Train control system

An example of a system specification and simulation using ASM and coreASM

Cocco Gabriele

The purpose of this example is to show the specification of a computerized system. The role of this system, called “train control system”, is to safely control trains crossing a track network.

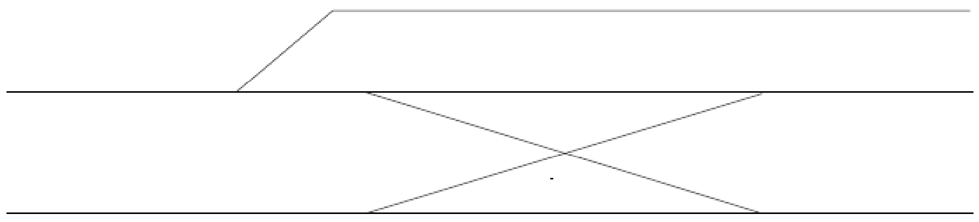
The example is taken from the Event-B book of J.R Abrial. To make the example runnable on coreAsm we introduced some modifications and refinements compared with the original specification. For example, in the original specification it is not specified how the system detects that a train has left or entered a route. In this case, we introduced the concept of sensors, located in particular points of the network.

All the details of the original specification of the train control system can be found in **[1]**. In the following pages we summarize of the most important requirements of the system and show the specification written as ASM **[2]** and simulated using coreASM **[3]**.

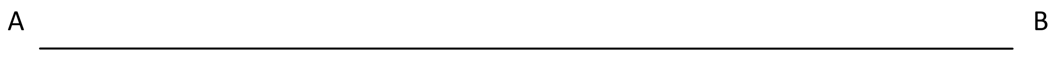
1. Description of the system

1.1. Network and tracks

The figure below shows a typical network.



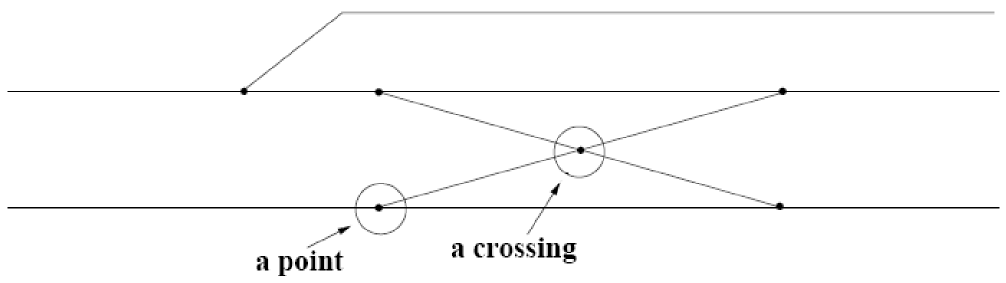
A network is composed by multiple tracks. *Every track is composed by two routes*, which represent the directions that a train can follow while travelling on the track. For example, the following track is composed by two routes: the route "from A to B" and the route "from B to A".



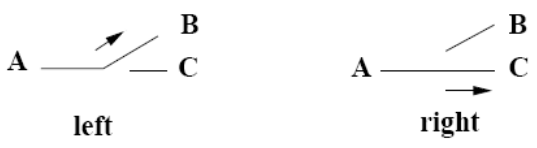
1.2. Special components

A network contains also a number of special components: these are the *points and the crossings* as illustrated in the following figure**.**

A point is a device allowing a track to split into two distinct directions. A crossing is a device that makes two different tracks crossing each other.



A point component can be in two different positions: *left or right*. This is illustrated in the following figure.

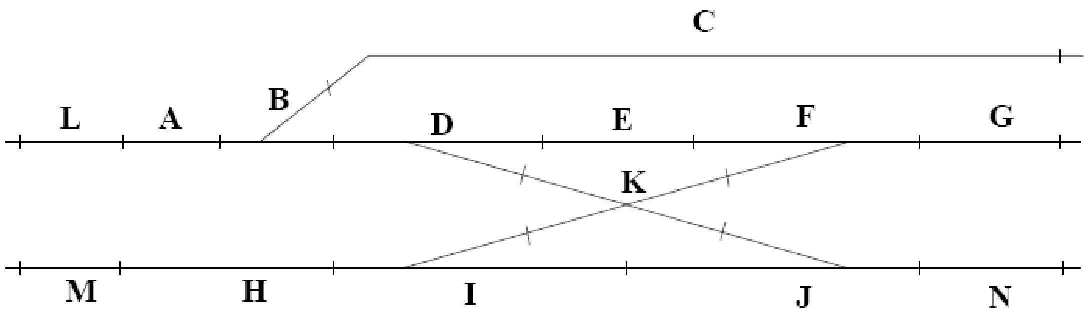


We suppose, as a simplification, that a point moves instantaneously from its current position to the other one. In other words, the unknown position, thinkable as the position of a point that is moving, is not treated.

