Modeling and simulating ERTMS/ETCS using VDM++

Christian M. Lillelund (201408354)

School of Engineering, Aarhus University

Course: E18 - Modeling of Critical Systems

Abstract

Modeling of critical systems is an important tool in software engineering, as it allows us to design a system from specification and verify its properties in a unambiguous way early in the development process. In this report, we use the object-oriented Vienna Development Method (VDM++) to model, simulate and verify a subset of the real specifications for the safety-critical European Rail Traffic Management System (ERTMS) standard focusing on those properties of ERTMS that concern interlocking of tracks and safety of the trains. In ERTMS, these features are controlled by the European Train Control System (ETCS) component. With VDM++, we can perform system design analysis and model verification to catch potential faults early before the system is constructed in an implementation language like Java and deployed in a real life context. We start by introducing requirements and invariants based on the ERTMS/ETCS specification and then use VDM++ to design a formal model followed by model-checking. Finally we apply model validation within Overture using VDM++ to verify the correctness of our model and perform a simulation of inputs. The abstract implementation of ERTMS detailed in this report provide a simple but fundamental understanding of the signaling system's behavior and how to describe its constraints.

Keywords: ERTMS, ETCS, VDM++, Formal methods, Interlocking, Safety

1. Introduction

The railway domain was identified as a grand challenge of computing science by (Bjørner [2]) in 2004 because it is understandable by the general public, provides useful features in terms of transportation and pose many concerns for design and controllability. One part of the challenge is improving the feasibility and capacity of modern railway as the world's population is increasing and rail traffic

now moves cross borders.

The European Rail Traffic Management System (ERTMS) is a signaling and control standard developed in the start 2000's to address the interoperability issues with cross-border rail traffic in Europe and lack of capacity with legacy systems, as detailed in (Ghazel [4]). Currently many European countries have their own national stand-alone signaling and control system implemented by a certain set of rules that differ from each country. ERTMS is designed to replace the national systems to make rail transport more frictionless, improve rail capacity and more attractive to consumers. It consists of two primary parts, the European Train Control and Command System (ETCS) to govern the positions and safety of trains, known as the Interlocking (ITL), and the GSM-R radio communications system to send messages between trains and the RadioBlockCenter (RBC). There a three different levels of ERTMS, L1, L2 and L3 detailed by (Goverde [5]). We consider L2 in this report. L2 introduces the RBC, a Eurobalise that register train movement and omits any track side signaling equipment. When a train computer wants to enter a new route (a set of tracks), it must requests a movement authority (MA) for that section from the RBC, which will see when we explore the model further. ERTMS defines end of authority (EoA) as the point the train is allowed to move to. We consider this aspect as well.

Figure 1 shows the essential components of ERTMS level 2. The train control (EVC) requests movement authorities from the RBC over GSM-R. A typical national railway will have multiple RBC's for each region, where they communicate with a central interlocking service, that receives and tracks the physical location of trains from the Eurobalises in order to either grant or deny movement authority to a train.

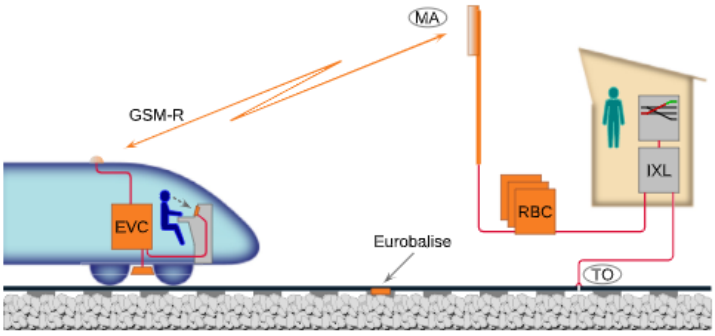

Figure 1: An overview of the ERTMS/ETCS level 2 components. Source: (Berger et al. [1])

2. Requirements

The purpose of the model is to design and verify the interlocking properties and operating rules of the ETCS system, a part of ERTMS. With the model we aim to show how VDM++ can we used to create and satisfy a subset of the real requirements of ETCS interlocking, abstracting away most of the physical constraints you would have in a real system and focusing on the control mechanisms you find in the RadioBlockCenter and Interlocking components, described in the introduction. Our model should demonstrate a simulated train that interacts with the control system when traversing multiple routes with sets of tracks, as it would in real life, while simultaneously monitoring and adhering to the safety features of the system. We address the following requirements:

R1: A train shall not enter a track section which is occupied by another train (same track).
R2: A train shall not pass a track boundary without being given a movement authority (MA) to do so.
R3: A train shall respect the maximum permitted speed of its current track.
R4: Two trains cannot have a MA for the same track at the same time.
R5: The system shall not provide an MA for a track that is occupied by a train.
R6: The RBC shall not answer MA's for tracks it is not responsible for.
R7: The Eurobalise shall report train movement to the Interlocking.

The requirements and the behavior of the ETCS control architecture, that we elaborate on in section 3, are inspired from literature (Berger et al. [1]) and (Ghazel [4]), that describe, model and verify many of the real functional properties of ERTMS. In this report we choose a portion of these to model and test our requirements *R1-R7*. From the specification of ERTMS one can define a typical usage scenario for a train that wishes to enter a route. Given a train *T1*, a route *Rt* with tracks *Tr_1* from a set of tracks (i=1,..,n), an *RBC* responsible for tracks TR_n, a eurobalise *Eb* mounted on the track that registers physical movement and a central interlocking service *ITL*:

- Train *T* wants to start a route *Rt* and enter track *Tr*. *T* requests MA for route *Rt* from RBC.
- RBC contacts the ITL to verify the route is available and clear.
- RBC grants MA to train *T* for route *Rt*.
- *T* enters track *Tr*.

- Eb registers movement and informs ITL.

This functionality will be further explored in the model architecture and design.

3. Model architecture

The architecture of the model represents the entire system as entities, later defined in VDM++ as classes, that encapsulate functionality related to the individual entity. We consider the purpose of our model, that is to express and simulate several important aspects of ERMTS/ETCS, and use the primary components of such system as our entities. These can seen in figure 2 that show a way to model the control architecture of ERTMS based on the real specification set for ERTMS level 2 detailed in (Berger et al. [1]), but junction control and route guidance has been omitted.

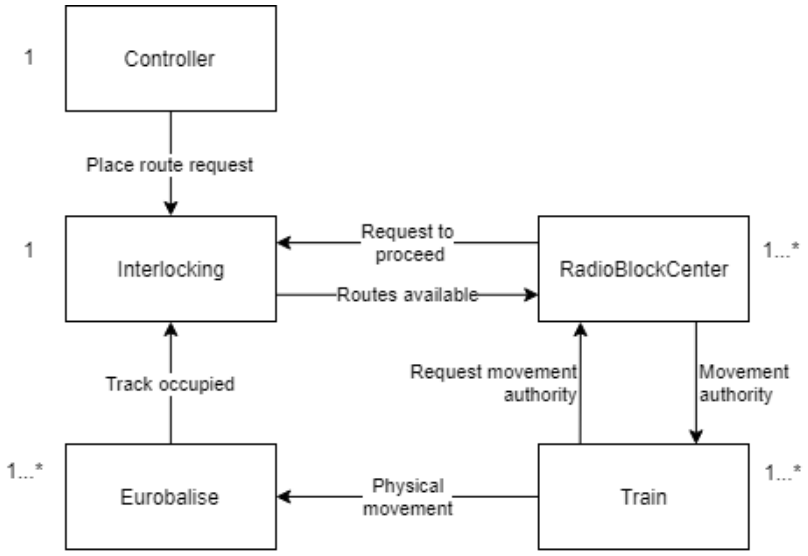

Figure 2: The ETCS control architecture partly from the real specification.

