Models of Railway Systems: Domain

Editors:

Dines Bjørner, Jakob Braad, and Karin S. Mogensen

Abstract

This report presents a domain model of railway systems. It is the intension to model as wide a spectrum as possible, including nets, trains, traffic, passenger handling, etc. The domain model is described both informally and formally in The RAISE Specification Language.
The work of this report is still in progress. Many section are not finished. The purpose of early dissemination of this report is to inform on possibilities and sulicit comments.

Draft

Contributors:

Dines Bjørner (DB): Prof., Dept. of IT, Techn.Univ.of Denmark, 1992–1997;

Chris W. George (CWG): Senior Research Fellow, UNU/IIST;

Bo Stig Hansen (BSH): Assoc.Prof., Dept. of IT, Techn.Univ.of Denmark;

Hans Laustrup (HL): Ericsson, Denmark, formerly Fellow, UNU/IIST;

Søren Prehn (SP): Terma Electronics, Denmark; formerly Senior Research Fellow, UNU/IIST;

Jakob Braad (JB): M.Sc. Student, Dept. of IT, Techn.Univ.of Denmark, 1998-1999;

Jon Fuglsang (JF): M.Sc. Student, Dept. of IT, Techn.Univ.of Denmark, 1998;

Karin S. Mogensen (KSM): M.Sc. Student, Dept. of IT, Techn.Univ.of Denmark, 1998-1999;

Ulrik Svanlundh (US): M.Sc. Student, Dept. of IT, Techn.Univ.of Denmark, 1998.

Contents

Chapter 1

Railways

1.1 Synopsis

Here a railway system consists of a rail net, timetables, trains (with passengers and freight) which run on the net (thus constituting a traffic) and according to schedules determined by the net and the timetables. We are interested in long (i.e. strategic), medium (i.e. tactical) and short term (day-to-day) planning and operational aspects of railways.

1.2 Narrative

We treat the question of *What is a Railway?* by decomposing the answer into a number of components: *What is a Rail Net?*, *What is a Timetable?*, *What is a Schedule?*, *What is Traffic?*, *What is Rescheduling?*, *What are Resources and their Allocation?*, *What is Shunting & Marshalling?*, *What is Station Management?*, and *What are Customer Services?*

Chapter 2

Rail Net

A rail net is composed from Lines and Stations. Lines are composed from Units. Sequences of units form blocks or segments. Stations are composed from units and contain tracks which are either platforms, sidings, or other.

The above description was "top-down": most composite notions were mentioned first, and defined in terms of successively less composite quantities. Finally we mentioned units. *We shall now reverse the presentation into "bottom-up": starting with* units *and* connectors.

2.1 Units & Connectors

A rail net consists of connected units. Units are the blocks, from which the rail net is build. Units have connectors. Connectors are further undefined quantities. If two units have a connector in common, they are said to be connected at that connector.

type
  U, C
value
  obs_U_Cs: U → **C-set**

Units are either linear units (pairs of straight or curved rails), junctions (switches), crossovers, or other.

value
  is_Linear_U: U → **Bool**,

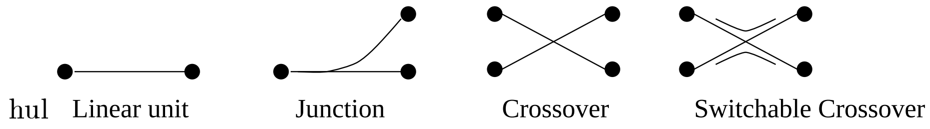

Figure 2.1: Different kinds of units

is_Junction_U: U → **Bool**,
is_Crossover_U: U → **Bool**
axiom
forall u:U •
is_Linear_U(u) ⇒ **card** obs_U_Cs(u) = 2,
is_Junction_U(u) ⇒ **card** obs_U_Cs(u) = 3,
is_Crossover_U(u) ⇒ **card** obs_U_Cs(u) = 4

Linear units always have exactly two connectors, junctions have 3 etc. Later we will state further axioms for different kinds of units. The number of connectors is not sufficient to describe the internal layout of units.

2.1.1 Auxiliary Unit Attributes

With units we can associate a large variety of attributes (types), and for each attribute a range of values. Examples are:

1. Lengths: The lengths, say in meters, of a unit, may be given as a map from paths to lengths.

2. Topology: The topology, from which the lengths, of a unit could be derived, describes — for example as a sequence of Bezier curve triples — the three dimensional layout of the unit: its co-ordinates so-to-speak. Included would also be additional information on the relative "tilting" of rails in curves, etc.

3. Context: The context of a unit tells us whether it is positioned on a bridge, in a tunnel, along a platform, along a quay, etc. Context information may determine maximum and minimum train speeds.

4. Etc.

2.2 Unit Paths & States

A path (through a unit) is a pair of connectors. A path designates a possible direction of train traffic through a unit.

The physical state of a unit is a set of paths. The state contains the paths that are current possible directions of travel through the unit. The physical state of a unit may depend on the topology of the unit, states of switch points, etc.

A path through a unit is physically open if it is in the physical state of the unit. If not in the physical state, the path is physically closed.

The managed state of a unit is a subset of the paths in the physical state of the unit. The managed state contains the paths that are intended directions of travel through the unit, i.e. the rail net management only allow traffic to use paths in the managed states of units. The managed state will for instance depend on states of light signals, laws of traffic, signs at the rail etc.

A path through a unit is managed open if it is in the managed state of the unit. If not in the managed state, the path is managed closed.

An empty managed state designates a closed unit, i.e. no traffic is intended through the unit.

The managed state of a unit depends on management decisions. The position of the unit in the rail net will often have effect on the managed state. For instance, units before the hump of a marshalling yard are typically only open in the direction of the hump, and after the hump only in the direction away from the hump. The managed states of units in the rail net are known to the rail net management.

Every unit has a set of possible (physical) states, the state space. These possible states are determined by for instance the shape and physical layout of the unit. The set of possible states may also contain states that are not intended and should never appear on the rail net. These may include situations of broken switchpoints, etc. Never the less, these states may occur and should therefore be included in the intrinsic model. The physical state of a unit will allways be one of the possible states of that unit.

type
  $\mathrm{P} = \mathrm{C} \times \mathrm{C}$,
  $\Sigma = \mathrm{P}$-**set**,
  $\Omega = \Sigma$-**set**
value
  obs_U_Ω: $\mathrm{U} \to \Omega$,
  obs_U_Physical_Σ: $\mathrm{U} \to \Sigma$,
  obs_U_Managed_Σ: $\mathrm{U} \to \Sigma$,

```
/* All physically possible paths through a unit */
U_Ps: U → P-set
U_Ps(u) ≡
   { p | p:P • ∃ σ:Σ •
      σ ∈ obs_U_Ω(u) ∧ p ∈ σ
   },

/* All connectors of a set of units */
Us_Cs: U-set → C-set
Us_Cs(us) ≡
   { c | c:C •
      ∃ u:U • u ∈ us ∧ c ∈ obs_U_Cs(u)
   }

axiom
/* The physical state is in the set of all states */
∀ u:U • obs_U_Physical_Σ(u) ∈ obs_U_Ω(u),

/* All connectors of paths in states are connectors of the unit */
∀ u:U, σ:Σ, (c,c′):P •
   σ ∈ obs_U_Ω(u) ∧ (c,c′) ∈ σ ⇒
   {c,c′} ⊆ obs_U_Cs(u),

/* Managed states are subsets of physical states */
∀ u:U • obs_U_Managed_Σ(u) ⊆ obs_U_Physical_Σ(u)
```

A linear unit with connectors c, c' will usually only have one possible physical state:

$$\{(c,c'),(c',c)\}$$

The unit gives rise to potentially four different managed states (see figure 2.2):

$$\{\},\{(c,c')\},\{(c',c)\},\{(c,c'),(c',c)\}$$

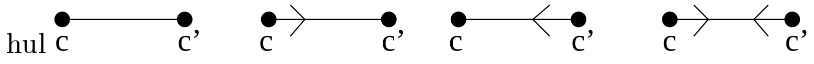

Figure 2.2: States of a linear unit

In the last state the unit is open for traffic in both directions!

There are several kinds of junction units. A certain junction unit, u, with connectors c', c'' at one end and connector c at the other end may for instance have three possible physical states:

$$\{(c',c),(c'',c)\},\{(c,c'),(c',c)\}and\{(c,c''),(c'',c)\}$$

A junction unit potentially has eight possible managed states:

1. $\sigma_0 : \{\}$ (closed),
2. $\sigma_1 : \{(c,c')\}$ (open in one direction, from "tongue" to left fork),
3. $\sigma_2 : \{(c,c'')\}$ (open in one direction, from "tongue" to right fork),
4. $\sigma_3 : \{(c',c)\}$ (open in one direction, from left fork to "tongue"),
5. $\sigma_4 : \{(c'',c)\}$ (open in one direction, from right fork to "tongue"),
6. $\sigma_5 : \{(c',c),(c'',c)\}$ (open in two directions, from either fork to "tongue")
7. $\sigma_6 : \{(c,c'),(c',c)\}$ (open in two directions, from right fork to "tongue" and from "tongue" to right fork)
8. $\sigma_7 : \{(c,c''),(c'',c)\}$ (open in two directions, from left fork to "tongue" and from "tongue" to left fork)

There are also several kinds of crossover units. A crossover unit with connectors c, c' and c'', c''' at respective ends may for instance have only one possible physical state:

$$\{(c,c'''),(c''',c),(c',c''),(c'',c')\}$$

The unit will have 16 possible managed states.

closed: {}

four open in one direction:

$$\{(c,c''')\},\{(c''',c)\},\{(c',c'')\},\{(c'',c')\}$$

six open in two directions:

$$\{(c,c'''),(c''',c)\},\{(c',c''),(c'',c')\},\{(c,c'''),(c',c'')\},\{(c''',c),(c'',c')\},\{(c'',c'),(c,c''')\},\{(c',c''),(c''',c)\}$$

four open in three directions:

$$\{(c,c'''),(c''',c),(c',c'')\},\{(c,c'''),(c''',c),(c'',c')\},\{(c',c''),(c'',c'),(c,c''')\},\{(c',c''),(c'',c'),(c''',c')\}$$

and one open in four directions:

$$\{(c,c'''),(c''',c),(c',c''),(c'',c')\}$$

Similarly for other forms of units.

