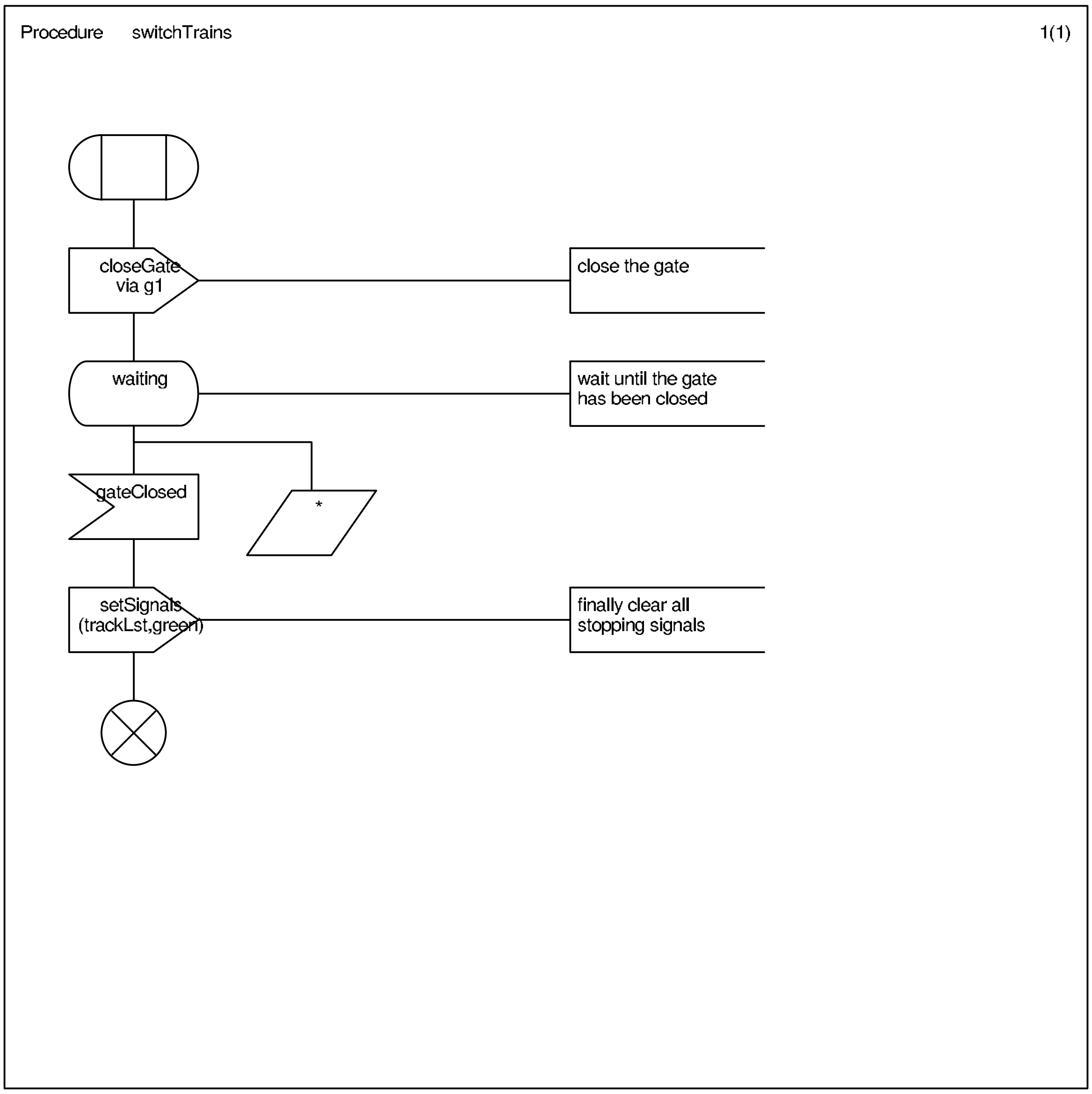

Service theSensorDebouncer
1(1)
/* timer */
timer t1(TrackId);
timer t2(TrackId);
/* variables */
DCL track TrackId;
synonym threshold Duration=1;
/* debouncing threshold */
no sensor signal for at least threshold time units
idle
detectApproaching (track)
detectLeaving (track)
t1(track)
t2(track)
set(NOW+threshold, t1(track));
set(NOW+threshold, t2(track))
sensor signals in the next threshold time units will not be passed to theCentralController
trackTbl(track).count:= trackTbl(track).count+1
trackTbl(track).count:= trackTbl(track).count-1
trainApproaching (track)
trainLeaving (track)
leaving
-
wake up theSignalControl