We shall describe the components in more detail before designing the model. The **Controller** places route requests into the interlocking at certain time *t* interval to control the flow of traffic and how many routes can be available concurrently in the rail network, a way to limit congestion. A route is simply a set of tracks. **Interlocking** receives route requests and stores them in a control table containing

available routes. It receives track occupation events from physical units along the track side known as eurobalises and uses this information to update the control table with available routes. The **RadioBlockCenter** receives MA requests from trains and forwards these to the interlocking. RBC's are responsible for multiple tracks, i.e. a block. The **Train** sends MA requests to the RBC before entering a new track in a route. For each route with one or many tracks (a line segment with a geographic starting and terminal point) the train must request and obtain a MA. This is also stated in the requirements. Each track has a certain speed limit, and the train must never exceed this. The **Eurobalise** registers physical movement when trains enter and leave tracks and reports this to the interlocking.

In a real scenario, the eurobalise sends track and speed information to passing trains and the ETCS would monitor this and eventually slow down trains, if they go too fast. We do not consider this part in our model. The ETCS architecture in figure 2, requirements and functionality will be used to create VDM++ classes, functions and invariants in the next section.

4. Model design

In this section we will review the VDM++ model in a top-down approach and elaborate on interesting design choices made. The entities in our architecture are now VDM++ classes, types represent information state objects, and events and requests are described with operations and functions. A full UML class diagram can be found in Appendix C.

4.1. Controller

The controller controls at what rate new routes are placed into the interlocking. In a real setting, it regulates the flow of trains and adds a detail of security, as new routes that become available (free of trains) are not immediately placed into the interlocking. Before ERTMS, this track state was represented by a signal with a flashing or constant yellow light. Listing 1 shows the VDM++ class. Operation SendRouteRequest() is executed sequentially, as it picks a random route from a sequence of routes and requests the route at the interlocking.

```
class Controller

instance variables
private itl : Interlocking;
private routes : seq of Interlocking`Route;
```

```
operations
public Step:() ==> ()
Step() == SendRouteRequest();

private SendRouteRequest:() ==> ()
SendRouteRequest() ==
if(len routes > 0) then
  (dcl rn : nat := MATH`rand(len routes)+1;
    itl.RequestRoute(routes(rn));
);
...
end Controller
```

Listing 1: State variables and the SendRouteRequest() operation in the Controller.

4.2. Interlocking

The interlocking has three main operations defining core functionality, made as such because they change the state of the class. First operation accepts new route requests from the controller for routes R_n and place it into its set of available routes, $availableRoutes$, if tracks are not already occupied. We choose a set for these, as they are finite, the order is not important and duplicate elements should not be significant. Sets also support some useful operators shown later. The second operation receive and store information from corresponding eurobalises (track trackers) about the physical movement of trains in a map. We choose a map for these, as they define lookup operators that allow us to assert if a particular track is occupied or not. The $trackTable$ maps tracks with a given coordinate to either a true or false value, if the track is occupied or not. Third operation respond to the RadionBlockCenter requesting a MA (Movement Authority) on the behalf of trains. We start be defining the implemented types and instance variables shown in listing 2.

Notice the atomic use in the constructor to defer invariant assertion after the assignment.

```
class Interlocking
...
types
public Order = <PROCEED_GRANTED> | <PROCEED_DENIED>;
public ProceedReply :: message : Order
routesAvaliable : set of Route;
public Track :: startX : nat
```

```
        startY : nat
        endX : nat
        endY : nat
        description : seq of char
        maxSpeed : nat;
public Route = set of Track;

instance variables
private trackTable : map Track to bool;
private availableRoutes : set of Route;
inv InvNoDuplicateTrack(availableRoutes);
inv InvNoTrackAvailableAndOccupied(trackTable,
 availableRoutes);

public Interlocking: set of Route ==> Interlocking
Interlocking(rts) ==
atomic (
  availableRoutes := rts;
  trackTable := { tr |-> false |
     tr in set GetTracksInRoutes(rts)};
);
...
end Interlocking
```

Listing 2: Types and state information and constructor for the Interlocking class.

The interlocking has two invariants, $InvNoTrackAvailableAndOccupied$ that makes sure occupied tracks in $trackTable$ and the set of available routes $availableRoutes$ are mutually exclusive, and $InvNoDuplicateTrack$ that makes sure no duplicate track (with the same start and end coordinates) exists in the set of available routes (listing 3)

```
public InvNoTrackAvailableAndOccupied: map Track to bool
 * set of Route -> bool
InvNoTrackAvailableAndOccupied(trmap, rts) ==
forall rt in set rts & forall tr in set rt
  & tr not in set dom trmap or trmap(tr) = false;

public InvNoDuplicateTrack: set of Route -> bool
InvNoDuplicateTrack(srt) ==
forall rt1 in set srt &
  forall tr1,tr2 in set rt1 & tr1 <> tr2
  => tr1.startX <> tr2.startX or tr1.endX <> tr2.endX
  or tr1.startY <> tr2.startY or tr1.endY <> tr2.endY;
```

Listing 3: Two invariants that verify the integrity of the available routes and checks a route for duplicate track.

Next is the $RequestRoute()$ operation called by the controller (listing 4). If the route rt is not in available routes and its tracks are not occupied, we add it to $availableRoutes$ and clear the track.

```
public RequestRoute: (Route) ==> ()
RequestRoute(rt) ==
if (rt not in set availableRoutes and
  forall tr in set rt & tr not in set dom trackTable
   or trackTable(tr) = false)
   then (
    availableRoutes := availableRoutes union {rt};
    trackTable := trackTable ++ { tr |-> false |
            tr in set GetTracksInRoutes({rt})};
   ) else skip;
pre card GetTracksInRoute(rt) > 0;
```

Listing 4: Definition of the SendRouteReques() operation.

$GetTracksInRoute()$ and $GetTracksInRoutes()$ (listing 5) are utility functions that return the tracks in a route or routes using set comprehensions.

```
public GetTracksInRoute: Route -> set of Track
GetTracksInRoute(rt) ==
  {tr | tr in set rt};

public GetTracksInRoutes: set of Route -> set of Track
GetTracksInRoutes(rts) ==
  dunion {tr | tr in set {rt | rt in set rts}};
```

Listing 5: Definition of two functions that return the tracks.

The next operation $SetTrackState()$ is called by the Eurobalise when a train enters a new track tr or leaves a track. This is passed as an argument by the Eurobalise. We set the track as occupied on enter and remove the routes that intersects with the occupied track from $availableRoutes$. On leave, the track is cleared in the $trackTable$ (listing 6). A precondition states that the track must be known in the track table.

```
public SetTrackState: Track * Eurobalise'TrainState ==> ()
SetTrackState(tr, sta) ==
if(sta = <TRAIN_ENTER>) then (
  atomic (
  trackTable := trackTable ++ {tr |-> true};
  availableRoutes := {rts | rts in set availableRoutes
    & rts inter {tr} = {}};
  );
) else if (sta = <TRAIN_LEAVE>) then
  trackTable := trackTable ++ {tr |-> false}
pre tr in set dom trackTable;
```

Listing 6: Definition of the SetTrackState() operation.