A crossing component is completely static: it has no state as points.

1.3. Blocks , routes and sensors

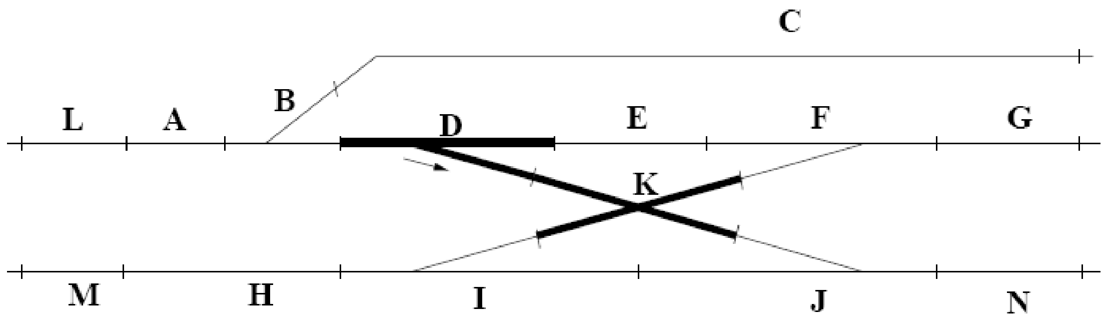
The controlled network is statically divided into a fixed number of blocks as indicated in the following figure where we have 14 blocks named by single letters from A to N:



Each block may contain at most one special component (points or crossings).

A block can be *free* (thus, can be reserved for a train), *committed* (reserved for a train which has not yet traveled over the block) and *occupied*. A block is said to be *reserved* if it is not free. In the original example the a block reserved for a train but not occupied is called “unoccupied”. We use the term “committed” because it seems to be more intuitive.

In the following figure, you can see that a train is occupying the two adjacent blocks D and K. Suppose that the train is going from L to N. The blocks L, A and B are free, because the train has already traveled over them. Therefore, these blocks can be reserved for another train. The Blocks D and K are occupied, because the train is currently over them. The blocks J and N are committed, because the train is not currently over them, but it will be, therefore J and N cannot be reserved for another train.



.

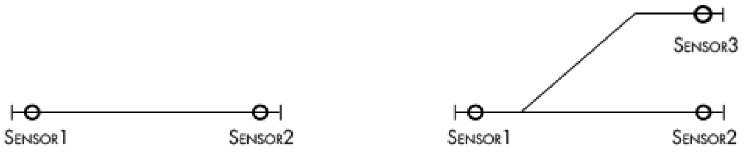
The blocks are always structured in a number of statically pre-defined routes. Each route represents a possible path that a train may follow within the network controlled by the train agent. In other words, the routes define the various ways a train can traverse the network. A route is composed of a number of adjacent blocks forming an ordered sequence.

A train following a route is supposed to occupy in turn each block of that route. Note that a train may occupy several adjacent blocks at the same time (the number of blocks occupied at the same time depends on the length of the train). Also note that a given block can be part of several routes.

A route cannot start where another route ends except when the second one shares the same set of blocks as the first one (so the first and the second one are the two routes composing the same track).
Another assumption is that a route contains no cycle.

A route can be *free, reserved or formed*. A route is free if all its blocks are free. As we explain in the section dedicated to the route reservation, a route becomes reserved when a train asks to travel on that route and the route can be reserved for the train. A route becomes formed after becoming reserved and after moving all its components in the right position. When the train arrives at the destination the route returns free.

Every block is equipped with a number of sensors. The sensors are used to detect a train that enters and exits a block. Each sensor is placed on a different bound of the block.
The following figures shows two examples of a single block with the relative sensors.