Using the possible states of units, one can put further constraints on different kinds of units. For instance, there should be a physical state of any linear unit, such that it is open from one end to the other. For a junction, travel should be possible from or to both forks and travel should not be possible between forks.

axiom
  forall u:U •
    is_Linear_U(u) ⇒ U_Ps(u)≠{},
    is_Junction_U(u) ⇒
      ∃ c1,c2,c3:C • **card** {c1,c2,c3} = 3 ∧
        {(c1,c2),(c2,c1)} ∩ U_Ps(u) ≠ {} ∧
        {(c1,c3),(c3,c1)} ∩ U_Ps(u) ≠ {} ∧
        {(c2,c3),(c3,c2)} ∩ U_Ps(u) = {},
    is_Crossover_U(u) ⇒
      ∃ c1,c2,c3,c4:C • **card** {c1,c2,c3,c4} = 4 ∧
        {(c1,c4),(c4,c1)} ∩ U_Ps(u) ≠ {} ∧
        {(c2,c3),(c3,c2)} ∩ U_Ps(u) ≠ {} ∧
        {(c1,c3),(c3,c1)} ∩ U_Ps(u) = {} ∧
        {(c2,c4),(c4,c2)} ∩ U_Ps(u) = {}

2.3 Rail Nets

A rail net is build from units. Not any composition of units is allowed though. A connector can never connect more than two units. Also, two units of a rail net share no paths. These rules express how one may compose units into rail nets. For example the unit compositions of figure 2.3 will not be legal in any rail net.

type

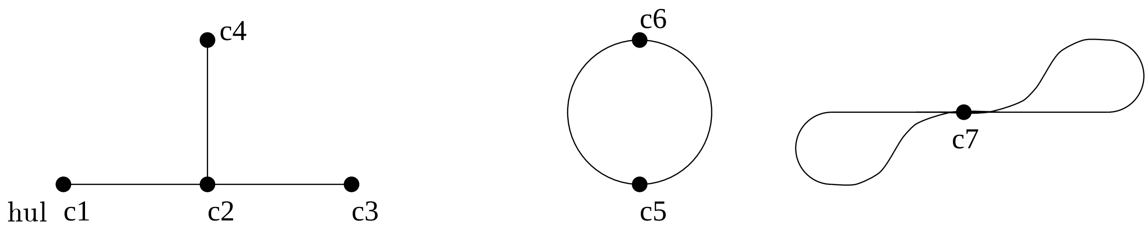

Figure 2.3: Illegal compositions of units

N
value
obs_N_Us: N $\rightarrow$ U-**set**
axiom
/$*$ In a rail net, a connector connects no more than two units $*$/
$\forall$ n:N, c:C $\bullet$
card $\{$ u $|$ u:U $\bullet$ u $\in$ obs_N_Us(n) $\wedge$ c $\in$ obs_U_Cs(u) $\}$ $\leq$ 2,

/$*$ In a rail net, two units do not contain the same path $*$/
$\forall$ n:N, u,u$'$:U $\bullet$
$\{$u,u$'\}$ $\subseteq$ obs_N_Us(n) $\wedge$ u$\neq$u$'$ $\Rightarrow$ U_Ps(u) $\cap$ U_Ps(u$'$) = $\{\}$

2.4 Routes: Open and Closed

The concept of routes play an important role in speaking about train journeys. A route is a sequence of connectors. The connectors of a route designate paths in some rail net, i.e. directions of travel.

A route is feasible in a rail net, if the route describes only possible paths through units of the rail net.

The rule that two units of a rail net share no paths ensures that a feasible route of a rail net describes a unique sequence of unit-paths through the rail net, i.e. given a feasible route of a rail net, it is possible to find the units of the route in that rail net.

A route is physically open in a given rail net, if the connectors of the route designate physically open paths in units of the rail net, i.e. the units are open in direction of the route.

A route is managed open in a given rail net, if the connectors of the route designate managed open paths in units in the rail net.

A routable set of units is a set of units, such that there is a route through the units that includes all units in the set, i.e. it is physically possible to travel along a route through the units, though it may not be allowed by the current states of the units of the route.

A route is cyclic in a rail net if it contains two or more paths through the same unit, such that these paths end in the same connector, i.e. an acyclic route may very well contain several paths through the same unit, as long as the exit-connectors of these paths are distinct.

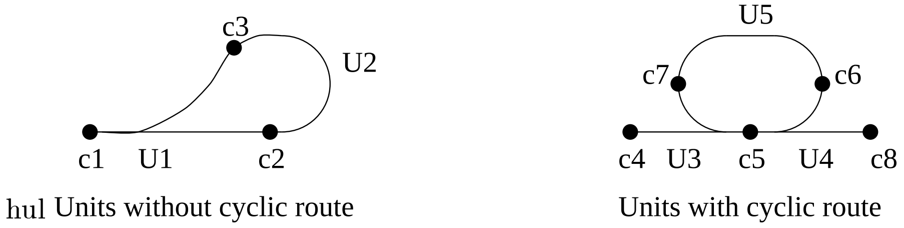

Figure 2.4: Cyclic and acyclic routes

This means that in figure 2.4, the route

$\langle$ c1,c2,c3,c1 $\rangle$

is an acyclic route, while the route

$\langle$ c4,c5,c6,c7,c5,c8 $\rangle$

is cyclic.

type
  $\mathrm{Rt}' = \mathrm{C}^*$,
  Rt = {| rt:Rt$'$ • wf_Rt(rt) |}
value
  /* Wellformed routes have lenght at least two and
    are feasible in some rail net */
  wf_Rt: Rt$'$ $\rightarrow$ **Bool**
  wf_Rt(rt) $\equiv$ **len** rt $\geq$ 2 $\wedge$ $\exists$ n:N • feasible_Rt(rt,n),

  /* A route is feasible wrt. a rail net if the route designates
    possible paths in the rail net and the route does not
    designate two succesive paths through the same unit */
  feasible_Rt: Rt$'$ $\times$ N $\rightarrow$ **Bool**

```
feasible_Rt(rt,n) ≡
   Rt_possible_paths(rt,n) ∧
   let ul = Rt_Ul(rt,n) in
      ∼∃ i:Nat • {i,i+1} ⊆ inds ul ∧ ul(i)=ul(i+1)
   end
pre len rt ≥ 2,

/∗ Route describes possible paths of units in a rail net ∗/
Rt_possible_paths: Rt′ × N → Bool
Rt_possible_paths(rt,n) ≡
   ∀ i:Nat • {i,i+1} ⊆ inds rt ⇒
      ∃ u:U • u ∈ obs_N_Us(n) ∧ (rt(i),rt(i+1)) ∈ U_Ps(u),

/∗ The list of units designated by a route ∗/
Rt_Ul: Rt × N ⥲ U∗
Rt_Ul(rt,n) as ul
post
   len ul = (len rt)−1 ∧
   elems ul ⊆ obs_N_Us(n) ∧
   ∀ i:Nat • {i,i+1} ⊆ inds rt ⇒ (rt(i),rt(i+1)) ∈ U_Ps(ul(i))
pre Rt_possible_paths(rt,n) ∧ len rt ≥ 2,

/∗ The list of paths designated by a route ∗/
Rt_Pl: Rt → P∗
Rt_Pl(rt) ≡ ⟨ (rt(i),rt(i+1)) | i in ⟨ 1 .. (len rt)−1 ⟩ ⟩,

/∗ All units of a route ∗/
Rt_Us: Rt × N ⥲ U-set
Rt_Us(rt,n) ≡ elems Rt_Ul(rt,n)
pre feasible_Rt(rt,n),

/∗ Examine if a route is physically open ∗/
is_Physical_OpenRt: Rt × N ⥲ Bool
is_Physical_OpenRt(rt,n) ≡
   ∀ i:Nat • {i,i+1} ⊆ inds rt ⇒
      (rt(i),rt(i+1)) ∈ obs_U_Physical_Σ(Rt_Ul(rt,n)(i))
pre feasible_Rt(rt,n),

/∗ Examine if a route is managed open ∗/
is_Managed_OpenRt: Rt × N ⥲ Bool
is_Managed_OpenRt(rt,n) ≡
   ∀ i:Nat • {i,i+1} ⊆ inds rt ⇒
      (rt(i),rt(i+1)) ∈ obs_U_Managed_Σ(Rt_Ul(rt,n)(i))
pre feasible_Rt(rt,n),
```

```
/* The first connector of a route */
Rt_firstC: Rt → C
Rt_firstC(rt) ≡ hd rt,

/* The last connector of a route */
Rt_lastC: Rt → C
Rt_lastC(rt) ≡ rt(len rt),

/* The first unit of a route */
Rt_firstU: Rt × N ~→ U
Rt_firstU(rt,n) ≡ hd Rt_Ul(rt,n)
pre feasible_Rt(rt,n),

/* The last unit of a route */
Rt_lastU: Rt × N ~→ U
Rt_lastU(rt,n) ≡ let ul = Rt_Ul(rt,n) in ul(len ul) end
pre feasible_Rt(rt,n),

/* All feasible routes of a rail net */
N_Rts: N → Rt-set
N_Rts(n) ≡ { rt | rt:Rt • feasible_Rt(rt,n) },

/* A route does not go through the same unit twice */
Rt_DisjUs: Rt × N ~→ Bool
Rt_DisjUs(rt,n) ≡ card Rt_Us(rt,n) = len Rt_Ul(rt,n)
pre feasible_Rt(rt,n),

/* Two routes are disjoint */
Rt_Disj: Rt × Rt × N ~→ Bool
Rt_Disj(rt,rt′,n) ≡ Rt_Us(rt,n) ∩ Rt_Us(rt′,n) = {}
pre feasible_Rt(rt,n) ∧ feasible_Rt(rt′,n),

/* All possible routes through a set of units */
Us_Rts: U-set ~→ Rt-set
Us_Rts(us) ≡
   { rt | rt:Rt •
      ∃ n:N •
         us ⊆ obs_N_Us(n) ∧
         feasible_Rt(rt,n) ∧
         Rt_Us(rt,n) ⊆ us
   }
pre net_Us(us),

/* Examine if a set of units is part of some rail net */
net_Us: U-set → Bool
```

```
net_Us(us) ≡ ∃ n:N • us ⊆ obs_N_Us(n),

/* All possible routes that use all units in a set */
Us_complete_Rts: U-set ~→ Rt-set
Us_complete_Rts(us) ≡
   { rt | rt:Rt •
      ∃ n:N •
         rt ∈ N_Rts(n) ∧
         feasible_Rt(rt,n) ∧
         Rt_Us(rt,n) = us
   }
pre net_Us(us),

/* There is a route through all units in a set */
is_RoutableUs: U-set → Bool
is_RoutableUs(us) ≡ Us_complete_Rts(us) ≠ {}
pre net_Us(us),

/* Route is cyclic */
is_Cyclic_Rt: Rt × N → Bool
is_Cyclic_Rt(rt,n) ≡
   ∃ i,j:Nat • {i,i+1,j,j+1} ⊆ inds rt ∧ i≠j ∧
      (Rt_Ul(rt,n)(i),rt(i+1)) = (Rt_Ul(rt,n)(j),rt(j+1))
pre feasible_Rt(rt,n)
```

2.5 Lines and Stations

A rail net consists of lines and stations, i.e.the units of a rail net can be decomposed into those belonging to stations and those belonging to lines.

A line is a routable sequence of linear units, i.e. a connectable sequence of linear units, such that there is a route through the units (in one or two directions).

A station is any set of units, including linear, junctions (switches), crossovers, etc. Two lines meeting in a junction thus gives rise to a station. This station may just consist of that one junction though. The sets of units of a station can be decomposed into those belonging to tracks, that is routable sequences of linear units, and the rest. Among the rest there may still be identifiable routable sequences of linear units. Part of tracks form platforms, sidings, etc.

A line always connects exactly two distinct stations, i.e. a line always forms a route between two stations.