Finally the $RequestToProceed()$ operation called by the RadionBlockCenter (listing 7). It returns a proceed granted if the route ptr is in $availableRoutes$ and for all tracks in route ptr none are in the $trackTable$ as occupied. Before a proceed is granted, it removes the route prt from the list of available routes as well all other routes that may contain the same track as ptr. Functions $ProceedGranted()$ and $ProceedDenied()$ return the available routes that intersects with the responsible tracks of the RadioBlockCenter.

```
public RequestToProceed: Route * set of Track ==> ProceedReply
RequestToProceed(prt, respTrs) ==
if (card availableRoutes > 0) then (
  if(exists rt in set availableRoutes
  & prt subset rt and forall tr in set prt
  & tr in set dom trackTable) then
    if(forall tr in set prt & trackTable(tr) = false) then (
      availableRoutes := {rts | rts in set availableRoutes
      & forall ptrs in set {prt} & rts inter ptrs = {}};
      return ProceedGranted(respTrs);
    )
    else return ProceedDenied(respTrs)
  else return ProceedDenied(respTrs)
) else ( return ProceedDenied(respTrs) )
pre card prt > 0 and card GetTracksInRoutes({prt}) > 0;
```

Listing 7: Definition of the RequestToProceed() operation.

4.3. RadioBlockCenter

The RadioBlockCenter receives MA requests from trains and forwards these to the interlocking, basically acting as a gateway but with its own state of available routes and responsible tracks initialized in the constructor. Similar to the interlocking, we use sets here as the order is unimportant and duplicate elements should not count in the model. Also at no point do we need to select certain elements, but operators like membership, intersection and union are important to us. Listing 8 shows the declarations for the RadioBlockCenter.

```
class RadioBlockCenter
types
public MovementAuthorityReply =
 <MovementAuthorityGranted> | <MovementAuthorityDenied>;

instance variables
private respTracks : set of Interlocking`Track;
private availableRoutes : set of Interlocking`Route := {};
private itl : Interlocking;

public RadioBlockCenter : set of Interlocking`Track
* Interlocking ==> RadioBlockCenter
RadioBlockCenter(trs,pitl) ==
  respTracks := trs;
  itl := pitl;
);
...
end RadioBlockCenter
```

Listing 8: Definition of class state and constructor for RadioBlockCenter.

To receive the MA's, we define a operation $RequestMovementAuthority()$ that takes a certain route Rt, checks that the radio block center is responsible for tracks in Rt, then forwards a synchronous proceed request to the interlocking and sends the reply back to the train for route Rt. The interlocking replies with all available routes that intersects with the responsible tracks, hence it is passed as a parameter (listing 9)

```
RequestMovementAuthority: Interlocking`Route
 ==> MovementAuthorityReply
RequestMovementAuthority(rt) ==
(
if(rt subset respTracks) then (
```

```
  dcl msg : Interlocking`Order;
  def mk_Interlocking`ProceedReply(message,rtr)
     = itl.RequestToProceed(rt, respTracks)
  in ( msg := message; availableRoutes := rtr; );
    if (msg = <PROCEED_GRANTED>)
      then ( return <MovementAuthorityGranted>; )
    else (
      return <MovementAuthorityDenied>;
    )
  ) else return <MovementAuthorityDenied>;
) pre card Interlocking`GetTracksInRoute(rt) > 0;
```

Listing 9: Definition of the public.

4.4. Eurobalise

The Eurobalise is a piece of track side equipment that registers the physical movement of trains, as described previously. The class is straightforward, as it simply defines some track it is situated at and the global interlocking system in listing 10.

```
class Eurobalise
types
public TrainState = <TRAIN_ENTER> | <TRAIN_LEAVE>;

instance variables
private itl : Interlocking;
private track: Interlocking`Track;

public Eurobalise : Interlocking * Interlocking`Track
==> Eurobalise
Eurobalise(pitl, tr) ==
(
  itl := pitl;
  track := tr;
);
end Eurobalise
```

Listing 10: Types, variables, constructor for the Eurobalise.

The two operations $Enter()$ and $Leave()$ invoke the interlocking for a track $track$ when a train either enters or leaves it (listing 11. $Enter()$ returns the speed limit of the track to the train.

```
public Enter: () ==> nat1
Enter() == (
  itl.SetTrackState(track, <TRAIN_ENTER>);
  return track.maxSpeed;
);

public Leave: () ==> ()
Leave() == (
  itl.SetTrackState(track, <TRAIN_LEAVE>);
);
```

Listing 11: The two operations that trains call to register their movement.

4.5. Train

The Train is the driver of our model and has a route sequence with routes it must traverse. When starting, it invokes the RadioBlockCenter by sending a MA request for the first route Rt in its route sequence, and if the MA is granted, it will enter the first track Tr in route Rt calling the track's respective Eurobalise from the $transponders$ map, set its current speed to match the track, move onto the end and ultimately leave it by calling the Eurobalise. The train will halt should it not obtain a MA for any Rt. In our simulation we follow the movement of trains by printing a log of their X and Y positional coordinates as simulation progresses. The train has a lot of state declarations, so for clarity, here we simply show the sequential $Drive()$ operation in listing 12. The full code for Train is found in Appendix A.

```
private Drive: () ==> ()
Drive() ==
(
if (len routeTable > 0
  and state = <Running> or state = <WaitingForSignal>) then (
  dcl currentRoute : Interlocking`Route := hd routeTable;
  if(rbc.RequestMovementAuthority(currentRoute)
   = <MovementAuthorityGranted>)
  then (
    for track in GetTracksInRoute(currentRoute) do (
      dcl currentEb : Eurobalise := transponders(track);
      atomic (
        state := <Running>;
        currentSpeed := currentEb.Enter()
      );
```

```
        posX := track.endX;
        posY := track.endY;
        currentEb.Leave();
        );
      routeTable := tl routeTable;
    ) else (
      currentSpeed := 0;
      state := <WaitingForSignal>;
      )
    ) else (
      currentSpeed := 0;
      state := <Finished>;
    )
  );
```

Listing 12: Drive() moves the train along a sequence of routes, requests MA for each route and obeys track speed limits.

5. Validation

This section describes some of the validation techniques used to ensure correctness of the model. Unit testing was used to test functional properties of classes, combinatorial testing to check logical assertions of the model and lastly proof obligations to find points of interest in the model where state is changed and where these has to guarantee some invariant or logical expression holds.

5.1. Unit testing

A typical unit test verifies the functional properties of a class's functions and operations, often conducted in isolation with stubs for any external dependencies. A unit test invokes the unit under test (a class) with some input and evaluates whether the output satisfies what was expected. VDM++ offers unit testing options with VDMUnit, which we have used to test the four control classes in our system and shall explore in a bit. Additionally VDM++ provides a framework for combinatorial testing used to verify pre, post and invariant conditions in our classes. Both methodologies have been applied in this project and the complete set of sets can be found in the attached code, see Appendix A.

5.2. Combinatorial testing

The combinatorial tests are defined in the classes as traces and can be seen to the full extend in Appendix A. With these tests we verify preconditions, postconditions and state invariants. As opposed to the unit tests, that fail if any of these

are violated, the combinatorial tests print inconclusive in such cases and succeeds no matter the assertion was true or false. Looking at the interlocking, listing 13 defines four combinatorial tests that check our invariants. All tests can be seen in Appendix A.

```
T1: let trmap in set
 {{mk_Track(10,5,15,5,"AB",100) |-> false}} in
InvNoTrackAvailableAndOccupied(trmap, routes);

T2: let trmap in set
 {{mk_Track(10,5,15,5,"AB",100) |-> true}} in
InvNoTrackAvailableAndOccupied(trmap, routes);

T3: let rts in set {{{mk_Track(5,5,10,5,"AB",100),
  mk_Track(2,5,10,5,"BA",100)},
  {mk_Track(20,5,10,5,"AB",100),
  mk_Track(30,5,10,5,"DE",100)}}} in
    InvNoDuplicateTrack(rts);

T4: let rt in set {{mk_Track(10,5,15,5,"AB",100),
    mk_Track(15,5,20,5,"BC",100)},
    {mk_Track(20,5,25,5,"AB",100),
     mk_Track(25,5,30,5,"DE",100)}} in
let trmap = {mk_Track(10,5,15,5,"AB",100) |-> false} in
InvIsTrackOccupied(rt, trmap);
```

Listing 13: Four combinatorial tests that exercise our invariant functions in the Interlocking class.