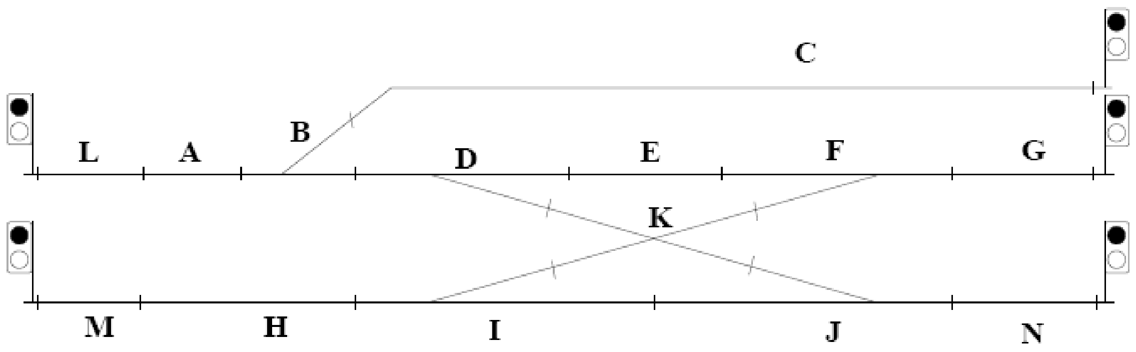
If we consider a block and a route, we can define a start-sensor and an end-sensor. The start-sensor/end-sensor is the first/last sensor of the block encountered when a train travels on that route.

Every sensor can produce two types of events: out-in and in-out events. An out-in event is produced as soon as the train reaches a sensor. An in-out event is produced as soon as the train leaves a sensor.
A block can be freed when the last sensor of the block, relative to the route the train is moving on, produces an in-out event.
A block can be set as occupied when the first sensor of the block, relative to the route the train is moving on, produces an out-in event.

1.4. Signals

Each route is protected by a signal, which can be *red or green*. This signal is situated just before the first block of each route. When a signal is red, then by convention the corresponding route cannot be used by an incoming train.

In the next figure, you can see the signals protecting each route.



Notice that a given signal can protect several routes. This is because each of these routes starts with the same block.

A last important property of a signal protecting the first block of a route is that, when green*, it turns back automatically to red as soon as a train enters the first block of the route*.

1.5. Route reservation

When a train is approaching the network, *the train agent is told that this train will cross the network* by using a certain route. The train agent *tries to reserve that route*. Note that other trains might already be crossing the network while the train agent is trying to reserve a route for a waiting train.

The reservation process of a route R consists of three phases:

1. The individual reservation of the various blocks composing route R is performed,
2. The positioning of the relevant points of route R is accomplished,
3. The turning to green of the signal protecting route R is done.

When the first phase is not possible the reservation fails and the two other phases are cancelled. In this case, the reservation has to be re-tried later by the train agent.

1) *Block reservation*. If all the blocks of the route which the train asked to travel on are free, the route can be reserved for that train. Therefore the route becomes reserved and all its blocks are marked as "not free but the train has not traveled over them", thus becomes committed.

This is expressed by the ASM *ReserveRoute(t, currentRoute(t))*, defined below, where t is the train selected for a reservation.

2) *Point positioning*. When the reservation of all blocks of a route R is successful, the reservation process proceeds with the second phase, namely the positioning of the corresponding points in the direction corresponding to the route R. When all points of R are properly positioned, the route is said to be *formed*. Note that a formed route remains reserved for the train.

This is expressed by the ASM *MoveSpecialComponent(currentRoute(t))*, defined below.

3) *Turning signal to green*. Once a route R is formed, the third and last phase of the reservation can be done; the signal controlling route R is turned green: a train can be accepted in it. A train driver, looking at the green signal, then leads the train within the reserved and formed route.

This is expressed by the ASM *GivePermission(t, currentRoute(t))*, defined below.

Therefore below we will define a train agent rule of the following form:

```
if not exists block in blocks(currentRoute(t)) with reserved(block) then
   ReserveRoute(t, currentRoute(t))
   MoveSpecialComponent(currentRoute(t))
   GivePermission(t, currentRoute(t))
```

We want to allow a large number of trains to be present in the network at the same time without danger. For this, we *allow each block of a reserved route to be freed* as soon as the train does not occupy it any more.

As a result, the only reserved blocks of a formed route are those blocks which are occupied by the train or those blocks of the route which are not yet occupied by the train. *A route remains formed as long as there are some reserved blocks in it*.

When no block of a formed route is reserved any more for that route, it means that the train has left the route, which can thus be made free.

1.6. Trains

We define three possible states for a train: *waiting, moving and stopped*. A waiting train is a train that requested to travel on a route and is waiting for the signal to turn green.
A moving train is a train that is travelling on a route. A stopped train is a train that is standing on a route. A train becomes stopped when it tries to move backward or enters the route before the signal protecting that route becomes green. The state of a train that exits the route (arrives at the destination) is undef (we are not interested in what a train does after the end of its travel).

1.7. Security

As several trains can cross the network at the same time and as the train agent is sometimes re-positioning a point when forming a route, there are clearly some risks of bad incidents. There are three main risks, which are the following:

1) Two (or more) trains traversing the network at the same time hit each other.
2) A point may change position under a train.
3) The point of a route may change position in front of a train using that route. In other words, the train has not yet occupied the block of this point but it will do so in the near future since that block is situated on that route.
The first risk (train hitting) is avoided by ensuring two safety conditions:

1) A given block can only be reserved for at most one route at a time.
2) The signal of a route is green only when the various blocks of that route are all reserved for it and are committed, and when all points of that route are set in the proper direction.

As a consequence several trains never occupy the same block at the same time, provided, of course, that train drivers do not overpass a red signal.

The second and third risks are avoided by ensuring that a point can only be moved when the corresponding block is that of a route which is reserved (all its blocks being reserved) but not yet formed.

We assume the a train cannot move backwards while in the network. This is so because in this case a freed block can be made occupied again.

The train system takes into account two abnormal cases which could happen. The first and most important case of failures is the one where, for some reasons, the driver of a train does not obey the red signal guarding a route.

This case is detected by a device called the *Automatic Train Protection*. As soon as this device detects that the train passes a red signal, it automatically activates the emergency brakes of the train.

Another case raised is that *a train cannot move backwards while in the network*. Here again, the Automatic Train Protection system is used. It detects immediately any backward move and in that case it activates automatically the emergency brakes. But one has to be sure that the train does not occupy again a block that it has freed recently. We assume that the backward-moving of a train is *detected by a sensor located at the beginning of each block*. When the sensor detects that a train is “entering” the sensor (production of an out-in event), the system controls the state of the block the sensor belongs to. Since the sensor is a start-sensor, relative to the route the block is reserved to, the production of an out-in event should signify that the train is entering the block. If the block is occupied the train has already “completely” entered the block (the back of the train passed the beginning of the block) and thus the train is moving backward. In this case the protection system is activated and the train is stopped. We assume that there is enough space between the beginning of the block and the first sensor to allow the train to be stopped before entering the preceding block.

2. The coreASM specification

2.1. General choices

An out-in event on a sensor is modeled by setting the value of a monitored function *inOutEvent(sensor)* to the value “true”. Similarly, an in-out event on a sensor is modeled by setting the value of a monitored function *outInEvent(sensor)* to the value “true”.

The set of trains that are waiting to travel on the network is modeled with a queue, called *waiting-queue*.
The request for the permission to enter a route is therefore modeled setting the state of the train to “waiting” and enqueuing it to the waiting-queue.
The usage of a queue permits requests of trains that cannot (at the moment) travel the route not to indefinitely delay.

2.2. Agents

The ASM specification is formed by three agents: the *train agent*, the *track control* and the *train simulator*.

2.2.1. The train agent

The role of the train agent is to listen to the requests of the trains that want to cross the network and try to reserve the routes to those trains.
The train agent, at every step, takes the first train in the waiting-queue and tries to reserve the route the train is waiting to travel on. If the reservation fails, the train is enqueued in the waiting-queue, thus the request of the train is delayed.

2.2.2. The track control

The track-control is the agent whose role is to perform the evolution (of the state) of the blocks that are parts of the routes the trains are travelling on. Therefore, the track control listens to the events produced by the sensors and, based on the type of the event and on the route the block is reserved for, updates the state of the blocks and, if it detects an abnormal situation (the train enters the first block and the signal is red, or the train is moving backward) it stops the train.

2.2.3. The train simulator

The train simulator is an agent whose role is to simulate the environment. In this system, the environment is formed by the sensors. The train-system listens to the events produced by the sensors and updates the state of the various blocks of the track-network. At the same time, based of the current state of the blocks, the system decides which trains can travel on the network.

Practically, the simulator has to produce events on the sensors forming the route a train in travelling on. To do this, we define, for every route *R* and every train *T*, the sequence of the sensors that must be considered for the production of the events when *T* travels on *R*. Moreover, for all the sensors we store in a location the last event produced. Initially, the last event produced by every sensor is undef. The first event that each sensor produces is obviously an out-in event (the train travels over the sensor). The second one is an in-out event. The third one is another out-in (probably caused by another train), and so on.

