Editors: Dines Bjørner and Martin Pěnička

Towards a TRain Book

— for The RAilway DomaIN

August 5, 2004

Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

.

0

Editorial Remarks

Dines Bjørner

Contents

0.1 State of Document

- This document contains an assortment of draft chapters around the abstract modelling of various aspects of the railway domain.
- The chapters all have different groupings of authors.
- Authorships are noted on the first page of each chapter.
- The document is issued, for the first time, on the occasion of the Topic Session: TRain: The Railway Domain held as part of the IFIP World Computer Congress in Toulouse, France, 23–27 August 2004, `http://www.wcc2004.org/congress/topical_days/top11.htm`.
- It is expected that subsequent editions of this document will be distributed at the FORMS/-FORMAT'2004 event, Braunschweig, Dec. 2–3, 2004, `http://www.forms-2004.de/`.

0.2 Editor & Author Affiliations

- **Dines Bjørner:** School of Computing, Department of Computer Science, National University of Singapore, 3 Science Drive 2, Singapore 117543, bjorner@comp.nus.edu.sg
- **Chris George:** UNU–IIST (UN University's Intl. Inst. f. Software Technology), P.O.Box 3058, Macau SAR, China, cwg@iist.unu.edu
- **Li Yang Fang:** School of Computing, Department of Computer Science, National University of Singapore, 3 Science Drive 2, Singapore 117543, liyf@comp.nus.edu.sg
- **Martin Pěnička:** Faculty of Transportation Sciences, Czech Technical University, Na Florenci 25, CZ-11000 Prague 1, The Czech Republic, penicka@fd.cvut.cz
- **Søren Prehn:** Terma Inc., Vasekær 12, DK–2730 Herlev, Denmark, spn@terma.com
- **Albena Strupchanska:** Linguistic Modelling Department, Central Laboratory for Parallel Processing, Bulgarian Academy of Sciences, Acad. G. Bonchev Str. Bl. 25A, 1113 Sofia, Bulgaria, albena@lml.bas.bg

0.3 Chapter Acknowledgements

2. **Railway Net:** This chapter has evolved over the last 10 years. This is reflected, not only in the co–authorships, but also in the cryptic "&c.": Reflecting that many students, too numerous, and otherwise, to list by names. The chapter is believed to form a viable basis for a proper domain model of railway nets.

3. **Modelling Rail Nets and Time Tables using OWL:** This chapter is a draft. As its contents reveal, there is much work needed to be done. We include it so that no–one can come and say: *"Aha ! All very good and well, but you really are building an ontology, and ought do so (also) in the proper tradition of knowledge representation, cum ontology languages."* We are doing so.
4. **Rolling Stock Maintenance:** This chapter represents work partially sponsored by the EU IST Research Training Network AMORE: Algorithmic Models for Optimising Railways in Europe: http://www.inf.uni-konstanz.de/algo/amore/. Contract no. HPRN-CT-1999-00104, Proposal no. RTN1-1999-00446. It was done during several stays, by the first two authors, at the Institute for Informatics and Mathematical Modelling at the Technical University of Denmark during 2002 and 2003.
   The chapter is a slightly revised version of the published conference paper [43] for FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems. Published by L'Harmattan Hongrie. Conf. held at Techn.Univ. of Budapest, Hungary, May 2003.
5. **Rostering:** This chapter represents work partially sponsored by the EU IST Research Training Network AMORE: Algorithmic Models for Optimising Railways in Europe: http://www.inf.uni--konstanz.de/algo/amore/. Contract no. HPRN-CT-1999-00104, Proposal no. RTN1-1999-00446. It was done during several stays, by the first two authors, at the Institute for Informatics and Mathematical Modelling at the Technical University of Denmark during 2002 and 2003.
   The chapter is a slightly revised version of the published conference paper [49] for FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems. Published by L'Harmattan Hongrie. Conf. held at Techn.Univ. of Budapest, Hungary, May 2003.
6. **Station Interlocking:** This chapter represents work partially sponsored by FET, the Future and Emerging Technologies arm of the IST Programme, FET-Open scheme, as part of CoLogNET, the Computational Logic Network.
   This chapter will be part of Martin Pěnička's forthcoming PhD Thesis. A precursor was published as part of [9] and presented by Dines Bjørner at *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering* at ETAPS, Barcelona, Spain, March 2004.
7. **Signalling on Lines:** This chapter represents work partially sponsored by FET, the Future and Emerging Technologies arm of the IST Programme, FET-Open scheme, as part of CoLogNET, the Computational Logic Network.
   This chapter will be part of Martin Pěnička's forthcoming PhD Thesis. A precursor was published as part of [9] and presented by Dines Bjørner at *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering* at ETAPS, Barcelona, Spain, March 2004.
8. **Line Direction Agreement:** This chapter represents work partially sponsored by FET, the Future and Emerging Technologies arm of the IST Programme, FET-Open scheme, as part of CoLogNET, the Computational Logic Network.
   This chapter will be part of Martin Pěnička's forthcoming PhD Thesis. A precursor was published as part of [9] and presented by Dines Bjørner at *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering* at ETAPS, Barcelona, Spain, March 2004.
9. **Towards a Formal Model of CyberRail:** The work reported here was in response to Takahiro Ogino's extensive and exciting work on a concept of ubiquitous computing & communication in the context of passenger railway service. Our presentation here marks the first semi–public appearance of our model.

[illegible]

∅

National University of Singapore, August 5, 2004

Contents

Part I

Opening

1

Introduction

Dines Bjørner

The aim of collecting the documents, that appear in this "compendium", is to demonstrate the conjecture: *"It appears highly plausible that one can develop major parts of a set of formal descriptions of a domain — such as, for example, the railway domain."*

We have elsewhere, `www.RailwayDomain.org`, outlined a Grand Challenge for computing science, namely that of developing, over the next N years, where N may be 20 or so, a comprehensive and reasonably complete set of commensurate, ie., integrated formal models of *"All things Railways !"*

The present document shall serve to make this claim plausible, shall serve to indicate what might me meant by such a set of *comprehensive and reasonably complete set of commensurate, ie., integrated formal models.*

It is hope that the mere, somewhat inofficial appearance of this compendium — and its limited physical distribution together with its Internet posting, search under `www.RailwayDomain.org`'s `Repository` entry — will help us all better find out what it really is we want !

Most of the chapters of this compendium make use of

- the `RAISE` Specification Language, `RSL,` well documented in several books: [10, 16, 17].

  A brief primer on `RSL` is given in Appendix A.
  References to other formalisms are given in [6].
  Some of these, as directly built upon in some chapters of the present compendium are:

- `Petri Nets` [44, 31, 45, 46, 48],
- `Statecharts` [18, 19, 22, 21, 24, 26], and
- `Live Sequence Charts` [11].

Part II

Basic Railway Domain Model

2

Railway Net

Dines Bjørner, Chris George, Søren Prehn, et al.

Contents

2.1 Basic Static Attributes

We introduce the phenomena of railway nets, lines, stations, tracks, (rail) units, and connectors. We designate such components of the rail net which can be physically demonstrated, but we abstract from number of physical attributes at the moment – they can be always be simply "added" later on.

This description is "top-down": most composite notions are mentioned first, and defined in terms of successively less composite quantities.

Our natural, professional railway language description proceeds as follows:

1. A railway net consists of one or more lines and two or more stations.
2. A railway net consists of rail units.
3. A line is a linear sequence of one or more linear rail units.
4. The rail units of a line must be rail units of the railway net of the line.
5. A station is a set of one or more rail units.
6. The rail units of a station must be rail units of the railway net of the station.
7. No two distinct lines and/or stations of a railway net share rail units.
8. A station consists of one or more tracks.
9. A track is a linear sequence of one or more linear rail units.
10. No two distinct tracks share rail units.
11. The rail units of a track must be rail units of the station (of that track).
12. A rail unit is either a linear, or is a switch, or a is simple crossover, or is a switchable crossover.
13. A rail unit has two or more connectors.
14. A linear rail unit has two distinct connectors, a switch rail unit has three distinct connectors, crossover rail units have four distinct connectors (whether simple or switchable), etc.
15. For every connector there are at most distinct two rail units which have that connector in common.
16. Every line of a railway net is connected to exactly two, distinct stations of that railway net.
17. A linear sequence of (linear) rail units is a non-cyclic sequence of linear units such that neighbouring units share connectors.
18. A path, $p : P$, is a pair of connectors, (c, c'),
19. which are distinct,
20. and of some unit.
21. A state, $\sigma : \Sigma$, of a unit is the set of all possible paths of that unit (at the time observed).
22. A route is a sequence of pairs of units and paths —
23. such that the path of a unit/path pair is a possible path of some state of the unit, and such that “neighbouring” connectors are identical.

type
  N, L, S, Tr, U, C

18. $\mathrm{P}' = \mathrm{C} \times \mathrm{C}$
19. $\mathrm{P} = \{|\ (\mathrm{c},\mathrm{c}'):\mathrm{P}' \bullet \mathrm{c} \neq \mathrm{c}'\ |\}$
21. Σ = P-**set**
22. $\mathrm{R}' = (\mathrm{U} \times \mathrm{P})^*$
23. $\mathrm{R} = \{|\ \mathrm{r}:\mathrm{R}' \bullet \mathrm{wf\_R(r)}\ |\}$

value
1. obs_Ls: N → L-**set**,
1. obs_Ss: N → S-**set**,
2. obs_Us: N → U-**set**,
3. obs_Us: L → U-**set**,
5. obs_Us: S → U-**set**,
8. obs_Trs: S → Tr-**set**,
9. obs_Us: Tr → U-**set**,
12. is_Linear: U → **Bool**,
12. is_Switch: U → **Bool**,
12. is_Simple_Crossover: U → **Bool**,
12. is_Switchable_Crossover: U → **Bool**,
13. obs_Cs: U → C-**set**,
17. lin_seq: U-**set** → **Bool**

```
    lin_seq(us) ≡
        ∀ u:U • u ∈ us ⇒ is_Linear(u) ∧
        ∃ q:U* • len q = card us ∧ elems q = us ∧
    ∀ i:**Nat** • {i,i+1} ⊆ **inds** q ⇒ ∃ c:C •
        obs_Cs(q(i)) ∩ obs_Cs(q(i+1)) = {c} ∧
    **len** q > 1 ⇒
        obs_Cs(q(i)) ∩ obs_Cs(q(**len** q)) = {}
```

21. obs_Σ: U → Σ

Some formal axioms are now given:

```
axiom
1. ∀ n:N • card obs_Ls(n) ≥ 1,

1. ∀ n:N • card obs_Ss(n) ≥ 2,

3. ∀ n:N, l:L • l ∈ obs_Ls(n) ⇒ lin_seq(obs_Us(l))

4. ∀ n:N, l:L • l ∈ obs_Ls(n) ⇒ obs_Us(l) ⊆ obs_Us(n)

5. ∀ n:N, s:S • s ∈ obs_Ss(n) ⇒ card obs_Us(s) ≥ 1

6. ∀ n:N, s:S • s ∈ obs_Ls(n) ⇒ obs_Us(s) ⊆ obs_Us(n)

7. ∀ n:N, l,l′:L •
       {l,l′} ⊆ obs_Ls(n) ∧ l ≠ l′
           ⇒ obs_Us(l) ∩ obs_Us(l′) = {}

7. ∀ n:N, l:L, s:S •
       l ∈ obs_Ls(n) ∧ s ∈ obs_Ss(n)
           ⇒ obs_Us(l) ∩ obs_Us(s) = {}

7. ∀ n:N, s,s′:S •
       {s,s′} ⊆ obs_Ss(n) ∧ s≠s′
           ⇒ obs_Us(s) ∩ obs_Us(s′) = {}

8. ∀ s:S • card obs_Trs(s) ≥ 1

9. ∀ n:N, s:S, t:Tr •
       s ∈ obs_Ss(n) ∧ t ∈ obs_Trs(s) ⇒ lin_seq(obs_Us(t))

10. ∀ n:N, s:S, t,t′:Tr •
       s ∈ obs_Ss(n) ∧ {t,t′} ⊆ obs_Trs(s) ∧ t ≠ t′
           ⇒ obs_Us(t) ∩ obs_Us(t′) = {}

11. ∀ s:S, t:Tr, u:U • u ∈ obs_Us(t) ∧ t ∈ obs_Trs(s) ⇒ u ∈ obs_Us(s)

13. ∀ u:U • card obs_Ls(n) ≥ 2

14. ∀ u:U •
       is_Linear(u) ⇒ card obs_Cs(u) = 2,
       is_Switch(u) ⇒ card obs_Cs(u) = 3,
       is_Simple_Crossover(u) ⇒ card obs_Cs(u) = 4,
       is_Switchable_Crossover(u) ⇒ card obs_Cs(u) = 4

15. ∀ n:N • ∀ c:C •
```

```
    c ∈ ⋃ { obs_Cs(u) | u:U • u ∈ obs_Us(n) }
       ⇒ card{ u | u:U • u ∈ obs_Us(n) ∧ c ∈ obs_Cs(u) } ≤ 2

16. ∀ n:N, l:L • l ∈ obs_Ls(n) ⇒
    ∃ s,s′:S • {s,s′} ⊆ obs_Ss(n) ∧ s≠s′ ⇒
       let sus = obs_Us(s), sus′ = obs_Us(s′), lus = obs_Us(l) in
          ∃ u:U • u ∈ sus, u′:U • u′ ∈ sus′, u′′,u′′′:U • {u′′,u′′′} ⊆ lus •
          let scs = obs_Cs(u), scs′ = obs_Cs(u′), lcs = obs_Cs(u′′), lcs′ = obs_Cs(u′′′) in
             ∃ ! c,c′:C • c ≠ c′ ∧ scs ∩ lcs = {c} ∧ scs′ ∩ lcs′ = {c′} end end

20. ∀ u:U • let ω = obs_Ω(u), σ = obs_Σ(u) in
    σ ∈ ω ∧ ∀ (c,c′):P • (c,c′) ∈ ⋃ ω ⇒ {c,c′} ⊆ obs_Cs(u) end

23.  wf_R: R′ → Bool
   wf_R(r) ≡
      len r > 0 ∧
      ∀ i:Nat • i ∈ inds r let (u,(c,c′)) = r(i) in
         (c,c′) ∈ ⋃ obs_Ω(u) ∧ i+1 ∈ inds r ⇒ let (_,(c′′,_)) = r(i+1) in c′ = c′′ end end
```

2.2 Further Static Attributes

2.2.1 Networks

A network 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 network share no paths. These rules express how one may compose units into networks. For example the unit compositions of figure 2.1 will not be legal in any network.

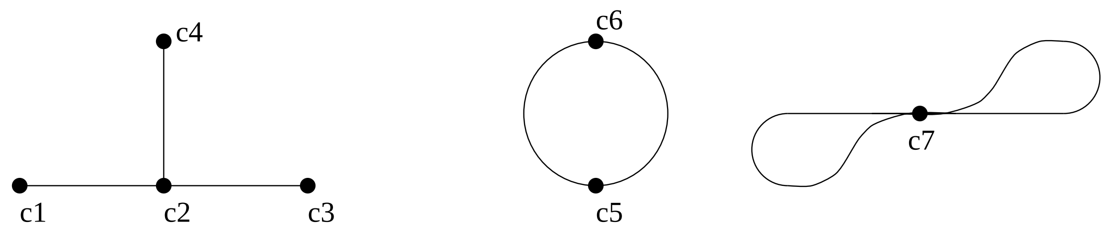


Fig. 2.1. Illegal compositions of units

```
type
   N
value
   obs_Us: N → U-set
axiom
   /∗ In a network, a connector connects no more than two units ∗/
   ∀ n:N • ∀ c:C • c ∈ ⋃ { obs_Cs(u) | u:U • u ∈ obs_Us(n) }
      ⇒ card{u | u:U • u ∈ obs_Us(n) ∧ c ∈ obs_Cs(u)} ≤ 2

   /∗ In a network, two units do not contain the same path ∗/
   /∗ Needs to bee fixed  ∗/
    ∀ n:N, u,u′:U •
      {u,u′} ⊆ obs_Us(n) ∧ u≠u′ ⇒ obs_Σ(u) ∩ obs_Σ(u′) = {}
```

2.2.2 Lines and Stations

A network consists of lines and stations. That is, the units of a network can be decomposed into those belonging to stations and those belonging to lines. A line is a linear sequence of linear units. A station is any set of units, including linear, junctions (switches), crossovers and switchable crossovers. 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. Part of tracks form platforms, sidings, etc. A line always connects exactly two distinct stations.

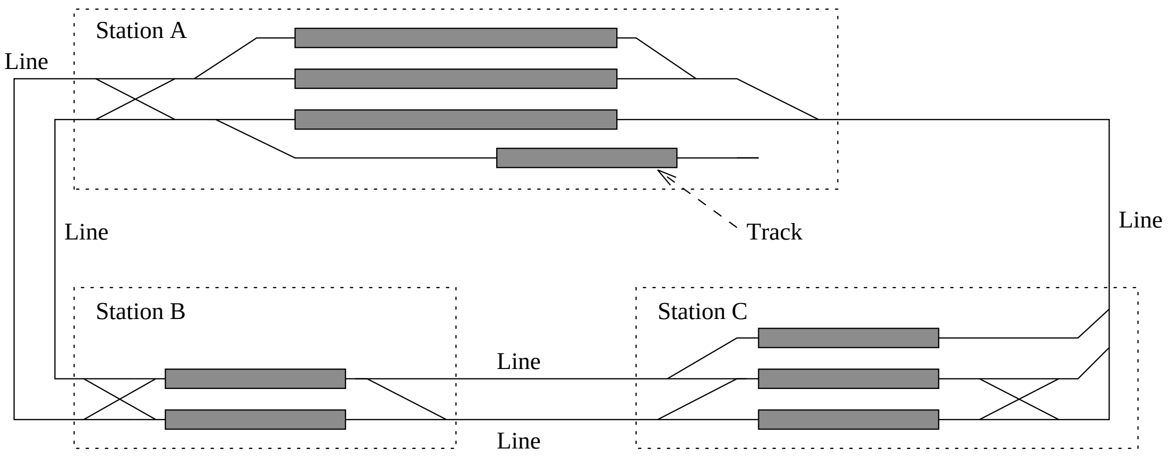


Fig. 2.2. A network of lines and stations

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 (to a unit of which some [end] line [unit] is connectable) 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
   N, L, S, Tr

value
   obs_Ls: N → L-set,
   obs_Ss: N → S-set,
   obs_Us: L → U-set,
   obs_Us: S → U-set,
   obs_Us: Tr → U-set,
   obs_Trs: S → Tr-set,

   /∗ Examine if a line connects to a station ∗/
   LS_Connection: N × L × S → Bool
   LS_Connection(n,l,s) ≡
      ∃ u,u′:U • u ∈ obs_Us(l) ∧ u′ ∈ obs_Us(s) ∧
         ∃ c:C • obs_Cs(u) ∩ obs_Cs(u′) = {c}
      pre l ∈ obs_Ls(n) ∧ s ∈ obs_Ss(n)


   /∗ Examine if two stations are connected via a line ∗/
```

```
    SLS_Connection: N × S × L × S → Bool
    SLS_Connection(n,s,l,s′) ≡
       LS_Connection(n,l,s) ∧ LS_Connection(n,l,s′),


    /∗ Examine if a line connects to a track in a station ∗/
    LTr_Connection: N × L × S × Tr → Bool
    LTr_Connection(n,l,s,t) ≡
       ∃ q:U∗ • ∀ i:Nat • {i,i+1} ⊆ inds q ⇒
          ∃ c:C • obs_Cs(q(i)) ∩ obs_Cs(q(i+1)) = {c} ∧
             q(1) ∈ obs_Us(l) ∧ q(len(q)) ∈ obs_Us(t) ∧
             ∀ u:U • u elems tl q ⇒ u ∈ obs_Us(s)
       pre l ∈ obs_Ls(n) ∧ s ∈ obs_Ss(n) ∧ t ∈ obs_Trs(s),


    /∗ All lines that can be reached directly from
       a given track in a given station ∗/
    TrLs: N × S × Tr →̃ L-set
    TrLs(n,s,t) ≡
       {l | l:L • l ∈ obs_Ls(n) ∧ LTr_Connection(n,l,s,t)}
       pre t ∈ obs_Trks(s) ∧ s ∈ obs_Ss(n),


    /∗ All tracks in a given station that can be reached
       directly from a given line ∗/
    LTrs: N × L × S →̃ Trk-set
    LTrs(n,l,s) ≡
       {t | t:Tr • t ∈ obs_Trs(s) ∧ LTr_Connection(n,l,s,t)}
       pre l ∈ obs_Ls(n) ∧ s ∈ obs_Ss(n),


    /∗ Examine if a set of units is part of some network ∗/
    IsInNet: U-set → Bool
    IsInNet(us) ≡ ∃ n:N • us ⊆ obs_Us(n),

axiom
    ∀ n:N, l,l′:L, s,s′:S, t,t′:Tr, c:C, u:U •

    /∗ Lines are part of some network ∗/
    IsInNet(obs_Us(l)),

    /∗ Lines consist only of linear units ∗/
    u ∈ obs_Us(l) ⇒ is_Linear(u),

    /∗ Tracks are part of some station ∗/
    ∃ s:S • obs_Us(t) ⊆ obs_Us(s),

    /∗ Tracks consist of linear units ∗/
    u ∈ obs_Us(t) ⇒ is_Linear(u),

    /∗ Tracks of a station do not intersect ∗/
    {t,t′} ⊆ obs_Trs(s) ∧
       t≠t′ ⇒ obs_Us(t) ∩ obs_Us(t′) = {},
    /* Lines in a network do not intersect */
    {l,l′} ⊆ obs_Ls(n) ∧ l≠l′ ⇒
       obs_Us(l) ∩ obs_Us(l′) = {},

    /* Stations are part of some network */
    IsInNet(obs_Us(s)),

    /* Stations in a network do not intersect */
    {s,s′} ⊆ obs_Ss(n) ∧ s≠s′ ⇒
       obs_Us(s) ∩ obs_Us(s′) = {},

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

    /* Lines connect exactly two stations */
    l ∈ obs_Ls(n) ⇒
       ∃ s,s′:S •
          s≠s′ ∧ {s,s′} ⊆ obs_Ss(n) ∧
          SLS_Connection(n,s,l,s′),

    /* Stations do not have common connectors */
    {s,s′} ⊆ obs_Ss(n) ∧ s≠s′ ⇒
       Us_Cs(obs_Us(s)) ∩ Us_Cs(obs_Us(s′)) = {}
```

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

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

   Sns: N → Sn-set
   Sns(n) ≡ dom obs_SnSm(n)
axiom
   ∀ n:N • obs_Ss(n) = rng obs_SnSm(n) ∧ card obs_Ss(n) = card Sns(n)
```

2.2.3 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 we could derive the lengths, of a unit, 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. *&c.*

2.2.4 Path

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 all the paths, that are possible directions of travel through the unit.