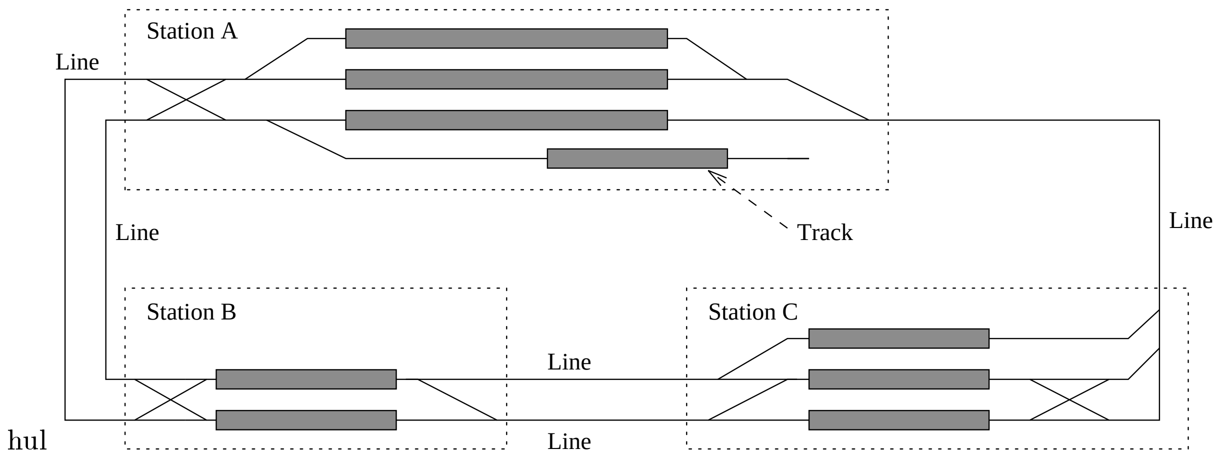

Figure 2.5: A rail net of lines and stations

If it is possible to find a route from a unit u to another unit u', possibly via other units, then u can reach u'. Reachability extends, mutually, to lines, tracks, and stations. Given a line and a station, it is possible to identify exactly which tracks of the station can be reached from the line; and given a track of a station it is likewise possible to identify the lines that can be reached from the track.

type
  L, S, Trk

value
  obs_N_Ls: N → L-**set**,
  obs_N_Ss: N → S-**set**,
  obs_L_Us: L → U-**set**,
  obs_S_Us: S → U-**set**,
  obs_S_Trks: S → Trk-**set**,
  obs_Trk_Us: Trk → U-**set**,

  /∗ Examine if a route of a line connects to a station ∗/
  LS_connection: L × S → **Bool**
  LS_connection(l,s) ≡
    ∃ rt:Rt •
      rt ∈ L_Rts(l) ∧
      Rt_lastC(rt) ∈ Us_Cs(obs_S_Us(s)),

  /∗ Examine if a station connects to a route of a line ∗/
  SL_connection: S × L → **Bool**
  SL_connection(s,l) ≡

```
  ∃ rt:Rt •
     rt ∈ L_Rts(l) ∧
     Rt_firstC(rt) ∈ Us_Cs(obs_S_Us(s)),

/* Examine if two stations are connected via a line */
SLSConnection: S × L × S → Bool
SLSConnection(s,l,s′) ≡
  SL_connection(s,l) ∧ LS_connection(l,s′),

/* All lines that can be reached from a track
    in a given station */
TrkLs: N × S × Trk ~→ L-set
TrkLs(n,s,t) ≡
  { l | l:L • l ∈ obs_N_Ls(n) ∧
     ∃ rt:Rt •
        rt ∈ S_Rts(s) ∧
        Rt_firstC(rt) ∈ Us_Cs(obs_Trk_Us(t)) ∧
        Rt_lastC(rt) ∈ Us_Cs(obs_L_Us(l)) }
pre t ∈ obs_S_Trks(s) ∧ s ∈ obs_N_Ss(n),

/* All tracks in a station that can be reached
    from a given line */
LTrks: N × L × S ~→ Trk-set
LTrks(n,l,s) ≡
  { t | t:Trk • t ∈ obs_S_Trks(s) ∧
     ∃ rt:Rt •
        rt ∈ S_Rts(s) ∧
        Rt_firstC(rt) ∈ Us_Cs(obs_L_Us(l)) ∧
        Rt_lastC(rt) ∈ Us_Cs(obs_Trk_Us(t)) }
pre l ∈ obs_N_Ls(n) ∧ s ∈ obs_N_Ss(n),

S_Rts: S → Rt-set
S_Rts(s) ≡ Us_Rts(obs_S_Us(s)),

L_Rts: L → Rt-set
L_Rts(l) ≡ Us_Rts(obs_L_Us(l)),

Trk_Rts: Trk → Rt-set
Trk_Rts(trk) ≡ Us_Rts(obs_Trk_Us(trk)),

/* All units of the lines in a rail net */
N_L_Us: N → U-set
N_L_Us(n) ≡
  { u | u:U • ∃ l:L • l ∈ obs_N_Ls(n) ∧ u ∈ obs_L_Us(l)},
```

/* All units of the stations in a rail net */
N_S_Us: N → U-**set**
N_S_Us(n) ≡
{ u | u:U • ∃ s:S • s ∈ obs_N_Ss(n) ∧ u ∈ obs_S_Us(s)}

axiom
forall n:N, l,l′:L, s,s′:S, t,t′:Trk, c:C, u:U •
/* Lines are part of some rail net */
net_Us(obs_L_Us(l)),

/* Lines are routable */
is_RoutableUs(obs_L_Us(l)),

/* Lines consist of linear units */
u ∈ obs_L_Us(l) ⇒ is_Linear_U(u),

/* Tracks are part of some station */
∃ s:S • obs_Trk_Us(t) ⊆ obs_S_Us(s),

/* Tracks are routable */
is_RoutableUs(obs_Trk_Us(t)),

/* Tracks consist of linear units */
u ∈ obs_Trk_Us(t) ⇒ is_Linear_U(u),

/* Lines in a rail net do not intersect */
{l,l′} ⊆ obs_N_Ls(n) ⇒
obs_L_Us(l) ⊆ obs_N_Us(n) ∧
l≠l′ ⇒ obs_L_Us(l) ∩ obs_L_Us(l′) = {},

/* Stations are part of some rail net */
net_Us(obs_S_Us(s)),

/* Stations in a rail net do not intersect */
{s,s′} ⊆ obs_N_Ss(n) ⇒
obs_S_Us(s) ⊆ obs_N_Us(n) ∧
s≠s′ ⇒ obs_S_Us(s) ∩ obs_S_Us(s′) = {},

/* Lines and stations do not intersect */
l ∈ obs_N_Ls(n) ∧ s ∈ obs_N_Ss(n) ⇒
obs_L_Us(l) ∩ obs_S_Us(s) = {},

/* Lines connect stations */
l ∈ obs_N_Ls(n) ⇒
∃ s,s′:S •

```
      s≠s′ ∧ {s,s′} ⊆ obs_N_Ss(n) ∧
      SLSConnection(s,l,s′),

  /∗ Tracks of a station do not intersect ∗/
  {t,t′} ⊆ obs_S_Trks(s) ⇒
    obs_Trk_Us(t) ⊆ obs_S_Us(s) ∧
    t≠t′ ⇒ obs_Trk_Us(t) ∩ obs_Trk_Us(t′) = {},

  /∗ Stations do not have common connectors ∗/
  {s,s′} ⊆ obs_N_Ss(n) ∧ s≠s′ ⇒
    Us_Cs(obs_S_Us(s)) ∩ Us_Cs(obs_S_Us(s′)) = {}
```

Stations have names (or identifiers). No two stations share the same name, though, and no station has two names. From a rail net, a map from station names to stations can be extracted.

```
type
  Sn
value
  obs_N_SnSm: N → (Sn →m S),

  obs_N_Sns: N → Sn-set
  obs_N_Sns(n) ≡ dom obs_N_SnSm(n)
axiom
  ∀ n:N •
    obs_N_Ss(n) = rng obs_N_SnSm(n) ∧
    card obs_N_Ss(n) = card obs_N_Sns(n)
```

It is possible to find all physically open routes and all managed open routes of a rail net.

```
value
  Physical_Open_N_Rts: N → Rt-set
  Physical_Open_N_Rts(n) ≡
    { rt | rt:Rt • rt ∈ N_Rts(n) ∧ is_Physical_OpenRt(rt,n) },

  Managed_Open_N_Rts: N → Rt-set
  Managed_Open_N_Rts(n) ≡
    { rt | rt:Rt • rt ∈ N_Rts(n) ∧ is_Managed_OpenRt(rt,n) }
```

2.6 Train Routes

A train route is a route. The intuition behind a train route is that a train occupies exactly the units designated by its train route in some rail net.

A wellformed move of a train route is that of adding a connector to the end of the route, removing a connector from the beginning of the route, simultaniously adding a connector to the end and removing a connector from the beginning of the route, or not changing the route at all. Thus, a train route may only be moved in the "forward" direction.

type
  TR = Rt
value
  wf_TR_move: TR × TR → **Bool**
  wf_TR_move(tr,tr′) ≡
    tr′=tr ∨
    tr′=**tl** tr ∨
    ∃ c:C • tr′=tr⌢⟨c⟩ ∨ tr′=(**tl** tr)⌢⟨c⟩

It is possible to determine if a train is in a given station or at a given track. This can be done by inspecting the train route that contains the train.

value
  TR_at_S: TR × S → **Bool**
  TR_at_S(tr,s) ≡ tr ∈ S_Rts(s),

  TR_at_Trk: TR × Trk → **Bool**
  TR_at_Trk(tr,trk) ≡ tr ∈ Trk_Rts(trk),

  TR_at_StaTrk: TR × S → **Bool**
  TR_at_StaTrk(tr,s) ≡
    ∃ trk:Trk • trk ∈ obs_S_Trks(s) ∧ TR_at_Trk(tr,trk)

2.7 Managed Rail Nets

A managed rail net "snap shot", i.e. a managed rail net state, gives the managed state of all the units in the rail net.

We do not, in this description of the 'intrinsics', define what sets and changes the state. But we prepare the reader for it: it is, of course, the combined setting of junctions (switches), light *signals (semaphores) and conventions, that determine the state. Take a line, as an example, it may be subdivided into segments or blocks, each consisting of e.g. one unit, and each such segment or block being delineated by a signal. (i.e.: the signal is at or about the point where two segments (units) are connected.) A green signal means that the segment right after that signal is open. Etc!*

Since rail nets are regularly being updated: new line and station units are added, old removed entirely, or put under repair, etc., a managed rail net is a function from time to rail net states.

Since changes (extensions, reductions) to the rail net are incremental: most of a rail net remains unchanged while a "small" part undergoes change, we impose some reasonable rule of monotonicity of managed rail nets. To define the monotonicity concept for managed rail nets, we introduce the concept of a rail net change.

A simple change may remove a proper subset of (closed) units, or may insert, i.e. connect a new set of (initially closed) units:

- A simple removal involves the proper closing of all affected units: those to be removed and possibly also all immediately connected (i.e. neighbouring) units, followed by removal.

  (After removal, previously neighbouring units may be reopened.)

- A simple insertion involves a sequence of up to four rail net actions: closing of some units, their removal, insertion of a set of new, but closed units, and the possible opening of these (new) units.

  The set of units removed and the set of units inserted usually have no units in common. For a unit to be inserted it must share a number of connectors with already existing rail net units.