For each train *T* that is moving on the network, the simulator takes the route *R* the train is moving on, and the sequence of the events that must be produced then *T* is moving on *R*. Then it removes the first sensor from that sequence and produces the “next event” for that sensor. Therefore, if the last event produced by the sensor is undef or an in-out event, the simulator produces an out-in event, otherwise it produces an in-out event. The production of the event is modeled by setting the value of the location i*nOutEvent(sensor)* or *outInEvent(sensor)* to “true”.

Note that the sequence of sensors considered for the production of events when a particular train travels on a particular route express also the length of the train (relative to the lengths of the blocks of that route).

The last note regards the initial state of the system.

1. Every route if free
2. Every block is free
3. For each block, the route which the block is reserved for is undef
4. The *inOutEvent* and the *outInEvent* of all the sensors is set to false.
5. The last event produced by each sensor is undef
6. The state of each signal is red

3. The code

```
/* CoreASM specification */

CoreASM TrainSystem

use StandardPlugins
use TimePlugin
use MathPlugin
use ListPlugin
use QueuePlugin
use IOPlugin

universe Route = {route1, route2}
universe Block = {block1, block2}
universe SpecialComponent = {comp1}
universe Train = {train1, train2}
universe Signal = {signal1, signal2}
universe Sensor = {sensor1, sensor2, sensor3, sensor4}

enum TrainStatus = {WAITING, MOVING, STOPPED}
enum BlockStatus = {FREE, COMMITTED, OCCUPIED}
enum RouteStatus = {UNRESERVED, RESERVED, FORMED}
enum SignalStatus = {GREEN, RED}
enum SpecialComponentType = {CROSSING, POINT}
enum SpecialComponentPosition = {UP, DOWN}
enum EventType = {OUTIN, INOUT}

/**/
function currentRoute: Train -> Route
function currentTrain: Route -> Train
function blocks: Route -> LIST
function specialComponents: Route -> LIST
function currentPosition: SpecialComponent -> SpecialComponentPosition
function position: SpecialComponent * Route -> SpecialComponentPosition
function trainStatus: Train -> TrainStatus
function blockStatus: Block -> BlockStatus
function routeStatus: Route -> RouteStatus
function signalStatus: Signal -> SignalStatus
function type: SpecialComponent -> SpecialComponentType
function signal: Route -> Signal
function startSensor: Block * Route -> Sensor
function endSensor: Block * Route -> Sensor
function inOutEvent: Sensor -> BOOLEAN
function outInEvent: Sensor -> BOOLEAN
function lastEvent: Sensor -> EventType
function currentOwnerRoute: Block -> Route
function sensorsSequence: Train * Route -> LIST

//Functions used to simulate the time taken by trains to travel on the routes
function iteration: -> NUMBER
function lastIteration: -> NUMBER
/**/

universe Agents = {trackAgent, trainAgent, trainSimulator}

derived reserved(block) =
   not(blockStatus(block) = FREE)
```

```
derived firstBlock(route) =
   head(blocks(route))

derived lastBlock(route) =
   last(blocks(route))

init InitRule


rule InitRule =
   par
   //Define labels for debugging
   label(train1) := "1"
   label(train2) := "2"
   label(route1) := "1"
   label(route2) := "2"
   label(block1) := "1"
   label(block2) := "2"

   label(sensor1) := "1"
   label(sensor2) := "2"
   label(sensor3) := "3"
   label(sensor4) := "4"

   //Define the blocks of each route
   blocks(route1) := [block1, block2]
   blocks(route2) := [block2, block1]
   //Define the special components (poins and crossing) of each route
   specialComponents(route1) := []
   specialComponents(route2) := []
   //Define the signal of each route
   signal(route1) := signal1
   signal(route2) := signal2

   //Define the type of each components
   type(comp1) := POINT
   //Define the position of each component
   position(comp1) := UP
   //Define the position of each component for all the routes
   position(comp1, route1) := UP
   position(comp1, route2) := DOWN

   //Define the sensors of each route
   startSensor(block1, route1) := sensor1
   endSensor(block1, route1) := sensor2
   startSensor(block2, route1) := sensor3
   endSensor(block2, route1) := sensor4

   startSensor(block1, route2) := sensor4
   endSensor(block1, route2) := sensor3
   startSensor(block3, route2) := sensor2
   endSensor(block3, route2) := sensor1
   /*
    * Define the sequence considered in simulating the travelling
    * of a train on a route. The first time a sensor is dequeued
    * the simulator produces an OUTIN event regarding it. The
    * second time the simulator produces an INOUT event for that
    * sensor, and so on.
    * The production of events on the sensors permits to simulate
    * the travelling of a train
    */
   sensorsSequence(train1, route1) := [sensor1, sensor1, sensor2, sensor3,
sensor2, sensor3, sensor4, sensor4]
```

```
   sensorsSequence(train2, route2) := [sensor4, sensor4, sensor3, sensor2,
sensor3, sensor2, sensor1, sensor1]

   //Initialize the signals
   forall signal in Signal do
      signalStatus(signal) := RED

   //Initialize the sensors
   forall sensor in Sensor do
      par
         outInEvent(sensor) := false
         inOutEvent(sensor) := false
         lastEvent(sensor) := undef
      endpar

   //Initialize the blocks
   forall block in Block do
      par
         blockStatus(block) := FREE
         currentOwnerRoute(block) := undef
      endpar

   //Initialize the routes
   forall route in Route do
      routeStatus(route) := FREE

   //Initalize the trains
   trainStatus(train2) := WAITING
   currentRoute(train2) := route2

   currentRoute(train1) := route1
   trainStatus(train1) := WAITING

   /**
    * Enqueuing a train in the waiting-queue means
    * that the train requests to travel on the network
    */
   waitingTrains := [train1,train2]

   iteration := 0
   lastIteration := 0

   program(trackAgent) := @TrackControl
   program(trainAgent) := @TrainAgent
   program(trainSimulator) := @TrainSimulator
   program(self) := undef
   Agents(self) := false
endpar


rule TrainAgent =
if size(waitingTrains) > 0 then
   seqblock
      dequeue t from waitingTrains
      if not exists block in blocks(currentRoute(t)) with reserved(block) then
         seqblock
            print "\nRoute " + label(currentRoute(t)) + " can be reserved to the
train " + label(t)
            ReserveRoute(t, currentRoute(t))
            print "Route " + label(currentRoute(t)) + " reserved to the train "
+ label(t)
            MoveSpecialComponent(currentRoute(t))
            print "Components of the route " + label(currentRoute(t)) + "
correctly moved (" + routeStatus(currentRoute(t)) + ")"

            GivePermission(t, currentRoute(t))
            print "Train " + label(t) + " starts\n"
         endseqblock
      else
         enqueue t into waitingTrains
   endseqblock


rule ReserveRoute(train, route) =
par
   forall b in blocks(route) do
      par
         blockStatus(b) := COMMITTED
         currentOwnerRoute(b) := route
      endpar
   currentTrain(route) := train
   routeStatus(route) := RESERVED
endpar


rule MoveSpecialComponent(route) =
par
   forall comp in specialComponents(route) do
      if type(comp)=POINT and not(currentPosition(comp)=position(comp, route))
then
         SwitchComponent(comp)

      routeStatus(route) := FORMED
endpar

rule GivePermission(train, route) =
par
   trainStatus(train) := MOVING
   signalStatus(signal(route)) := GREEN
endpar


rule SwitchComponent(comp) =
   if(currentPosition(comp) = UP) then
      currentPosition(comp) := DOWN
   else
      currentPosition(comp) := UP


rule TrackControl =
par
   forall block in Block do
   par
      if outInEvent(startSensor(block, currentOwnerRoute(block))) then
      par
         print "Train " + label(currentTrain(currentOwnerRoute(block))) + "
enters block " + label(block)
         blockStatus(block) := OCCUPIED

         //The train is moving backward
         if not(blockStatus(block) = COMMITTED) then
         par
            print "Train " + label(currentTrain(currentOwnerRoute(block))) + "
is going backward!!! Stop it"
```

```
            trainStatus(currentTrain(currentOwnerRoute(block))) := STOPPED
         endpar


         if block = firstBlock(currentOwnerRoute(block)) then
            //The train has started even if signal was red
            if signalStatus(signal(currentOwnerRoute(block))) = RED then
               trainStatus(currentTrain(currentOwnerRoute(block))) := STOPPED
            else
               signalStatus(signal(currentOwnerRoute(block))) := RED

         outInEvent(startSensor(block, currentOwnerRoute(block))) := false
      endpar

      if inOutEvent(endSensor(block, currentOwnerRoute(block))) then
      par
         print "Train " + label(currentTrain(currentOwnerRoute(block))) + " exit
block " + label(block)
         blockStatus(block) := FREE
         currentOwnerRoute(block) := undef

         if block = lastBlock(currentOwnerRoute(block)) then
         par
            print "Train " + label(currentTrain(currentOwnerRoute(block))) + "
arrived at destination\n"
            routeStatus(currentOwnerRoute(block)) := FREE
            trainStatus(currentTrain(currentOwnerRoute(block))) := undef
            currentRoute(currentTrain(currentOwnerRoute(block))) := undef
            currentTrain(currentOwnerRoute(block)) := undef
         endpar

         inOutEvent(endSensor(block, currentOwnerRoute(block))) := false
      endpar

      if outInEvent(endSensor(block, currentOwnerRoute(block))) then
      par
         print "Train " + label(currentTrain(currentOwnerRoute(block))) + "
reached the end of the block " + label(block)
         outInEvent(endSensor(block, currentOwnerRoute(block))) := false
      endpar

      if inOutEvent(startSensor(block, currentOwnerRoute(block))) then
      par
         print "Train " + label(currentTrain(currentOwnerRoute(block))) + " is
going to exit the block " + label(block)
         inOutEvent(startSensor(block, currentOwnerRoute(block))) := false
      endpar

   endpar
endpar


rule TrainSimulator =
choose train in Train with trainStatus(train) = MOVING do
   if size(sensorsSequence(train, currentRoute(train))) > 0 and iteration -
lastIteration >= 5 then
      seqblock
         dequeue next_sensor from sensorsSequence(train, currentRoute(train))
         par
            if lastEvent(next_sensor)=undef or lastEvent(next_sensor)=INOUT then
            par
               print "Train " + label(train) + " fired OUTIN event on the sensor
" + label(next_sensor)
```

```
                  outInEvent(next_sensor) := true
                  //Remember the last fired event type
                  lastEvent(next_sensor) := OUTIN
               endpar

               else
               par
                  print "Train " + label(train) + " fired INOUT event on the sensor
" + label(next_sensor)
                  inOutEvent(next_sensor) := true
                  //Remember the last fired event type
                  lastEvent(next_sensor) := INOUT
               endpar

            lastIteration := iteration
            endpar
         endseqblock
      else
         iteration := iteration + 1
```

4. Three scenarios

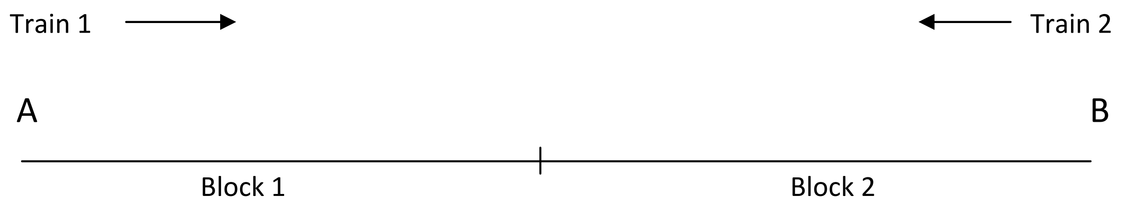
A scenario represents a particular structure of the train network and a particular set of trains travelling on it. The structure of the networks regards the routes that form the network, the blocks, the signals and the special components belonging to the various routes and the sensors of each block.

To complete the scenario, we have to establish the set of the trains and define:

1) The subset of the trains that are waiting to move on the network,
2) The route that each waiting train asked to travel on,
3) For each train T that requests to travel on a route R, we define the sequence of the sensors considered for the production of events when T travels on R.

4.1. Scenario 1

The first scenario is that reported in the code. We consider a network formed by two routes, two blocks and two trains. Each train is waiting to travel on a different route.



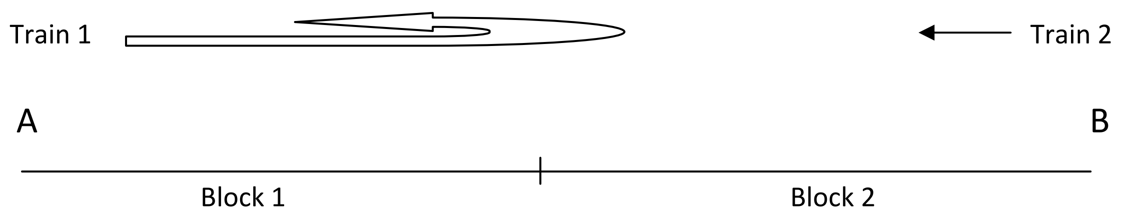
The first train in the waiting queue is that waiting to move from A to B.