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.

```
type
  P = C × C,
  $\Sigma$ = P-set,
value
  obs_$\Sigma$: U → $\Sigma$,

  /* All physically possible paths through a unit */
  U_Ps: U → P-set
  U_Ps(u) ≡
    { p | p:P • ∃ $\sigma$:$\Sigma$ •
        $\sigma$ ∈ obs_U_$\Omega$(u) ∧ p ∈ $\sigma$
    },

  /* 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_$\Sigma$(u) ∈ obs_U_$\Omega$(u),

  /* All connectors of paths in states are connectors of the unit */
  ∀ u:U, $\sigma$:$\Sigma$, (c,c′):P •
    $\sigma$ ∈ obs_U_$\Omega$(u) ∧ (c,c′) ∈ $\sigma$ ⇒
    {c,c′} ⊆ obs_U_Cs(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:

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

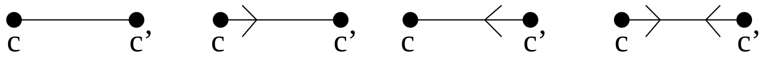


Fig. 2.3. States of a linear unit

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:

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

The 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:

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

Etcetera 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
  ∀ 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) = {}
```

2.2.5 Routes

The concept of routes play an important role in speaking about train journies. A route is a sequence of connectors. The connectors of a route designate paths in some network. That is, directions of travel.

A route is feasible in a network, if the route describes only possible paths though units of the network.

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

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

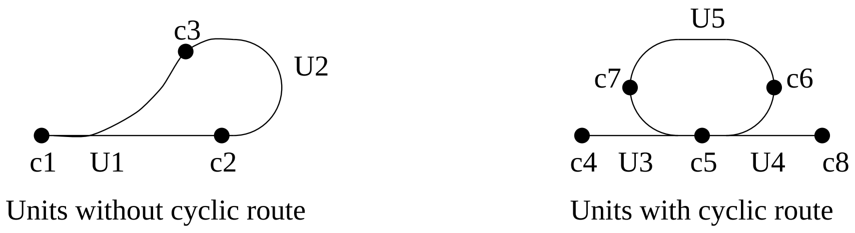


Fig. 2.4. Cyclic and acyclic routes

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
   Rt′ = C*,
   Rt = {| rt:Rt′ • wf_Rt(rt) |}
value
   /* Wellformed routes have lenght at least two and
      are feasible in some network */
   wf_Rt: Rt′ → Bool
   wf_Rt(rt) ≡ len rt ≥ 2 ∧ ∃ n:N • feasible_Rt(rt,n),

   /* A route is feasible wrt a network if the route designates
      possible paths in the network and the route does not
      designate two succesive paths through the same unit */
   feasible_Rt: Rt′ × N → 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 network */
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 $\overset{\sim}{\rightarrow}$ 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 $\overset{\sim}{\rightarrow}$ U-set
Rt_Us(rt,n) ≡ elems Rt_Ul(rt,n)
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 $\overset{\sim}{\rightarrow}$ 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 $\overset{\sim}{\rightarrow}$ 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 network */
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 $\overset{\sim}{\rightarrow}$ 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 $\overset{\sim}{\rightarrow}$ 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 $\overset{\sim}{\rightarrow}$ 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),


/* All possible routes that use all units in a set */
Us_complete_Rts: U-set $\overset{\sim}{\rightarrow}$ 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)
```

2.3 Basic Dynamic Attributess

We introduce defined concepts such as paths through rail units, state of rail units, rail unit state spaces, routes through a railway network, open and closed routes, trains on the railway net, and train movement on the railway net.
[1]

1. A unit may, over its operational life, attain any of a (possibly small) number of different states ω, Ω.

[1] A path of a unit designate that a train may move across the unit in the direction from c to c'. We say that the unit is open in the direction of the path.

2. An open route is a route such that all its paths are open.
3. A train is modelled as a route.
4. Train movement is modelled as a discrete function (ie., a map) from time to routes such that for any two adjacent times the two corresponding routes differ by at most one of the following:
   a) a unit path pair has been deleted (removed) from one end of the route;
   b) a unit path pair has been deleted (removed) from the other end of the route;
   c) a unit path pair has been added (joined) from one end of the route;
   d) a unit path pair has been added (joined) from the other end of the route;
   e) a unit path pair has been added (joined) from one end of the route, and another unit path par has been deleted (removed) from the other end of the route;
   f) a unit path pair has been added joined) from the other of the route, and another unit path par has been deleted (removed) from the one end of the route;
   g) or there has been no changes with respect to the route (yet the train may have moved);

   and such that the new route is a well–formed route.

We shall arbitrarily think of "one end" as the "left end", and "the other end", hence, as the "right end" — where 'left', in a model where elements of a list is indexed from 1 to its length, means the index 1 position, and 'right' means the last index position of the list.

```
type
1. Ω = Σ-set
3. Trn = R
4. Mov′ = T →m Trn
4. Mov = {| m:Mov′ • wf_Mov(m) |}

value
1. obs_Ω: U → Ω

axiom
2. open_R: R → Bool
    open_R(r) ≡
       ∀ (u,p):U×P • (u,p) ∈ elems r ∧ p ∈ obs_Σ(u)

4. wf_Mov: Mov → Bool
    wf_Mov(m) ≡ card dom m ≥ 2 ∧
       ∀ t,t′:T • t,t′ ∈ dom m ∧ t < t′
       ∧ adjacent(t,t′) ⇒
          let (r,r′) = (m(t),m(t′))
             (u,p):U×P • p ∈ ⋃ obs_Ω(u) in
4a.          (l_d(r,r′,(u,p)) ∨
4b.           r_d(r,r′,(u,p)) ∨
4c.           l_a(r,r′,(u,p)) ∨
4d.           r_a(r,r′,(u,p)) ∨
4e.           l_d_r_a(r,r′,(u,p)) ∨
4f.           r_d_l_a(r,r′,(u,p)) ∨
4g.           r=r′)
          end
```

The last line's route well–formedness ensures that the type of Move is maintained.

```
value
   adjacent: T × T → Bool
   adjacent(t,t′) ≡ ∼∃ t″:T • t″ ∈ dom m ∧ t < t″ < t′

   l_d,r_d,l_a,r_a,l_d_r_a,r_d_l_a: R × R × P → Bool
```

```
l_d(r,r′,(u,p)) ≡ r′ = tl r pre len r>1
r_d(r,r′,(u,p)) ≡ r′ = fst(r) pre len r>1
l_a(r,r′,(u,p)) ≡ r′ = ⟨(u,p)⟩⌢r
r_a(r,r′,(u,p)) ≡ r′ = r⌢⟨(u,p)⟩
l_d_r_a(r,r′,(u,p)) ≡ r′ = tl r⌢⟨(u,p)⟩
r_d_l_a(r,r′,(u,p)) ≡ r′ = ⟨(u,p)⟩⌢fst(r)

fst: R ~→ R′
fst(r) ≡ ⟨ r(i) | i in ⟨1..len r−1⟩ ⟩
```

If r as argument to fst is of length 1 then the result is not a well–formed route, but is in R′.

2.4 Further Dynamic Attributes

2.4.1 Path: Open and Closed

A path through a unit is physically open, if it is in the physical state of the unit. If not in the 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. That is, 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. That is, no traffic is intended through the unit.

The managed state of a unit depends on management decisions. The position of the unit in the network 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 away from the hump. The managed states of units in the network are known to the rail net management.

```
type
   P = C × C,
   Σ = P-set,
   Ω = Σ-set
value
   obs_U_Ω: U → Ω,
   obs_U_Physical_Σ: U → Σ,
   obs_U_Managed_Σ: U → Σ,

axiom
   /∗ Managed states are subsets of Physical states ∗/
   ∀ u:U • obs_U_Managed_Σ(u) ⊆ obs_U_Physical_Σ(u)
```

2.4.2 Routes: Open and Closed

A route is physically open in a given network, if the connectors of the route designate physically open paths in units of the network. That is, the units are open in direction of the route.

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

```
type
    Rt' = C*,
    Rt = {| rt:Rt' • wf_Rt(rt) |}
value

    /* Examine if a route is physically open */
    is_Physical_OpenRt: Rt × N $\xrightarrow{\sim}$ 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 $\xrightarrow{\sim}$ 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),
```

2.4.3 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 network.

A wellformed move of a train route is that of not changing the route, adding a connector to the end of the route, removing a connector from the beginning of the route or simultaniously adding a connector to the end and removing a connector from the beginning of the route. 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)
```

2.4.4 Managed Rail Nets

A managed rail net “snap shot”, i.e. a managed rail net state, is a rail net such that all units are in each their own state.

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, say, of one unit, and each such segment or block being delineated by a signal. (That is: 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. Etcetera!

Since rail nets are regularly being updated: new line and station units are added, old removed entirely, or put under repair, etc., we have that 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, each change defined in terms of rail net closing, removal, insertion and opening actions.

```
  T,
  MR′ = T → N,
  MR = {| mr:MR′ • wf_MR(mr) |}
value
  wf_MR: MR′ → Bool
  wf_MR(mr) ≡
      ∀ t:T • ∃ t′:T • t′>t ∧
          ∀ t″:T • t≤t″≤t′ ⇒ MoN(mr(t),mr(t″)),

  MoN: N × N → 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′)
```

2.4.5 Stable, Transition and Re–organisation States

A unit is at any one time either in a stable state, or in a transition state, or in a reconfiguration state. A rail unit event is one where a rail unit changes from one kind of state to another. In all: Three kinds of states and four kinds of events.

We have decided to model "transitions" from stable states to stable states as not taking place instantaneously, but having some time duration. During that time of change we say that the rail unit is in a transition state.

Reconfiguration states are like transition states, but, in addition, the rail units changes basic characteristics.

2.4.6 Time and State Durations

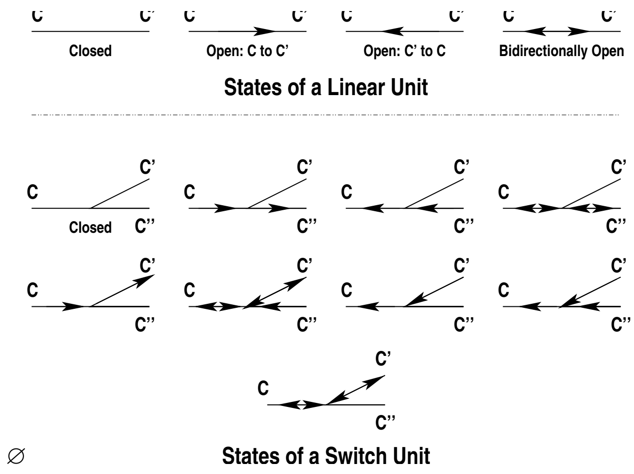
Units remain in stable, transition and reconfiguration states "for some time" ! We decide to endow each unit with possibly different minimum stable state, and maximum transition and reconfiguration state durations: A unit, irrespective of its state, must remain in any stable state for a minimum duration of time. A unit, irrespective of its state, at most remains in any transition state for a maximum duration of time. A unit, irrespective of its state, at most remains in any reconfiguration state for a maximum duration of time. The stable state minimum duration is (very much) larger than the maximum re–configuration duration, which again is (very much) larger than the maximum transition duration.

```
type
   T /∗ T is some limited, dense time range ∗/
   Δ /∗ δ: Δ is some time duration ∗/
   sΔ = Δ, tΔ = Δ, rΔ = Δ
value
   lo_T: U → T
   obs_ℓ_T: U → T
   obs_sΔ: U → sΔ,
   obs_tΔ: U → tΔ,
   obs_rΔ: U → rΔ
   ≪,<,>,≫: Δ × Δ → Bool
   ≪,<,>,≫: T × T → Bool
   +,−: T × Δ → T
   ∗: Δ × Real → Δ  pre δ∗r: r>0
axiom
   ∀ u:U •
      obs_sΔ(u)≫obs_rΔ(u)≫obs_tΔ(u),
   ∀ 1δ,2δ:Δ •
      1δ≪2δ⇒1δ<2δ ∧ 1δ≫2δ⇒1δ>2δ
```

2.4.7 Stable States

A stable state (of a unit) is a possibly empty set of pairs of connectors of that unit. At any one time, when in a stable state, a unit is willing to be in any one of a number of states, its (current) state space. If a pair of connectors is in some stable state then that means that a train can move across the unit in the direction implied by the pairing: from the first connector to the second connector. A unit in a stable state has been so for a duration — which we assume can be observed.

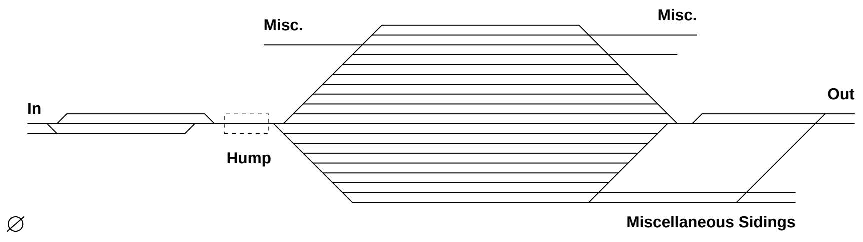
Figure 2.4.7 shows two kinds of rail units and the possible stable states they may 'occupy'.



The arrows of Figure 2.4.7 shall designate possible ("open") directions of (allowed, "free") movement. To be able to compare units, and to say that a unit at one time, in some state, "is the same" as a unit, at another time, in another state, we introduce a "normalisation" function: nor_Σ. It behaves as if it "resets" the current state of a unit to the empty state, and as if the elapsed time is "zero" — leaving all else unchanged. [2]

```
type
  $PS = P\textbf{-set}$
  $s\Sigma = PS$
value
  $is\_s\Sigma: U \to \textbf{Bool}$
  $obs\_s\Sigma: U \xrightarrow{\sim} s\Sigma$
  $obs\_s\Omega: U \to s\Omega$
  $obs\_\Delta: U \to \Delta$
  $obs\_s\Delta: U \to s\Delta$
  $nor\_\Sigma: U \to U$
axiom
  $\forall u:U \bullet$
    $is\_s\Sigma(u) \Rightarrow$
      $obs\_s\Sigma(u) \in obs\_s\Omega(u) \wedge$
      $obs\_s\Sigma(u) \subseteq obs\_Ps(u) \wedge$
      $obs\_s\Sigma(nor\_\Sigma(u)) = \{\}$
```

[2] The latter is, however, not formalised. But ought be.



2.4.8 Transition States

When a unit is in a transition state it is making a transition from one stable state to another.

We now make the following crucial modelling decision: Since we are dealing, throughout, with man–made phenomena, with entities most of whose properties we “design into” these physical “gadgets” we can assume the following: That we can observe from the rail units “their intention”: Namely, in this case, that they are to make a transition from one, known, stable state to another, known, stable state, and that, at any one time of observing such a transition, we can also observe the elapsed time duration since the start of a transition.

```
type
   tΣ={|(s′σ,s″σ):(sΣ×sΣ)•s′σ≠s″σ|}
   tΩ = tΣ-set
value
   is_tΣ: U → Bool
   obs_tΣ: U → tΣ
   obs_tΩ: U → tΩ
   obs_Δ: U → Δ
   obs_tΔ: U → tΔ
axiom
   ∀ u:U,s′σ,s″σ:Σ •
    is_tΣ(u) ⇒ (s′σ,s″σ) = obs_tΣ(u) ⇒
      (s′σ,s″σ) ∈ obs_tΩ(u)∧{s′σ,s″σ}⊆obs_sΩ(u)
```

The dynamics of this change will be elaborated upon later. Suffice it to hint that the change from a stable state to the “beginning” of a transition state is an event, likewise is the change from a transition state to the stable state, and the stable state of the unit “just” before the transition state must be the same as the first stable state of the pair of the transition state, while the stable state of the unit “just” after the transition state must be the same as the second stable state of the pair of the transition state.[3]

2.4.9 Reconfiguration States

A rail unit may be subject to reconfiguration: In a net some existing (ie., “old”) rail units need be “changed” by allowing “additional”, or dis–allowing “previously valid” paths, hence changing the state space, or by allowing new kinds of transitions, or both. Reconfiguration additionally permits new units to be “connected” to existing units’ “dangling” connectors.

A rail unit reconfiguration thus changes its state space — from a past to a future state space, and therfore also by changing into a future transition state space, while possibly changing the unit from one stable state (of the past state space) to another (of the future state space) — where we impose the seemingly arbitrary constraint that the transition state (ie., the pair of before and after stable states) must be in both the “old” and the “new” set of transition states.

[3] We allow this seeming redundancy of representation in order to simplify some subsequent formalisations.

```
type
    $r\Sigma' = (s\Omega \times s\Sigma \times t\Omega) \times (t\Omega \times s\Sigma \times s\Omega)$
    $r\Sigma = \{|\ r\sigma{:}r\Sigma' \bullet wf\_r\Sigma(r\sigma)\ |\}$
value
    $wf\_r\Sigma$: $r\Sigma' \rightarrow$ Bool
    $wf\_r\Sigma((s'\omega,s'\sigma,t'\omega),(t''\omega,s''\sigma,s''\omega)) \equiv$
        $s'\sigma \in s'\omega \wedge s''\sigma \in s''\omega \wedge$
        $(s'\sigma,s''\sigma) \in t'\omega \cap t''\omega \wedge$
        $\bigcup s'\omega \cup \bigcup s''\omega \subseteq obs\_Ps(u) \wedge$
        $\forall\ (sa\sigma,sb\sigma){:}t\Sigma \bullet$
            $(sa\sigma,sb\sigma) \in t'\omega \Rightarrow \{sa\sigma,sb\sigma\} \subseteq s'\omega \wedge$
            $(sa\sigma,sb\sigma) \in t''\omega \Rightarrow \{sa\sigma,sb\sigma\} \subseteq s''\omega$
    $is\_r\Sigma$: U $\rightarrow$ Bool
    $obs\_r\Sigma$: U $\xrightarrow{\sim}$ $r\Sigma$
    $obs\_\Delta$: U $\rightarrow \Delta$
    $obs\_r\Delta$: U $\rightarrow r\Delta$
```

We thus see that a reconfiguration state embodies also a transition state. And thus we inherit many of the constraints expressed earlier. Now they are part of the well–formedness of any reconfiguration state. For the other state types sorts were constrained via the axioms. A number of decisions have been made: We have decided, in this model, to maintain “redundant” “information”: The before and after stable state spaces, as well as transition state spaces. And we have decided to impose a further “commonality” constraint: The actual state transition taken (“undergone”) during reconfiguration must be one that was allowed before, as well as being allowed after, reconfiguration.

2.5 Dynamical Units: Continuity

A railway net of many units, all timed to the same clock and time period, can be considered ideally an programmed, dynamic active system, less ideally, a dynamic reactive system.

These terms ‘programmed, dynamic active system’, respectively ‘dynamic reactive system’ are, for the realm of computing science and software engineering, that is: Programming methodology, described in [30].

In this section we shall consider railway nets to be ‘programmed’. That is: It is us, the managers of railway nets, who control the time–wise behaviour of the net — to a first approximation. To a second approximation, when ordering the rail units to undergo a reconfiguration and/or a transition, such changes may involve a time duration, such as modelled above. During those durations the rail units behave reactively: Over the time period of the duration they “switch state” in reaction to a control signal.

Although we shall thus primarily consider railway nets as programmed, active dynamic systems, we shall bring a model which appears to model railway nets as more general dynamic, active systems. But one should understand these models appropriately: As reflecting what can be observed from outside the system of railway nets plus their control. We shall subsequently review the above distinctions.

The behaviour of a unit, as seen from outside the railway net and its control, is that it changes from being in stable states and making transitions between these. A state transition is from the stable state before to the stable state after the transition. The stable state components of transition states must be in the current state space. A reconfiguration state transition has its stable states be in the intersection of, ie., in both, the before and after stable state spaces. (This constraint has already been formally expressed.)

2.5.1 Timed Units

We now "lift" a unit to be a timed unit: That is, a function from time, in some dense interval, to "almost the same" unit ! We assume that we can delimit time intervals so that each timed unit is described as from some lower (lo_T) time upwards !

```
type
    T /* T dense, with lower boun */
    TU = T → U
value
    lo_T: TU → T
    ℓ_T: (TU|U) → T
axiom
    ∀ tu:TU • unique_TU(tu)
value
    unique_TU: TU → Bool
    unique_TU(tu) ≡ ∀ t,t′:T • {t,t′}⊆𝒟 tu
        ∧ no_rΣs(tu)(t,t′)⇒same_Us({tu(t),tu(t′)})

    no_rΣs: TU → (T × T) → Bool
    no_rΣs(tu)(t,t′) ≡ is_sΣ(tu(t)) ∧ is_sΣ(tu(t′))
        ∧ ∼∃ t″:T • t<t″<t′ ∧ is_rΣ(tu(t″))

    same_Us: U-set → Bool
    same_Us(us) ≡ ∀ u,u′:U • us ⊆ obs_sΩ(u)
     ∧ obs_sΣ(nor_U(u)) = obs_sΣ(nor_U(u′))
     assert: ∀ u″:U•u″ ∈ us ⇒ us ⊆ obs_sΩ(u″)
```

tu:TUs are continuous functions over their lower limited, although infinite definition set of times.

2.5.2 Operations on Timed Units

In the following we will abstract from the two operations that are implied by the transition state, and the reconfiguration state. That is: We think, now, of these states as having been brought about by controls, ie., by external events and communication between an environment and the net (or, as in the case of timed rail units, between an environment and respective units).

So an operation on a timed unit is something that takes place, at some time, say τ, and which involves an operator. The meaning of the operator is what we model, not the syntax that is eventually needed in order to concretely implement the operation. And that meaning we take to involve the following entities: A function, ϕ, which is like a timed unit, except that its lower time limit is like "0". And a time duration, oδ, for the operation.