5.3. Proof obligations

Overture can generate proof obligations for VDM++ that can be used after model creation to find points of interest in regards to invariant checks, logical assertions, map applications and type compatibilities. The obligations of the model were evaluated and asserted at each point to either help improve the model (eg. by strengthening a precondition), understand where state variables were changed and applied, or to get an overview of invariant checks. Below are three examples from our proof obligation review:

- 1. Interlocking-L40: State invariant holds.
- 2. Interlocking-L134: Legal map application.
- 3. Train-L68: Operation establishes postcondition.

The first obligation tells us that the state invariant imposed $availableRoutes$ holds given the assignment of the previously available routes apart from those now reserved in the route request. The second state that the map application done in $InvIsTrackOccupied()$ is legal as the index exists in the domain set of the map. The third states that the operation $AddRoute()$ upholds its postcondition by adding the route to its sequence.

6. Simulation

To simulate trains interacting with the control system, we create a $World$ class with two trains running different routes, but sharing a mutual one over a bridge, which they cannot pass at the same time. The World class creates objects for all parts of the environment needed and performs a sequential looping, where we advance the trains and the controller by calling a operation $Step()$. Initially the trains do not have a route set, so we construct our routes based on tracks and use the VDMIO library to print log reports of train movement to the console. Figure 3 shows the track terrain with colors indicating routes for the trains. Both must traverse the line indicated by their color in order to finish. The ERTMS icons show boundaries between tracks. The $World$ class is detailed in Appendix A. The output of the simulation can be viewed in Appendix B.

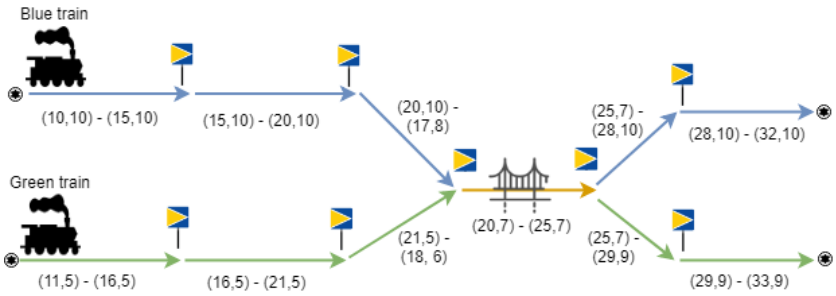

Figure 3: Two trains on a test track used in the simulation. One follows the green-orange-green route, another the blue-orange-blue route.

7. Alternatives to the model

The nature of a real-life ERTMS application is obviously real time, concurrent and multi-threaded. The model of ERTMS level 2 presented in this report is made sequential, that is the execution is performed in steps, as the trains iteratively requests routes to traverse from their route table and cannot move simultaneously,

to make it simple for its purpose of conveying the essentials of ERTMS. VDM++ offers a real-time approach to model creation where we define threads as concurrent classes of execution and let our $World$ class implement a $TimeStamp$ class that introduce the notion of time in our model. An obvious benefit of making the model threaded and influenced by time is the transferability to the real-world domain, but introducing this complexity requires us to consider synchronization aspects of threads, shared data and timing constraints. This is a evident next step for this model.

8. Conclusion

In this report we have argued why a standardized system like ERTMS and ETCS are indeed needed to cope with interoperability issues presented in current nation-based railway technologies, noted in (Commission [3]) and why it is suitable for formal modeling. We proceeded to described the architecture behind it and presented a model that all-though simplified, demonstrate the primary entities and interactions in the VDM++ language using Overture. VDM++ constructs were used where appropriate, as we used a set to represent the list of available routes at the Interlocking component, a map to model the physical track table and used a sequence to hold the routes each train has to traverse. We used three techniques Overture offers to validate the correctness of our model and ran a simulation to show the mutual exclusivity of routes and tracks that ETCS impose.

References

[1] Ulrich Berger, Phillip James, Andrew Lawrence, Markus Roggenbach, and Monika Seisenberger. Verification of the European Rail Traffic Management System in Real-Time Maude. *Science of Computer Programming*, 154:61–88, 2018. ISSN 01676423. doi: 10.1016/j.scico.2017.10.011. URL `https://doi.org/10.1016/j.scico.2017.10.011`.

[2] Dines Bjørner. TRain : The Railway Domain. *TRain*, page 5, 2004.

[3] The European Commission. ERTMS Benefits for stakeholders, 2018. URL `https://ec.europa.eu/transport/modes/rail/ertms/general-information/ertms{_}benefits{_}for{_}stakeholders{_}en`.

[4] Mohamed Ghazel. Formalizing a subset of ERTMS/ETCS specifications for verification purposes. *Transportation Research Part C: Emerging Technologies*, 42:60–75, 2014. ISSN 0968090X. doi: 10.1016/j.trc.2014.02.002. URL http://dx.doi.org/10.1016/j.trc.2014.02.002.

[5] Rob M.P. Goverde. ERTMS: a means to an end. *Nature*, page 23, 2016.

Appendix A

ERTMS

December 10, 2018

Contents

1 Controller

```
class Controller
types

instance variables
 private itl : Interlocking;
 private routes : seq of Interlocking`Route;

operations

 public Step:() ==> ()
 Step() ==
  SendRouteRequest();


 private SendRouteRequest:() ==> ()
 SendRouteRequest() ==
  -- send a random selected route into the Interlocking as a route request.
  if(len routes > 0) then
```

```
  (dcl rn : nat := MATH`rand(len routes)+1;
   itl.RequestRoute(routes(rn));
  );


 public Controller : seq of Interlocking`Route * Interlocking ==> Controller
 Controller(rts, pitl) ==
 atomic (
  routes := rts;
  itl := pitl;
 );

end Controller
```

2 Eurobalise

```
class Eurobalise
types
 public TrainState = <TRAIN_ENTER> | <TRAIN_LEAVE>;

instance variables
 private itl : Interlocking;
 private track: Interlocking`Track;

operations

 public Enter: () ==> nat1
 Enter() ==
 (
  -- notify interlocking that train has entered track
  -- and return permitted speed for the track
  itl.SetTrackState(track, <TRAIN_ENTER>);
  return track.maxSpeed;
 );


 public Leave: () ==> ()
 Leave() ==
   -- notify interlocking that train has left track
  itl.SetTrackState(track, <TRAIN_LEAVE>);


 public GetTrack: () ==> Interlocking`Track
 GetTrack() ==
  -- get track
  return track;


 public Eurobalise : Interlocking * Interlocking`Track ==> Eurobalise
 Eurobalise(pitl, tr) ==
 (
  itl := pitl;
  track := tr;
 );

end Eurobalise
```

3 Interlocking

```
class Interlocking

values
tracks = {mk_Track(10,5,15,5,"AB",100),
 mk_Track(15,5,20,5,"BC",110),
 mk_Track(20,5,25,5,"CD",110),
 mk_Track(25,5,30,5,"DE",110),
 mk_Track(25,5,30,5,"EF",110)};
routes = {tracks};

types
 public Order = <PROCEED_GRANTED> | <PROCEED_DENIED>;
 public ProceedReply :: message : Order
             routesAvaliable : set of Route;
 public Track :: startX : nat
         startY : nat
         endX : nat
         endY : nat
         description : seq of char
         maxSpeed : nat1;
 public Route = set of Track;

instance variables
 private trackTable : map Track to bool; -- track table with physical state of track
 private availableRoutes : set of Route; -- routes from controller to be used

 inv InvNoDuplicateTrack(availableRoutes);
 inv InvNoTrackAvailableAndOccupied(trackTable, availableRoutes);

operations

 public RequestToProceed: Route * set of Track ==> ProceedReply
 RequestToProceed(prt, respTrs) ==
  -- if route is in avaliableRoutes and the route tracks are not in the dom of trackOccupied
   -- grant proceed by returning true, else return false
  if (card availableRoutes > 0) then (
   if(exists rt in set availableRoutes
    & prt subset rt and forall tr in set prt
     & tr in set dom trackTable) then
     if(forall tr in set prt & trackTable(tr) = false) then (
      availableRoutes := {rts | rts in set availableRoutes
       & forall ptrs in set {prt} & rts inter ptrs = {}};
       return ProceedGranted(availableRoutes, respTrs);
     )
     else
      return ProceedDenied(availableRoutes, respTrs)
   else return ProceedDenied(availableRoutes, respTrs)
   ) else (
    return ProceedDenied(availableRoutes, respTrs)
   )
 pre card prt > 0 and card GetTracksInRoutes({prt}) > 0;


 public RequestRoute: (Route) ==> ()
 RequestRoute(rt) ==
  -- if route is not in routesAvaliable and the tracks are not occupied ,
  -- then add it to routesAvaliable and clear trackTable -- else do nothing
  if (rt not in set availableRoutes and
   forall tr in set rt & tr not in set dom trackTable
    or trackTable(tr) = false) then (
       availableRoutes := availableRoutes union {rt};
```

```
      trackTable := trackTable ++ { tr |-> false | tr in set GetTracksInRoutes({rt})};
    ) else skip
 pre card GetTracksInRoute(rt) > 0;


 public SetTrackState: Track * Eurobalise`TrainState ==> ()
  SetTrackState(tr, sta) ==
  -- if train enters, set specific track as occupied and update routes
  -- if train leaves, clear track
  if(sta = <TRAIN_ENTER>) then (
   atomic (
    trackTable := trackTable ++ {tr |-> true};
    availableRoutes := {rts | rts in set availableRoutes
        & rts inter {tr} = {}};
   );
  ) else if (sta = <TRAIN_LEAVE>) then
    trackTable := trackTable ++ {tr |-> false}
  pre tr in set dom trackTable;


 pure public GetAvaliableRoutes: () ==> set of Route
 GetAvaliableRoutes() ==
  return availableRoutes;

 pure public GetAvaliableRoutes: set of Track ==> set of Route
 GetAvaliableRoutes(trs) ==
  return (availableRoutes inter {trs});


 pure public GetOccupiedTracks : () ==> set of Track
 GetOccupiedTracks() ==
  return {tr | tr in set dom trackTable & trackTable(tr) = true};


 public Interlocking: set of Route ==> Interlocking
 Interlocking(rts) ==
 (
  atomic (
   availableRoutes := rts;
   trackTable := { tr |-> false | tr in set GetTracksInRoutes(rts)};
  );
 )

functions

 private ProceedGranted: set of Route * set of Track -> ProceedReply
 ProceedGranted(rts, respTrs) ==
  mk_ProceedReply(<PROCEED_GRANTED>, rts inter {respTrs});


 private ProceedDenied: set of Route * set of Track -> ProceedReply
 ProceedDenied(rts, respTrs) ==
  mk_ProceedReply(<PROCEED_DENIED>, rts inter {respTrs});


 public GetTracksInRoute: Route -> set of Track
 GetTracksInRoute(rt) ==
  {tr | tr in set rt};


 public GetTracksInRoutes: set of Route -> set of Track
 GetTracksInRoutes(rts) ==
  dunion {tr | tr in set {rt | rt in set rts}};
```

```
 public InvNoTrackAvailableAndOccupied: map Track to bool * set of Route -> bool
 InvNoTrackAvailableAndOccupied(trmap, rts) ==
 -- A track cannot be available and occupied.
 forall rt in set rts & forall tr in set rt
  & tr not in set dom trmap or trmap(tr) = false;


 public InvNoDuplicateTrack: set of Route -> bool
 InvNoDuplicateTrack(rts) ==
 -- a route cannot contain the same track twice,
 -- but two routes can contain the same track
 forall rt1 in set rts &
   forall tr1,tr2 in set rt1 & tr1 <> tr2
    => tr1.startX <> tr2.startX or tr1.endX <> tr2.endX
    or tr1.startY <> tr2.startY or tr1.endY <> tr2.endY;


 public InvIsTrackOccupied: Route * map Track to bool -> bool
 InvIsTrackOccupied(rt,trmap) ==
  exists i in set rt & if i in set dom trmap
   then (trmap(i) = true) else false
 pre card rt > 0

traces
T1: let trmap in set {{mk_Track(10,5,15,5,"AB",100) |-> false}} in
   InvNoTrackAvailableAndOccupied(trmap, routes);

T2: let trmap in set {{mk_Track(10,5,15,5,"AB",100) |-> true}} in
   InvNoTrackAvailableAndOccupied(trmap, routes);

T3: let trmap in set {{mk_Track(90,5,90,5,"XX",100) |-> true}} in
   InvNoTrackAvailableAndOccupied(trmap, routes);

T4: InvNoDuplicateTrack(routes);

T5: let rts in set {{{mk_Track(5,5,10,5,"AB",100),
    mk_Track(2,5,10,5,"BA",100)},
     {mk_Track(20,5,10,5,"AB",100),
      mk_Track(30,5,10,5,"DE",100)}}} in
     InvNoDuplicateTrack(rts);

T6: let rts in set {{{mk_Track(5,5,10,5,"AB",100),
    mk_Track(5,5,10,5,"BA",100)},
     {mk_Track(5,5,10,5,"CD",100),
      mk_Track(5,5,10,5,"DE",100)}}} in
     InvNoDuplicateTrack(rts);

T7: let rt in set {{mk_Track(10,5,15,5,"AB",100), mk_Track(15,5,20,5,"BC",100)},
    {mk_Track(20,5,25,5,"AB",100), mk_Track(25,5,30,5,"DE",100)}} in
     let trmap = {mk_Track(10,5,15,5,"AB",100) |-> false} in
      InvIsTrackOccupied(rt, trmap);

T8: let rt in set {{mk_Track(10,5,15,5,"AB",100), mk_Track(15,5,20,5,"BC",100)},
    {mk_Track(20,5,25,5,"CD",100), mk_Track(25,5,30,5,"DE",100)}} in
     let trmap = {mk_Track(25,5,30,5,"DE",100) |-> true} in
      InvIsTrackOccupied(rt, trmap);

T9: let trmap = {mk_Track(10,5,15,5,"AB",100) |-> false} in
     InvIsTrackOccupied({}, trmap);


end Interlocking
```

4 RadioBlockCenter

```
class RadioBlockCenter
types
 public MovementAuthorityReply = <MovementAuthorityGranted> | <MovementAuthorityDenied>;

values
tracks = {mk_Interlocking`Track(10,5,15,5,"AB",100),
 mk_Interlocking`Track(15,5,20,5,"BC",110),
 mk_Interlocking`Track(20,5,25,5,"CD",110),
 mk_Interlocking`Track(25,5,30,5,"DE",110),
 mk_Interlocking`Track(25,5,30,5,"EF",110)}

instance variables
 private respTracks : set of Interlocking`Track := {};
 private availableRoutes : set of Interlocking`Route := {}; -- local state of available routes
 private itl : Interlocking;
 inv Interlocking`InvNoDuplicateTrack(availableRoutes);

operations

 public RequestMovementAuthority: Interlocking`Route ==> MovementAuthorityReply
 RequestMovementAuthority(rt) ==
 (
  --dcl trs: set of Interlocking`Track := {let t in set rt in t | rt in set availableRoutes};
  --if(tr in set responsibleTracks and tr in set trs)
  if(rt subset respTracks) then (
    dcl msg : Interlocking`Order;
    def mk_Interlocking`ProceedReply(message,rtr) = itl.RequestToProceed(rt, respTracks)
    in ( msg := message; availableRoutes := rtr; );
     if (msg = <PROCEED_GRANTED>)
      then ( return <MovementAuthorityGranted>; )
      else (
      return <MovementAuthorityDenied>;
    )
  ) else return <MovementAuthorityDenied>
 ) pre card Interlocking`GetTracksInRoute(rt) > 0;


 public GetAvailableRoutes: () ==> set of Interlocking`Route
 GetAvailableRoutes() ==
  return availableRoutes;


 public GetResponsibleTracks: () ==> set of Interlocking`Track
 GetResponsibleTracks() ==
  return respTracks;


 public RadioBlockCenter : set of Interlocking`Track
  * Interlocking ==> RadioBlockCenter
 RadioBlockCenter(trs,pitl) ==
 atomic (
  respTracks := trs;
  itl := pitl;
 );

functions

 public IsTrackInSetOfTracks: Interlocking`Track * set of Interlocking`Track
  -> bool
 IsTrackInSetOfTracks(tr,trs) ==
  tr in set trs
```

```
 pre card trs > 0;


 public IsTrackInRoute: Interlocking`Track * Interlocking`Route
  -> bool
 IsTrackInRoute(tr, rt) ==
  forall rtt in set rt & tr = rtt
 pre card rt > 0;


 public IsTrackOccupied: Interlocking`Track * map Interlocking`Track to bool
  -> bool
 IsTrackOccupied(tr,routemap) ==
  routemap(tr) = true
 pre tr in set dom routemap;

traces
T1: let tr in set tracks in
   let trs = tracks in
    IsTrackInSetOfTracks(tr, trs);

T2: let tr = mk_Interlocking`Track(25,5,30,10,"AB",100) in
   let trs = tracks in
    IsTrackInSetOfTracks(tr, trs);

T3: let tr in set tracks in
   IsTrackInRoute(tr, tracks);

T4: let tr = mk_Interlocking`Track(25,5,30,10,"AB",100) in
   IsTrackInRoute(tr, tracks);

T5: let tr in set tracks in
    IsTrackInRoute(tr, {});

T6: let tr in set tracks in
   let trmap = {mk_Interlocking`Track(10,5,15,5,"AB",100) |-> true} in
     IsTrackOccupied(tr, trmap);

T7: let tr in set tracks in
   let trmap = {mk_Interlocking`Track(10,5,15,5,"AB",100) |-> false} in
     IsTrackOccupied(tr, trmap);

end RadioBlockCenter
```

5 ControllerTest

```
class ControllerTest  is subclass of TestCase
values
 routes = [{mk_Interlocking`Track(10,5,15,5,"AB",100)},
 {mk_Interlocking`Track(15,5,20,5,"BC",110)},
 {mk_Interlocking`Track(20,5,25,5,"CD",110)},
 {mk_Interlocking`Track(25,5,30,5,"DE",110)},
 {mk_Interlocking`Track(25,5,30,5,"EF",110)}];

 tracks = {mk_Interlocking`Track(10,5,15,5,"AB",100),
 mk_Interlocking`Track(15,5,20,5,"BC",110),
 mk_Interlocking`Track(20,5,25,5,"CD",110),
 mk_Interlocking`Track(25,5,30,5,"DE",110),
 mk_Interlocking`Track(25,5,30,5,"EF",110)}
```

```
instance variables
 private uut: Controller;
 private itl: Interlocking;

operations

 public ControllerTest: () ==> ControllerTest
 ControllerTest() ==
 (
  itl := new Interlocking({});
  uut := new Controller(routes, itl);
 );


 public Test_StepOneTime_OneRoutePlacedIntoItl: () ==> ()
 Test_StepOneTime_OneRoutePlacedIntoItl() ==
 (
  uut.Step();
  assertTrue(card itl.GetAvaliableRoutes() = 1);
 );


 public Test_StepOneTime_RandomRouteFromRoutesPlacedIntoItl: () ==> ()
 Test_StepOneTime_RandomRouteFromRoutesPlacedIntoItl() ==
 (
  uut.Step();
  assertTrue(itl.GetAvaliableRoutes() subset {rt|rt in seq routes});
 );


end ControllerTest
```

6 EurobaliseTest

```
class EurobaliseTest is subclass of TestCase
values
 routes = {{mk_Interlocking`Track(10,5,15,5,"AB",100)},
 {mk_Interlocking`Track(15,5,20,5,"BC",110)},
 {mk_Interlocking`Track(20,5,25,5,"CD",110)},
 {mk_Interlocking`Track(25,5,30,5,"DE",110)},
 {mk_Interlocking`Track(25,5,30,5,"EF",110)}};

 tracks = {mk_Interlocking`Track(10,5,15,5,"AB",100),
 mk_Interlocking`Track(15,5,20,5,"BC",110)};

 track = mk_Interlocking`Track(10,5,15,5,"AB",100);

instance variables
 private uut: Eurobalise;
 private itl: Interlocking;

operations

 public EurobaliseTest: () ==> EurobaliseTest
 EurobaliseTest() ==
 (
  itl := new Interlocking(routes);
  uut := new Eurobalise(itl, track);
 );
```

```
 public Test_CtorCalledWithTrack_TrackIsAssigned: () ==> ()
 Test_CtorCalledWithTrack_TrackIsAssigned() ==
 (
  assertTrue(track = uut.GetTrack());
 );


 public Test_EnterNotCalled_ItlDoesNotChangeStateForTrack: () ==> ()
 Test_EnterNotCalled_ItlDoesNotChangeStateForTrack() ==
 (
  dcl occupiedTracks : set of Interlocking`Track;
  occupiedTracks := itl.GetOccupiedTracks();
  assertTrue(track not in set occupiedTracks);
 );


 public Test_EnterCalled_ItlChangeStateForTrack: () ==> ()
 Test_EnterCalled_ItlChangeStateForTrack() ==
 (
  dcl occupiedTracks : set of Interlocking`Track;
  dcl speed  : nat1 := uut.Enter();
  occupiedTracks := itl.GetOccupiedTracks();
  assertTrue(track in set occupiedTracks);
 );


 public Test_LeaveCalled_ItlChangeStateForTrack: () ==> ()
 Test_LeaveCalled_ItlChangeStateForTrack() ==
 (
  dcl occupiedTracks : set of Interlocking`Track;
  dcl speed  : nat1 := uut.Enter();
  uut.Leave();
  occupiedTracks := itl.GetOccupiedTracks();
  assertTrue(track not in set occupiedTracks);
 );


end EurobaliseTest
```

7 InterlockingTest

```
class InterlockingTest is subclass of TestCase

values
 routes = {{mk_Interlocking`Track(10,5,15,5,"AB",100)},
 {mk_Interlocking`Track(15,5,20,5,"BC",110)},
 {mk_Interlocking`Track(20,5,25,5,"CD",110)},
 {mk_Interlocking`Track(25,5,30,5,"DE",110)},
 {mk_Interlocking`Track(25,5,30,5,"EF",110)}};

instance variables
 private uut: Interlocking;

operations

 public InterlockingTest: () ==> InterlockingTest
 InterlockingTest() ==
  uut := new Interlocking(routes);
```

```
public Test_CtorCalledWithRoutes_RoutesInAvaRoutes: () ==> ()
Test_CtorCalledWithRoutes_RoutesInAvaRoutes() ==
(
 assertTrue(uut.GetAvaliableRoutes() = routes);
);


public Test_GetAvaliableRoutesForTrack_RoutesReturned: () ==> ()
Test_GetAvaliableRoutesForTrack_RoutesReturned() ==
(
 dcl track : Interlocking`Track := mk_Interlocking`Track(10,5,15,5,"AB",100);
 dcl availableRoutes : set of Interlocking`Route := uut.GetAvaliableRoutes({track});
 assertTrue(availableRoutes subset routes);
);


public Test_GetAvaliableRoutesForTrack_NoRoutesAvaliable: () ==> ()
Test_GetAvaliableRoutesForTrack_NoRoutesAvaliable() ==
(
 dcl track : Interlocking`Track := mk_Interlocking`Track(30,5,35,5,"FG",100);
 dcl availableRoutes : set of Interlocking`Route := uut.GetAvaliableRoutes({track});
 assertTrue(card availableRoutes = 0);
);


public Test_RouteInAvailableRoutes_ProceedGranted: () ==> ()
Test_RouteInAvailableRoutes_ProceedGranted() ==
(
 dcl testRoute : Interlocking`Route := {mk_Interlocking`Track(10,5,15,5,"AB",100)};
 dcl msg : Interlocking`ProceedReply := uut.RequestToProceed(testRoute, testRoute);
 assertTrue(msg.message = <PROCEED_GRANTED>);
);


public Test_RouteNotInAvailableRoutes_ProceedDenied: () ==> ()
Test_RouteNotInAvailableRoutes_ProceedDenied() ==
(
 dcl testRoute : Interlocking`Route := {mk_Interlocking`Track(11,4,16,4,"AB",100)};
 dcl msg : Interlocking`ProceedReply := uut.RequestToProceed(testRoute, testRoute);
 assertTrue(msg.message = <PROCEED_DENIED>);
);


public Test_RouteIsRequested_RoutePlacedInAvailableRoutes: () ==> ()
Test_RouteIsRequested_RoutePlacedInAvailableRoutes() ==
(
 dcl reqRoute : Interlocking`Route := {mk_Interlocking`Track(11,4,16,4,"AB",100)};
 uut.RequestRoute(reqRoute);
 assertTrue(reqRoute in set uut.GetAvaliableRoutes());
);


public Test_TrainEntersAndLeavesTrack_TrackChangesState: () ==> ()
Test_TrainEntersAndLeavesTrack_TrackChangesState() ==
(
 dcl track : Interlocking`Track := mk_Interlocking`Track(10,5,15,5,"AB",100);
 uut.SetTrackState(track, <TRAIN_ENTER>);
 assertTrue(track in set uut.GetOccupiedTracks());
 uut.SetTrackState(track, <TRAIN_LEAVE>);
 assertTrue(track not in set uut.GetOccupiedTracks());
);


public Test_TrainLeavesTrack_TrackNotOccupied: () ==> ()
```

```
 Test_TrainLeavesTrack_TrackNotOccupied() ==
 (
  dcl tr : Interlocking`Track := mk_Interlocking`Track(10,5,15,5,"AB",100);
  uut.SetTrackState(tr, <TRAIN_ENTER>);
  uut.SetTrackState(tr, <TRAIN_LEAVE>);
  assertTrue(tr not in set uut.GetOccupiedTracks());
 );


 public Test_RouteHasNoDuplicateTrack_InvSucceed: () ==> ()
 Test_RouteHasNoDuplicateTrack_InvSucceed() ==
 (
  dcl testRoute : set of Interlocking`Route :=
   {{mk_Interlocking`Track(10,5,15,5,"AB",100)},
    {mk_Interlocking`Track(15,5,20,5,"AC",100)}};
  assertTrue(uut.InvNoDuplicateTrack(testRoute));
 );


 public Test_RouteHasDuplicateTrack_InvFailed: () ==> ()
 Test_RouteHasDuplicateTrack_InvFailed() ==
 (
  dcl testRoute : set of Interlocking`Route :=
   {{mk_Interlocking`Track(10,5,15,5,"AB",100),
    mk_Interlocking`Track(10,5,15,5,"AC",100)}};
  assertFalse(uut.InvNoDuplicateTrack(testRoute));
 );

end InterlockingTest
```

8 RadioBlockCenterTest

```
class RadioBlockCenterTest is subclass of TestCase
values
 routes = {{mk_Interlocking`Track(10,5,15,5,"AB",100)},
 {mk_Interlocking`Track(15,5,20,5,"BC",110)},
 {mk_Interlocking`Track(20,5,25,5,"CD",110)},
 {mk_Interlocking`Track(25,5,30,5,"DE",110)},
 {mk_Interlocking`Track(25,5,30,5,"EF",110)}};

 tracks = {mk_Interlocking`Track(10,5,15,5,"AB",100),
 mk_Interlocking`Track(15,5,20,5,"BC",110),
 mk_Interlocking`Track(20,5,25,5,"CD",110),
 mk_Interlocking`Track(25,5,30,5,"DE",110),
 mk_Interlocking`Track(25,5,30,5,"EF",110)}

instance variables
 private uut: RadioBlockCenter;

operations

 public RadioBlockCenterTest: () ==> RadioBlockCenterTest
 RadioBlockCenterTest() ==
 (
  uut := new RadioBlockCenter(tracks, new Interlocking(routes));
 );


 public Test_CtorCalledWithTracks_TracksInResTracks: () ==> ()
 Test_CtorCalledWithTracks_TracksInResTracks() ==
```

```
 (
  assertTrue(uut.GetResponsibleTracks() = tracks);
 );


 public Test_DoNotSetRouteAsAvailable_RouteIsUnavailable: () ==> ()
 Test_DoNotSetRouteAsAvailable_RouteIsUnavailable() ==
 (
  dcl route : Interlocking`Route := {mk_Interlocking`Track(25,5,90,5,"HG",100)};
  assertTrue(route not in set uut.GetAvailableRoutes());
 );


 public Test_RequestMoaForFreeRoute_MoaGranted: () ==> ()
 Test_RequestMoaForFreeRoute_MoaGranted() ==
 (
  dcl track : Interlocking`Track := mk_Interlocking`Track(10,5,15,5,"AB",100);
  dcl msg : RadioBlockCenter`MovementAuthorityReply
   := uut.RequestMovementAuthority({track});
  assertTrue(msg = <MovementAuthorityGranted>);
 );


 public Test_RequestMoaForNotFreeRoute_MoaDenied: () ==> ()
 Test_RequestMoaForNotFreeRoute_MoaDenied() ==
 (
  dcl track : Interlocking`Track := mk_Interlocking`Track(90,70,95,75,"HI",100);
  dcl msg : RadioBlockCenter`MovementAuthorityReply
   := uut.RequestMovementAuthority({track});
  assertTrue(msg = <MovementAuthorityDenied>);
 );

end RadioBlockCenterTest
```

9 Runner

```
class Runner
operations

 public Run : () ==> ()
 Run() ==
 (
  new TestRunner().run();
 );
end Runner
```

10 Train

```
class Train
types
 public State = <Running> | <Stopped> | <WaitingForSignal> | <Finished>;
 public TrainLogEntry :: id : seq of char
             state : State
             posX : real
             posY : real
```

```
instance variables
 private posX : nat := 0;
 private posY : nat := 0;
 private currentSpeed : nat := 0;
 private transponders : map Interlocking`Track to Eurobalise := {|->};
 private id : seq of char := "";
 private state: State := <Stopped>;
 private routeTable: seq of Interlocking`Route := [];
 private trainLog: seq of TrainLogEntry := [];
 private rbc : RadioBlockCenter;
-- inv forall rt in seq routeTable & InvTrackIsConnected(rt);
 inv state = <Running> and currentSpeed >= 0
  or state = <Stopped> and currentSpeed = 0
  or state = <WaitingForSignal> and currentSpeed = 0
  or state = <Finished> and currentSpeed = 0;

operations

 public Step: () ==> ()
 Step() ==
   Drive();


 private Drive: () ==> ()
 Drive() ==
  (
  if (len routeTable > 0
   and state = <Running> or state = <WaitingForSignal>) then (
   dcl currentRoute : Interlocking`Route := hd routeTable;
   UpdateStats();
   if(rbc.RequestMovementAuthority(currentRoute) = <MovementAuthorityGranted>)
   then (
     UpdateStats();
     for track in GetTracksInRoute(currentRoute) do (
      dcl currentEb : Eurobalise := transponders(track);
      atomic (state := <Running>; currentSpeed := currentEb.Enter());
      posX := track.endX;
      posY := track.endY;
      currentEb.Leave();
      UpdateStats();
     );
     routeTable := tl routeTable;
     UpdateStats();
   ) else (
    currentSpeed := 0;
    state := <WaitingForSignal>;
    UpdateStats();
   )
  ) else (currentSpeed := 0;
       state := <Finished>;
       UpdateStats();)
  );


 private UpdateStats: () ==> ()
 UpdateStats() ==
  (
     if(len trainLog > 3) then
     trainLog := tl trainLog;
    trainLog := trainLog ^ [mk_TrainLogEntry(id, state, posX, posY)];
  );


 public AddRoute: Interlocking`Route ==> ()
```

```
 AddRoute(rt) ==
   routeTable := routeTable ^ [rt]
  pre card rt > 0
  post rt in set elems routeTable;


 pure public GetStats: () ==> seq of TrainLogEntry
 GetStats() ==
  return trainLog;


 pure public GetTracksInRoute: Interlocking`Route ==> seq of Interlocking`Track
 GetTracksInRoute(rt) ==
  if (card rt > 0) then (
   dcl trs : seq of Interlocking`Track := [];
   for all tr in set rt do
    trs := trs ^ [tr];
   return trs;
  ) else return [];


 public Start: () ==> ()
 Start() ==
  state := <Running>;


 public Stop: () ==> ()
 Stop() ==
  state := <Stopped>;


 public GetId: () ==> seq of char
 GetId() ==
  return id;


 public IsRunning: () ==> bool
 IsRunning() ==
  if state = <Running> then return true
  else return false;


 public Train: map Interlocking`Track to Eurobalise
  * RadioBlockCenter * seq of char ==> Train
  Train(trans, prbc, pid) ==
  atomic (
   transponders := trans;
   rbc := prbc;
   id := pid;
  );

functions
-- public InvTrackIsConnected: Interlocking`Route -> bool
-- InvTrackIsConnected(rt) ==
--  forall tr1,tr2 in set rt & tr1 <> tr2 =>
--   tr1.endX = tr2.startX and tr1.endY = tr2.startY;

end Train
```

11 World

```
class World

values
 --First upper part of world
 private track1 = mk_Interlocking`Track(10,10,15,10,"U-AB",100);
 private track2 = mk_Interlocking`Track(15,10,20,10,"U-BC",100);
 private track3 = mk_Interlocking`Track(20,10,17,8,"U-CD",50);

 --First lower part of world
 private track4 = mk_Interlocking`Track(11,5,16,5,"L-AB",100);
 private track5 = mk_Interlocking`Track(16,5,21,5,"L-BC",100);
 private track6 = mk_Interlocking`Track(21,5,18,6,"L-CD",50);

 --Bridge
 private track7 = mk_Interlocking`Track(20,7,25,7,"B-AB",10);

 --Last upper part of world
 private track8 = mk_Interlocking`Track(25,7,28,10,"U-DE",90);
 private track9 = mk_Interlocking`Track(28,10,32,10, "U-EF",90);

 --Last lower part of world
 private track10 = mk_Interlocking`Track(25,7,29,9,"L-DE", 90);
 private track11 = mk_Interlocking`Track(29,9,33,9,"L-EF", 90);

 --five routes for all parts of the world
 private routeFU = {track1, track2, track3};
 private routeFL = {track4, track5, track6};
 private routeB = {track7};
 private routeLU = {track8, track9};
 private routeLL = {track10, track11};

 --all tracks in system
 private allTracks = {track1, track2, track3, track4, track5, track6,
 track7, track8, track9, track10, track11};

 --global interlocking system, initial empty table
 private itl = new Interlocking({});

 --global controller, feeds routes to ITL
 private controller = new Controller([routeFU, routeFL,
 routeB, routeLU, routeLL], itl);

 --global radioblockcenter, no available routes, covers all tracks
 private rbc = new RadioBlockCenter(allTracks, itl);

 -- Eurobalise maps for trains
 private trackEbMap = {track1 |-> new Eurobalise(itl, track1), track2 |-> new Eurobalise(itl,
     track2),
 track3 |-> new Eurobalise(itl, track3), track4 |-> new Eurobalise(itl, track4),
 track5 |-> new Eurobalise(itl, track5), track6 |-> new Eurobalise(itl, track6),
 track7 |-> new Eurobalise(itl, track7), track8 |-> new Eurobalise(itl, track8),
 track9 |-> new Eurobalise(itl, track9), track10 |-> new Eurobalise(itl, track10),
 track11 |-> new Eurobalise(itl, track11)};

 -- Routetables for trains
 private timeTable1 = [routeFU, routeB, routeLU];
 private timeTable2 = [routeFL, routeB, routeLL];

 -- Create trains
 private trains: seq of Train = [new Train(trackEbMap, rbc, "IC1"),
            new Train(trackEbMap, rbc, "IC2")];

operations
```

```
    public World: () ==> World
    World() == InitialiseSystem();


    private InitialiseSystem: () ==> ()
    InitialiseSystem() ==
    (
     for route in timeTable1 do (trains(1).AddRoute(route); trains(1).Start());
      for route in timeTable2 do (trains(2).AddRoute(route); trains(2).Start());
    );


    public Run: nat ==> ()
        Run(stepLimit) ==
             for all step in set { 1, ..., stepLimit } do
              (
               controller.Step();
                Print("Step: " ^ VDMUtil`val2seq_of_char[nat](step));
                for train in trains do
                (
                 train.Step();
                 -- print stats
                 Print("Stats for train: " ^
                 VDMUtil`val2seq_of_char[seq of char](train.GetId()) ^ " @ " ^
                 VDMUtil`val2seq_of_char[seq of Train`TrainLogEntry](train.GetStats()));
                )
               );


    private Print: seq of char ==> ()
            Print(text) ==
             def - = new IO().echo(text ^ "\n") in skip;

end World
```

Appendix B

See attached file SimulationResults.txt.