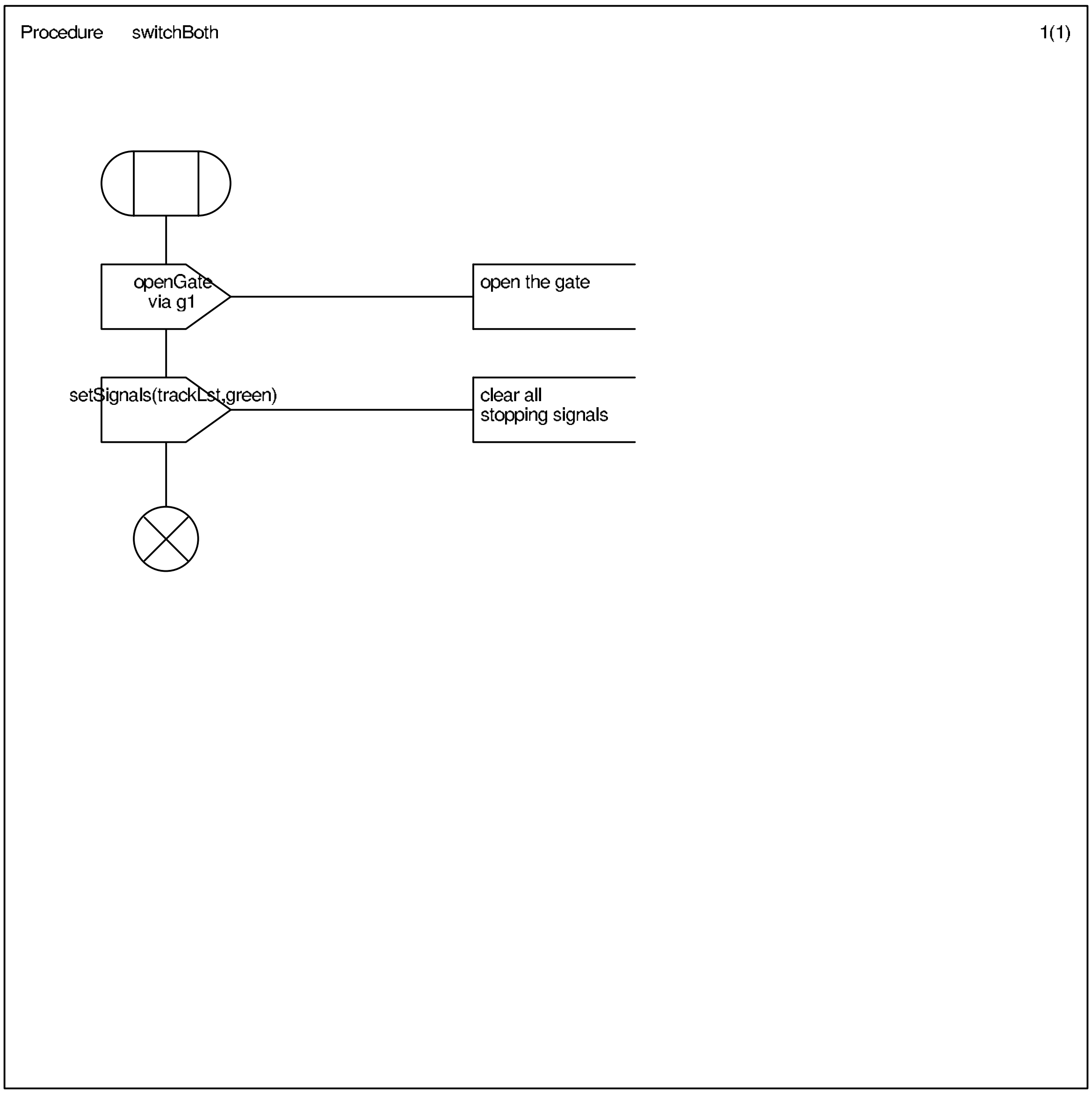

Service Type BasicController
1(4)
virtual
trains
initially the gate is closed and all signals are green
/* variables */
DCL train PId;
DCL track TrackId;
DCL speed Real;
*
trackAnnounce(track, speed)
trackLst:=trackLst // mkstring(track);
trackTbl(track).speed:=speed;
append track to track list, set speed information in table
-
g1
(gateStatus), (carSensor)
(gateControl)
g2
(trainDetection)
g3
trackAnnounce, inSight, (carSensor)
trainSignal
g4
SettingDone
setSignals
g5
(gateControl)

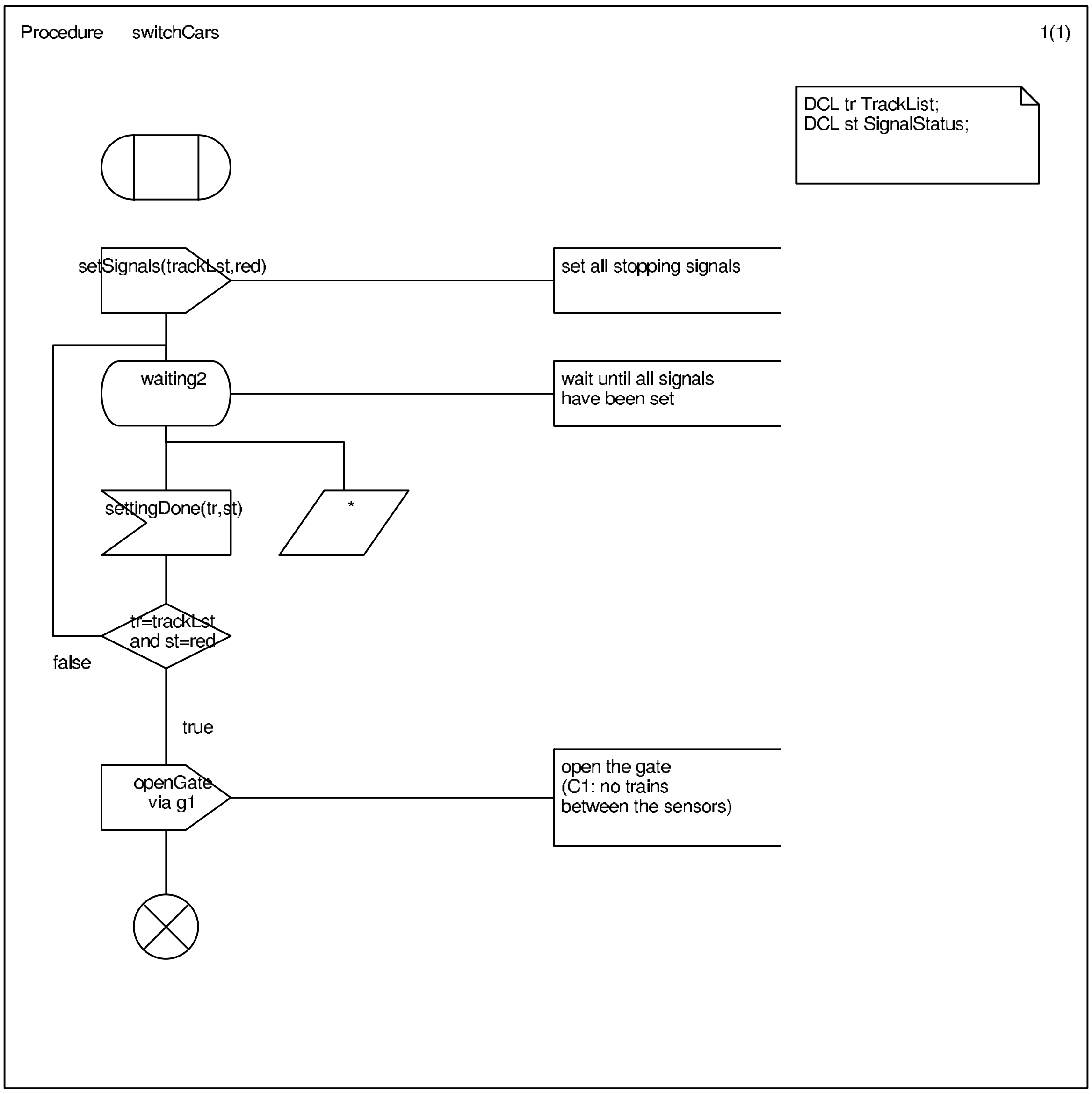

Service Type
BasicController
2(4)
*
inSight(track,train)
handover requested
trackTbl(track).sight:=train;
update table entry
train
null
else
trainSignal(trackTbl(track).sig)
to trackTbl(track).sight
send signal status to
train which is now in
range of sight
-