The idea is now that applying an operation ϕ at time τ, means that the timed unit function, tu, is "extended" by " glueing" the operation function ϕ to tu "chopped" at τ. After the operation has completed, at time τ+oδ, the unit remains in the state it was left in by ϕ at the end of its completion.

```
value
    lo_δ:Δ, hi_δ:Δ
axiom
    [ lo_δ ‵‵behaves like zero″ ]
type
    Θ
    Φ = ΦΔ → U
    /∗ $\Phi\Delta$: continuous relative time interval ∗/
  $\Phi\Delta$ = {lo$\delta$..hi$\delta$}
value
  obs_$\Phi$: $\Theta \to \Phi$
  obs_o$\Delta$: $\Theta \to$ {hi_$\delta$ .. lo_$\delta$}
  OP: $\Theta \to$ TU $\to$ T $\to$ TU
  OP($\theta$)(tu)($\tau$) $\equiv$
    let $\phi$ = obs_$\Phi$($\theta$),
      o$\delta$ = obs_o$\Delta$($\theta$) assert: o$\delta$=hi_$\delta$−lo_$\delta$,
      lo_t = lo_T(tu) in
    $\lambda$ t:T • if t<lo_t then chaos
       elsif lo_t$\leq$t$\leq\tau$ then tu(t)
       elsif $\tau$<t<$\tau$+o$\delta$ then $\phi$(t−$\tau$)
       elsif t$\geq\tau$+o$\delta$ then $\phi$(o$\delta$) end end
```

In the above — generalised — formulation of the effect of operations on timed units we have abstracted from whether these "stood" for state transitions or state reconfigurations. We have alkso made a number of general assumptions. These we now describe and formalise: The initial unit of the operation must be compatible with (for simplicity we here take it to be: the same as) the unit of the timed unit at the time the operations is applied.

```
  OP($\theta$)(tu)($\tau$) $\equiv$ ...
  pre obs_$\Phi$($\theta$)(lo_$\delta$) = tu($\tau$)
```

One can think of the following constraint being already "syntactically" expressed in the specification of transition and reconfiguration states. We refer to Section 2.4.8 and Section 2.4.9. These state change specifications ("redundantly") specified the "before" and "after" states, where specifying the "after", ie., the final state, would have sufficed.

We leave it to another occassion to provide a proper linkage between specifying the syntactics of the operations and the already specified state change types.

2.5.3 State Sequences

In the previous section we view timed units as something that changed only as the result of applying operations to the (timed) units. In this section we shall revert to looking at timed units as entities which have observable behaviour — ie., which can be observed from a vantage point "outside" the units and the "control machinery" that effects the operations.

Any one unit resides in a sequence of "adjacent" states: (i) For some time in a stable state, ψ, (ii) then, perhaps for a short time in a transation state: $\psi \mapsto \psi'$, (iii) then, as (i–ii): for some time in ψ', then $\psi' \mapsto \psi''$, etc.: (iv) ψ'', $\psi'' \mapsto \psi'''$, ψ''', $\psi''' \mapsto \psi''''$, etcetera. Maybe after a very long time compared to the time span from stable state ψ to stable state $\psi''''\cdots'$, the unit goes into a reconfiguration state. Whereupon (i–iv) is repeated, for a possibly other stable state and transition state sequence. One constraint that rules state changes with respect to state transitions (and, of course, stable states) is expressed below:

```
axiom
  $\forall$ tu:TU •
    $\forall$ t:T • t $\in \mathcal{D}$ tu $\Rightarrow$
      is_t$\Sigma$(tu(t)) $\Rightarrow$
        is_s$\Sigma$(tu(t−obs_t$\Delta$(tu(t)))) $\wedge$
        is_s$\Sigma$(tu(t+obs_t$\Delta$(tu(t)))) $\wedge$
        let (s$'\sigma$,s$''\sigma$) = obs_t$\Sigma$(tu(t)) in
        s$'\sigma$ = obs_s$\Sigma$(t−obs_t$\Delta$(tu(t))) $\wedge$
        s$''\sigma$ = obs_s$\Sigma$(t+obs_t$\Delta$(tu(t))) $\wedge$
        $\{s'\sigma, s''\sigma\} \subseteq$ obs_$\Omega$(tu(t))
        /* last property follows */
        /* from earlier axiom */
        **end**
```

We can formalise a similar constraint for the dynamic behaviour of units before and after undergoing, ie., residing in, reconfiguration states. We will leave that as an exercise.

2.5.4 Dynamical Nets

Railway nets consists of units — and otherwise possess many other properties. We now "lift" the conglomeration of all timed units to one timed net. This has to be understood as follows: Not only does the thus timed net consist of timed units but also of other "things".

2.5.5 Timed Nets

Railway nets consists of units (and possibly more). A timed net is now a continuous function from time to nets. From a timed net (as from units and timed units) we can observe "its" lowest (its "begin" or "start") time.

```
type
  N, U, T
  TU = T $\rightarrow$ U
  TN = T $\rightarrow$ N
value
  lo_T: (U|TU|TN) $\rightarrow$ T
  obs_Us: N $\rightarrow$ U-set
```

For the purposes of our ensuing discussion we make the following simplifying, but not substantially limiting assumptions: For a given timed net, at any time after its "begin" time, it contains the same units as when first "started".

```
  assume: TN $\rightarrow$ T $\rightarrow$ Bool
  assume(tn)($\tau$) $\equiv \forall$ t:T•lo_T(tn)<t$\leq\tau$
      $\Rightarrow$ nor_Us(tn(lo_T(tn)))=nor_Us(tn(t))

  nor_Us: N $\rightarrow$ U-set
  nor_Us(n)$\equiv$\{nor_U(u)|u:U•u $\in$ obs_Us(n)\}

  nor_Us: TN $\rightarrow$ U-set
  nor_Us(tn)$\equiv\bigcup$\{nor_Us(tn(t))|t:T•lo_T(tn)$\leq$t$\leq\tau$\}
```

nor_Us defines an equivalence class over any set of "different" units.

2.5.6 Relations between Timed Nets and Timed Units

From a timed net we can "construct" a set of timed units reflecting the timed behaviour of all the units of the timed net.

```
value
  TN_2_TUs: TN $\rightarrow$ TU-set
  TN_2_TUs(tn) $\equiv$
     { $\lambda$ t:T • if t<lo_T(tu) then chaos
            else capture_U(tn)(u)(t) end
```

```
        | u:U • u ∈ obs_Us(tn(lo_T(u))) }
    pre ∀ t:T • t>lo_T(tn) assume(tn)(t)

  capture_U: TN → U → T → U
  capture_U(tn)(u)(t) ≡
    let n′ = tn(t) in
    let us′ = obs_Us(n′) in
    let u′:U • u′ ∈ us′ ∧ nor_U(u′)=nor_U(u)
    in u′  end end end
```

We can not, alas, define the inverse function:

```
value
  TUs_2_TN: TU-set → TN
  conjecture:
    ∀ tn:TN • ∀ t:T • t>lo_T(tn) assume(tn)(t)
      ⇒ TUs_2_TN(TN_2_TUs(tn)) = tn
```

The reason is that the net is more than the sum of all its units. Had we defined a net to just be the set of all units, then a TUs_2_TN could be defined which satisfies the conjecture. Why is a net more than the sum total of all its units ?

The answer to that question can, for example, be found in [5] We also wish to be able to observe, from a net, The delineations between lines and stations, the embedding, within stations, of tracks within the units of the stations, *&c.*

2.5.7 Selecting Timed Units

Given a timed net and a "prototype" rail unit, that is, a normalised rail unit, we sometimes have a need to find that unit in the net, or, rather, to find "its" timed version:

```
value
  select_TU: TN → U $\xrightarrow{\sim}$ TU
  select_TU(tn)(u) ≡
    let tus = TN_2_TUs(tn) in
    if ∃ tu:TU • tu ∈ tus ∧
      nor_U(tu(lo_T(tu)))=nor_U(u)
      then
        let tu:TU • tu ∈ tus ∧
            nor_U(tu(lo_T(tu)))=nor_U(u) in
        tu end
      else chaos end end
```

2.5.8 Operations on Timed Nets

We have, in Section 2.5.2, defined the general idea of operations on timed units. We now wish to examine what the meaning of these operations are in the context of timed nets. Suppose we could say: Performing an operation on a timed unit of a timed net only affects that timed unit, and not any of the other timed units of the timed net, then performing "that same" operation, somehow identifying the unit, would have to express the above, as is done below:

```
type
  $\Theta$
value
```

```
OP: $\Theta \to (\mathrm{TN} \times \mathrm{U}) \to \mathrm{T} \to \mathrm{TN}$
OP: $\Theta \to \mathrm{TU} \to \mathrm{T} \to \mathrm{TU}$

OP($\theta$)(tn,u)($\tau$) **as** tn$'$
   **pre** $\exists$ u$'$:U •
      u$'$ $\in$ obs_Us(tn($\tau$))$\wedge$nor_U(u$'$)=nor_U(u)
   **post let** tu:U = select_TU(tn)(u) **in**
            tus = TN_2_TUs(tn),
            tus$'$ = TN_2_TUs(tn$'$) **in**
        tus\tus$'$=tus$'$\tus={OP($\theta$)(tu)($\tau$)}
     $\wedge$ ... **end**
```

The ∧ ... part of the above pre/post characterisation of operations on timed units of a timed net refers to the fact that *the whole is more than the sum of its parts*, that is: There may be aspects of the net which are affected by an operation, but not captured by the individual rail units.

2.6 Discussion of the Continuous Model

A model of certain aspects of a railway net has been presented. We could have chosen many different ways of formulating this model.

Next we shall discuss two aspects: Why we have not spoken about the unique identification of units. And whether the model of time (and timing) is the right model.

2.6.1 Why no Unique Unit Identification ?

Perhaps most controversially is our tacit decision not to endow rail units with a unique identification. It is indeed true that each rail unit is unique. It is unique simply by the choice of its connectors. We never made that explicit. But it is indeed contained in the model of railway nets we referred to earlier. See [5]. We could have instead endowed each unit with a unique identifier, but then we would have to express a lot of "book–keeping" constraints to secure that the already existing uniqueness of rail units was not being interfered with by the additional "unique" identifier.

2.6.2 Is it the Right Model of Timing ?

When time is involved in a phenomenon, a good advice, in computing science circles, is usually not to introduce time explicitly in the model till the latest possible step of development — if at all ! It is obviously not an advice we have followed. So: Why not ? For two reasons: The first is, that we would otherwise have modelled timing by means of some combination [51, 27] of RSL, as we have used it, [16, 17], and explicit timing constructs of an extended RSL [51], or, [27], of any one, or more, of the Duration Calculi [53, 55, 56, 54, 52]. Either approach might have "complicated" the presentation of the notations — which we have kept as Annotations in footnotes. So — in anticipation of such a possible complication — we have "cowardly" refrained. The other reason for not choosing to also use the above mentioned blends of RSL and either explicit RSL extending timing constructs, or one or more of the Duration Calculi, is that we wish, in a separate publication to perform those experiments: Of using exactly such "extensions", and then compare the two–three approaches.

In other words: It may not be the right model that we have presented in the current paper. "Time (!) will tell !" (Pun intended.)

2.6.3 Possible Relations to Control Theory

The whole purpose of Section 2.5.4 has been to present a model of a domain that is of interest to both software engineering and control engineering. We have presented “one side of the coin”, the computing science facets of the models of such domains. It now remains to put forward, informally, some ideas that might relate to control theory, and to suggest that classical ideas of control theory, or just plain, simple calculus (ie., the differential and integral calculi) — that ideas from these disciplines — might be of use in further extending the computational models that are encountered when developing software.

The crucial phenomenon that forces us, so to speak, to raise the issue of possible relations — as far as the domain of transport goes — between computing science and control engineering is that of our model of traffic: Let us take a look at the concept of train traffic:

```
type
   Train, Pos
   TF = T → (N × (Train ₘ→ Pos))
```

where T is time, N is the time–varying net, Train stands for trains, and Pos is the position of trains. The timed net follows from traffic:

```
value
   timed_net: TF → TN
   timed_net(tf)≡λt:T•let (n,tps)=tf(t) in n end
```

In control engineering we are used to monitor and control the net and the trains. Here they are brought together in one model. Something that can be done by means of the techniques of computing science, but something that does not seem to be so easy, as here, to express in usual control theoretic ways.

For a given train, say of identity tn:Tn, we may wish to observe its dynamics:

```
type
Tn
TRAIN = T → (N × Train × Pos)
value
obs_Tn: Train → Tn
monitor_Train: Tn → TF → TRAIN
monitor_Train(tn)(tf) ≡
    λt:T•(let (n,tps) = tf(t) in
          let (trn,pos) = select(tn)(tps(t)) in
          (n,trn,pos) end end)

select: Tn → (Train ₘ→ Pos) → (Train × Pos)
select(tn)(tps) ≡
   let trn:Train•trn ∈ dom tps∧obs_Tn(trn)=tn
   in (trn,tps(trn)) end
```

3

Modelling Rail Nets and Time Tables using OWL

Yuan Fang Li and Dines Bjørner

Contents

3.1 Some Introductory Remarks

This chapter is but a very first draft.

In this chapter we present another model of what has earlier, in Sect. 2.1 been presented as a model of rail nets: Nets, lines, stations, units, connectors, paths, routes, etc.

This other, the present model, is expressed in `OWL`, the Web Ontology Language[1]. `OWL` is a semantic markup language for publishing and sharing ontologies on the World Wide Web.

The aim of the draft work presented here is severalfold:

- To create a semantic web ontology for railways. Initially for nets and time tables. An ontology that could spur railway infrastructure owners and train operators to endow the Internet with near–exhaustive information on their nets and train time tables. This might facilitate the automatic extraction of such information as could help train traffic planning across national borders, across continents, on one hand, and, on the other hand, help passengers plan detailed, complex train travels — supported by more–or–less automated tools.
- This automation is predicated upon the assumption that the language, here `OWL`, is decidable. The “subset” of `RSL` used in Sect. 2.1 is not decidable. Expressing a model of rail nets, lines, stations, units, connectors, paths and routes, in `OWL` should thus, potentially lead to automatic verification of a number of properties of nets and time tables. Such automation could help in the design and further (evolutionary) development of models like those in this section and in Sect. 2.1.

[1] `http://www.w3.org/TR/owl-ref/`

- Further aims are more on the scientific side: To develop semantic models, in one and the same "third" specification language, viz.: Z, of the subsets of `RSL` and of `OWL` used in the mutual descriptions of rail nets, lines, stations, units, connectors, paths and routes — to show that they are equivalent. Also to show that alternative ways of modelling rail nets etc., in either of the two styles, are either equivalent, or lead to weaker, or stronger models.

3.2 On Notation

Table 3.1. Legend

$\sqsubseteq$	Denotes "sub class of" relationship.
$\equiv$	Denotes "equivalent class" relationship.
Time	The data type representing time.
$\sqcup$	Denotes "class union" relationship.
$Pr \geq (\leq, =) n$	Denotes "For a given domain value, the property *Pr* maps it to at least (at most, exactly) *n* range values.

3.3 Nets

A *Net* is a railway net.

$Net : Class$

3.3.1 Line

A *Net* consists of one or more *Line*s.

$Line : Class$

$ConsistsOf : ObjectProperty$
$domain(ConsistsOf\ \ Net)$
$range(ConsistsOf\ \ Line)$
$Net \sqsubseteq ConsistsOf \geq 1$

The last statement above states that *Net* must have at least one *Line*.

3.3.2 Station

*Station*s are linked by *Line*s. A *Line* has exactly 2 *Station*s.

$Station : Class$

$Links : ObjectProperty$
$domain(Links\ Line)$
$range(Links\ Station)$
$Line \sqsubseteq Links = 2$

It is, at the moment of writing this, August 5, 2004, strongly believed that one must also express the property that nets consists of stations.

3.3.3 Unit

Unit is a class of basic railway units that cannot be further divided. It is the super class of 4 sub classes, which are pair-wise disjoint. *Unit* is equivalent as the union of the four disjoint classes.

$Unit : Class$

$LinearUnit : Class$
$Switch : Class$
$SwitchableCrossover : Class$
$Crossover : Class$

$LinearUnit \sqsubseteq Unit$
$Switch \sqsubseteq Unit$
$SwitchableCrossover \sqsubseteq Unit$
$Crossover \sqsubseteq Unit$
$Unit \equiv LinearUnit \sqcup Switch \sqcup$
$\quad SwitchableCrossover \sqcup Crossover$

$disjointWith(LinearUnit\ Switch)$
$disjointWith(LinearUnit\ SwitchableCrossover)$
$disjointWith(LinearUnit\ Crossover)$
$disjointWith(Switch\ SwitchableCrossover)$
$disjointWith(Switch\ Crossover)$
$disjointWith(SwitchableCrossover\ Crossover)$

We need to specify that any *Line* should have at least one *Unit*.

$HasUnit : ObjectProperty$
$domain(HasUnit\ Line)$
$range(HasUnit\ Unit)$
$Line \sqsubseteq HasUnit \geq 1$

It is, at the moment of writing this, August 5, 2004, strongly believed that one must also express the property that stations have units, and that units of lines and stations are disjoint and are part of the net.

3.3.4 Connector

A number of (2, 3 or 4) *Connector*s terminate a railway *Unit*. Any *Connector* can only terminate at most 2 *Unit*s.

$Connector : Class$
$HasConnector : ObjectProperty$

$domain(HasConnector\ Unit)$
$range(HasConnector\ Connector)$
$Unit \sqsubseteq (HasConnector \geq 1) \sqcap (HasConnector \leq 4)$

$Terminates : ObjectProperty$
$domain(Terminates\ Connector)$
$range(Terminates\ Unit)$
$Connector \sqsubseteq Terminates \leq 2$
$inverseOf(HasConnector\ Terminates)$

Every *Unit* has a status associated with it.

3.3.5 Path

A *Path* is a state that a particular railway *Unit* could be in at any given time point. This state can be characterized using the two connectors terminating the unit. The constraint that a path should have distinct connectors cannot be captured by OWL (but maybe SWRL [29])!

$$
\begin{aligned}
&\mathit{Path} : \mathit{Class}\\
&\mathit{HasConnectors}, \mathit{FirstConnector},\\
&\quad \mathit{SecondConnector} : \mathit{ObjectProperty}\\
\\
&\mathit{domain}(\mathit{HasConnectors}\ \mathit{Path})\\
&\mathit{range}(\mathit{HasConnectors}\ \mathit{Connector})\\
&\mathit{Path} \sqsubseteq \mathit{HasConnectors} = 2\\
\\
&\mathit{domain}(\mathit{FirstConnector}\ \mathit{Path})\\
&\mathit{range}(\mathit{FirstConnector}\ \mathit{Connector})\\
&\mathit{domain}(\mathit{SecondConnector}\ \mathit{Path})\\
&\mathit{range}(\mathit{SecondConnector}\ \mathit{Connector})
\end{aligned}
$$

3.3.6 Route

A *Route* is a sequence of *Path*s through *Unit*s such that adjacent *Unit*s are connected. The connectedness of *Path*s cannot be specified by OWL!

$$
\begin{aligned}
&\mathit{Route} : \mathit{Class}\\
\\
&\mathit{PathUnit} : \mathit{Class}\\
&\mathit{RoutePath} : \mathit{ObjectProperty}\\
&\mathit{domain}(\mathit{RoutePath}\ \mathit{Route})\\
&\mathit{range}(\mathit{RoutePath}\ \mathit{PathUnit})\\
&\mathit{Route} \sqsubseteq \mathit{RoutePath} \geq 0\\
\\
&\mathit{RPPath} : \mathit{ObjectProperty}\\
&\mathit{domain}(\mathit{RPPath}\ \mathit{RoutePath})\\
&\mathit{range}(\mathit{RPPath}\ \mathit{Path})\\
&\mathit{RoutePath} \sqsubseteq \mathit{RPPath} = 1\\
\\
&\mathit{RPUnit} : \mathit{ObjectProperty}\\
&\mathit{domain}(\mathit{RPUnit}\ \mathit{RoutePath})\\
&\mathit{range}(\mathit{RPUnit}\ \mathit{Unit})\\
&\mathit{RoutePath} \sqsubseteq \mathit{RPUnit} = 1
\end{aligned}
$$

3.3.7 Train

Each *Train* occupies some *Route*.

$$
\begin{aligned}
&\mathit{Train} : \mathit{Class}\\
&\mathit{Occupy} : \mathit{ObjectProperty}\\
&\mathit{domain}(\mathit{Occupy}\ \mathit{Train})\\
&\mathit{range}(\mathit{Occupy}\ \mathit{Route})\\
&\mathit{Train} \sqsubseteq \mathit{Occupy} = 1
\end{aligned}
$$

3.4 Time Table

A *Timetable* contains a number of records of train names and train journeys. Each train journey consists of an *ArrivalTime*, a *DepartureTime* and a *Station*.

3.4.1 An RSL Model

type
  TT = Tn −m−Z TJ
  $TJ' = SV^*$
  TJ = {| tj:TJ′ • **len** tj $\geq$ 2 $\wedge$ no_repeat_visits(tj) |}
  SV = T × S × T
value
  no_repeat_visits: TJ′ → **Bool**

3.4.2 An OWL Model

$Timetable : Class$
$HasEntry : ObjectProperty$
$domain(HasEntry\ Timetable)$
$range(HasEntry\ TrainEntry)$
$Timetable \sqsubseteq HasEntry \geq 1$

$TrainEntry : Class$
$HasTrain : ObjectProperty$
$domain(HasTrain\ TrainEntry)$
$range(HasTrain\ Train)$
$TrainEntry \sqsubseteq HasTrain = 1$

$HasJourney : ObjectProperty$
$domain(HasJourney\ TrainEntry)$
$range(HasJourney\ TrainJourney)$
$TrainEntry \sqsubseteq HasJourney = 1$

$TrainJourney : Class$
$HasSJourney : ObjectProperty$
$domain(HasSJourney\ TrainJourney)$
$range(HasSJourney\ SingleJourney)$
$TrainJourney \sqsubseteq HasSJourney \geq 1$

$SingleJourney : Class$

$HasATime : DatatypeProperty$
$domain(HasATime\ SingleJourney)$
$range(HasATime\ Time)$
$SingleJourney \sqsubseteq HasATime = 1$

$HasDTime : DatatypeProperty$
$domain(HasDTime\ SingleJourney)$
$range(HasDTime\ Time)$
$SingleJourney \sqsubseteq HasDTime = 1$

$HasStation : ObjectProperty$
$domain(HasStation\ SingleJourney)$
$range(HasStation\ Station)$
$SingleJourney \sqsubseteq HasStation = 1$

3.5 Some Remarks

We pointed out, in two places, that the above OWL model need be carefully validated. We are sure much more work has to be done !

Part III

Allocation & Scheduling

4

Rolling Stock Maintenance

Martin Pěnička, Albena Strupchanska and Dines Bjørner

Contents

4.1 INTRODUCTION

Railway planners handle time–consuming tasks of railway operations. There are a number of tasks which can be solved by computers using operation research algorithms. These tasks are mainly being solved separately without any relations or integrations between them. We would like to focus not only at their integration, but we would like to find some common parts of these tasks already during the software development stages.

This is the main reason for presenting, in this paper, a formal methods approach to one of the railway optimisation problems — train maintenance routing. Formal methods approaches to crew rostering is presented in Chapter 5, and to optimal train length/train composition and decomposition respectively in a forthcoming report.

In future, this approach should lead to much deeper, better and easier integration of railway applications in and among all tasks of the railway monitoring, control and planning processes.