The run of the ASM shows that the other train begins to move only after the first one has arrived at the destination.

4.2. Scenario 2

The second scenario permits to see the behavior of the Automatic Train Protection when a train begins moving backward. The network is the same as in the preceding scenario. However, this time, we define the sequence of the sensors considered for the production of events when Train 1 travels from A to B as follows:

1) start sensor of block 1: out-in event
2) end sensor of block 1: out-in event
3) start sensor of block 2: out-in event
4) start sensor of block 2: in-out event
5) start sensor of block 2: out-in event



The run of the ASM shows that, when the fifth event is produced by the simulator, the track control detects that block 2 is occupied. Therefore, the out-in event is recognized as the sign that Train 1 is moving backward. Train 1 is stopped and Train 2 will never enter the network , because Train 1 never leave the block 1.

The only change to the code consists in substituting this update:

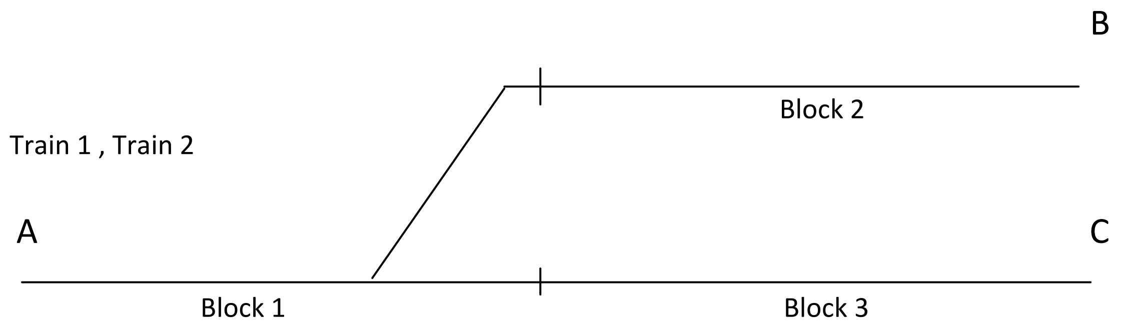
```
sensorsSequence(train1, route1) := [sensor1, sensor2, sensor3, sensor1, sensor4,
sensor2, sensor3, sensor4]
```

with this one:

```
sensorsSequence(train1, route1) := [sensor1, sensor2, sensor3, sensor3, sensor3]
```

4.3. Scenario 3

The third scenario is a little more complex and permits to show the behavior of the individual reservation of the blocks. The network is shown in the following figure. Train 1 travels from A to B, train 2 from A to C. Train 1 is the first to be considered by the train agent. Following the specification, the train agent permits to train 2 to start as soon as train 1 exits the first block (the only block shared by the routes of the trains).



We show the changes in the initialization that allow to run this scenario.

```
rule InitRule =
   par
   …
   //Define the blocks of each route
   blocks(route1) := [block1, block2]
   blocks(route2) := [block1, block3]
   //Define the special components (poins and crossing) of each route
   specialComponents(route1) := [comp1]
   specialComponents(route2) := [comp1]
   //Define the signal of each route
   signal(route1) := signal1
   signal(route2) := signal1
   …

   //Define the sensors of each route
   startSensor(block1, route1) := sensor1
   endSensor(block1, route1) := sensor2
   startSensor(block2, route1) := sensor3
   endSensor(block2, route1) := sensor4

   startSensor(block1, route2) := sensor1
   endSensor(block1, route2) := sensor5
   startSensor(block3, route2) := sensor6
   endSensor(block3, route2) := sensor7

   sensorsSequence(train1, route1) := [sensor1, sensor1, sensor2, sensor3,
sensor2, sensor3, sensor4, sensor4]
   sensorsSequence(train2, route2) := [sensor1, sensor1, sensor5, sensor6,
sensor5, sensor6, sensor7, sensor7]

   //Initialize the signals
   …
```

Bibliography

[1] J.R. Abrial. Modeling in Event-B: System and Software Engineering. Cambridge University Press. (To be published in 2009)

[2] The CoreASM project. http://www.coreasm.org/

[3] Boerger E., Staerk R., Abstract state machines. A method for high-level system design and analysis. (Springer 2003)