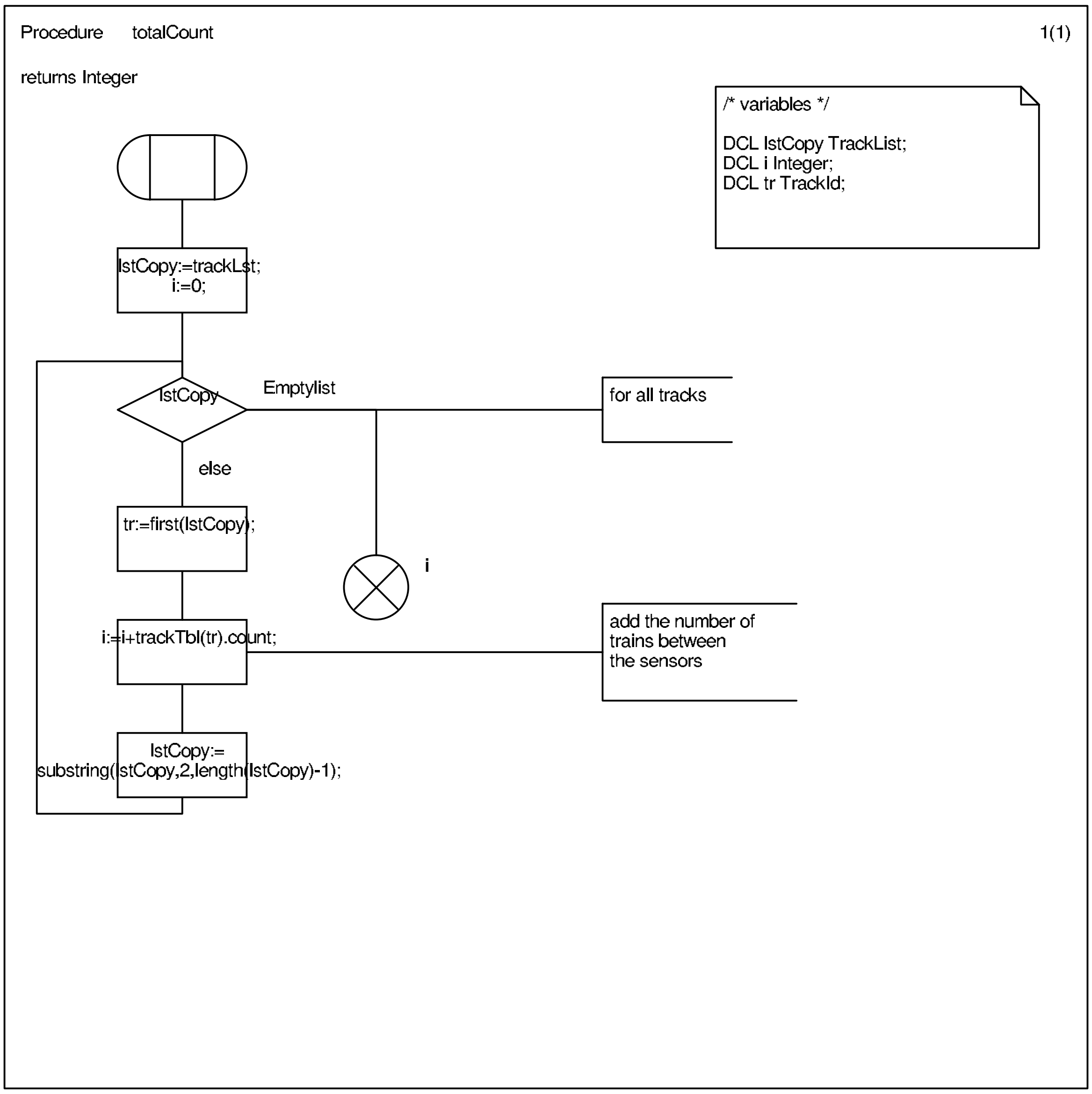

Service Type
BasicController
3(4)
*
standard behaviour
virtual trainLeaving(track)
virtual trainApproaching(track)
virtual carsWaiting
virtual manyCarsWaiting
-
-
-
-
switchTrains
close the gate, signals to green
switchBoth
open the gate, signals to green
switchCars
signals to red, open the gate
totalCount
returns total number of trains (all tracks) between the sensors


Service Type
BasicController
4(4)
cars,both
closeGate
"manual close" request
switchTrains
trains
trains
openGate
"manual open" request
call totalCount
safe to open the gate
0
else
switchCars
cars
-


Procedure switchTrains
1(1)
closeGate via g1
close the gate
waiting
wait until the gate has been closed
gateClosed
*
setSignals (trackLst,green)
finally clear all stopping signals


Procedure switchBoth
1(1)
openGate via g1
open the gate
setSignals(trackLst,green)
clear all stopping signals


Procedure switchCars
1(1)
DCL tr TrackList;
DCL st SignalStatus;
setSignals(trackLst,red)
set all stopping signals
waiting2
wait until all signals have been set
settingDone(tr,st)
*
tr=trackLst and st=red
false
true
openGate via g1
open the gate (C1: no trains between the sensors)


Procedure totalCount
1(1)
returns Integer
/* variables */
DCL lstCopy TrackList;
DCL i Integer;
DCL tr TrackId;
lstCopy:=trackList;
i:=0;
lstCopy
Emptylist
for all tracks
else
tr:=first(lstCopy);
i
i:=i+trackTbl(tr).count;
add the number of trains between the sensors
lstCopy:=
substring(lstCopy,2,length(lstCopy)-1);