Given two successive managed rail net states, there is a finite, possibly empty set of rail net removal and insertion changes which describes the differences between the two states. Each change is defined in terms of rail net closing, removal, insertion, and opening actions.

type
  T,
  $\mathrm{MR}' = \mathrm{T} \to \mathrm{N}$,
  $\mathrm{MR} = \{|\ \mathrm{mr}{:}\mathrm{MR}' \bullet \mathrm{wf\_MR}(\mathrm{mr})\ |\}$
value
  $\mathrm{wf\_MR}: \mathrm{MR}' \to \textbf{Bool}$
  $\mathrm{wf\_MR}(\mathrm{mr}) \equiv$
    $\forall\ \mathrm{t}{:}\mathrm{T} \bullet \exists\ \mathrm{t}'{:}\mathrm{T} \bullet \mathrm{t}'{>}\mathrm{t} \wedge$
      $\forall\ \mathrm{t}''{:}\mathrm{T} \bullet \mathrm{t}{\leq}\mathrm{t}''{\leq}\mathrm{t}' \Rightarrow \mathrm{MoN}(\mathrm{mr}(\mathrm{t}),\mathrm{mr}(\mathrm{t}''))$,

  $\mathrm{MoN}: \mathrm{N} \times \mathrm{N} \to \textbf{Bool}$,

/∗ Removed or inserted stations contain only closed units ∗/
rem_ins_S_closed: N × N → **Bool**
rem_ins_S_closed(n,n′) ≡
  ∀ s:S •
    s ∈ (obs_N_Ss(n)\obs_N_Ss(n′)) ∪ (obs_N_Ss(n′)\obs_N_Ss(n)) ⇒
      managed_closed_Us(obs_S_Us(s)),

/∗ Removed or inserted lines contain only closed units ∗/
rem_ins_L_closed: N × N → **Bool**
rem_ins_L_closed(n,n′) ≡
  ∀ l:L •
    l ∈ (obs_N_Ls(n)\obs_N_Ls(n′)) ∪ (obs_N_Ls(n′)\obs_N_Ls(n)) ⇒
      managed_closed_Us(obs_L_Us(l)),

managed_closed_Us: U-**set** → **Bool**
managed_closed_Us(us) ≡
  ∀ u:U • u ∈ us ⇒ obs_U_Managed_Σ(u) = {}

axiom
  ∀ n,n′:N • MoN(n,n′) ⇒
    rem_ins_S_closed(n,n′) ∧
    rem_ins_L_closed(n,n′)

. . .

2.8 Laws of Rail Nets

. . .

1. **Law of :**
2. **Law of :**
3. **Law of :**
4. **Law of :**

Chapter 3

Traffic

A traffic is a function from time to states of the rail net and all trains on the net. A train state contains information of the train position, direction of movement, velocity, acceleration, and possibly other information as needed. Thus, a traffic gives at any point in time the entire state of the rail net and all trains.

Traffic within stations may be due to shunting and marshalling. A traffic may also record other matters not covered here.

Certain constraints should be met by any traffic. At any point in time, all trains are on physically open routes of the rail net. These routes may be managed closed though. It is not physically impossible to cross a red light, for instance. Furthermore, trains may not at any time "jump" from one spot to another.

These given constraints for traffics do not reflect intentions or management decisions for the rail net. They only exclude some traffics that would never be possible, no matter how the rail net is managed.

type
  $\mathrm{TF}' = \mathrm{T} \to \mathrm{RS}$,
  $\mathrm{TF} = \{|\ \mathrm{tf}{:}\mathrm{TF}' \bullet \mathrm{wf\_TF}(\mathrm{tf})\ |\}$,
  RS,
  $\mathrm{TP} = \mathrm{Tn} \xrightarrow[m]{} \mathrm{TS}$,
  Tn, TS
value
  net: $\mathrm{RS} \to \mathrm{N}$,
  trns: $\mathrm{RS} \to \mathrm{TP}$,

  obs_TS_TR: $\mathrm{TS} \to \mathrm{TR}$,

obs_TS_Velocity: TS $\rightarrow$ ...
obs_TS_Acc: TS $\rightarrow$ ...

/$*$ Trains do not jump $*$/
wf_TF: TF$'$ $\rightarrow$ **Bool**
wf_TF(tf) $\equiv$ continous_movement(tf),

/$*$ The trains of a TP are on units of the rail net $*$/
TP_on_physical_open_routes: N $\times$ TP $\rightarrow$ **Bool**
TP_on_physical_open_routes(n,tp) $\equiv$
  $\forall$ tn:Tn $\bullet$ tn $\in$ **dom** tp $\Rightarrow$
    obs_TS_TR(tp(tn)) $\in$ Physical_Open_N_Rts(n),

/$*$ Trains do not jump from one spot to another $*$/
continous_movement: TF$'$ $\rightarrow$ **Bool**
continous_movement(tf) $\equiv$
 $\forall$ t:T, tn:Tn $\bullet$ tn $\in$ TF_Tns(tf,t) $\Rightarrow$
    train_removed(tf,tn,t) $\vee$
    train_wf_move(tf,tn,t),

/$*$ In the traffic tf, the train tn is removed at time t $*$/
train_removed: TF$'$ $\times$ Tn $\times$ T $\rightarrow$ **Bool**
train_removed(tf,tn,t) $\equiv$
  tn $\in$ TF_Tns(tf,t) $\wedge$
  $\exists$ t$'$:T $\bullet$ t$'$>t $\wedge$
    $\forall$ t$''$:T $\bullet$ t<t$''\leq$t$'$ $\Rightarrow$ tn $\notin$ TF_Tns(tf,t$''$),

/$*$ In the traffic tf, the train tn is performing a wellformed
  (continuous) move at time t $*$/
train_wf_move: TF$'$ $\times$ Tn $\times$ T $\rightarrow$ **Bool**
train_wf_move(tf,tn,t) $\equiv$
  $\exists$ t$'$:T $\bullet$ t$'$>t $\wedge$
    $\forall$ t$''$:T $\bullet$ t$\leq$t$''\leq$t$'$ $\Rightarrow$
      tn $\in$ TF_Tns(tf,t$''$) $\wedge$
      wf_TR_move(TF_TR(tf,tn,t),TF_TR(tf,tn,t$''$)),

TF_Tns: TF$'$ $\times$ T $\rightarrow$ Tn-**set**
TF_Tns(tf,t) $\equiv$ RS_Tns(tf(t)),

RS_Tns: RS $\rightarrow$ Tn-**set**
RS_Tns(rs) $\equiv$ **dom** trns(rs),

TF_TR: TF$'$ $\times$ Tn $\times$ T $\rightarrow$ TR
TF_TR(tf,tn,t) $\equiv$ RS_TR(tf(t),tn)
pre tn $\in$ TF_Tns(tf,t),

```
    RS_TR: RS × Tn → TR
    RS_TR(rs,tn) ≡ obs_TS_TR(RS_TS(rs,tn))
    pre tn ∈ RS_Tns(rs),

    RS_TS: RS × Tn → TS
    RS_TS(rs,tn) ≡ trns(rs)(tn)
    pre tn ∈ RS_Tns(rs),

    TF_N: TF × T → N
    TF_N(tf,t) ≡ net(tf(t)),

    TF_Sns: TF × T → Sn-set
    TF_Sns(tf,t) ≡ obs_N_Sns(TF_N(tf,t)),

    TF_S: TF × Sn × T → S
    TF_S(tf,sn,t) ≡ obs_N_SnSm(TF_N(tf,t))(sn)
    pre Sn ∈ TF_Sns(tf,t)
axiom
    ∀ rs:RS • TP_on_physical_open_routes(net(rs),trns(rs))
```

One could express further constraint on traffics. For instance the acceleration, maximum speed etc. of trains could be taken into account when describing wellformed moves for trains. Also the rail net topology (length of units, whether units go over bridges, through tunnels, or pass curves etc.) will have effect on the movement of trains.

3.1 Traffic Quality

In a set of traffics there may be some traffics that are in some sense better than others. In describing the intrinsics, we will not further specify how quality of a traffic is determined. This may depend on a lot of different management rules and decisions. We will however describe some functions that may be used when considering the quality of a traffic.

The quality of a traffic may depend on many different aspects. For instance if the traffic satisfies certain time conditions, trains are not delayed etc.

Also, one would usually only want trains to use managed open routes of the rail net and to be within lines or stations of the rail net. Note that a traffic may describe situations where these conditions are not met. It is in fact possible for trains to be on closed routes of the rail net for instance. However, when talking about quality of a traffic, the conditions described should be taken into account.

value
/* Trains are on open routes */
TF_Managed_Open_Rts: TF $\rightarrow$ **Bool**
TF_Managed_Open_Rts(tf) $\equiv$
$\forall$ t:T, tn:Tn • tn $\in$ TF_Tns(tf,t) $\Rightarrow$
TF_TR(tf,tn,t) $\in$ Managed_Open_N_Rts(TF_N(tf,t)),

/* Trains are on lines or within stations of the rail net */
TF_on_S_or_L: TF $\rightarrow$ **Bool**
TF_on_S_or_L(tf) $\equiv$
$\forall$ t:T, tn:Tn • tn $\in$ TF_Tns(tf,t) $\Rightarrow$
TF_TR(tf,tn,t) $\in$ Us_Rts(N_L_Us(TF_N(tf,t)) $\cup$ N_S_Us(TF_N(tf,t)))

One would also prefer certain safety conditions to be met by a traffic. Two trains should never occupy intersecting units of the rail net and a train route should never run through a unit twice. This would probably mean that a crash had occurred or was about to occur.

value
TF_no_collisions: TF $\rightarrow$ **Bool**
TF_no_collisions(tf) $\equiv$ TF_disj_TR(tf) $\wedge$ TF_trdisj(tf),

/* No two trains share units */
TF_disj_TR: TF $\rightarrow$ **Bool**
TF_disj_TR(tf) $\equiv$ $\forall$ t:T • RS_disj_TR(tf(t)),

RS_disj_TR: RS $\rightarrow$ **Bool**
RS_disj_TR(rs) $\equiv$
$\forall$ tn,tn$'$:Tn • {tn,tn$'$} $\subseteq$ RS_Tns(rs) $\wedge$ tn$\neq$tn$'$ $\Rightarrow$
Rt_Disj(RS_TR(rs,tn),RS_TR(rs,tn$'$),net(rs)),

/* A train route does not run through the same unit twice */
TF_trdisj: TF $\rightarrow$ **Bool**
TF_trdisj(tf) $\equiv$ $\forall$ t:T • RS_trdisj(rs),

RS_trdisj: RS $\rightarrow$ **Bool**
RS_trdisj(rs) $\equiv$
$\forall$ tn:Tn • tn $\in$ RS_Tns(rs) $\Rightarrow$
Rt_DisjUs(RS_TR(rs,tn),net(rs))

A traffic embodies the concept of managed rail nets. The managed net of a traffic should thus satisfy the notion of monotonicity outlined above.

value

/∗ The managed railnet of a traffic is wellformed ∗/
TF_wf_MR: TF → **Bool**
TF_wf_MR(tf) ≡ ∃ mr:MR • ∀ t:T • mr(t) = TF_N(tf,t)

3.2 Laws of Traffic

...

1. **Law of :**
2. **Law of :**
3. **Law of :**
4. **Law of :**

Chapter 4

Schedules

A schedule is a plan for train traffic. A schedule is a (possibly infinite) set of acceptable traffics, i.e. the actual traffic in the net is intended to be in the set of traffics of the schedule.

type
  SC = TF-**infset**

A schedule is a way for the rail net managers to describe which traffics should in fact occur on the rail net. Note that it may in some extreme cases be necessary to schedule otherwise not wanted states of the rail net. For instance if a light signal is not working it may be necessary to ignore the signal and let trains pass it no matter the state of the signal. In case of train crashes, it may be necessary to schedule unusual movements of trains. Furthermore, note that a schedule describes the states of the rail net at all points in time. Therefore, if for instance a train has at some point in time passed a closed path, traffics describing this passing of a closed path should be in the schedule, as this is what has in fact happened on the rail net.