4.1.1 Synopsis

Each railway company operating trains deals with the problem of maintenance of their rolling stock. By a circulation plan we shall understand a schedule of sequences of station visits. Each train is composed of carriages, which, according to a circulation plan, can be grouped into in–divisible parts — called assemblies. In other words, an assembly is one ore more carriages that have the same circulation plan. We do not discuss how to device circulation plans for assemblies in this paper. This is a task of rolling stock rostering and of optimal train length determination.

In this paper we define notions of rolling stock rosters and of maintenance events, and we show how maintenance events can be added into rolling stock rosters. By maintenance we do not mean only regular check of all systems (assemblies, etc.) in the depot, but we present maintenance in a more general sense. We understand maintenance as all activities, which must be done with rolling stock, regularly according to some rules, and which should be planned in advance except for the operation of rolling stock itself. That is the reason, why we also include outside and inside cleaning of carriages, refuelling diesel engines, refilling supplies into restaurant carriages, water and oil refilling, etc.

Each carriage, according to its type, has associated certain types of maintenance tasks. Each task has a defined frequency of necessary handling of this task upon the carriage. The frequency can be expressed by the elapsed number of kilometer or operating house since a previous maintenance, ie., are intervals, for which maintenance need be done.

Basically there are two different ways on how to add maintenance to the rolling stock plans.

Rostering with Maintenance is the first possible way. Maintenance is planned already in the rolling stock roster planning process (maintenance is seen just as a one of the tasks in the roster). All maintenance actions for all rolling stock are planned in advance. This approach seems to be appropriate for long–distance trains and also for maintenance types that have to be done quite often (eg., inside cleaning, diesel fuelling, etc.).

Maintenance Routing is the second possible way. In this case, the rolling stock roster has several maintenance opportunities only. That means, that not all carriages have maintenance in their plans. In this approach it is necessary to have on–line statistics about actually elapsed kilometers and operating hours for all assemblies. Later, during train operation, maintenance checks are planned for those assemblies which are close to reach a given kilometer or time limit. It is done by modifying previous plans in such a way, that all assemblies are routed to maintenance stations. These modifications are called night and day exchanges or empty rides between stations. Maintenance routing better fits short-distance trains, typically trains “around” big cities, and also for those types of maintenance, where irregularities can be expected. For example, a broken engine must be routed to the maintenance station immediately, with no care about kilometer distance for which it was slated at the time of the breakdown event.

In this paper we deal only with the second approach — maintenance routing. This means that our input is a rolling stock roster with several maintenance opportunities (not assigned to concrete assemblies yet). For all assemblies in the network we can find their position in the network according to the schedule at a given time, number of kilometers and hours elapsed from the most recent maintenance checks. Certain events (like breakdowns) must be recorded and taken into account as well.

Output from the maintenance routing planning is a list of changes in rolling stock roster for the next few days. Once a day, changes and recorded events are applied to the current rolling stock roster plan and are used as input for the following day’s maintenance routing planning.

4.1.2 The Major Functions

Given a railway net, N, a traffic schedule TS, and a planning period (from day-time, DT, to day-time, DT) the job is to formally characterize and generate all the possible sets of changes, CS, necessary and sufficient to secure finely maintenance. What we understand by terms net, traffic schedule and sets of changes can be found further on in the paper.

type
  N, TS, DT, CS
value
  gen_Changes: $\mathsf{N} \times \mathsf{TS} \times (\mathsf{DT} \times \mathsf{DT}) \stackrel{\sim}{\to}$ CS-**set**

Given these possible change sets, one is selected and applied to the traffic schedule to generate a new traffic schedule for a given period.

value
  ApplyChanges: TS × CS × (DT × DT) → TS

4.1.3 Requirements and Software Design

We emphasis that we formally characterize schedules, assembly plans and maintenance changes — such as they are "out there", in realty, not necessarily as we wish them to be. On the basis of such formal domain specifications we can then express software requirements, ie., such as we wish schedules, assembly plans and maintenance changes to be.

The actual software design relies on the identification of suitable operation research techniques (ie., algorithms), that can provide reasonably optimal solutions and at reasonable computing times.

It is not the aim of this paper to show such operations research algorithms. Instead we refer to [33, 34, 38].

4.1.4 Chapter Structure

The chapter is divided into two main parts. In the first part (Section 2) we give a full description of those railway domain terms that are relevant to the problem at hand. We start with a description of railway nets, lines and stations. Then it is explained what we mean by trains and assemblies. Further we explain the concept of traffic schedules and describe some functions on such schedules. The last part of Section 2 presents a detailed description of maintenance.

The second main part is Section 3: Planning. In it we explain the necessary and sufficient changes to rolling stock rosters, introducing the concepts of day and night changes and of empty rides. We explain their generation, as well as the application of these changes to the new traffic schedule.

4.2 FORMAL MODEL

In this section we introduce the actual domain phenomena and concepts of railway nets, trains, schedules, rolling stock rosters and maintenance, and we build a domain model in a 'formal methods' approach — step–by–step.

First we define the concept of railway nets. We describe railway nets as a set of lines and a set of stations and all properties, which belongs to these concepts.

4.2.1 Nets, Lines, Stations

In the first part basic concepts of railway net, lines and stations are described. We present it in natural English description as well as in RSL.

Narrative

Each net (`N`) is composed from two main parts: stations (`S`) and lines (`L`). Stations and lines can be observed from the net. Axioms (ie., constraints) are:

- There are at least two stations and one line in a net (α_1).
- Each line connects exactly two distinct stations (α_2).
- Each station is connected at least to one line (α_3).
- Each line has no zero length (α_4).

Formal Model

```
scheme NETWORK_S =
  class
    type N, L, S, KM

    value
      zero_km : KM,
      > : KM × KM → Bool,
      obs_S : N → S-set,
      obs_L : N → L-set,
      obs_S : L → S-set,
      obs_Length : L → KM

    axiom
      (α1) ∀ n : N •
            card obs_S(n) ≥ 2 ∧
            card obs_L(n) ≥ 1,
      (α2) ∀ n : N, ℓ: L •
            ℓ∈ obs_L(n) ⇒
              card obs_S(ℓ) = 2,
      (α3) ∀ n : N, s : S • s ∈ obs_S(n) ⇒
            ∃ ℓ: L ⇒ s ∈ obs_S(ℓ)
      (α4) ∀ n : N, ℓ:Lin •
            ℓ∈ obs_L(n) ⇒
              obs_Length(ℓ) > zero_km
  end
```

4.2.2 Time & Date

Narrative

In this part, basic functions about date (`D`), time (`T`) and time intervals (`TI`) are presented. Since it is not the main subject of this paper, no detailed description is given.

Formal Model

```
scheme TIME_S =
  class
    type T, TI

    value
      + : T × TI → T,
      + : TI × T → T,
      + : TI × TI → TI,
      − : T × TI → T,
      − : TI × TI → TI,
      > : T × T → Bool,
      > : TI × TI → Bool,
      < : T × T → Bool,
      < : TI × TI → Bool
  end

scheme DATE_S =
  class
    type
      D,
      Day,
      Month,
      Year,
      WD == mo | tu | we | th | fr | sa | su

    value
      obs_Day : D → Day,
      obs_Month : D → Month,
      obs_Year : D → Year,
      obs_WeekDay : D → WD
  end

scheme TIME_DATE_S =
  extend TIME_S, DATE_S with
  class
    type DT = D × T

    value
      + : DT × TI → DT,
      + : TI × DT → DT,
      − : DT × TI → DT,
      − : TI × TI → TI,
      > : DT × DT → Bool,
      < : DT × DT → Bool,
  end
```

4.2.3 Trains and Assemblies

Narrative

There are trains (`TR`) travelling in the network from station to station. Each train has a train number (`TRNo`) and a train name (`TRNa`). Each train is composed of an ordered list of assemblies

(A). In a real world assembly can be composed of cars, but in this task, the assembly is always the smallest part of the train and can never be divided into pieces.

Each assembly has its unique identification number (AID). There is a kilometer distance, which each assembly has run in total at certain time. There are several different types of assemblies (AT) operated by railway companies. These type could be passenger or cargo car, diesel or electric engine, double decker or sprinter unit, etc.

Since each train is a ordered list of assemblies, we can easily find out the position (POS) of an assembly in a train. In our case, we just distinguish, if an assembly is first, last or properly internal. If the train is composed of one assembly, then it is called solo.

There are some axioms: Each train is composed of least one assembly (α_5). Each assembly has its unique identification number (α_6).

Formal Model

```
scheme TRAIN_S =
  extend TIME_DATE_S with
  class
    type
      TR,
      A,
      TRNo,
      TRNa,
      AID,
      AT == el_loko | di_loko | cargo_car
      POS == fir | mid | las | sol | non

    value
      obs_TrnNa : TR → TRNa,
      obs_TrnNo : TR → TRNo,
      obs_Asml : TR → A*,
      obs_AId : A → AID,
      obs_AType : A → AT,
      obs_Km : A × DT → KM,
      Position : TR × A → POS
      Position(trn, a) ≡
        case obs_Asml(trn) of
          ⟨a⟩ → sol
          ⟨a⟩ ⌢ asl → fir
          asl ⌢ ⟨a⟩ → las
          asl ⌢ ⟨a⟩ ⌢ asm′ → mid
          _ → non
        end

    axiom
      (α5) ∀ trn : TR •
          len obs_Asml(trn) ≥ 1
      (α6) ∀ a : A • ∼∃ a′ •
          a ≠ a′ ∧
          obs_AId(a) = obs_AId(a′)
  end
```

4.2.4 Traffic Schedule

Traffic schedules together with network topologies and train descriptions are the main inputs into our application.

Narrative

Each railway company which operates trains needs to deal with schedules (SCH) from which traffic schedule (TS) can be extracted. Traffic schedules assign journies (J) to each train number and date . A journey is a sequence of rides (R). A ride is composed of departure time and station, arrival time and station, and the train, that serves the ride. Sequence of rides served always by the same assembly is called an assembly roster or an assembly plan (AP).

The function (APlan) extracts the assembly plan for a given assembly identification from given traffic schedule and in a given time interval. Function (AIDL) returns a list of assembly identifications from a given ride. Function (ActAsms) extracts the set of assemblies which are active according to a given traffic schedule in a given time interval.

There are other axioms. Each traffic schedule has at least one journey (α_7). In each ride, the arrival station is different from the departure station and the arrival time is “later” than the departure time (α_8). The train number is the same for all rides of a journey. The arrival station of any ride in a journey is equal to the departure station of the next ride in that journey (α_9).

Formal Model

```
scheme SCHEDULE_S =
  extend TRAIN_S with
  class
    type
      SCH,
      TS = TRNo →m (DT →m J),
      J = R*,
      R = (DT × S) × (S × DT) × TR,
      AP = R*

    value
      obs_TraSCH : SCH → TS,

      APlan :
        TS × AID × (DT × DT) → AP
      APlan(ts, aid, (t, t′)) as rl post
        ∀ i : Nat • {i, i+1} ⊆ indx rl ⇒
          aid ∈ elems AIDL(rl(i)) ∧
          aid ∈ elems AIDL(rl(i+1)) ∧
          {rl(i), rl(i+1)} ⊆ JSet(ts) ∧
          DepS(rl(i+1)) = ArrS(rl(i)) ∧
          t ≤ DeptT(rl(i)) <
            DepT(rl(i+1)) ≤ t′,

      AIDL : R → AID*
      AIDL(r) ≡
      let(_, _, _, _, trn) = r,
        a = obs_Asml(trn) in
      ⟨aid | aid in
      ⟨obs_AId(hd a)..obs_AId(a(len a))⟩⟩
      end

      ActAsms : TS × (DT × DT) → A-set
      ActAsms(ts, (t, t′)) ≡
        {a | a : A •
          len APlan(ts, a, (t, t′)) > 0},

      JSet : TS → J-set,
      JSet(ts) ≡
        ∪ {rng tn | tn : (DT →m J) •
          tn ∈ rng ts}

      DepS : R → S
      DepS(r) ≡
        let (_, s, _, _, _) = s in s end,

      DepT : R → DT
      DepT(r) ≡
        let (t, _, _, _, _) = r in t end,

      ArrS : R → S
      ArrS(r) ≡
        let (_, _, s, _, _) = r in s end,

      ArrT : R → DT
      ArrT(r) ≡
        let (_, _, _, t, _) = r in t end,

    axiom
      (α7) ∀ ts : TS • card JSet(ts) ≥ 1,
      (α8) ∀ r : Ride •
        let ((dt, ds), (ast, ast), _) = r
        in ds ≠ ad ∧ dst < ast end,
      (α9) ∀ j : J, i : Nat •
        {i, i+1} ∈ inds j ⇒
        let (t1, _, s, t2, trn) = j(i),
            (t1′, s′, _, t2′, trn′) = j(i+1)
        in
        obs_TrnNa(trn)=obs_TrnNa(trn′)∧
        obs_TrnNo(trn)=obs_TrnNo(trn′) ∧
            t1<t2≤t1′<t2′ ∧ s = s′
        end
  end
```

4.2.5 Maintenance

Narrative

We extend the general model of railway network. First we define different types of maintenance (MT). Some possible maintenance types can be:

Regular operation check: Each engine and carriage — according to given rules and safety regulations — must be checked regularly. There is a limited number of stations where this maintenance can be made (usually just one for each train type).

Inside cleaning is the most common maintenance operation for passenger carriages. It can be done at nearly every station, without additional shunting demands and costs.

Outside cleaning is also common maintenance, but usually not all stations in the network have required equipment.

Diesel engine refuel and *Water/sand/oil refill* are other examples of maintenance types.

We next define maintenance plans. They, (`MNTPLAN,`) are lists of actions (`ACTION`), which are temporally ordered. These action could be: 'Working Ride' (`WR`), 'Empty Ride' (`ER`), and 'Maintenance Check' (`MNT`).

Each assembly type has defined certain maintenance types that have to be done (`REQMNT`). There are also given upper limits (`MNTLIM`) for each assembly and maintenance type. These limits are given either in or kilometer or in time intervals. According to the position of a given assembly (in a train) and of its assembly type, we can find out how difficult it is to exchange the assembly in the train with another assembly of the same type (`EXDIF`). Each station in the set has defined costs (`COST`) and required time for each maintenance type (`MNTDUR`).

For each assembly and maintenance type one can observe where and when that assembly was last maintained according to that type. Different topologies and shunting possibilities of each station allow or does not allow exchange of two assemblies within certain time limits. This time need not be the same for nights and for days.

The functions below are explained, ie., narrated, after their definition.

Formal Model

```
extend SCHEDULE_S with
class
  type
    IMP,
    DIF,
    COST,
    MT ==
      regular_check | out_clean |
      in_clean | diesel_fuel,

    ACTION == WR | ER | MNT,

    WR = R,
    ER = R,
    MNT = DT × S × DT × MT,

    MNTPLAN = ACTION*,
    REQMNT = AT →m MT-set,
    MNTLIM =
      (AT × MT) →m (TI | KMS)
    MNTIMP = (AT × MT) →m IMP,
    MNTDUR = AT →m (MT →m (S →m TI)),
    EXDIF = (AT × POS) →m DIF

  value
    max_imp : IMP,
    req_mnt : REQMNT,
    mnt_lim : MNTLIM,
    mnt_imp : MNTIMP,
    mnt_dur : MNTDUR,
    exc_dif : EXDIF,

    obs_LastMnt: A × MT → (TI|KMS),
    obs_MinExDTime: S × DT→ TI,
    obs_MinExNTime: S × DT → TI,
    obs_ExDCost: S × DT ~→ COST,
    obs_ExNCost: S × DT ~→ COST,
    obs_MntCost: N×AT×MT×S×DT~→COST,

    ≤ : IMP × IMP → Bool,
    ≤ : DIF × DIF → Bool,

    RemDist : A × MT × DT → KMS
    RemDist(a, mt, t) ≡
      obs_LastMnt(a, mt) +
      mnt_lim(obs_AType(a), mt) −
      obs_Km(a, t),

    RemTime : A × MT × DT → TI
    RemTime(a, mt, t) ≡
      obs_LastMnt(a, mt) +
      mnt_lim(obs_AType(a), mt) − t,

    MStas : N × AT × MT → Sta-set
    MStas(n, at, mt) ≡
      {s | s : Sta •
          s ∈ obs_Sta(n) ∧
          s ∈ dom mnt_dur(at)(mt)},

    MTypes : N × S × AT → MT-set
    MTypes(n, s, at) ≡
      {mt | mt : MType •
          mt ∈ dom mnt_dur(at) ∧
          s ∈ dom mnt_dur(at)(mt) ∧
          s ∈ MStas(n, at, mt)},

    isMPos : N × AP × AT × MT → Bool
    isMPos(n, p, at, mt) ≡
      ∃ s : S, i : Nat •
         s ∈ MSta(n, at, mt) ∧
         {i, i+1} ⊆ inds p ∧
         s = ArrS(p(i)) ∧
```

```
        ArrT(p(i)) +
          mnt_dur(at)(mt)(s) <
          DepT(p(i+1)),

  isInSta: AP × S × DT → Bool
  isInSta(p, s, t) ≡
    ∃ i:Nat •
      {i, i+1} ⊆ inds p ⇒
      ArrT(p(i)) ≤ t ≤
      DepT(p(i+1)) ∧
      s = ArrS(p(i)) = DepS(p(i+1)),

  DisToMS:
    N × AP × AT × MT ⥲ (TI|KMS)
  DisToMS(n, p, at, mt) ≡
    if DepS(hd p) ∈ MStas(n, at, mt)
    then 0
    else
      RideDis(hd p) +
      DisToMS(n, tl p, at, mt)
    end,

  MntUrg : A × MT × DT → IMP
  MntUrg(a, mt, t) ≡
    if RemDist(a, mt, t) ≤ 0
    then max_imp
    else
      mnt_imp(obs_AType(a), mt) /
      RemDist(a, mt, t)
    end

  TMax : TS → DT
  TMax(ts) as tmax post
    ∀ j : J, i : Nat •
      j ∈ JSet(ts) ∧ i ∈ inds j ⇒
        ArrT(j(i)) < tmax

axiom
  (α10) ∀ i : IMP • i ≤ max_imp,
  (α11) ∀ n : N, at:AT •
          ∃ s : S, mt : MT ⇒
          s ∈ obs_S(n) ∧
          mt ∈ MTypes(n, s, at) ∧
          s ∈ MStas(n, at, mt),
  (α12) ∀ at : AT, mt : MT •
          mt ∈ req_mnt(at) ⇒
          0 < mnt_imp(at, mt) ≤
            max_imp,
  (α13) ∀ at : AT, mt : MT •
          mt ∉ req_mnt(at) ⇒
            mnt_imp(at, mt) = 0,
  (α14) ∀ at : AT •
          exc_dif(at, sol) ≤
          exc_dif(at, las) ≤
          exc_dif(at, fir) ≤
          exc_dif(at, mid),
end
```

We now explain the above planning functions.

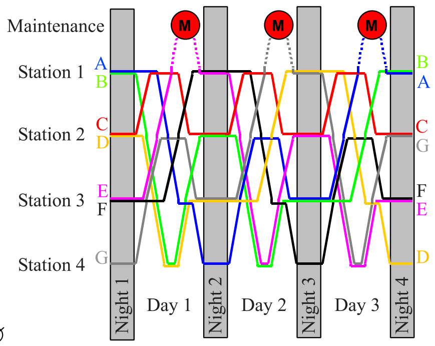
(RemDist) and (RemTime) calculate remaining distance to the maintenance of a given type either in kilometers or in time interval, for a given assembly at a given time. (MStas) yields the set of stations that are maintenance stations for a given assembly and maintenance type, and in a given network. (MTypes) yields all maintenance types, which a given assembly can undergo at a given station. (isMPos) checks if there is a maintenance opportunity in a given plan, for given assembly and maintenance types. (isInSta) checks if, according to a given plan, a given assembly is at a given station and at a given time. (DisToMS) espresses the "distance" to a maintenance opportunity, in a given plan for a given assembly and maintenance type. (MntUrg) expresses the importance of undergoing maintenance of a given type at a given time for a given assembly. (TMax) expresses the total time horizon of a given traffic schedule.

4.3 PLANNING

Every day, last-moment changes and updates must be applied to the previously planned rolling stock roster. In this section we describe two basic functions for rolling stock maintenance routing: Generation of changes to a rolling stock roster and application of changes to a roster.

4.3.1 Generation of Rolling Stock Roster Changes

Given a rolling stock roster and a net we can express sets of necessary rolling stock roster changes. An example of rolling stock roster for several consecutive days is shown in figure 1.

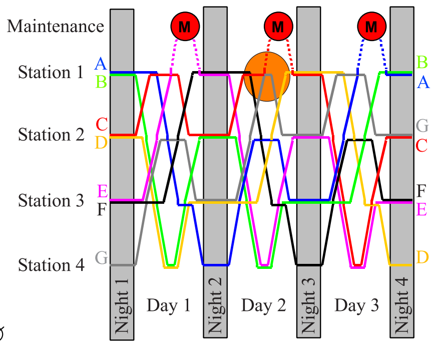


∅ Fig. 1: The Original Plan

Each change set is composed from three different types of changes. They are called: Day Change, Night Change and Empty Ride.

Day Change is composed of two assemblies, of a station and a time, where and when the interchange between these two assemblies takes place. Day change may occur when the first assembly is an ‘urgent’ assembly (needs to undergo maintenance check in couple of days) and the second assembly has a maintenance station in the plan, and when both assemblies are in the same station at the same time, during a day time, and there is enough time to interchange them.

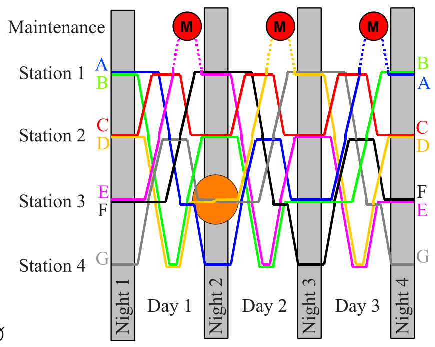
In Figure 2 the assembly ”C” is exchanged at station 1 with the assembly ”G” — designated, thus, to reach the maintenance station in two days.



∅ Fig. 2: Example of ‘Day Change’

Night Change is quite similar to the day change. The main difference is in the time when this change is applied. Night changes may occur when the first assembly is an ‘urgent’ assembly and the second assembly has a maintenance station in the plan, and when both assemblies are in the same depot or station during the same night. It is the least expensive way in which to add maintenance into the assembly plan.

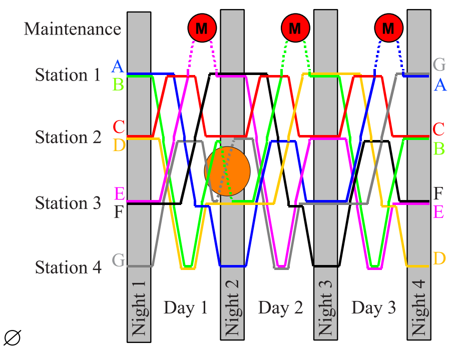
In Figure 3 the assembly ”D” is exchanged at station 3 with the assembly ”G” — designated, thus, to reach the maintenance station the second night.



∅ Fig. 3. Example of ‘Night Change’

Empty Ride is the last possible change that can be applied to the rolling stock roster. This change must be applied, when there is no day or night change possible (ie., when there is no such situation in the rolling stock roster where two assemblies are in the same station and time during their operations). In that case, two additional rides have to be added into the plan. An 'Empty Ride' is composed of two assemblies and two additional rides for these assemblies. In general, an 'Empty Ride' is always possible, but it has the highest cost.

In Figure 4 the assembly "B" is routed from station 2 to station 3 and the assembly "G" in opposite direction.



∅ Fig. 4. Example of Empty Ride

Formal model

```
scheme CHANGES_S =
    extend MAINTENANCE_S
    class
        type
            C == DC | NC | ER
            DC == mkDC(a1:A,a2:A,t:DT,s:S)
            NC == mkNC(a1:A,a2:A,t:DT,s:S)
            ER == mkER(a1:A,a2:A,r1:R,r2:R)
            CS = C-set

        value
            NPlan:
                TS × C × (DT × DT) → A
            NPlan(ts,c,(t,t′)) ≡
                APlan(ts,UAID(c),(t,CT(c)))
                ^APlan(ts,SAID(c),(CT(c),t′))

            UAID: C → AID
            UAID(c) ≡
            case c of
                mkDC(a,_,_,_) → obs_AsmID(a),
                mkNC(a,_,_,_) → obs_AsmID(a),
                mkER(a,_,_,_) → obs_AsmID(a)
            end

            SAID: C → AID
            SAID(c) ≡
            case c of
                mkDC(_,a,_,_) → obs_AsmID(a),
                mkNC(_,a,_ _) → obs_AsmID(a),
                mkER(_,a,_,_) → obs_AsmID(a)
            end

            CT: C → DT
            CT(c) ≡
            case c of
                mkDC(_,_,t,_) → t,
                mkNC(_,_,t,_) → t,
                mkER(_,_,r,_) →
                    let ((_,_),(_,t),_)=r
                    in t end
            end

            CS: C → S
            CS(c) ≡
            case c of
                mkDC(_,_,_,s) → s,
                mkNC(_,_,_,s) → s,
                mkER(_,_,r,_) →
                    let ((_,s),(_,_),_)=r
                    in s end
            end

            MinExT: C → TI
            MinExT(c) ≡
            case c of
                mkDC(_ _,t,s) →
                    obs_MinExDTime(s,t)
                mkNC(_,_,t,s) →
                    obs_MinExNTime(s,t)
                mkER(_,_,r,_) →
                    let ((_,s),(_,t),_) = r
    in obs_MinExDTime(s,t)          end
    end                          end
```

4.3.2 Generating changes

We now present the main function of this paper: (`gen_Changes`). This function expresses all possible sets of changes from a given net, traffic schedule and planning period. These change sets are limited by several constraints (ie., post conditions). All changes which are in the generated set must be possible, necessary and sufficient.

The 'possible' condition checks whether two assemblies are of the same type and whether these two assemblies are active assemblies in a given time period. Then it is checked if it is a case that both assemblies are in the same station at the same time for a long enough period according to their plans.

The change is 'necessary' when remaining distance to becoming an 'urgent' assembly is smaller than the distance to the maintenance station in the plan of this assembly.

The 'sufficiency' condition checks if an 'urgent assembly' can arrive at its maintenance station before exceeding its (time or kilometer) limit according to the new plan.

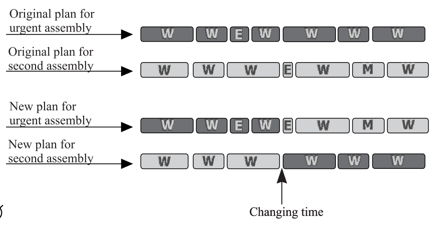
```
scheme PLANNING_S =
  extend CHANGES_S
  class
    type
    value
      gen_Chgs: N×TS×(DT×DT)→̃CS-set
      gen_Chgs(n, ts, (t, t′)) as css
      pre
        ∃ a : A, mt : MT •
         DisToMS(APlan(ts, obs_AID(a),
          (t, t′)), obs_AType(a), mt)
           > RemDis(a, mt, t)
      post
        ∀ cs : CS, c : C • c
        ∈ cs ∧ cs ∈ css
        ⇒ isPossible(ts, c, (t, t′)) ∧
          isNecessary(ts, c, (t, t′)) ∧
          isSufficient(ts, c, (t, t′))

      isPossible:
        TS × C × (DT×DT) → Bool
      isPossible(ts, c, (t, t′)) ≡
        obs_AType(UA(c)) =
          obs_AType(SA(c)) ∧
        {UA(c), SA(c)} ⊆
          ActAsms(ts, (t, t′)) ∧
        MntUrg(UA(c), mt, t′) >
          MntUrg(SA(c), mt, t′) ∧
        isInSta(APlan(ts, UAID(c),
          (t, t′)), CS(c), CT(c)) ∧
        isInSta(APlan(ts, UAID(c),
          (t, t′)), CS(c), CT(c) +
          MinExTU(c)) ∧
        isInSta(APlan(ts, SAID(c),
          (t, t′)), CS(c), CT(c)) ∧
        isInSta(APlan(ts, SAID(c),
          (t, t′)), CS(c), CT(c) +
          MinExTU(c)),


      isNecessary:
        TS × C × (DT×DT) → Bool
      isNecessary(ts, c, (t, t′)) ≡
        ∃ mt: M •
          DisToMS(APlan(ts,
          UAID(c), (t, t′)),
          obs_AType(UA(a)), mt) >
          RemaDist(a, mt, t),


      isSufficient:
        TS × C × (DT×DT) → Bool
      isSufficient(ts, c, (t, t′)) ≡
        ∀ mt: MT •
        mt ∈ req_mnt(obs_AType(a))
        ⇒
        DisToMS(NPlan(ts, c, (t, t′)),
        obs_AsmType(UA(c)), mt) ≤
        RemaDist(UA(c), mt, t)
  end
```

Applying changes: Once a day, specified changes are applied to the rolling stock roster traffic schedule. We get a new traffic schedule, which is used as an input to next day's operations. There can be only one difference between the old and the new traffic schedule: some trains in the new schedule can be served by different assemblies of the same type. That means, that assembly plans are modified as shown on Figure 5.


Original plan for urgent assembly
W W E W W W W
Original plan for second assembly
W W W E W M W
New plan for urgent assembly
W W E W E W M W
New plan for second assembly
W W W W W W
Changing time

∅ Fig. 5: Plan modification

The correct solution is when all assemblies which require maintenance in the given period, after application of calculated changes in the new traffic schedule can reach the maintenance station in the remaining distance.

```
scheme UPDATE_S =
  extend PLANNING_S
  class
    value
      AppChgs: TS×CS×(DT×DT)→TS
      AppChgs(ts, cs, (t, t′)) as ts′
      post
        ∀ c: C • c ∈ cs ⇒
        let aid1 = UAID(c),
              aid2 = SAID(c)
        in
          APlan(ts, aid1, (t, CT(c))) ⌢
          APlan(ts, aid2, (CT(c), t′)) =
            APlan(ts′, aid1, (t, t′))
          ∧
          APlan(ts, aid2, (t, CT(c))) ⌢
          APlan(ts, aid1, (CT(c), t′)) =
            APlan(ts′, aid2, (t, t′))
        end
        ∧
        ∀ a: A, mt: MT, cs: CS •
          a ∈ ActAsms(ts, (t, t′)) ∧
          mt ∈ req_mnt(obs_AType(a)) ∧
          MntUrgency(a, mt, t) > 0 ⇒
            ∃ c: C •
            c ∈ cs ∧ a = UA(c) ∧
            DisToMS(APlan(ts′,
              obs_AId(a), (t, t′)),
              obs_AType(a),mt) ≤
                RemDis(a, mt, t)
  end
```

4.4 SUMMARY

A formal model of maintenance routing have been shown. The task was divided into two basic steps:

- generation of possible, necessary and sufficient changes of the traffic schedule
- application of these changes to the traffic schedule

We emphasize that we formally characterized schedules, assembly plans and changes in the plan to meet maintenance demands. On the basis of such formal space software we can now prescribe requirements.

In the future, this formal approach, we claim, should lead to deeper, better and easier integration of all railway optimization applications in all the tasks of railway planning, monitoring and control processes.

5

Rostering

Albena Strupchanska, Martin Pěnička and Dines Bjørner

Contents

5.1 INTRODUCTION

5.1.1 Synopsis

Staff planning is a typical problem arising in the management of large transport companies, including railway companies. It is concerned with building the work schedules (duties and rosters) for staff members needed to cover a planned timetable. Each work schedule is built concerning a given staff type (engine men, conductors, cater staff, etc.).

There are two types of staff planning: long-term planning and short-term planning. We will be interested in long term planning. Normally the long term planning task is separated into two stages: staff scheduling and staff rostering. Staff scheduling is concerned with building short-term working schedules, called duties, for staff members such that they satisfy schedule demands. After this stage it is easy to determine the global number of staff members needed to hire such that the working schedules could be performed. Staff rostering is concerned with ordering of duties into long-term working schedules, called base rosters, and assignment of specific staff members to them such that each staff member performs a roster. During the stage of rostering we have the assumption that we have enough hired staff members such that we could assign rosters to them.

In this paper we will try to explain and analyze first informally and then formally the problem. Using a formal methods approach and RAISE Specification Language we will present a formal model of the domain of staff rostering.

5.1.2 The Major Functions

Given a schedule, a staff type, a depot and rules the task is to produce a set of rosters. What we understand in terms of schedule, depot and rules can be found further in the paper.

gen_sross: SCH × StfTp × Dep × eRS → Ros

The function above produces all the rosters for a staff type per depot. Usually rosters are generated per depot and we have the assumption that after the staff scheduling stage all duties generated per depot are shifted to the depot. If this is not the case we propose a function that integrates the two stages in staff rostering into one. So given a schedule, a staff type, set of depots and rules we produce all rosters per each depot for this staff type.

obtain_ross: SCH × StfTp × Dep-**set** × eRS →
  Ros-**set**

5.1.3 Requirements and Software Design

We emphasis that we formally characterized schedules, duties and rosters to meet staff rostering demands. On the basic of such formal characterization we can now express software requirements.

The actual software design relies on identification of suitable operation research techniques, that can provide reasonable optimal solution at reasonable computing times.

It is not the aim of this paper to show such operation research algorithms. Instead we formalize the domain of railway staff rostering such that later we could apply to it operation research techniques discovered in further research work done within AMORE group.

5.1.4 Paper Structure

The paper consists of five sections. Each section consist of formal description of the problem (narrative) and formalization of it (formal model). The first section introduces the topology of the railway net from staff management perspective. The second one introduces the notion of a staff member and related to it characteristics taken into account in the early stage of planning. And finally the last three sections are the ones which gradually show the creation of rosters from a schedule, a set of depots and rules. The first of them is concerned of separating the journeys observed from a schedule into trips. The notions of journey and trip are introduced there. The second one introduces the notion of a duty and produces the set of duties per each depot. Finally the third one introduces more characteristics of staff members and the notion of rosters. It generates the rosters for staff members too.

5.2 NETS, STATIONS AND DEPOTS

In this section we will introduce the notions of nets, stations and depots which are related to the topology of the railway net from a staff manager point of view.

5.2.1 Narrative

We take as base concept for the railway net the topology of that net. From a railway net (Net) we can observe stations (Sta) and depots (Dep). Depots are personnel bases i.e places where staff members are located. The notion of staff member will be introduced in more details in the next section. From a station we can observe a set of depots to which the station can belong. From a depot we can observe a set of stations from which it is easy to reach the depot. Given a depot and a station we can observe the distance in time (TInt) between them. We will be interested in these stations and depots which are 'close' to each other.

There are at least two stations in a net (α_1). There is at least one depot in a net (α_2). The set of depots observed from a station consists of depots of the same railway net (α_3). The set of stations observed from a depot consists of stations of the same railway net (α_4).

5.2.2 Formal Model

We first state some abstract types, ie. sorts, and some observer functions.

```
scheme NETWORK =
 class
  type Net, Sta, Dep, TInt, StaNm, DepNm
   value
    obs_Stas : Net → Sta-set,
    obs_StaNm : Sta → StaNm,
    obs_Deps : Net → Dep-set,
    obs_DepNm : Dep → DepNm,
    obs_StaDeps: Sta → Dep-set,
    obs_DepStas: Dep → Sta-set,
    obs_StDepDistance : Sta × Dep → TInt
 end
```

We will then illustrate some axioms:

```
(α1) axiom ∀ n : Net • card obs_Stas(n) ≥ 2

(α2) axiom ∀ n : Net • card obs_Deps(n) ≥ 1

(α3) axiom ∀ n: Net • ∀ s : Sta •
       s ∈ obs_Stas(n) ⇒ (∀ d : Dep •
        d ∈ obs_StaDeps(s) ⇒ d ∈ obs_Deps(n))

(α4) axiom ∀ n: Net • ∀ d : Dep •
       d ∈ obs_Deps(n) ⇒ (∀ s : Sta •
        s ∈ obs_DepStas(d) ⇒ s ∈ obs_Stas(n))
```

5.3 STAFF MEMBERS

We introduce the notions of staff members and related to them attributes according to a staff manager stake-holder's perspective.

5.3.1 Narrative

We call staff all those people who are employed in a railway company and who could perform some actions in order to fulfill a schedule demands.

At the first stage of staff rostering - staff scheduling we will be interested in a part of the characteristics that can be related to staff members. Staff members are exchangeable at staff scheduling stage that is why we will call them anonymous staff members (AnonStfMbr). From anonymous staff member we could observe his/her home depot (obs_SMDep). A home depot of some staff member is the depot of the railway net from where he/she starts and finishes his/her sequence of actions. There is a notion of a staff type (StfTp). Some possible staff types are: engine men (engS), conductors (condS), cater staff (catS) etc. From anonymous staff member we could observe his/her staff type (obs_SMStfTp). The set of anonymous staff members we will call anonymous staff (AnonStaff).

At the second stage of staff rostering we will take into account all the characteristics that can be related to a staff member. We assume that staff member's personal information makes him distinguishable from other staff members. So we will call specific staff member (SpecStfMbr) an anonymous staff member with added personal information. From a specific staff member we can observe his personal information as well as home depot and staff type.

From anonymous and specific staff member we can observe staff member's name. It makes the relation between two abstractions of a staff member - anonymous and specific.

Given a staff type we can observe all the depots which are home depots for staff members of a given staff type (function deps_staff below). Given a staff type and a depot we can observe all anonymous staff members at this depot of this staff type and respectively their number (functions dstft and dstft_num below).

5.3.2 Formal Model

We first state some types and some observer functions.

```
scheme STAFF =
 extend NETWORK with
 class
  type
   AnonStfMbr, Name,
   SpecStfMbr, PersInfo,
   StfTp == engS | condS | catS,
   AnonStaff = Name -m-> AnonStfMbr,
   Staff = Name -m-> SpecStfMbr

  value
   obs_Name: AnonStfMbr → Name,
   obs_Name: SpecStfMbr → Name,
   obs_SMStfTp : AnonStfMbr → StfTp,
   obs_SMStfTp: SpecStfMbr → StfTp,
   obs_SMDep : AnonStfMbr → Dep,
   obs_SMDep: SpecStfMbr → Dep,
   obs_PersInfo: SpecStfMbr → PersInfo
end
```

We will then illustrate some axioms and functions:

```
proj_SpecAnonStfMbr: SpecStfMbr →
     AnonStfMbr
proj_SpecAnonStfMbr(ssm) as asm
 kwpost obs_SMStfTp(ssm) = obs_SMStfTp(asm)
 ∧ obs_SMDep(ssm) = obs_SMDep(asm),

proj_AnonSpecStfMbr: AnonStfMbr × PersInfo
     → SpecStfMbr
proj_AnonSpecStfMbr(asm, pinf) as ssm
 post obs_Name(asm) = obs_Name(ssm) ∧
  obs_PersInfo(ssm) = pinf ∧
  obs_SMStfTp(asm) = obs_SMStfTp(ssm) ∧
  obs_SMDep(asm) = obs_SMDep(ssm),

 axiom ∀ asm: AnonStfMbr • ∃! ssm:
  SpecStfMbr • obs_Name(asm) =
    obs_Name(ssm)

axiom ∀ ssm, ssm′: SpecStfMbr • ssm ≠ ssm′
  ⇒ proj_SpecAnonStfMbr(ssm) =
    proj_SpecAnonStfMbr(ssm′)

value
 depStfMbrs : Dep → AnonStaff
 depStfMbrs(d) as astf
  post (∀ asm: AnonStfMbr •
  astf = [obs_Name(asm) ↦ asm] ∧
   obs_SMDep(asm) = d),

 deps_staff : StfTp → Dep-set
 deps_staff(stft) ≡
  {d | d : Dep • ∃ asm : AnonStfMbr •
     obs_SMStfTp(asm) = stft ∧
      obs_SMDep(asm) = d},

 dstft : Dep × StfTp → AnonStaff
 dstft(d, stft) as astf
 post (∀ asm: AnonStfMbr
  • astf = [obs_Name(asm) ↦ asm] ∧
   obs_SMDep(asm) = d
    ∧ obs_SMStfTp(asm) = stft),

 dstft_num : Dep × StfTp → Nat
 dstft_num(d, stft) ≡ card dom dstft(d, stft),

 dsstft_grs : Dep-set × StfTp →
   (Dep × Nat)-set
 dsstft_grs(ds, stft) ≡ {(dep, n) | dep : Dep,
   n : Nat • dep ∈ ds ∧
     n = dstft_num(dep, stft)}
```

5.4 SCHEDULE, JOURNEYS AND TRIPS

In this section we will explain the notions of schedule, journeys and trips that help us to introduce further the notion of duties.

5.4.1 Narrative

Schedule and Exchange stations: A schedule includes information about all train journeys such that each train journey is uniquely determined by a train number and a date and time. A train number is a unique identifier of a train which remains the same from the first to the last station of its journey. We don't consider train names here as not all the trains in a railway net have names.

Some of the stations in the net are special from staff management perspective because it is possible either to exchange staff members or a staff member to start or to finish his work there. We will call such stations exchange stations. From a station we could observe all the staff types for which this station is an exchange station (obs_ExchgStas). Given a station and a staff type we could check if the station is an exchange station or not for this staff type (is_exchgst). Exchange stations are located near the depots in the railway net.

Journeys and Trips: Staff members are responsible for performing some actions in order to fulfill the schedule demands. Some of the actions are related to train journeys. Train journeys could be both actual journeys with passengers or freights or empty trains journeys. A train journey is a sequence of rides with the same train number. A ride is characterized by a departure station, a departure time, an arrival station, an arrival time and a train between these two stations. Given a schedule we can extract a set of train journeys (journ_set).

There are some restrictions about the maximal working time for a staff member without a rest. Taking into account these restrictions it is natural to divide a journey into an indivisible pieces of work for staff members. That is why we introduce the notion of a trip. A trip is a sequence of rides of a train journey such that the first and the last station of a trip are exchange stations and the duration of a trip is less or equal to maximal allowed uninterrupted working time (maxUnIntWrkHr). Each trip is characterized by a train, a departure time, a departure station, an arrival time, an arrival station and possibly additional attributes. From a trip we can observe train characteristics for instance kind of the engine, staff types and their numbers needed to perform a trip etc.

5.4.2 Formal Model

First we will state some types(abstract and concrete) and some observer functions.

```
NETWORK, STAFF
scheme SCHEDULE =
extend STAFF with
class
 type
  Date, Hour,  Trn, TrnId, LongDistance,
  Urban, ICE, TGV, StfAttr, NoStf,
  TrnChar = LongDistance| Urban| ICE| TGV,
  DateTime = Date × Hour,
  Ride′ == rd(sta: Sta, dt: DateTime,
   nsta: Sta, at: DateTime, trn: Trn),
  Ride = {|rd: Ride′ • wf_rd(rd)|},
  Journey′ = Ride*,
  Journey = {|j: Journey′ • wf_journ(j)|},
  Trip = Ride*,
  TrpAttr == Overnight|Other,
  SCH= TrnId -m-> (DateTime -m-> Journey)

 value
  < : DateTime × DateTime → Bool,
   /* DateTime < DateTime*/
  ≤: TInt × TInt → Bool, /*TInt< TInt*/
  − : DateTime × DateTime → TInt,
  − : TInt × TInt → TInt,
  ≤: DateTime × DateTime → Bool,
  ≥: TInt × TInt → Bool,

  consec_intime: DateTime × DateTime →
    Bool,
  obs_TrnId: Trn → TrnId,
  trnchr: Ride → TrnChar,
  stfchr: TrnChar → StfTp -m-> Nat,
  obs_ExchgStas: Sta -~-> StfTp-set,
  techTime: Sta × Trn × StfTp → TInt,
  maxUnIntWrkHr: StfTp → TInt,
  /* from a staff type (rules taken into account
   implicitly) we can observe the maximal
   permitted working time in minutes without
   a rest */
  maxWrkHr: StfTp → TInt,
  /* from a staff type (rules taken into account
   implicitly) we can observe
  the maximal permitted working time*/
  tripAttr: Trip → TrpAttr,

  wf_rd : Ride′ → Bool
  wf_rd(rd) ≡ dt(rd) < at(rd),

  wf_journ: Journey′ → Bool
  wf_journ(j) ≡
  (∀ i: Nat • {i, i + 1} ⊆ inds j ⇒
   obs_TrnId(trn(j(i))) = obs_TrnId(trn(j(i+1)))
   ∧ nsta(j(i)) = sta(j(i+1)) ∧
   consec_intime(at(j(i)), dt(j(i+1)))),

  journ_set: SCH → Journey-set
  journ_set(sc) ≡ {j| j: Journey •(∀ trnid: TrnId,
   timdat: DateTime • trnid ∈ dom sc ∧ timdat
    ∈ dom sc(trnid)⇒ j=sc(trnid)(timdat))},
  journ_set1: SCH → Journey-set
  journ_set1(sc) ≡ ∪ {rng tn| tn:
   (DateTime →m Journey) • tn ∈ rng sc}
  end
```

Each train journey is divided into trips with subject to a staff type. The following is a function that divides a journey into trips.

```
trip_list : Journey × StfTp → Trip*
trip_list(j, stft) as trpl
 post (∀ i : Nat • i ∈ inds trpl ⇒
     wf_stft_trip(trpl(i), stft)) ∧
     check_separation(trpl, stft),
```

A trip is well formed if it consists of consecutive rides, the first and the last stations of a trip are exchangeable stations and the train during the trip has the same characteristics from a staff member perspective.

```
wf_stft_trip: Trip × StfTp → Bool
wf_stft_trip(trp , stft) ≡
is_exchgst(trip_fsta(trp), stft) ∧
is_exchgst(trip_lsta(trp), stft) ∧
∼(possible_exchg_inside(trp, stft)) ∧
trip_fnT(trp) − trip_stT(trp) ≤
maxUnIntWrkHr(stft) ∧ same_trn(trp, stft),

is_exchgst: Sta × StfTp → Bool
is_exchgst(s, stft) ≡ stft ∈ obs_ExchgStas(s),

possible_exchg_inside: Trip × StfTp → Bool
possible_exchg_inside(trp, stft) ≡
(∀ i: Nat • i ∈ {1..len trp −1} ⇒
 if is_exchgst(nsta(trp(i)), stft) then
  dt(trp(i + 1)) − at(trp(i)) ≥
   tech_time(trp(i), stft) else false end),

same_trn: Trip × StfTp → Bool
same_trn(trp, stft) ≡
(∀ i: Nat • {i, i + 1} ⊆ inds trp ⇒
 same_trnchr(trnchr(trp(i)),
       trnchr(trp(i + 1)), stft)),


same_trnchr: TrnChar × TrnChar × StfTp →
 Bool,
/∗checks if two trains are with the same
 characteristics from the staff point of view∗/

check_separation: Trip* × StfTp → Bool
check_separation(trpl, stft) ≡
(∀ i : Nat • {i, i + 1} ⊆ inds trpl ⇒
coincident_sta(trpl(i), trpl(i + 1)) ∧
div_sta(trpl(i), trpl(i + 1), stft)),

coincident_sta: Trip × Trip → Bool
coincident_sta(trp1, trp2) ≡
trip_lsta(trp1) = trip_fsta(trp2),
```

On the station where we separate the train journey there should be enough time for exchanging the staff members or a staff member to change a train. The time interval between departure and arrival time of a train at this station should be greater or equal to the technical time. Technical time is the smallest interval of time for which it is possible to exchange staff members or a staff member to change a train.

```
div_sta: Trip × Trip × StfTp→ Bool
div_sta(trp1, trp2, stft) ≡
trip_stT(trp2) − trip_fnT(trp1) ≥
 tech_time(last(trp1), stft),

tech_time: Ride × StfTp → TInt
tech_time(rd, stft) ≡
techTime(sta(rd), trn(rd), stft),

last: Trip ∼→ Ride
last(trp) ≡ trp(len trp)
pre len trp ≥ 1,
```

Finally given a schedule and a staff type we produce the trip set such that each journey that can be extracted from a schedule is divided into trips.

```
gen_tripss: SCH × StfTp → Trip-set
gen_tripss(sc, stft) ≡ ∪ {trips| trips: Trip-set •
                        trips = gen_trips(sc, stft)},
```

```
gen_trips : SCH × StfTp → Trip-set
gen_trips(sc, stft) as trps
post (∀ j: Journey • j ∈ journ_set(sc) ⇒
    trps = elems trip_list(j, stft))
```

The following are some functions that extract some characteristics of a trip.

```
trip_stT: Trip → DateTime
trip_stT(trp) ≡ dt(hd trp),

trip_fnT: Trip → DateTime
trip_fnT(trp) ≡ at(last(trp)),

trip_fsta: Trip → Sta
trip_fsta(trp) ≡ sta(hd trp),

trip_lsta: Trip → Sta
trip_lsta(trp) ≡ nsta(last(trp)),
```

```
trip_trn: Trip → Trn
trip_trn(trp) ≡ trn(hd trp),

trip_trnchr: Trip → TrnChar
trip_trnchr(trp) ≡ trnchr(hd trp),

trip_stfchr: Trip → StfTp →m Nat
trip_stfchr(trp) ≡ stfchr(trip_trnchr(trp)),

trip_WrkTm: Trip → TInt
trip_WrkTm(tp) ≡ trip_fnT(tp) − trip_stT(tp)
```

5.5 ACTIONS AND DUTIES

5.5.1 Narrative

Actions: Each staff member performs some actions. Actions could be sequence of trips, rests and some human resource activities. Rests could be rest between trips, meal rests, rests away from home depot including sleeping in dormitories (external rest) etc. By human resource activities we mean activities performing from a staff member in order to increase his qualification (seminars, courses etc.).

The sequence of trips is characterized with a start time, an end time and a list of rides. A rest is characterized by a start and an end time, station name and also some attributes. We will assume that a rest starts and ends at the same station. Human resource activities has the same characteristics as rests.

Duties: Each staff member is related to a given depot, home depot, in a railway net, which represents starting and ending point of his work segments. A natural constraint imposes that each staff member must return to his home depot within some period of time. This leads to the introduction of the concept of duty as a list of actions spanning L consecutive days such that its start and end actions are related to the same depot. A duty conforms to some rules and satisfy some requirements like continuance, working hours per duty etc. Each duty is concerned with members of the same staff type. From a duty we can observe duty attributes for example: 'duty with external rest', 'overnight duty', 'heavy overnight duty', 'long duty' etc. Also each duty has some characteristics as:

- Start time: it is given explicitly when the first action of a duty is either rest or human resource activity; in case of a trip it is defined as the departure time of its first ride minus the sum of technical departure time and briefing time,
- End time: it is given explicitly when the last action of a duty is either rest or human resource activity; in case of a trip it is defined as the arrival time of its last ride plus the sum of technical arrival time and debriefing time,
- Paid time: it is defined as the elapsed time from the start time to the end time of the duty,
- Working time: it is defined as the duration of the time interval between the start time and the end time of the duty, minus the external rest, if any.

Mentioned above characteristics are common for every duty. There are other possible characteristics of a duty but they strictly depend on a staff type. For instance taking into account engine men staff type we could observe:

- Driving time: it is defined as the sum of the trip durations plus all rest periods between consecutive trips which are shorter than M minutes e.g. 30 minutes,

Duties attributes and characteristics are taken into account in scheduling process while selecting feasible, efficient and acceptable duties per each depot and in sequencing duties into rosters. This will be introduced in the next sections.

Given the schedule, staff type, set of depots and rules we can generate duty sets per each depot.

5.5.2 Formal Model

```
scheme DUTY =
extend SCHEDULE with
class
 type
  RestAttr, HRAttr, DtChar,
  Ac == mk_trip(st: DateTime, tripl: Trip*,
     et: DateTime)|
    mk_rest(sr: DateTime, rsta: Sta,
     ratt: RestAttr, er: DateTime)|
    mk_hra(sh: DateTime, hsta: Sta,
     hatt: HRAttr, eh: DateTime),
  Duty = Ac*,
  DtAttr==ExtRest|Long|Overnight|
   HeavyOvernight,
  AcR = Ac × StfTp → Bool,
  AcRS = AcR-set,
  DuR = Duty × StfTp → Bool,
  DuRS = DuR -set,
  DepR = Dep × Duty-set × StfTp→ Bool,
  DepRS = DepR-set,
  OvDR = (Duty-set)-set × StfTp → Bool,
  OvDRS = OvDR-set,
  RS == check_acr(ar: AcRS)|
   check_dur(dur: DuRS)|
   check_dpr(dpr: DepRS)|
   check_ovdsr(ovdsr: OvDRS)
 value
  dt_maxlenght: StfTp → TInt,
  dt_char: Duty → DtChar,
  dt_attr: Duty → DtAttr
end
```

Each duty is generated taking into account some depot and some staff type. The following is a function which generates a duty set for a depot. It generates all possible duties for the depot.

```
gendep_dutys : Trip-set × StfTp × Dep × RS
 → Duty-set
gendep_dutys(trps, stft, dep, rs) as ds
post (∀ d: Duty • d ∈ ds ⇒
   d = gen_duty(trps, stft, dep, rs)) ∧
 ∼(∃ d′: Duty • d′ =
    gen_duty(trps, stft, dep, rs) ∧ d′ ∉ ds),
```

Each duty has to start and to end at the same depot and has to conform some rules. Rules are related to the sequence of actions in a duty, maximal number of actions with a given characteristics, rest time between actions, overall rest time, overall working time etc. These rules we will call rules at a duty level. Given a trip set, a staff type, a depot and rules we can generate a duty for the depot. The function below generates a duty such that its fist and its last action starts and respectively finishes at the depot, the depot is a home depot for staff members of the given staff type and the duty satisfy the rules.

```
gen_duty : Trip-set × StfTp × Dep × RS → Duty
gen_duty(trps, stft, dep, srs) as d
   post is_wfd(d, stft, srs) ∧ ac_dep(hd d, stft) = dep ∧
    dt_endt(d) − dt_startt(d) ≤ dt_maxlenght(stft) ∧
    (∃ trpl : Trip* •
        belong(trpl, d) ⇒ trip_stft(trpl, stft, dep)),

is_wfd: Duty × StfTp × RS → Bool
is_wfd(dt, stft, rs) ≡
  ac_dep(hd dt, stft) = ac_dep(dt(len dt), stft) ∧
   comp_dtTrips(dt, stft) ∧ conf_dt_rules(dt, stft, rs),

ac_dep : Ac × StfTp ~→ Dep
ac_dep(ac, stft) as dep
  post (∃ dep′:Dep •
   case ac of
     mk_trip(st, tripl, et) →
             dep ∈  st_stftdep(sta(hd (hd tripl)), stft),
     mk_rest(sr, rsta, ratt, er) →
             dep ∈  st_stftdep(rsta, stft),
     mk_hra(sh, hsta, hatt, eh) →
           dep ∈ st_stftdep(hsta, stft)
      end
   ∧ dep = dep′),

  st_stftdep: Sta × StfTp → Dep-set
  st_stftdep(st, stft) ≡
   {dep| dep: Dep • dep ∈ obs_StaDeps(st)∧
     is_exchgst(st, stft)},

  /∗ checks if all the trips in a duty has the same characteristics from staff point of view ∗/
  comp_dtTrips: Duty × StfTp → Bool
  comp_dtTrips(dt, stft) ≡
   (∀ i: Nat • i ∈ inds dt ⇒
      case dt(i) of
        mk_trip(sti, tripli, eti) →
        (∀ j: Nat • j ∈ inds dt ∧ j ≠ i ⇒
         case dt(j) of
           mk_trip(stj, triplj, etj) →
            same_trpchr(hd tripli, hd triplj, stft)
         end)
        end),

  same_trpchr: Trip × Trip × StfTp → Bool
  same_trpchr(trp1, trp2, stft) ≡
   same_trnchr(trip_trnchr(trp1), trip_trnchr(trp2), stft),

  conf_dt_rules: Duty × StfTp × RS → Bool
  conf_dt_rules(dt, stft, rs) ≡ satf(dt, stft, rs) ∧
   (∀ i : Nat • i ∈ inds dt ⇒ conf_ac(dt(i), stft, rs)),

  conf_ac: Ac × StfTp × RS → Bool
  conf_ac(ac, stft, rs) ≡
   case rs of
     check_acr(acrs) →
        (∀ acr: AcR • acr ∈ acrs ⇒ acr(ac, stft))
   end,
  /* checks if the rules for sequencing */
  /* actions in a duty are satisfied */
  satf: Duty × StfTp × RS → Bool
  satf(dt, stft, rs) ≡
   case rs of
     check_dur(durs) →
        (∀ dur:DuR • dur ∈ durs ⇒ dur(dt, stft))
   end,

  belong: Trip* × Duty → Bool
  belong(tpl,dt) ≡ (∃ ac: Ac • ac ∈ elems dt ∧
   case ac of
     mk_trip(st,tpl,et) → true
   end),

  trip_stft: Trip* × StfTp × Dep → Bool
  trip_stft(trpl, stft, dep) ≡
     let stfm = trip_stfchr(hd trpl) in
      stft ∈ dom stfm ∧ dstft_num(dep, stft) > 0 end,
```

The set of all duties for a depot has to obey to some rules too. The rules/restrictions could be related to a maximal number of duties with specific characteristics per depot, maximal number of duties per depot etc. We will call these rules rules on a depot level.
The function below selects a subset of a duty set, generated on previous stage, such that it satisfies the rules on the depot level and there is enough staff at the depot to perform the duty set.

```
seldep_dutys : Trip-set × StfTp × Dep × RS
 → Duty-set
seldep_dutys(trps, stft, dep, rs) as ds
post let ds1 = gendep_dutys(trps, stft, dep, rs)
  in ds ⊆ ds1 ∧
   conf_dts_deprules(dep, ds, stft, rs) ∧
   enough_staff(ds, stft, dep) end,

conf_dts_deprules: Dep × Duty-set × StfTp
 × RS → Bool
conf_dts_deprules(dep,ds,stft, rs) ≡
 case rs of
  check_dpr(dprs) → (∀ dpr: DepR • dpr ∈
   dprs ⇒ dpr(dep, ds, stft))
 end,

enough_staff: Duty-set × StfTp × Dep →
 Bool
enough_staff(ds, stft, dep) ≡
 dutys_staff_numb(ds, stft) ≤
  dstft_num(dep, stft),

dutys_staff_numb: Duty-set × StfTp → Nat,
/*the number of people should be equal to
the number of duties, but in case of a
conductor staff type the number of people
may be more than the number of duties as two
conductors may have the same duties*/
```

Finally given a trip set, a staff type, a depot set and rules we can generate a set of duties per each depot.

```
gen_dutys : Trip-set × StfTp × Dep-set × RS
 → (Duty-set)-set
gen_dutys(trps, stft, deps, rs) as dss
 post (∀ ds: Duty-set • ds ∈ dss ⇒
   (∃! dep: Dep • dep ∈ deps ∧
    ds = seldep_dutys(trps, stft, dep, rs))) ∧
    card dss = card deps,
```

The union of generated sets of duties per each depot has to conform to some overall rules e.g. the number of duties as a whole with a given characteristics not to exceed some defined number etc. Also the generated duties as a whole has to cover all the trips that can be observed from a schedule. Finally given a schedule, a staff type, set of depots and rules we can generate set of duties per each depot such that the mentioned above constraints are satisfied.

```
sel_dutyss : SCH × StfTp × Dep-set × RS →
 (Duty-set)-set
sel_dutyss(sc, stft, deps, rs) as dss
 post let trps = gen_tripss(sc, stft) in
    dss = gen_dutys(trps, stft, deps, rs) ∧
  cover(dss, trps) ∧ conf_dts_ovr(dss, stft, rs)
   end,

cover : (Duty-set)-set × Trip-set → Bool,

conf_dts_ovr: (Duty-set)-set  × StfTp × RS →
 Bool
conf_dts_ovr(dss, stft, rs) ≡
 case rs of
 check_ovdsr(ovdsrs) → (∀ ovdsr: OvDR •
  ovdsr ∈ ovdsrs ⇒ ovdsr(dss, stft))
 end,
```

The following are some functions concerning a duty and its characteristics.

```
duty_dep: Duty × StfTp → Dep
duty_dep(dt, stft) as dep
 post dep ∈ st_stftdep(dt_fsta(dt),stft),

dt_fsta: Duty → Sta
dt_fsta(dt) ≡
 case hd dt of
 mk_trip(_, tripl, _) → trip_fsta(hd tripl),
 mk_rest(_, rsta, _, _) → rsta,
 mk_hra(_, hsta, _, _) → hsta
end,

dt_lsta: Duty → Sta
dt_lsta(dt) ≡
 case dt(len dt) of
 mk_trip(_, tripl, _) → trip_fsta(hd tripl),
 mk_rest(_, rsta, _, _) → rsta,
 mk_hra(_, hsta, _, _) → hsta
end,

dt_starttime: Duty → DateTime
dt_starttime(dt) ≡
 case hd dt of
 mk_trip(st, tripl, et) → st,
 mk_rest(sr, rsta, ratt, er) → sr,
 mk_hra(sh, hsta, hatt, eh) → sh
 end,

dt_endtime: Duty → DateTime
dt_endtime(dt) ≡
 case dt(len dt) of
 mk_trip(st, tripl, et) → et,
 mk_rest(sr, rsta, ratt, er) → er,
 mk_hra(sh, hsta, hatt, eh) → eh
 end,

duty_stft_num: Duty → StfTp →~ Nat
duty_stft_num(dt) as stfm
 post (∃ trpl: Trip* • belong(trpl, dt) ⇒
 stfm = trip_stfchr(hd trpl))
```

5.6 ROSTERS AND STAFF MEMBERS

In this section we will explain the notion of a roster and how it is related to staff members.

5.6.1 Narrative

Rosters: During the second stage of staff rostering the duties generated at previous stage are ordered in rosters which are long term working schedules assigned to specific staff members. For each depot in a depot set, a separate staff rostering problem is solved considering only the corresponding duties. We will introduce two help notions in order to explain the concept of roster and its stages of generation.

A plan roster is a sequence of duties generated for anonymous staff members of the same staff type. A base roster is a cyclic sequence of a plan roster such that it spans trough a planning period determined by a schedule. In other words, a plan roster is that part of the base roster which is repeated several times and a base roster is just a cyclic sequence of duties. Each base roster has to satisfy some rules. The rules are about the order of duties in a consecutive days and their attributes. Also there are some constraints concerning number of duties in a base roster with determined attributes. These rules we will call conventionally rules at the roster level.

So given a schedule, a staff type, a depot and rules we can generate base rosters for the given depot. These base rosters have to cover all the duties corresponding to this depot and have to conform to some rules. The rules at this level we will call conventionally rules at the overall roster level.

All the duties in a base roster has to be performed by a specific staff member. We will call roster a cyclic sequence of duties (base roster) for a specific staff member such that he/she could perform them. So from a base roster and a staff type we can generate rosters. The number of staff members assigned to the base roster is equal to the length of the plan roster. All staff members perform the base roster but starting at a different day.

Staff Members:

During the assignment of duties in a base roster to staff members we consider specific staff members. At this stage we are working with specific staff members as we are interested in their personal information. From a staff member personal information we could observe his/her private information (obs_PrInf) as date of birth, place of living, address etc. Also we could observe his qualification (obs_Qual), special work requirements (obs_SpWrkReq) and the list of his/her previous duties (obs_PrevDuty).

Given a base roster and a staff member we can observe his roster which is considered to his/her attributes.

5.6.2 Formal Model

```
scheme ROSTER =
 extend DUTY with
 class
  type
   Info, WrkReq, Qualification,
   PlRos = Duty*,
   BRos = PlRos × Nat,
   RoR = PlRos × StfTp → Bool,
   RoRS = RoR-set,
   OvR = BRos × StfTp → Bool,
   OvRS = OvR-set,
   eRS == RS | check_ror(rrs : RoRS) |
    check_ovrs(ovrs : OvRS),
   Ros = SpecStfMbr →m BRos
  value
   f : eRS → RS,
   obs_PrInf : PersInfo → Info,
   obs_SpWrkReq : PersInfo → WrkReq,
   obs_PrevDuty : PersInfo → Duty*,
   obs_PostDuty : PersInfo → Duty*,
   obs_Qualf : PersInfo → Qualification,
   obs_PlPer : SCH → Nat,

   bros_length : BRos → Nat
   bros_length(bros) ≡
    let (plros, rnumb) = bros in
    len plros end
 end
```

The following function generates all possible base rosters for a given duty set (related to a depot).

```
   gen_dep_bross : SCH × StfTp × Dep × eRS
    → BRos-set
   gen_dep_bross(sc, stft, dep, rs) as bross
    post (∀ bros : BRos •
       bros ∈ bross ⇒
       bros = genbros_dep(sc, stft, dep, rs)) ∧
    ∼(∃ bros′ : BRos •
      bros′ = genbros_dep(sc, stft, dep, rs) ∧
      bros′ ∉ bross),

   genbros_dep : SCH × StfTp × Dep × eRS
    → BRos
   genbros_dep(sc, stft, dep, rs) as bros post
    let ds = dep_dutyset(dep, stft) in
     cover_rds(bros, ds)
    end ∧ wf_bros(bros, sc, stft, rs),

   cover_rds : BRos × Duty-set → Bool,

   wf_bros : BRos × SCH × StfTp × eRS
    → Bool
   wf_bros(bros, sc, stft, rs) ≡
```

```
  let (plros, rnumb) = bros in
 same_qualific(plros, stft) ∧
 conform_cplrosrs(plros, stft, rs) ∧
 len plros ∗ rnumb = obs_PlPer(sc)
 end,

same_qualific : PlRos × StfTp → Bool
same_qualific(plros, stft) ≡
 (∀ i : Nat • {i, i + 1} ⊆ inds plros ⇒
  sm_qual(plros(i), plros(i + 1))),

sm_qual : Duty × Duty → Bool,

conform_cplrosrs : PlRos × StfTp × eRS
 → Bool
conform_cplrosrs(plros, stft, rs) ≡
 conform_plrosrs(plros, stft, rs) ∧
 let cycros = ⟨plros(len plros)⟩ ⁀
  ⟨hd plros⟩ in
   conform_plrosrs(cycros, stft, rs)
 end,

conform_plrosrs : PlRos × StfTp × eRS
 → Bool
conform_plrosrs(plros, stft, rs) ≡
 case rs of
  check_ror(rrs) →
   (∀ rr : RoR • rr ∈ rrs ⇒ rr(plros, stft))
 end,
```

The generated, on previous stage, set of base rosters has to conform to some rules as maximal percentage of base rosters with particular characteristics etc.

```
sel_dep_bross: SCH × StfTp × Dep × eRS
 → BRos-set
sel_dep_bross(sc, stft, dep, rs) as bross
 post let bross1 = gen_dep_bross(sc, stft,
   dep, rs) in bross ⊆ bross1 ∧
    conform_bros_rules(bross, stft, rs)
   end,

conform_bros_rules : BRos-set × StfTp × eRS
 → Bool
conform_bros_rules(bross, stft, rs) ≡
 (∀ bros: BRos • bros ∈ bross ⇒
 case rs of
  check_ovrs(ovrs) →
   (∀ ovr : OvR •
     ovr ∈ ovrs ⇒ ovr(bros, stft))
 end),
```

Having a base roster and a staff type and a depot we can produce rosters for the specific staff members of the given staff type.

```
gen_ssmros : BRos × StfTp × Dep → Ros
gen_ssmros(bros, stft, dep) as ros
 post let sms = dstft_gr(dep, stft) in
    ros = assignment(bros, sms) ∧
    card dom ros = bros_length(bros)
   end,

dstft_gr : Dep × StfTp → Staff
dstft_gr(dep, stft) ≡
 let anstaff = dstft(dep, stft) in
  get_staff(anstaff)
 end,

get_staff: AnonStaff → Staff
get_staff(anstaff) as staff
 post (∀ asm: AnonStfMbr • asm ∈
  rng anstaff ⇒
   (∃! ssm: SpecStfMbr • ssm ∈ rng staff ∧
   obs_Name(asm) = obs_Name(ssm)))
```

Given a base roster and staff we assign specific staff members to the base roster such that we receive a set of rosters. The number of rosters is equal to the length of the base roster. All the rosters are permutations of the base roster. So at this stage of planning we assign specific staff members to duties in the plan roster (cyclic part of the base roster).

```
assignment : BRos × Staff → Ros
assignment(bros, staff) as ros
 post (∀ dt : Duty • duty_in_bros(dt, bros)⇒
    (∃! ssm : SpecStfMbr • ssm ∈ dom ros ∧
    dt = first_bros_duty(ros(ssm)) ∧
    conform_rsm(ros(ssm),ssm) ∧
     permutation(ros(ssm), bros)))
 pre card rng staff > bros_length(bros),

duty_in_bros: Duty × BRos → Bool
duty_in_bros(dt, bros) ≡
 let (plros, rnumb) = bros in
```

```
   dt ∈ elems plros end,

 first_bros_duty: BRos → Duty
 first_bros_duty(bros) ≡
  let (plros, rnumb) = bros in
  hd plros end,
```

Each roster is assigned to a specific staff member according to his/her qualification, special work requirements and previous duties such that he/she could perform it.

```
 conform_rsm : BRos × SpecStfMbr → Bool
 conform_rsm(bros, ssm) ≡
  satisfy_qual(bros,
   obs_Qualf(obs_PersInfo(ssm))) ∧
  satisfy_predt(bros,
   obs_PrevDuty(obs_PersInfo(ssm))) ∧
  satisfy_swr(bros,
   obs_SpWrkReq(obs_PersInfo(ssm))),

 satisfy_qual : BRos × Qualification → Bool,
 satisfy_predt : BRos × Duty* → Bool,
 satisfy_swr : BRos × WrkReq → Bool,

 permutation: BRos × BRos → Bool,
```

Finally we generate the rosters for the given depot and staff type such that for each base roster generated at the previous stage we generate rosters.

```
 gen_sross : SCH × StfTp × Dep × eRS → Ros
 gen_sross(sc, stft, dep, rs) as ros
  post let bross =
    sel_dep_bross(sc, stft, dep, rs) in
    (∀ bros : BRos • bros ∈ bross ⇒
     ros = gen_ssmros(bros, stft, dep))
  end,
```

All rosters are generated taking into account a staff type. So using the function above we can generate all rosters per depot for all staff types related to this depot. In this case to generate rosters per depot we will need only the schedule, the depot and the rules.

```
 dep_rosters : SCH × Dep × eRS →
  StfTp $\underset{m}{\rightarrow}$ Ros
 dep_rosters(sc, dep, rs) as stft_ross
  post (∃! stft : StfTp •
     stft ∈ dep_stftypes(dep) ⇒
     let rset = gen_sross(sc, stft, dep, rs) in
      stft_ross = [stft ↦ rset] end)
     ∧ card dep_stftypes(dep) =
      card dom stft_ross,

 dep_stftypes : Dep → StfTp-set
 dep_stftypes(dep) ≡ {stft| stft: StfTp •
  ∃ ssm: SpecStfMbr •
   obs_SMDep(ssm) = dep},
```

Base rosters and respectively rosters are generated per depot and we have the assumption that after the staff scheduling stage all duties generated per depot are shifted to the depot. If this is not the case we could observe all the duties generated in staff scheduling stage per depot (dep_dutyset) which will help us to integrate the two stages in staff planning into one. So given a schedule, a staff type, a set of depots and rules we will produce all rosters per each depot in the depot set for the given staff type.

```
 obtain_ross : SCH × StfTp × Dep-set × eRS
  → Ros-set
 obtain_ross(sc, stft, deps, rs) as rosset
  post let dtss = sel_dutyss(sc, stft, deps, f(rs))
   in (∀ ross : Ros • ross ∈ rosset ⇒
     (∃! dep : Dep • dep ∈ deps ⇒
      ross = gen_sross(sc, stft, dep, rs) ∧
      dep_dutyset(dep, stft) ∈ dtss)) end ∧
     card rosset = card deps,

 dep_dutyset: Dep × StfTp → Duty-set
 dep_dutyset(dep, stft) ≡
  {dt| dt: Duty • dep = duty_dep(dt, stft)}
```

The rest is a small part of the possible functions for operating with staff members in depots.

```
hire_sm: SpecStfMbr × Staff $\xrightarrow{\sim}$ Staff
hire_sm(ssm, stf) ≡ stf ∪
  [obs_Name(ssm) ↦ ssm]
 pre (∀ ssm′: SpecStfMbr • ssm′ ∈ rng stf ⇒
    obs_Name(ssm′) ≠ obs_Name(ssm)) ∧
    ssm ∉ rng stf,

fire_sm: SpecStfMbr × Staff $\xrightarrow{\sim}$ Staff
fire_sm(ssm, stf) ≡ stf \ {obs_Name(ssm)}
 pre obs_Name(ssm) ∈ dom stf,

hired_sm: SpecStfMbr × Staff → Bool
hired_sm(ssm, stf) ≡ ssm ∈ rng stf,

add_specsm: AnonStfMbr × PersInfo × Name
 → SpecStfMbr
add_specsm(asm, pinf, nm) as ssm
 post obs_Name(asm) = nm ∧
  obs_SMStfTp(asm) = obs_SMStfTp(ssm) ∧
  obs_SMDep(asm) = obs_SMDep(ssm) ∧
  obs_PersInfo(ssm) = pinf,

get_specsm : AnonStfMbr × PersInfo
 → SpecStfMbr
get_specsm(asm, pinf) as ssm
 post obs_Name(asm) = obs_Name(ssm) ∧
 obs_PersInfo(ssm) = pinf,

dep_staff : Dep → Staff
dep_staff(dep) ≡
 let anstaff = depStfMbrs(dep) in
  get_staff(anstaff)
 end
```

Part IV

Train Monitor & Control

6

Station Interlocking

Martin Pěnička and Dines Bjørner

Contents

6.1 Route Descriptions

Routes are described in terms of Units, Switches, Signals and Interlocking tables. In the previous section 6.1 in paragraphs 23 and 24 the route is defined as a sequence of pairs of units and paths, such that the path of a unit/path pair is a possible path of some state of the unit, and such that "neighboring" connector are identical. There can be many such routes in the station. We are only interested in routes which start at the signal and end either at the track or on the line. In the example station in Figure 6.1 you can find 16 such routes.

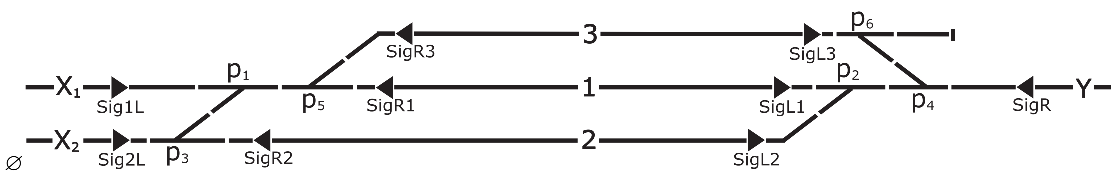


Fig. 6.1. Example Station.

All routes for each station are written in Interlocking tables. For each route inside the station the required setting of each switch and signal in the station are written in the Interlocking table. If there are no requirements on the setting, it is marked with –.

In paragraph 25 in section 6.1 you can find that route can be open or close. The route can be open only when all requirement for the route in the Interlocking table are full-filled.

The Interlocking table for the example station in Figure 6.1 is shown in Table 6.1.

We can now start to construct Petri Net for the interlocking routes inside the station. We build this Petri Net from for subparts: for a Unit, for a Switch and for a Signal.

6.2 Petri Net for a Unit

The Petri Net for a Unit is extremely simple – it consists of a single place. If the place is marked, the Unit is free, otherwise it is either blocked or occupied. A unit is free if and only if it is not part of a route.

Requirements:	Switches						Signals								
Routes	p1	p2	p3	p4	p5	p6	Sig1L	Sig2L	SigL1	SigL2	SigL3	SigR	SigR1	SigR2	SigR3
1. $Sig1L - 1$	S	–	S	–	S	–	G	–	–	–	–	–	R	–	R
2. $Sig1L - 3$	S	–	S	–	T	–	G	–	–	–	–	–	R	–	R
3. $Sig2L - 1$	T	–	T	–	S	–	R	G	–	–	–	–	R	R	R
4. $Sig2L - 2$	S	–	S	–	–	–	–	G	–	–	–	–	–	R	–
5. $Sig2L - 3$	T	–	T	–	T	–	R	G	–	–	–	–	R	R	R
6. $SigL1 - Y$	–	S	–	S	–	S	–	–	G	R	R	R	–	–	–
7. $SigL2 - Y$	–	T	–	S	–	S	–	–	R	G	R	R	–	–	–
8. $SigL3 - Y$	–	–	–	T	–	T	–	–	R	R	G	R	–	–	–
9. $SigR - 1$	–	S	–	S	–	S	–	–	R	R	R	G	–	–	–
10. $SigR - 2$	–	T	–	S	–	S	–	–	R	R	R	G	–	–	–
11. $SigR - 3$	–	–	–	T	–	T	–	–	R	R	R	G	–	–	–
12. $SigR1 - X_1$	S	–	S	–	S	–	R	–	–	–	–	–	G	–	R
13. $SigR1 - X_2$	T	–	T	–	S	–	R	R	–	–	–	–	G	R	R
14. $SigR2 - X_2$	S	–	S	–	–	–	–	R	–	–	–	–	–	G	–
15. $SigR3 - X_1$	S	–	S	–	T	–	R	–	–	–	–	–	R	–	G
16. $SigR3 - X_2$	T	–	T	–	T	–	R	R	–	–	–	–	R	R	G

Table 6.1. Interlocking table for routes through the example station.

6.3 Petri Net for a Switch

A typical switch has two settings: Straight and Turn. A switch may be required to be set in certain position in two ways: as a direct part of the route, or because it must be set for side protection. In the both cases, the switch is blocked. A blocked switch may not change setting.

The Petri Net for a switch has two places representing the two settings Straight and Turn. The initial marking consists of n tokens at the Straight place, where n is the number of routes which require setting of that switch. The switch can change state if and only if all n tokens are available. With the example in Figure 6.1 and its Interlocking table (Table 6.1), one finds that for switch p1, n=10, for switch p2, n=4, etc. If less than n tokens are available, the Switch is blocked. This will ensure that the switch can only change when no route that requires a particular setting is active, but still the switch can be part of several routes, as long as these routes require the switch to be in the same setting. These requirements are captured by the Petri Net in Figure 6.2(a).

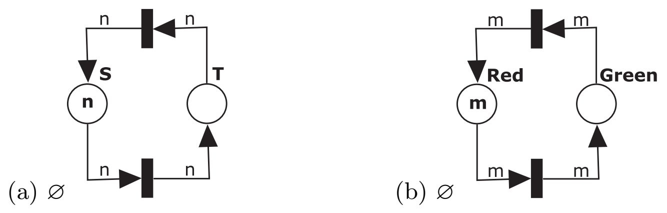


Fig. 6.2. (a) Petri Net for a Switch, (b) Petri Net for a Signal

6.4 Petri Net for a Signal

The Petri Net for a signal also has two places representing the two settings Green and Red[1]. The initial marking consists of m tokens at the Red place, where m is the number of routes which require setting of that signal. The signal can only change state if all m tokens are available. With the example 6.1 and its Interlocking table 6.1 one can easily find that for signal $Sig1L$, $m = 8$, for

[1] This is a simplistic view – a real signal is able to indicate the speed with which it may be passed.

signal $Sig2L$, $n = 6$, etc. The signal can only change setting if all m tokens are available. This will ensure that it can only change when no route that require a particular setting is active, but still the signal can be part of several routes, as long as these routes require the signal to be in the same setting. These requirements are captured by the Petri Net in Figure 6.2(b).

6.5 Constructing the Petri Net for a Route

The Petri Net for a route also has two places representing the two states: Open and Closed. The initial marking consists of one token at the Closed place. The basic Petri Net for a route is shown in Figure 6.3(a). This corresponds to the route that has no requirements on switches, signals or units.

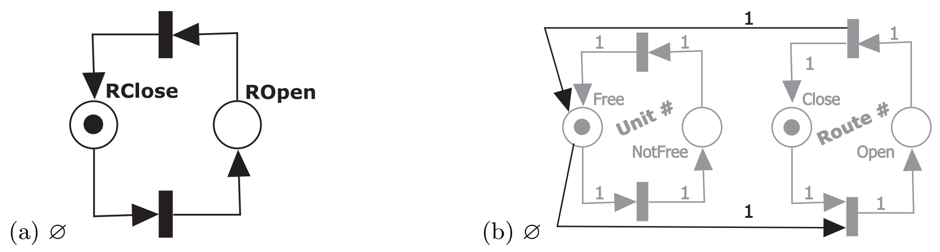


Fig. 6.3. Additions for (a) Petri Net for a Route , (b) Additions for unit requirement

The route can be open, when all units that the route is composed of, are not occupied by train or blocked by another route in the station. Figure 6.3(b) shows how to add this unit requirement.

For each switch requirement it must be ensured that the switch cannot change setting while the route is open. This requirement is captured in the Petri Net in Figure 6.4. Note, that in the figure it is assumed the route requires the switch to be set to Turn. The case for Straight is obvious.

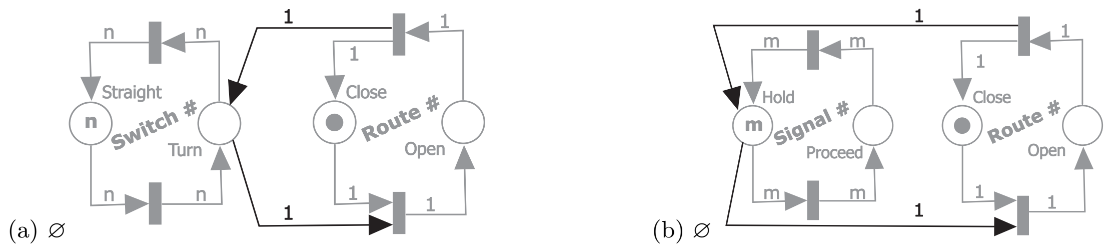


Fig. 6.4. Additions for (a) switch requirement (b) signal requirement

Figure 6.4(b) illustrates adding a signal requirement. The Figure illustrates the situation where the signal is required to be Red. The case for Green is obvious.

6.6 Discussion

Elsewhere it is shown how to formally integrate specifications expressed in `RSL` with such which are expressed, as in this chapter, using Petri Nets [37, 36, 9]. Chapters 12 of Vol. 2 of [10] consolidates the above.

7

Signalling on Lines

Martin Pěnička and Dines Bjørner

Contents

The problem with high train speed and low coefficient of friction between train wheels and track is that the drivers cannot stop their trains within sighting distance of another train or within sighting distance of a signal. This is the reason why automatic signaling is used on some lines. If there are junctions or turnouts then 'semi-automatic signalling' is required. Station interlocking systems are described in chapter 6.

In this chapter we first describe in natural language (as apposed to formal description) the principle of automatic line signalling. Then we give formal description examples using State Charts [18] of what we have described. Finely we give precise formal description of the charts and description of the State Charts in RAISE ([16]).

7.1 Narrative

Lines are usually divided into segments $l = \langle s_1, s_2, ..., s_{i-1}, s_i, s_{i+1}, ..., s_n \rangle$. Line l connects exactly two stations - *staA* and *staB*. A line can be in one of three possible states: '*OpenAB*', '*OpenBA*' and '*Close*'. These states and their possible transition are described in detail in chapter 8 on Line Direction Agreement.

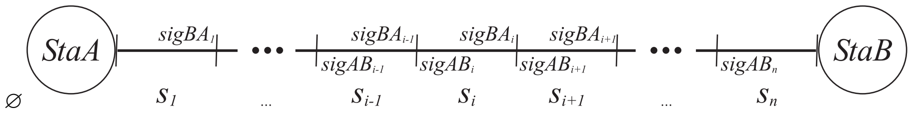


Fig. 7.1. Automatic Line Signalling

Each segment can be in two states: '*segFree*' and '*segOccupied*'. Segment s_i is in '*segFree*' when no train is detected in the segment. Segment s_i is in '*segOccupied*' when a train is detected in the segment.

7.1.1 General Line Segment

For each inner segment s_i, where $i = \langle 2, ..., n-1 \rangle$, there are two signals $sigAB_i$ and $sigBA_i$ (one in each direction of travel).

With each signal we associate four possible states: '*sigOnRed*', '*sigOnYellow*', '*sigOnGreen*' and '*sigOff*'.

Signal $sigAB_i$ is in

'*sigOnRed*'	state, when line l is in '*OpenAB*' state and segment s_i is in '*segOccupied*' state,
'*sigOnGreen*'	state, when line l is in '*OpenAB*' state and both segment s_i and s_{i+1} are in '*segFree*' state,
'*sigOnYellow*'	state, when line l is in '*OpenAB*' state and segment s_i is in '*segFree*' and segment s_{i+1} are in '*segOccupied*' state,
'*sigOff*'	state, when line l is in '*OpenBA*' or '*Closed*' state.

Signal $sigBA_i$ is in

'*sigOnRed*'	state, when line l is in '*OpenBA*' state and segment s_i is in '*segOccupied*' state,
'*sigOnGreen*'	state, when line l is in '*OpenBA*' state and both segment s_i and s_{i-1} are in '*segFree*' state,
'*sigOnYellow*'	state, when line l is in '*OpenBA*' state and segment s_i is in '*segFree*' and segment s_{i-1} are in '*segOccupied*' state,
'*sigOff*'	state, when line l is in '*OpenAB*' or '*Closed*' state.

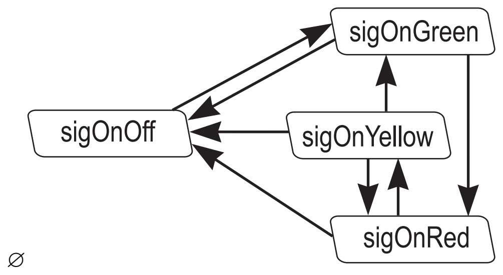


Fig. 7.2. Possible transmissions of signal states

Each segment has two signals, each signal can be in four states. One can calculate total number of 16, but possible combinations are:

$sigAB_i$	$sigBA_i$
'*sigOnRed*'	'*sigOff*'
'*sigOnYellow*'	'*sigOff*'
'*sigOnGreen*'	'*sigOff*'
'*sigOff*'	'*sigOff*'
'*sigOff*'	'*sigOnRed*'
'*sigOff*'	'*sigOnYellow*'
'*sigOff*'	'*sigOnGreen*'

7.1.2 First Line Segment

For segment s_1 there is only only one signal $sigBA_1$, for segment s_n there is only one signal $sigAB_n$ (see figure 7.1). The signals in the opposite directions ($sigAB_1$ and $sigBA_n$) are controlled manually in the stations. The details description of the station interlocking is given in chapter 6.

7.1.3 Last Line Segment

To increase the total capacity of line these states can be extended by one more state.

7.2 State Charts

In this section, we show how description of automatic line signalling can be expressed by using state charts [18].

7.2.1 General Model

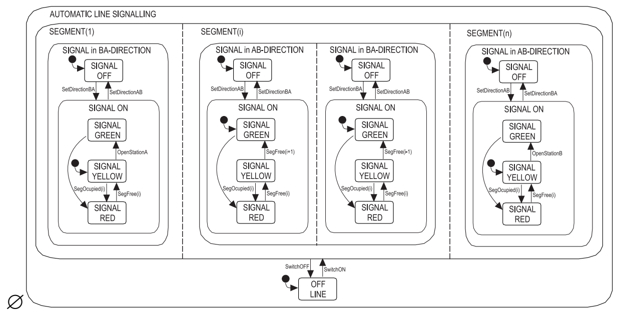


Fig. 7.3. General State Charts for Automatic Line Signalling

7.2.2 First, General and Last Segments

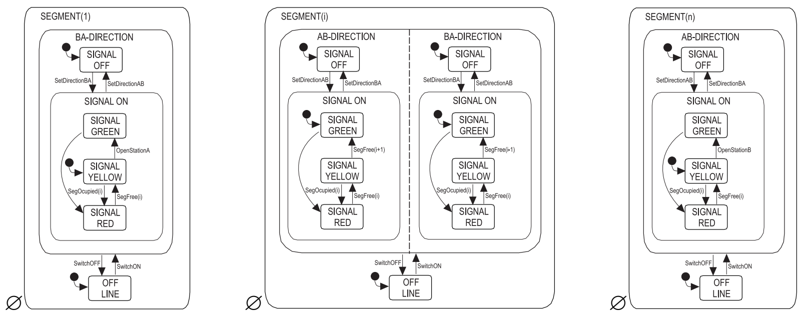


Fig. 7.4. First, General and Last Segments

7.2.3 Line with one Segment

There are no signals to be controlled automatically.

7.2.4 The line with two segments

We refer to Fig. 7.5 on the following page.

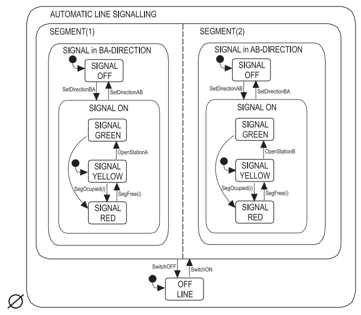


Fig. 7.5. Two Segments

7.3 Discussion

Elsewhere it is shown how to formally integrate specifications expressed in `RSL` with such which are expressed, as in this chapter, using Statecharts [37, 36, 9]. Chapter 14 of Vol. 2 of [10] consolidates the above.

8

Line Direction Agreement

Martin Pěnička and Dines Bjørner

Contents

In this chapter we first describe in natural language the principle of Line Direction Agreement Device. Then we give formal description examples using State-Charts and using Live Sequence Charts of what we have described. Finely we give precise formal description of the charts and description of the State-Charts and Live Sequence Charts in RAISE ([16]).

Each line connects exactly two stations. At any point in time, the line can be open in at most one direction. This is to protect head-on train crashes on the line.

In the old days, a sheet of paper was used and only that station, which had the sheet, could send trains to the line. The sheet was sent by trains between stations. Later on, the sheet of paper was replaced by abstract token transited electronically (Electric Token Block or Radio Electronic Token Block).

8.1 Narrative

The Line Direction Agreement System (LDAS) is a device that is responsible for fail-safe communication (token transition) and train direction control on the line between two stations.

Let us have a line l that connects two stations - called *staA* and *staB*. The line can be in either of three states: '*OpenAB*', '*OpenBA*' and '*Close*'. On that line LDAS device is installed.

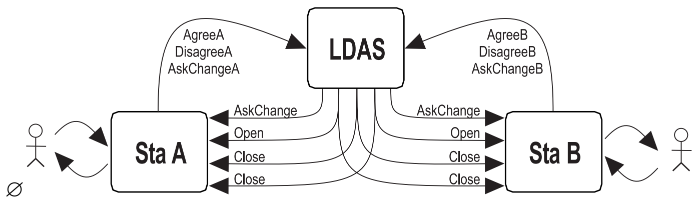


Fig. 8.1. Comunication with LDAS

Both stations communicate with the LDAS (see figure 8.1). From the first station *StaA* to LDAS there are three types of commands, which can be sent: '*AskChangeA*', '*AgreeA*' and '*DisagreeA*'. From the second station *StaB* to LDAS there are three types of commands, which can be sent: '*AskChangeB*', '*AgreeB*' and '*DisagreeB*'. LDAS sends either of three different commands '*Open*', '*Close*' or '*AskChange*'.

8.2 State Chart

The behavior of the LDAS in response to internal and external stimuli depends on the state(s) it is currently in. That is the reason, why we for graphical representation of internal behavior introduce StateChart [18]. StateCharts are represented graphically as so-called *higraphs.* Complete StateChart that represent internal behavior of LDAS is shown in Fig. 8.2.

LDAS can be either one of several states during its operation. The five most important states are '*LockedAB*', '*LockedBA*', '*AskedAB*', '*AskedBA*' and '*Dead*'. All possible transmissions between these states are shown as an arrow with a label.

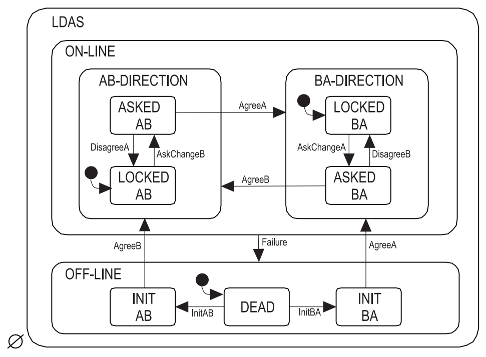


Fig. 8.2. LDAS - State Chart

8.3 Live Sequence Charts

In this section possible scenarios of communications be graphical representation of LDAS is described. All possible scenarios can be expressed by Live Sequence Charts. In total, there are eight possible scenarios, four in each direction. These scenarios

A station receiving '*Open*' command for line l from LDAS is thus told that the line l is open from that station (trains can travel from that station to the line).

A station receiving '*Close*' command for line l from LDAS is thus told means, that the line is close from that station (no train is allowed to leave from that station to the line).

A station receiving '*AskChange*' command for line l from LDAS is thus told that the line is open from that station but that other station is asking for direction change. A reply is expected.

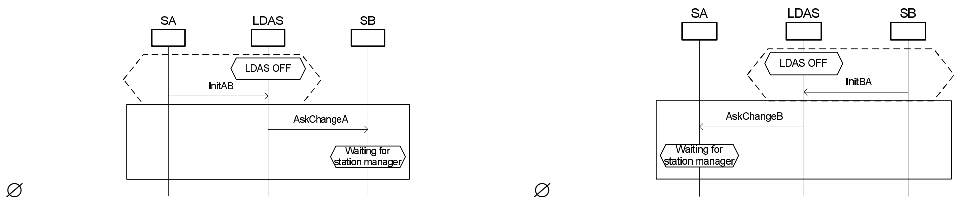


Fig. 8.3. Initializations to AB- and BA-Direction

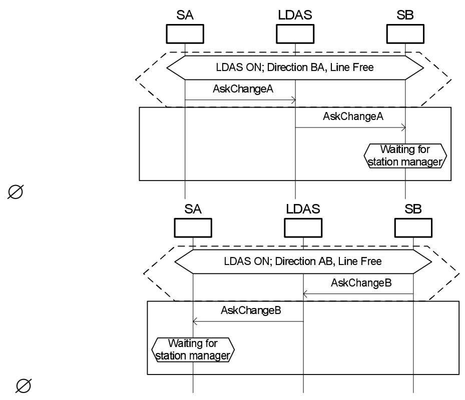

SA
LDAS
SB
LDAS ON; Direction BA, Line Free
AskChangeA
AskChangeA
Waiting for station manager
∅
SA
LDAS
SB
LDAS ON; Direction AB, Line Free
AskChangeB
AskChangeB
Waiting for station manager
∅

Fig. 8.4. Change Direction Requests

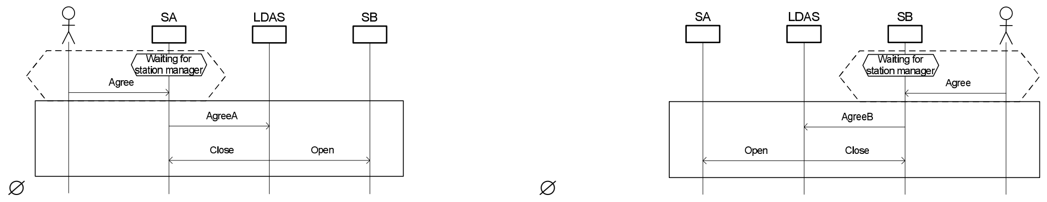


Fig. 8.5. Change Direction Approvals

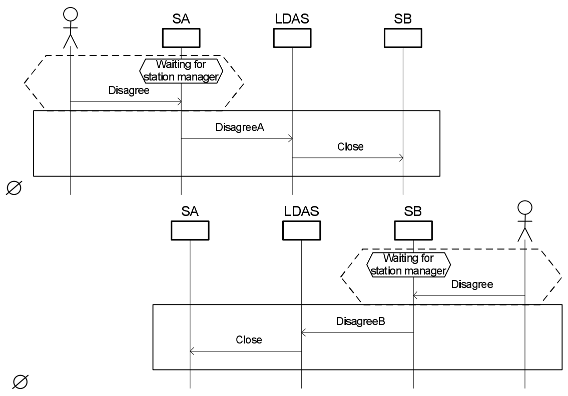


Fig. 8.6. Change Direction Disapprovals

8.4 Discussion

Elsewhere it is shown how to formally integrate specifications expressed in `RSL` with such which are expressed, as in this chapter, using Statecharts and Live Sequence Charts [37, 36, 9]. Chapters 13 and 14 of Vol. 2 of [10] consolidates the above.

Part V

The CyberRail Concept

9

Towards a Formal Model of CyberRail

Dines Bjørner et al.[1]

Contents

9.1 Background

Based on a number of reports and publications, primarily by Takahiko Ogino [39], [40], [41] (in these proceedings), and [42], on the emerging concept of CyberRail, we attempt to show what a formal domain model of CyberRail might look like, and what benefits one might derive from establishing and having such a formal model.

The background for the work reported in this chapter is threefold: (i) Many years of actual formal specification as well as research into how to engineer such formal specifications, by the first author, of domains, including the railway domain [1] [2] [3] [4] [5] [8] [6] [7] [9] — using abstraction and modelling principles and techniques extensively covered in three forthcoming software engineering textbooks [10]. (ii) A term project with four MSc students. And (iii) Some fascination as whether one cold formalise an essence of the novel ideas of CyberRail. We strongly believe that we can capture one crucial essence of CyberRail — such as this paper will show.

The formalisation of CyberRail is expressed in the `RAISE` [17] Specification Language, `RSL` [16]. `RAISE` stands for `R`igorous `A`pproach to `I`ndustrial `S`oftware `E`ngineering. In the current abstract model we especially make use of `RSL`'s parallel process modeling capability. It builds on, ie., borrows from Tony Hoare's algebraic process concept of `C`ommunicating `S`equential `P`rocesses, `CSP` [28].

9.2 A Rough Sketch Formal Model

9.2.1 An Overall CyberRail System

CyberRail consists of an index set of traveller behaviours and one cyber behaviour "running" in parallel. Each traveller behaviour is uniquely identified, p:Tx. Traveller behaviours communicate

[1] Work done together with students: Peter Chiang, Morten S.T. Jacobsen, Jens Kielsgaard Hansen, and Michael P. Madsen, Section of Computer Science and Engineering, Institute of Informatics and Computer Engineering, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, and with Martin Pěnička

with the cyber behaviour. We abstract the communication medium as an indexed set of channels, ct[p], from the cyber behaviour to each individual traveller behaviour, and tc[p], from traveller behaviours to the cyber behaviour. Messages over channels are of respective types, CT and TC. The cyber behaviour starts in an initial state ω_i, and each traveller behaviour, p, starts in some initial state $m\sigma_i(p)$.

```
type
    Tx, $\Sigma$, $\Omega$, CT, TC
    M$\Sigma$ = Tx $\overrightarrow{m}$ $\Sigma$
channel
    {ct[ p ]:CT,tc[ p ]:TC|p:Tx}, cr:CR, rc:RC
value
    m$\sigma_i$:M$\Sigma$, $\omega_i$:$\Omega$

    cyberrail_system: Unit → Unit
    cyberrail_system() ≡ ‖ { traveller(p)(m$\sigma_i$(p)) | p:Tx } ‖ cyber($\omega$)

    cyber: $\Omega$ → in {tc[ p ]|p:Tx},cr out {ct[ p ]|p:Tx},rc Unit
    cyber($\omega$) ≡
        cyber_as_server($\omega$) ⌈⌉ cyber_as_proactive($\omega$) ⌈⌉ cyber_as_co_director($\omega$)

    traveller: p:Tx → $\Sigma$ → in ct[ p ] out tc[ p ]  Unit
    traveller(p)($\sigma$) ≡ active_traveller(p)($\sigma$) ⌈⌉ passive_traveller(p)($\sigma$)
```

The cyber behaviour either acts as a server: Ready to engage in communication input from any traveller behaviour; or the cyber behaviour acts pro–actively: Ready to engage in performing output to one, or some traveller behaviours; or the cyber behaviour acts in consort with the "rest" of the transportation market (including rail infrastructure owners, train operators, etc.), in improving and changing services, and in otherwise responding to unforeseen circumstances of that market.

Similarly any traveller behaviour acts as a client: Ready to engage in performing output to the cyber behaviour; or its acts passively: Ready to accept input from the cyber behaviour.

9.2.2 Travellers

Active Travellers

Active traveller behaviours alternate internally non–deterministically, ie., at their own choice, between *start (travel) planning st_pl, select (among suggested) travel plan(s) se_pl, change (travel) planning ch_pl, begin travel be_tr, board train bo_tr, leave train lv_tr, ignore train ig_tr, cancel travel ca_tr, seeking guidance se_gu, notifying cyber no_cy, entertainment ent, deposit resource de_re* (park car, ...), *claim resource cl_re* (retreive car, ...), *get resource ge_re* (rent a car, ...), *return resource re_re* (return rent-car, ...), *going to restaurant rest* (or other), *change travel ch_tr, interrupt travel in_tr, resume travel re_tr, leave train le_tr, end travel en_tr,* and many other choices. Each of these normally entail an output communication to the cyber behaviour, and for those we can assume immediate response from the cyber behaviour, where applicable.

```
value
    active_traveller: p:Tx → $\Sigma$ → out tc[ p ] in ct[ p ]  Unit
    active_traveller(p)($\sigma$) ≡
        let choice = st_pl ⌈⌉ ac_pl ⌈⌉ ch_pl ⌈⌉ en_tr ⌈⌉ ...  ⌈⌉ le_tr ⌈⌉ te_tr in
        let $\sigma'$ = case choice of
                    st_pl → start_planning(p)($\sigma$),
                    se_pl → select_travel_plan(p)($\sigma$),
                    ch_pl → change_trael_plan(p)($\sigma$),
                    be_tr → begin_travel(p)($\sigma$),
                    bo_tr → board_train(p )($\sigma$),
                    ... → ..,
                    le_tr → leave_train(p)($\sigma$),
                    en_tr → end_travel(p)($\sigma$),
                    ... → ..
                 **end in**
        traveller(p)($\sigma'$) **end end**

    start_planning: p:Tx → $\Sigma$ → **out** tc[p] **in** ct[p] $\Sigma$
    start_planning(p)($\sigma$) ≡
        **let** ($\sigma'$,plan) = magic_plan($\sigma$) **in**
        tc[p]!plan;
        **let** sps = ct[p]? **in** update$\Sigma$((plan,sps))($\sigma'$) **end end**
    ...
    update$\Sigma$: Update → $\Sigma$ → $\Sigma$
type
    Update == mkInPlRes(ip:InitialPlan,ps:Plan-**set**) | ...
```

Passive Travellers

When not engaging actively with the cyber behaviour, traveller behaviours are ready to accept any cyber initated action. The traveller behaviour basically “assimilates” messages received from cyber — and may make use of these in future.

```
value
  passive_traveller: p:Tx → $\Sigma$ → **in** ct[p] **out** tc[p]  **Unit**
  passive_traveller(p)($\sigma$) ≡ **let** res = ct[p]? **in** update$\Sigma$(res)($\sigma$) **end**
```

Active Traveller Actions

The *active_traveller* behaviour performs either of the internally non–deterministically chosen actions: *start_planning, select_travel_plan, change_travel_plan, begin_travel, board_train,* . . . , *leave_train,* or *end_travel.* They make use only of the “sum total state” (σ) that that traveller behaviour “is in”. Each such action basically communicates either of a number of plans (or parts thereof, here simplified into plans). Let us summarise:

```
type
  Plan
  Request = Initial_Plan | Selected_Plan | Change_Plan | Begin_Travel
             | Board_Train | ... | Leave_Train | End_Travel | ...
  Initial_Plan == mkIniPl(pl:Plan)
  Selected_Plan == mkSelPl(pl:Plan)
  Change_Plan == mkChgPl(pl:Plan)
  Begin_Travel == mkBTrav(pl:Plan)
  Board_Train == mkBTrai(pl:Plan)
  ...
  Leave_Train == mkLeTr(pl:Plan)
  End_Travel == mkEnTr(pl:Plan)
value
  ∀ f: p:Tx → $\Sigma$ → **out** tc[p] $\Sigma$
  magic_f: $\Sigma$ → $\Sigma$ × Request

  f(p)($\sigma$) ≡ **let** ($\sigma'$,req) = magic_f($\sigma$) **in** tc[p]!req;$\sigma'$ **end**
```

The magic_functions access and changes the state while otherwise yielding some request. They engage in no events with other than the traveller state. There are the possibility of literally “zillions” such functions, all fitted into the above sketched traveller behaviour.

9.2.3 cyber

cyber as Server

cyber is at any moment ready to engage in actions with any traveller behaviour. cyber is assumed here to respond immediately to “any and such”.

```
value
   cyber_rail_as_server: $\Omega \to$ in {tc[p]|p:Tx} out {ct[p]|p:Tx} Unit
   cyber_rail_as_server($\omega$) $\equiv$
      $\lceil\rceil$ {let req = tc[p]? in cyber(serve_traveller(p,req)($\omega$)) end | p:Tx}

   serve_traveller: p:Tx $\times$ Req $\to \Omega \to$ in {tc[p]|p:Tx} out {ct[p]|p:Tx} $\Omega$
   serve_traveller(p,req)($\omega$) $\equiv$
      case req of
         mkIniPl(pl) $\to$
            let ($\omega'$,pls) = sugg_pls(p,pl)($\omega$) in ct[p]!pls;cyberrail($\omega'$) end
         mkSelPl(pl) $\to$
            let ($\omega'$,res) = res_pl(p,pl)($\omega$) in ct[p]!book;cyberrail($\omega'$) end
         mkChgPl(pl) $\to$
            let ($\omega'$,pl$'$) = chg_pl(p,pl)($\omega$) in ct[p]!pl$'$;cyberrail($\omega'$) end
         mkBTrav(pl) $\to$ ...
         mkBTrai(pl) $\to$ ...
         ...
         mkLeTr(pl) $\to$ ...
         mkEnTr(pl) $\to$ ...
      end
```

cyber as Pro–Active

cyber, on its own volition, may, typically based on its accumulated knowledge of traveller behaviours, engage in sending messages of one kind or another to selected groups of travellers. Section 9.2.3 rough sketch–formalises one of these.

```
type
   CR_act == gu_tr | no_tr | co_tr | wa_tr | ...
value
   cyber_as_proactive: $\Omega \to$ out {ct[p]|p:Tx} Unit
   cyber_as_proactive($\omega$) $\equiv$
      let cho = gu_tr $\lceil\rceil$ no_tr $\lceil\rceil$ co_tr $\lceil\rceil$ wa_tr $\lceil\rceil$ ... in
      let $\omega'$ = case cho of gu_tr $\to$ guide_traveller($\omega$),
                              no_tr $\to$ notify_traveller($\omega$),
                              co_tr $\to$ commercial_to_travellers($\omega$),
                              wa_tr $\to$ warn_travellers($\omega$),
                              ... $\to$ ... end in
      cyber($\omega'$) end end
```

cyber as Co–Director

We do not specify this behaviour. It concerns the actions that cyber takes together with the "rest" of the transportation market. One could mention input from cyber_as_co_director to the train operators as to new traveller preferences, profiles, etc., and output from the rail (ie., net) infrastructure owners or train operators to cyber_as_co_director as to net repairs or train shortages, etc. The decomposition of CyberRail into cyber and the "rest", may — to some — be articificial, namely in countries where there is no effective privatisation and split–up into infrastructyre owners and train operators. But it is a decomposition which is relevant, structurally, in any case.

cyber Server Actions

We sketch:

```
value
    sugg_plans: p:Tx × Plan → Ω → Ω × Plan-set
    res_pl: p:Tx × Plan → Ω → Ω × Plan
    chg_pl: p:Tx × Plan → Ω → Ω × Plan
    ...
```

There are many other such traveller instigated cyber actions.

Pro–Active cyber Actions

We rough sketch just a single of the possible "dozens" of cyber inititated actions versus the travellers.

```
value
    guide_traveller: Ω → out {ct[p]|p:Tx} Ω
    guide_traveller(ω) ≡
        let (ω′,(ps,guide)) = any_guide(ω) in broadcast(ps,guide) ; ω′ end

    any_guide: Ω → Ω × (Tx-set × Guide)

    notify_traveller: Ω → out {ct[p]|p:Tx} Ω
    commercial_to_travellers: Ω → out {ct[p]|p:Tx} Ω
    warn_traveller: Ω → out {ct[p]|p:Tx} Ω
    ...

    broadcast: Tx-set × CT → Unit
    broadcast(ps,msg) ≡
        case ps of {}→skip,{p}∪ ps′→ct[p]!msg;broadcast(ps′,msg) end

type
    CT = Guide | Notification | Commercial | Warning | ...
    Guide == mkGui(...)
    Notification == mkNot(...)
    Commercial == mkCom(...)
    Warning == mkWar(...)
    ...
```

9.3 Conclusion

A formalisation of a crucial aspect of CyberRail has been sketched. Namely the interplay between the rôles of travellers and the central CyberRail system.

Next we need analyse carfully all the action functions with respect to the way in which they use and update the respective states ($\sigma : \Sigma$) of traveller behaviours and the cyber behaviour ($\omega : \Omega$). At the end of such an analysis one can then come up with precise, formal descriptions, including axioms, of what the title of [41] refers to as the *Information Infrastructure.* We look forward to report on that in a near future.

The aim of this work is to provide a foundation, a domain theory, for CyberRail. A set of models from which to "derive", in a systematic way, proposals for computing systems, including software architectures.

9.4 A **CyberRail** Bibliography

1. Takahiko Ogino: "Advanced Railway Transport Systems and ITS", RTRI Report Vol 13, No. 1, January 1999 (in Japanese)
2. Takahiko Ogino, Ryuji Tsuchiya: "CyberRail: A Probable Form of ITS in Japan", RTRI Report Vol 14, No. 7, July 2000 (in Japanese)
3. Takahiko Ogino: "CyberRail: An Enhanced Railway System for Intermodal Transportation", Quarterly Report of RTRI, Vol 42, No. 4, November 2001
4. Takahiko Ogino: "When Train Stations become cyber Stations", Japanese Railway Technology Today, pp 209-219, December 2001
5. Takahiko Ogino: "CyberRail Study Group Activities and Achievements", RTRI Report Vol 16, No.11, November 2002 (in Japanese)
6. Takahiko Ogino, Ryuji Tsuchiya, Akihiko Matsuoka, Koichi Goto: "A Realization of Information and Guidance function of cyber", RTRI Report Vol 17, No. 12, December 2003 (in Japanese)
7. Takahiko Ogino:"CyberRail - In search of IT infrastracture in intermodal transport", JREA, Vol.45., No.1 (2002) (in Japanese)
8. Ryuji Tsuchiya, Koichi Goto, Akihiko Matsuoka, Takahiko Ogino, "CyberRail and its significance in the coming ubiquitous society", Proc. of the World Congress on Railway Research 2003 (2003-9) (in Japanese)
9. Takashi Watanabe, :"Experiment ofCyberRail Passenger guidance using Bluetooth", Preprint of RTRI Annual Lecture Meeting in 2000 (in Japanese)
10. Takashi Watanabe, et. al.: "Personal Navigation System Using Bluetooth", Technical Report of ITS-SIG,IPSJ(2001-ITS-4), p.55 (in Japanese)
11. Ryuji Tsuchiya, Koichi Goto, Akihiko Matsuoka, Takahiko Ogino, "Deriving interoperable traveler support system specification through requirements engineering process", Proc. of the 7th World Multiconference on Systemics, Cybernetics and Informatics (July, 2003)
12. Ryuji Tsuchiya, Takahiko Ogino, Koichi Goto, Akihiko Matsuoka, "Personalized Passenger Information Services and cyber", Technical Report of SIG-IAC, IPSJ (2002-IAC-4), p15 (in Japanese)
13. Akihiko Matsuoka, Ryuji Tsuchiya: "Current Status of cyber SIG",Technical Report of SIG-ITS, IPSJ, p.45 (2002-ITS-11) (in Japanese)
14. Akihiko Matsuoka, Koichi Goto, Ryuji Tsuchiya, Takahiko Ogino: "CyberRail and new passengers information services", IEE Japan, TER-03-22 (2003-6) (in Japanese)
15. Yuji Shinoe, Ryuji Tsuchiya, "Personalized Route Choice Support System for Railway Passengers", Technical Report of SIG-ITS, IPSJ (2001-ITS-6), p.23 (in Japanese)
16. Hiroshi Matsubara, Noriko Fukasawa, Koichi Goto, "Development of Interactive Guidance System for Visually Disabled", Technical Report of SIG-ITS, IPSJ (2001-ITS-6), p.75 (in Japanese)
17. Ryuji Tsuchiya, Takahiko Ogino, Koichi Goto, Akihiko Matsuoka, "Location-sensitive Itinerary-based Passenger Information System", Technical Report of ITS-SIG, IPSJ, p.85 (2003-ITS-6) (in Japanese)
18. Ryuji Tsuchiya, Kiyotaka Seki, Takahiko Ogino, