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
```