As can be seen from the cases described above, schedules must be able to describe any traffic. Therefore there will not be stated further axioms describing which traffics could be included in schedules.

4.1 Schedule Quality

It may be useful to compare the quality of schedules, i.e. which schedule is best, in some sense. This may be done by comparing the traffics allowed by the schedules. As for traffic quality, the schedule quality may depend on many aspects though, and it will not be described in the intrinsic model.

4.2 Laws of Schedules

...

1. **Law of :**
2. **Law of :**
3. **Law of :**

Chapter 5

Passenger & Freight Train Timetables

Timetables describe certain points in time where some conditions on the traffic are to be met. A condition could for instance be the position of a train at a given station, a train passing a specific point on the rail net, etc. Timetables will usually only make few conditions to the traffic. There will for instance not be information regarding how the trains should move from one spot in the rail net to another. Timetables will also usually only contain limited information on the layout of the rail net.

type
  TT

5.1 Schedules and Timetables

For any timetable it is possible to find the schedule describing those traffics that adhere to the timetable.

value
  TT_SC: TT $\rightarrow$ SC

For a schedule to satisfy a timetable, any traffic allowed by the schedule must adhere to the timetable.

value
$\quad$ SC_sat_TT: SC $\times$ TT $\rightarrow$ **Bool**
$\quad$ SC_sat_TT(sc,tt) $\equiv$ sc $\subseteq$ TT_SC(tt)

5.2 Timetable Descriptions

```
Editors remark.
Maybe this section should be rewritten or deleted.
It appears not to follow the structure of the context.
```

Many different specifications of timetables could be used. The following will show an example of such a description and define the semantics of this description.

This section only deals with the timetables of passenger and freight trains: plans for shunting and marshalling will be covered later.

A timetable lists all trains that may engage in traffic on the rail net. For each train its journey is specified. A journey is a set of station visits. A station visit describes arrival and departure times — in addition to the station name!

For a station visit the departure time is always equal to or larger than the arrival time. If the two times are equal, it means that the train does not stop at the designated station. For the first (last) station in a journey the arrival (departure) time shall designate the time the train first (last) appears at the station.

At any given time, a train cannot be at two distinct stations at once, i.e. for a train the time-intervals of station visits may not intersect.

type
$\quad$ /$*$ An example of a timetable specification $*$/
$\quad$ TT = Tn $\overrightarrow{m}$ J,

$\quad$ J$'$ = SV-**set**,
$\quad$ J = {| j:J$'$ • wf_J(j) |},

$\quad$ SV$'$ :: sta : Sn arrival : T depart : T,
$\quad$ SV = {| sv:SV$'$ • wf_SV(sv) |}
value
$\quad$ /$*$ Arrival before departure $*$/
$\quad$ wf_SV: SV$'$ $\rightarrow$ **Bool**

wf_SV(sv) $\equiv$ arrival(sv) $\leq$ depart(sv),

/* Station visits are disjoint */
wf_J: J$'$ $\rightarrow$ **Bool**
wf_J(j) $\equiv$
 $\forall$ sv,sv$'$:SV • {sv,sv$'$} $\subseteq$ j $\wedge$ sv$\neq$sv$'$ $\Rightarrow$ disj_SV(sv,sv$'$),

disj_SV: SV $\times$ SV $\rightarrow$ **Bool**
disj_SV(sv,sv$'$) $\equiv$ arrival(sv)>depart(sv$'$) $\vee$ arrival(sv$'$)>depart(sv)

The timetables of a railway system may consist of one, two, or more timetables. These may be linked to specific calendar periods and cover low and high seasons, vacation and holiday seasons, etc. Any timetable may be concretely expressed in terms of modula: a quarterly timetable which shows train journeys modulo working and week-end days, from Monday through Sunday.

Timetable journey entries may further indicate train types, capacities and services: passenger or freight trains; intercity, express, regional or other trains; number of first, second, etc. class seats (smoking or no-smoking, window, aisle, etc.); sleepers (berths: upper, lower, medium), dining, snack and refreshment cars (tables, seats, menu offerings), etc.; whether seat and berth reservations are required, etc.; whether household (domesticated) pets can accompany passengers; etc.

The semantics of a timetable is a schedule. The schedule contains exactly those traffics which adhere to the timetable.

value
 TT_SC: TT $\rightarrow$ SC
 TT_SC(tt) $\equiv$ { tf | tf:TF • TF_sat_TT(tf,tt) },

 TF_sat_TT: TF $\times$ TT $\rightarrow$ **Bool**
 TF_sat_TT(tf,tt) $\equiv$
  $\forall$ tn:Tn, sv:SV •
   tn $\in$ **dom** tt $\wedge$ sv $\in$ tt(tn) $\Rightarrow$
   TF_sat_SV(tf,tn,sv),

 /* For the duration of the stationvisit, the station
  is part of the rail net and the train is at a track
  within the station */
 TF_sat_SV: TF $\times$ Tn $\times$ SV $\rightarrow$ **Bool**
 TF_sat_SV(tf,tn,sv) $\equiv$
  $\forall$ t:T • arrival(sv)$\leq$t$\leq$depart(sv) $\Rightarrow$
   sta(sv) $\in$ obs_N_Sns(TF_N(tf,t)) $\wedge$
   TR_at_StaTrk(TF_TR(tf,tn,t),obs_N_SnSm(TF_N(tf,t))(sta(sv)))

Note, that it is acceptable that trains arrive at stations too early or that they leave too late. It is not acceptable, however, for trains to depart too early or arrive too late.

5.3 Laws of Timetables

...

Chapter 6

Rescheduling

One can talk about a traffic being on schedule or not. In particular one can talk about traffic being delayed.

For a traffic to be on schedule, the traffic must be one of the traffics allowed by the schedule.

value
  TF_on_SC: TF $\times$ SC $\rightarrow$ **Bool**
  TF_on_SC(tf,sc) $\equiv$ tf $\in$ sc

At any given point in time, the rail net managers will probably not know the exact state of the entire rail net including all train positions, accelerations, speeds, etc. The net managers will get information from the sensors on the rail net and messages from engine men, local managers, etc. Furthermore, the managers will for sure only have limited knowledge of the future states of the rail net. Therefore, the net manager will only know a (possibly infinite) set, ts, of possible traffics, i.e. the manager knows that the actual traffic is in the set ts.

For such a set of possible traffics, it is possible to determine if the traffic can be concluded on schedule or not.

value
  /* Examine if all possible traffics are on schedule */
  TFs_on_SC: TF-**set** $\times$ SC $\rightarrow$ **Bool**
  TFs_on_SC(tfs,sc) $\equiv$
    $\forall$ tf:TF $\bullet$ tf $\in$ tfs $\Rightarrow$ TF_on_SC(tf,sc),

  /* Examine if no possible traffics are on schedule */

TFs_not_on_SC: TF-**set** × SC → **Bool**
TFs_not_on_SC(tfs,sc) ≡
  ∀ tf:TF • tf ∈ tfs ⇒ ∼TF_on_SC(tf,sc)

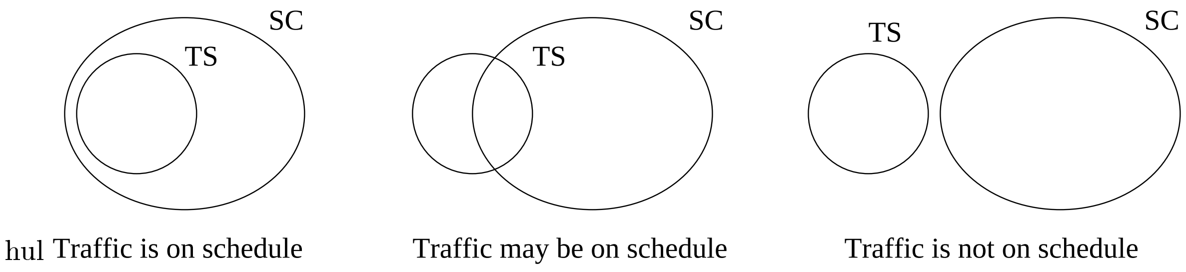

Figure 6.1: Schedules and sets of possible traffics

As time passes, the set of possible traffics gets smaller, i.e. caused by more and more information being known. That is, more and more traffics can be excluded as not possible traffics of the net.

A disruption is defined as anything that makes the traffic not adhere to the schedule. So there is an earliest time, t, up to, but not including which traffic did adhere to the schedule. It may not be possible to determine this time exactly though. A disruption may only be concluded when no possible traffics adhere to the schedule.

value
  disruption: TF-**set** × SC → **Bool**
  disruption(tfs,sc) ≡ TFs_not_on_SC(tfs,sc)

Disruption may give rise to rescheduling. Rescheduling constructs a new timetable and a schedule such that the traffic adhere to the this schedule.

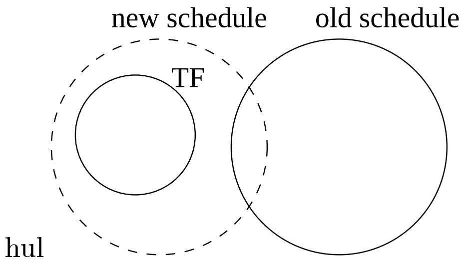

Figure 6.2: Rescheduling

Traffic should adhere to the new schedule.

value
  new_SC: TF-**set** $\times$ SC $\rightarrow$ **Bool**
  new_SC(tfs,sc) $\equiv$ ~disruption(tfs,sc)

The new schedule should satisfy the new timetable.

value
  new_TT: TT $\times$ SC $\rightarrow$ **Bool**
  new_TT(tt,sc) $\equiv$ SC_sat_TT(sc,tt)

Note that disruptions may be observed even before any train is delayed. If for instance a junction breaks down making it impossible for future traffic to be on schedule, it could be observed as a disruption, i.e. rescheduling may be done even before any delay is observed.

6.1 Laws of Rescheduling

...

1. **Law of :**
2. **Law of :**
3. **Law of :**
4. **Law of :**

Chapter 7

Shunting and Marshalling

A train body denotes a sequence of freight and specialty cars and passenger wagons. To allocate, i.e. compose train bodies, one uses shunting and marshalling.

7.1 Shunting

Shunting is the decomposition and composition of train bodies outside marshalling yards, i.e. in other than marshalling parts of stations, typically across platform and siding tracks.

7.2 Marshalling

A marshalling yard is a special part of a station. Here train bodies are decomposed in such a way that individual wagons and cars of one incoming train body are sequentially removed from the head, and sent along a tree structured net of units to either one of a usually large number of standing (other) train bodies to which they join their tails.

A marshalling yard consists of three parts: a possibly tree structured set of incoming sidings, a hump and a tree structured set of typically several outgoing sidings. The hump is the top of the tree of incoming sidings, and is the root of the tree of outgoing sidings, i.e. each incoming siding leads directly to the hump and from there to any outgoing siding. All units of a marshalling yard, when in a managed open state, features exactly one path, and the direction of this path, for a unit of an in-going siding, is always towards the hump, and away from the hump if it is a unit of an outgoing siding. The hump, when managed open, is opened in the direction from the in-going sidings toward the outgoing sidings.

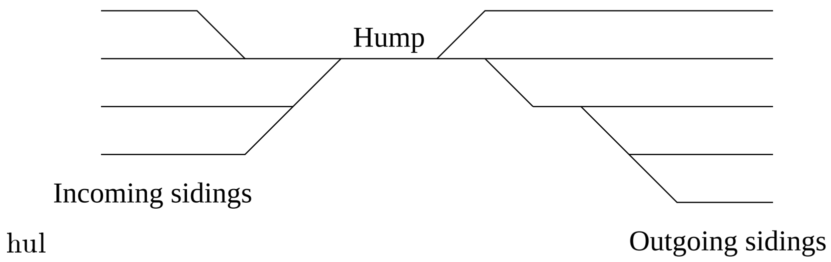

Figure 7.1: A marshalling yard

```
type
  MY
value
  obs_MY_Us: MY → U-set,
  obs_MY_incoming: MY → C-set,
  obs_MY_outgoing: MY → C-set,
  obs_MY_hump: MY → U,

  MY_Rts: MY → Rt-set
  MY_Rts(my) ≡ Us_Rts(obs_MY_Us(my)),

  MY_inout_Rts: MY → Rt-set
  MY_inout_Rts(my) ≡
    { rt | rt:Rt •
      rt ∈ MY_Rts(my) ∧
      Rt_firstC(rt) ∈ obs_MY_incoming(my) ∧
      Rt_lastC(rt) ∈ obs_MY_outgoing(my)
    }
axiom
  forall my:MY, u:U, c,c′:C •
    /∗ Marshalling Yards are part of some rail net ∗/
    net_Us(obs_MY_Us(my)),

    /∗ Humps are linear ∗/
    is_Linear_U(obs_MY_hump(my)),

    /∗ Incoming and outgoing connectors are disjoint ∗/
    obs_MY_incoming(my) ∩ obs_MY_outgoing(my) = {},

    /∗ There is a route from any incoming connector
       to any outgoing connector ∗/
    c ∈ obs_MY_incoming(my) ∧ c′ ∈ obs_MY_outgoing(my) ⇒
      ∃ rt:Rt • rt ∈ MY_inout_Rts(my) ∧
```

$(c,c') = (Rt\_firstC(rt),Rt\_lastC(rt)),$

/* Between any two connectors in a marshalling yard
there is no more than one route */
$\sim\exists\ rt,rt':Rt \bullet rt \neq rt' \wedge \{rt,rt'\} \subseteq MY\_Rts(my) \wedge$
$(Rt\_firstC(rt),Rt\_lastC(rt)) = (Rt\_firstC(rt'),Rt\_lastC(rt')),$

/* All connectors in a marshalling yard are part of a route
from an incoming connector to an outgoing connector
which goes through the hump */
$c \in Us\_Cs(obs\_MY\_Us(my)) \Rightarrow$
$\exists\ rt:Rt \bullet rt \in MY\_Rts(my) \wedge$
$\{c\} \cup obs\_U\_Cs(obs\_MY\_hump(my)) \subseteq \textbf{elems}\ rt,$

/* Units are never managed open in direction from the outgoing side
towards the incoming side */
$\forall\ rt:Rt \bullet$
$(rt \in MY\_inout\_Rts(my) \wedge$
$u \in obs\_MY\_Us(my) \wedge$
$(c,c') \in \textbf{elems}\ Rt\_Pl(rt)) \Rightarrow (c',c) \notin obs\_U\_Managed\_\Sigma(u)$

A train body list (or set) description has a name and a list (respectively set) of car and wagon descriptions.

type
  $TB = W^*,$
  W,
  Wtype
value
  $obs\_TS\_TB: TS \rightarrow TB,$
  $obs\_Wtype: W \rightarrow Wtype$

A marshalling plan is a pair of train body descriptions: A set of incoming train body list (or set) descriptions and a set of outgoing train body list (respectively set) descriptions.

type
  MP
value
  obs_MP_incoming: MP → TB-**set**,
  obs_MP_outgoing: MP → TB-**set**

A marshalling description is a list of marshalling states. A marshalling state is a set of incoming train bodies and the set of outgoing train bodies.

For any state in a marshalling description, the next state is reached by moving exactly one wagon from the front end of an incoming train body to the rear end of an outgoing train body.

type
  $\mathrm{MD}' = \mathrm{MS}^*$,
  $\mathrm{MD} = \{|\ \mathrm{md}{:}\mathrm{MD}' \bullet \mathrm{wf\_MD}(\mathrm{md})\ |\}$,
  MS :: incoming: TB-**set** outgoing: TB-**set**
value
  wf_MD: $\mathrm{MD}' \to$ **Bool**
  wf_MD(md) $\equiv$
    $\forall$ i:**Nat** $\bullet$ $\{i,i+1\} \subseteq$ **inds** md $\Rightarrow$
      wf_state_shift(md(i),md(i+1)),

  wf_state_shift: MS $\times$ MS $\to$ **Bool**
  wf_state_shift(ms,ms$'$) $\equiv$
    $\exists$ w:W, tb,tb$'$:TB $\bullet$
    let ims=incoming(ms)$\backslash\{$tb$\}$ $\cup$ $\{$**tl** tb$\}$,
      oms=outgoing(ms)$\backslash\{$tb$'\}$ $\cup$ $\{$tb$'{}^\frown\langle$**hd** tb$\rangle\}$
    in ms$'$ = mk_MS(ims,oms) **end**

It is possible to determine if a marshalling plan is feasible, i.e. if it is possible to assemble the outgoing train bodies from the incoming train bodies on some marshalling yard.

value
  feasible_MP: MP $\to$ **Bool**
  feasible_MP(mp) $\equiv$
    $\exists$ md:MD $\bullet$
      incoming(**hd** md) = obs_MP_incoming(mp) $\wedge$
      outgoing(md(**len** md)) = obs_MP_outgoing(mp)

Furthermore, the marshalling yard must be big enough. There must be enough incoming and outgoing sidings for the train bodies involved.

value
  feasible_MP_wrt_MY: MP $\times$ MY $\to$ **Bool**
  feasible_MP_wrt_MY(mp,my) $\equiv$
    feasible_MP(mp) $\wedge$

card obs_MY_incoming(my) $\geq$ **card** obs_MP_incoming(mp) $\wedge$
card obs_MY_outgoing(my) $\geq$ **card** obs_MP_outgoing(mp)

7.3 Laws of Marshalling

Laws of marshalling could be:

1. **Law of Preservation:**

   In any time interval such that there were no in-going and no outgoing train bodies in the marshalling yard at the beginning and at the ending of the interval, the cars and wagons arriving at the in-going sidings equals the cars and wagons departing from an outgoing siding.

2. **Law of No–Reversal:**

   With the above marshalling yard topology and possible unit states it is not possible to reverse, in a train body of an outgoing siding, cars or wagons of a train body of an in-going siding: i.e. if rolling stock items A and B were in one train body when passing the hump, and A came before B, and if A and B is in a same train body of an outgoing siding, then A will be in front of B.

3. **Law of Ordering:**

   More specifically: If A, B and C were in the same train body passing the hump, and in the order A–B–C, and if they are all in a same train body of an outgoing siding, then B will be between A and C.

Chapter 8

Passengers

Trains of the rail net may carry passengers. At any time there is a set of passengers in trains of the rail net system.

8.1 Passenger Ticketing:

Any passenger or an agent acting on behalf of any group of one or more passengers may inquire about:

- a journey, i.e. whether it is possible or not, etc.

  A journey might be simple or composite.

- price of tickets.
- current traffic, i.e. whether a certain train is delayed.

The passenger or agent may also want to:

- reserve tickets, seats, etc.
- buy tickets.
- cancel tickets.

A passenger or group of passenger may board a train or alight from a train.

Chapter 9

Freight

The domain model similarly only speaks about the booking and occupancy status of freight trains. This chapter concentrates on the stake holder perspective as seen from the customers, i.e. on the activities which is needed to fulfil the customers needs and wishes.
In the following Freight Handling is used to refer to the system which handles the booking etc.

type
  FTn = Tn,
  Freight,
  FreightId,
  FreightSc,
  FreightInfo,
  FreightSystem,
  FreightISI = FreightId $\times$ FreightSc $\times$ FreightInfo

value
  /* Observer functions */
  obs_FreightId : FreightSystem $\rightarrow$ FreightId-**set**,

  /* obs_Freight, obs_FreightInfo and obs_Schedule are partial */
  /* since the FreightId might not exist in the FreightSystem */
  obs_Freight : FreightSystem $\times$ FreightId $\stackrel{\sim}{\rightarrow}$ Freight,
  obs_FreightInfo : FreightSystem $\times$ FreightId $\stackrel{\sim}{\rightarrow}$ FreightInfo,
  obs_Schedule : FreightSystem $\times$ FreightId $\stackrel{\sim}{\rightarrow}$ FreightSc,

  obs_FreightSystem_from_Net : N $\rightarrow$ FreightSystem

axiom

$/*$ For each FreightId in a FreightSystem exists exactly one Freight and one FreightSc $*/$
$\forall$ f:FreightSystem, fid, fid$'$: FreightId $\bullet$
    {fid,fid$'$} $\subseteq$ obs_FreightId(f) $\Rightarrow$
    fid = fid$'$ $\equiv$
    let
        freight = obs_Freight(f,fid),
        freight$'$ = obs_Freight(f,fid$'$),
        finfo = obs_FreightInfo(f,fid),
        finfo$'$ = obs_FreightInfo(f,fid$'$),
        fsc = obs_Schedule(f,fid),
        fsc$'$ = obs_Schedule(f,fid$'$)
    in
        freight = freight$'$ $\wedge$
        finfo = finfo$'$ $\wedge$
        fsc = fsc$'$
    end

In the following the functions which directly reflect the communication with the customers are described. These include checking if freight is delivered at the station or if it has reached its destination. The customers can request price, etc. for a piece of freight, confirm a request if this is possible, and reserve a place for a piece of freight.

value
    $/*$ Test if Freight is delivered. False if FreightId doesn't exist in FreightSystem $*/$
    Is_Delivered : N $\times$ FreightId $\to$ **Bool**,

    $/*$ Test if Freight is at destination. False if FreightId doesn't exist in FreightSystem $*/$
    At_Destination : N $\times$ FreightId $\to$ **Bool**,

    $/*$ Deliver freight $*/$
    Deliver_Freight : N $\times$ FreightId $\xrightarrow{\sim}$ N,

    $/*$ Receive freight $*/$
    Receive_Freight : N $\times$ FreightId $\xrightarrow{\sim}$ N,

    $/*$ Freight request - Do not make a reservation $*/$
    Request : N $\times$ TT $\times$ Sn $\times$ Sn $\times$ T $\times$ Freight $\to$ FreightISI,

    $/*$ Confirm (reserve) a request if possible $*/$
    Confirm : N $\times$ FreightISI $\to$ N $\times$ **Bool**,

/∗ Freight reservation ∗/
Reservation : N × TT × Sn × Sn × T × Freight → N × FreightISI,

/∗ Cancel reservation. ∗/
Cancel : N × FreightId $\xrightarrow{\sim}$ N

axiom
/∗ Deliver ∗/
∀ n : N, fid : FreightId •
Deliver_Freight(n, fid) **as** n′
post Is_Delivered(n′, fid)
pre fid ∈ obs_FreightId(obs_FreightSystem_from_Net(n)),

/∗ Receive ∗/
∀ n : N, fid : FreightId •
Receive_Freight(n, fid) **as** n′
post fid ∉ obs_FreightId(obs_FreightSystem_from_Net(n′))
pre
fid ∈ obs_FreightId(obs_FreightSystem_from_Net(n)) ∧
At_Destination(n, fid),

/∗ Request ∗/
∀ n : N, tt : TT, sn1, sn2 : Sn, t : T, freight : Freight •
Request(n, tt, sn1, sn2,t,freight) **as** (fid′, fsc′, finfo′)
post fid′ ∉ obs_FreightId(obs_FreightSystem_from_Net(n)),

/∗ Confirm Request ∗/
∀ n : N, fid : FreightId, fsc : FreightSc, finfo : FreightInfo •
Confirm(n, (fid, fsc, finfo)) **as** (n′, b′)
post b′ =
(obs_Schedule(obs_FreightSystem_from_Net(n′), fid) = fsc) ∧
(obs_FreightInfo(obs_FreightSystem_from_Net(n′), fid) = finfo),

/∗ Make reservation ∗/
∀ n : N, tt : TT, sn1, sn2 : Sn, t : T, freight : Freight •
Reservation(n, tt, sn1, sn2, t, freight) **as** (n′, (fid′, fsc′, finfo′))
post freight = obs_Freight(obs_FreightSystem_from_Net(n′), fid′),

/∗ Cancel reservation ∗/
∀ n : N, fid : FreightId •
Cancel(n, fid) **as** n′
post fid ∉ obs_FreightId(obs_FreightSystem_from_Net(n′))
pre fid ∈ obs_FreightId(obs_FreightSystem_from_Net(n))

From the Freight System one can extract information about the pieces of freight. This information is needed to decide which wagons are needed to fulfil the agreement with the customers.

value
  /* Find which wagons are needed for the freight */
  GetTrain : N $\rightarrow$ Wtype*

As a special treat, Freight Handling also offers the customers the ability to trace their freight. The customers can request where the freight is at any time and will either get a station name or a train name and the location of that train.

type
  FreightTraffic,
  Location,
  Place,
  Pos

value
  /* Observer functions */
  obs_FreightSet : FreightTraffic $\rightarrow$ FreightId-**set**,

  /* obs_Location is partial since the loacation of a certain freight */
  /* might not be known, e.g. if it is lost */
  obs_Location : FreightTraffic $\times$ T $\times$ FreightId $\xrightarrow{\sim}$ Location,

  obs_FreightTraffic_from_Net : N $\rightarrow$ FreightTraffic,

  /* Test if Place is a Station Name */
  Is_Sn : Place $\rightarrow$ **Bool**,

  /* Test if Place is a Freight Train Name and a Position */
  Is_FTnPos : Place $\rightarrow$ **Bool**,

  /* Test if Location is a Station Name */
  Is_Sn : Location $\rightarrow$ **Bool**,

  /* Test if Location is a Train Name */
  Is_FTn : Location $\rightarrow$ **Bool**,

  Can_obs_Location : FreightTraffic $\times$ T $\times$ FreightId $\rightarrow$ **Bool**,

/* Freight Tracing */
Freight_Trace : N × TF × T × FreightId $\xrightarrow{\sim}$ Place

axiom
/* A Place is either a Station Name or a Freight Train Name and a Position */
∀ place : Place •
~ (Is_Sn(place) ∧ Is_FTnPos(place)),

/* A Location is either a Station Name or a Freight Train Name */
∀ location : Location •
~ (Is_Sn(location) ∧ Is_FTn(location)),

/* Freight Tracing */
∀ n : N, tf : TF, time : T, fid : FreightId •
Freight_Trace(n, tf, time, fid) **as** place
post
let loc = obs_Location(obs_FreightTraffic_from_Net(n), time, fid)
in if Is_Sn(loc) **then** Is_Sn(place) **elsif** Is_FTn(loc) **then** Is_FTnPos(place) **end**
end
pre
Can_obs_Location(obs_FreightTraffic_from_Net(n),time,fid)

9.1 Scheduling of Freight

The scheduling of freight is modelled here but some of the issues are mentioned briefly. For example, if the freight is flammable or radioactive, the schedule shouldn't take it through residental areas. This could be modelled by predicates which test freight wrt. some given criteria. The predicates could then be used to decide on the best schedule but this is left to the Freight Traffic and Train Management which is in charge of making schedules.

9.2 Lost & Found

It may happen that freight is lost while in transit from one place to another, e.g. a wagon might accidently get separated from a train. Freight might also reappear, e.g. when railway personnel find any freight that was lost earlier.

This can be modelled by using the following functions:

type
  Freights,
  FT

value
  obs_Freights_from_Train: FT $\rightarrow$ Freights,
  Load_Freights: FT $\times$ Freights $\rightarrow$ FT,
  /* Merge takes two Freights and combines them */
  Merge: Freights $\times$ Freights $\rightarrow$ Freights

Freights is a type containing pieces of freight. *FT* is a type representing a freight train.

The following three cases are possible:

1. **No freight is lost :** In this case all the freight loaded on the freight train can be observed from the train.

   Formally, this can be formulated as:

   axiom
     $\forall$ ft : FT, frs, frs$'$: Freights $\bullet$
     let frs$'$ = obs_Freights_from_Train(ft) **in**
     obs_Freights_from_Train(Load_Freights(ft,frs)) = merge(frs,frs$'$)
     end

2. **Freight is lost :** In this case it is not possible to observe all the freight which were loaded on the train.

   Formally, this can be formulated as:

   axiom
     $\forall$ ft : FT, frs, frs$'$: Freights $\bullet$
     let frs$'$ = obs_Freights_from_Train(ft) **in**
     obs_Freights_from_Train(Load_Freights(ft,frs)) $\subset$ merge(frs,frs$'$)
     end

3. **Freight is found:** In this case more freight can be observed on the train than the freight that were actually loaded on the train.

   Formally, this can be formulated as:

axiom
$\forall$ ft : FT, frs, frs$'$: Freights $\bullet$
let frs$'$ = obs_Freights_from_Train(ft) **in**
obs_Freights_from_Train(Load_Freights(ft,frs)) $\supset$ merge(frs,frs$'$)
end

When freight is lost, it should be possible to store information about it, its *FreightInfo* and *FreightSc*, etc. It should also be possible to check if freight is lost etc.

In the following it is assumed that *FreightId* can be obtained on freight found. This is used to remove the recovered freight from *LostFreight*.

Formally, this can be formulated as:

type
LostFreight

value
/$*$ Observer function $*$/
obs_LostFreight: FreightSystem $\rightarrow$ LostFreight,
obs_Freight_from_LostFreight: LostFreight $\rightarrow$ FreightId-**set**,

/$*$ obs_FreightInfo_from_LostFreight and obs_FreightSc_from_LostFreight $*$/
/$*$ are partial since FreightId might not exist in LostFreight $*$/
obs_FreightInfo_from_LostFreight: LostFreight $\times$ FreightId $\stackrel{\sim}{\rightarrow}$ FreightInfo,
obs_FreightSc_from_LostFreight: LostFreight $\times$ FreightId $\stackrel{\sim}{\rightarrow}$ FreightSc,

Is_Freight_Lost: N $\times$ FreightId $\rightarrow$ **Bool**
Is_Freight_Lost(n,fid) $\equiv$
let
fs = obs_FreightSystem_from_Net(n),
lf = obs_LostFreight(fs),
lostfid = obs_Freight_from_LostFreight(lf)
in fid $\in$ lostfid
end,

Insert_LostFreight: N $\times$ FreightId $\stackrel{\sim}{\rightarrow}$ N
Insert_LostFreight(n,fid) **as** n$'$
post
let
fs = obs_FreightSystem_from_Net(n$'$),
lf = obs_LostFreight(fs),

```
            lostfid = obs_Freight_from_LostFreight(lf)
        in fid ∈ lostfid
        end
    pre ~Is_Freight_Lost(n,fid),

Remove_Freight: N × FreightId ~→ N
Remove_Freight(n,fid) as n′
    post
        let
            fs = obs_FreightSystem_from_Net(n′),
            lf = obs_LostFreight(fs),
            lostfid = obs_Freight_from_LostFreight(lf)
        in fid ∉ lostfid
        end
        pre Is_Freight_Lost(n,fid)
```

9.3 Laws of Freight Customers

A law of Freight Customers could be:

1. **Law of Preservation:**

   In any timeinterval such that no freight is delivered or received, the sum of traceable freight and lost freight is constant.

Chapter 10

Resources & Allocations

The railway system resources, in addition to the rail net, consists of the rolling stock, the wagons waiting for or under maintenance, and the wagons in actual use.

10.1 Rolling Stock

The rolling stock contains information about, among other things, the available wagons and the units on which they are located.
It is possible to extract information about the type of each wagon in the rail net. The type of a wagon could be locomotive, passenger car, etc. This information is needed in rolling stock when composing train bodies consisting of wagons of certain types.

type
  RollingStock,

value
  /$*$ Observer functions $*$/
  obs_Wagons : RollingStock $\rightarrow$ W-**set**,

  /$*$ obs_Units_from_Wagon and obs_Train_from_Wagon are partial $*$/
  /$*$ since they are undefined if the wagon doesn't exist in RollingStock $*$/
  obs_Units_from_Wagon : RollingStock $\times$ W $\xrightarrow{\sim}$ U*,
  obs_Train_from_Wagon : RollingStock $\times$ W $\xrightarrow{\sim}$ TB,

  obs_RollingStock_from_Net : N $\rightarrow$ RollingStock, During the work with the specification of plans for composing a certain train body from the rolling stock, it was necessary to use a bag type. Instead of specifying a specific bag type, the bag specification is parameterized. This makes it possible to use the specification in other contexts.

scheme BAG(E : **class type** Elem **end**) =
class
type
  bag = E.Elem $\overrightarrow{m}$ **Nat**

value
  /* A list is converted to a bag */
  List_to_Bag′ : E.Elem* → bag → bag
  List_to_Bag′(elist)(bag) ≡
    case elist **of**
      ⟨⟩ → bag,
      ⟨e⟩ ^ list →
        List_to_Bag′(list)(
        if e ∈ **dom** bag **then**
          bag † [e ↦ bag(e) + 1]
        else
          bag † [e ↦ 1]
        end)
    end,

  List_to_Bag : E.Elem* → bag
  List_to_Bag(elist) ≡
    List_to_Bag′(elist)([ ])

end

The bag specification is used to specify test functions. These test if a certain train body can be composed by the wagons in the rolling stock. As a special case, it is tested if the desired train body already exists as a train body somewhere in the rolling stock.

object WTYPE: **class type** Elem = Wtype **end**
object WTYPE_BAG: BAG(WTYPE)

value
  /* Test if a certain train can be composed */
  Train_Exists : N × Wtype* → **Bool**

Train_Exists(n, wtypelist) $\equiv$
let
  r = obs_RollingStock_from_Net(n),
  rs_ws = obs_Wagons(r),
  rs_ls = Convert(rs_ws),
  rs_wtypebag = WTYPE_BAG.List_to_Bag(rs_ls),
  wtypebag = WTYPE_BAG.List_to_Bag(wtypelist)
in
  $\forall$ wtype : Wtype $\bullet$
  wtype $\in$ **dom** wtypebag $\Rightarrow$
  wtype $\in$ **dom** rs_wtypebag $\wedge$ (wtypebag(wtype) $\leq$ rs_wtypebag(wtype))
end,

/* Converts a W-set to a Wtype-list. Needed to convert a W-set to a Wtype-bag */
Convert: W-**set** $\rightarrow$ Wtype*
Convert(ws) **as** wtypelist
  post
    card ws = **len** wtypelist $\wedge$
    ($\forall$ w:W $\bullet$ w $\in$ ws $\equiv$ obs_Wtype(w) $\in$ **elems** wtypelist),

/* Test if a certain train already exists in RollingStock */
Train_Already_Exists : N $\times$ Wtype* $\rightarrow$ **Bool**

One need to be able to compose train bodies. This requires knowledge of where wagons of a certain type are located (or where a certain train body with specific wagons is located if such exists).

value
  /* The locations of wagons of a certain type */
  Wagons_Locations : N $\times$ Wtype $\rightarrow$ (U*)-**set**
  Wagons_Locations(n, wtype) $\equiv$
    { u | u : U* $\bullet$
      let r = obs_RollingStock_from_Net(n), rs_ws = obs_Wagons(r)
      in $\forall$ w : W $\bullet$ w $\in$ rs_ws $\wedge$ u = obs_Units_from_Wagon(r,w)
          $\wedge$ obs_Wtype(w) = wtype
      end
    },

  /* The locations of an entire trainbody of a certain type */
  Trainbody_Location : N $\times$ Wtype* $\rightarrow$ (U*)-**set**,

  /* The locations of a trainbody of a certain type */
  Train_Location : N $\times$ Wtype* $\rightarrow$ ((U*)*)-**set**

axiom

$\forall$ n : N, wtypelist : Wtype* $\bullet$
  Train_Already_Exists(n, wtypelist) $\Rightarrow$ Train_Exists(n, wtypelist),

$\forall$ n : N, wtypelist : Wtype* $\bullet$
  Train_Already_Exists(n, wtypelist) $\equiv$ (Trainbody_Location(n, wtypelist) $\neq$ {}),

$\forall$ n : N, wtypelist : Wtype* $\bullet$
  Train_Exists(n, wtypelist) $\equiv$ (Train_Location(n, wtypelist) $\neq$ {})

It is now possible to extract enough information from the rolling stock to make plans for composing a certain train body. A plan contains information about how to compose a certain train body which is needed at a specific place. The place is given as a unit on which either the whole train body or the front of the train body (if the train body is longer than the unit) is to be located after executing the plan. From a plan it is possible to extract information about which wagons the train body is to be composed of and where they are located.

type
  Plan,
  PlanId

value
  /* Observer functions */
  obs_destinationUnit_from_Plan : Plan $\rightarrow$ U,
  obs_Wagons_from_Plan : Plan $\rightarrow$ (W $\xrightarrow[m]{}$ U*),

  /* Make plan for composing train */
  /* The function is partial; there might not exist a Plan for Wtype-list */
  Plan_for_Composing_Train : N $\times$ Wtype* $\times$ U $\stackrel{\sim}{\rightarrow}$ Plan

Often, it is possible to make several plans for a certain train body. In practice one often wants to choose the plan which is considered best wrt. some criteria, e.g. cost, environmental issues, time, distance, etc. In the following is given a predicate *Better_Plan* which tests if one plan is better than another. No specification of what makes one plan better than another is given.

value
  /* A plan is better than another */

Better_Plan : N × Plan × Plan → **Bool**,

/* The set of best plans: A plan is in this set if no plan is better than it */
Make_Set_of_Good_Plans : N × Wtype* × U → Plan-**set**
Make_Set_of_Good_Plans(n, wtypelist, unit) ≡
{ p | p : Plan •
p = Plan_for_Composing_Train(n, wtypelist, unit) ∧
(∀ p′ : Plan • p′ ≠ p ∧ p′ = Plan_for_Composing_Train(n, wtypelist, unit)
⇒ ∼Better_Plan(n, p′, p))
},

/* Choose one of the best plans */
Best_Plan : N × Wtype* × U → Plan
Best_Plan(n, wtypelist, unit) **as** plan
post plan ∈ Make_Set_of_Good_Plans(n, wtypelist, unit)

axiom
/* Better_Plan */
∀ n : N, p, p′, p″ : Plan •
∼ Better_Plan(n, p, p) ∧
(Better_Plan(n, p, p′) ⇒ ∼ Better_Plan(n, p′, p)) ∧
((Better_Plan(n, p, p′) ∧ Better_Plan(n, p′, p″)) ⇒ Better_Plan(n, p, p″))

One could also test whether a plan is better than another plan e.g. wrt. cost, distance, or time, etc:

value
Better_Plan_wrt_Cost : N × Plan × Plan → **Bool**,
Better_Plan_wrt_Distance : N × Plan × Plan → **Bool**,
Better_Plan_wrt_Time : N × Plan × Plan → **Bool**

After deciding which plan to use, this plan should be registered. This can be done by inserting it in the plan for the rolling stock. This plan, called *rsPlan*, contains information about all the chosen plans.
One also has to consider what to do with the wagons in the inserted plan. For example, what if a plan is made for composing another train body and that plan uses some of the wagons in the inserted plan. This would result in a conflict.
This is resolved by having a reservoir which contains information about the wagons already committed to a plan. Whenever a plan is inserted in *rsPlan*, the wagons committed to that plan

is removed from the wagon pool in the rolling stock and inserted in the reservoir. When a plan is executed, it is removed from the rsPlan.

type
  rsPlan,
  Reservoir

value
  /* Observer functions */
  obs_rsPlan : RollingStock $\rightarrow$ rsPlan,

  /* obs_Plan_from_rsPlan is partial since PlanId might not exist in rsPlan */
  obs_Plan_from_rsPlan : rsPlan $\times$ PlanId $\stackrel{\sim}{\rightarrow}$ Plan,

  obs_PlanId_from_rsPlan : rsPlan $\rightarrow$ PlanId-**set**,
  obs_Reservoir_from_Net : N $\rightarrow$ Reservoir,
  obs_Wagons_from_Reservoir : Reservoir $\rightarrow$ W-**set**,

  /* When a Plan is inserted in rsPlan, the wagons in the Plan is moved to Reservoir */
  Insert_Plan : N $\times$ Plan $\stackrel{\sim}{\rightarrow}$ N $\times$ PlanId
  Insert_Plan(n,p) **as** (n′,pid′)
    post
      let
        wm = obs_Wagons_from_Plan(p),
        ws = **dom** wm,
        r = obs_RollingStock_from_Net(n′),
        rs_ws = obs_Wagons(r),
        res = obs_Reservoir_from_Net(n′),
        res_ws = obs_Wagons_from_Reservoir(res),
        rsplan = obs_rsPlan(r)
      in
        $\sim$(ws $\subseteq$ rs_ws) $\wedge$ (ws $\subseteq$ res_ws) $\wedge$
        obs_Plan_from_rsPlan(rsplan, pid′) = p
      end
    pre Test_Plan(n,p),

  /* When a Plan is cancelled, it's removed from rsPlan */
  Cancel_Plan: N $\times$ PlanId $\stackrel{\sim}{\rightarrow}$ N
  Cancel_Plan(n,pid) **as** n′
    post
      let
        r = obs_RollingStock_from_Net(n),
        rsplan = obs_rsPlan(r),

```
            r′ = obs_RollingStock_from_Net(n′),
            rsplan′ = obs_rsPlan(r′)
        in
            ∼ ( ∃ pid′ : PlanId •
            obs_Plan_from_rsPlan(rsplan′, pid′) = obs_Plan_from_rsPlan(rsplan,pid)
            )
        end
    pre pid ∈ obs_PlanId_from_rsPlan(obs_rsPlan(obs_RollingStock_from_Net(n))),

/* Test if a Plan is legal. A Plan is legal wrt. rsPlan */
/* if it doesn't conflict with the Plans in rsPlan */
Test_Plan : N × Plan → Bool,

/* Merge two plans */
Merge_Plans : N × PlanId × PlanId → N × PlanId,

/* A Plan is removed from rsPlan when it is executed */
Execute_Plan : N × PlanId ~→ N
Execute_Plan(n,pid) as n′
    post
        let
            r = obs_RollingStock_from_Net(n),
            rsplan = obs_rsPlan(r),
            r′ = obs_RollingStock_from_Net(n′),
            rsplan′ = obs_rsPlan(r′)
        in
            ∼ ( ∃ pid′ : PlanId •
            obs_Plan_from_rsPlan(rsplan′, pid′) = obs_Plan_from_rsPlan(rsplan,pid)
            )
        end
    pre pid ∈ obs_PlanId_from_rsPlan(obs_rsPlan(obs_RollingStock_from_Net(n)))
```

When the wagons in a train are no longer used, the wagons are tranferred to the wagon pool in the rolling stock. This happens e.g. when a train reaches its destination and is "unloaded", whether it is a passenger or freight train.

```
/* A wagon is inserted in RollingStock */
Insert_Wagon: N × W × U* ~→ N
Insert_Wagon(n,w,ulist) as n′
    post
        let r = obs_RollingStock_from_Net(n′)
```

```
    in w ∈ obs_Wagons(r) ∧ ulist = obs_Units_from_Wagon(r,w)
    end
  pre w ∉ obs_Wagons(obs_RollingStock_from_Net(n))
```

10.2 Laws of Resources & Allocation

Laws of resources & allocation could be:

1. **Law of Best Plan:**

   In any time interval, starting when a plan for a certain train body is executed, such that no wagons are inserted in the rolling stock during the interval, any plan for composing the same train body cannot be better than that plan.

2. **Law of Preservation:**

   In any time interval such that no wagons are inserted or removed from the rolling stock or the reservoir (e.g. when a plan is executed), the sum of wagons in the rolling stock and the reservoir is constant.

Chapter 11

Station and Line Management

Station management consists of:

1. Train Dispatch consists of the monitoring and controlling train traffic according to schedules. This usually involves the opening and closing of station routes, reschedulings if required, etc.

   Train dispatch involves route planning and signalling, i.e. the setting of unit states.

   Train dispatch also involves determination of train positions.

2. Shunting and Marshalling: Given description of the status (whereabouts, availability, etc.) of rolling stock, including train bodies waiting to be decomposed, and given description of train bodies to be composed, shunting and marshalling implies both the planning for and the execution of plans of shunting and marshalling.

   Shunting and marshalling involves route planning and signalling: i.e. the setting of unit states.

   Shunting and marshalling also involves determination of train body, car and wagon positions.

3. Passenger & Freight Information: Real-time dissemination of the time and other status of all incoming, arrived or departed trains, whether passenger or freight trains, and if the latter, what freight has been received or passed on, and to where.

4. Other Resource Allocation & Scheduling:

   Staff, money, and auxiliary resources need also be managed.

   Among auxiliary resources are: car and wagon cleaning; car and wagon maintenance and repair equipment; freight loading & unloading trucks; etc.

   Their physical and temporal availability, i.e. allocation and scheduling, subject to various rules and regulations, is part of station management.

5. Line Management: Lines may be blocked into several open routes (properly separated by closed routes). The planning and actual setting (i.e. signalling) of corresponding unit states is an essential function of line management.

   Line management also involves determination of train positions.

11.1 Laws of Management

1. **Law of :**
2. **Law of :**
3. **Law of :**
4. **Law of :**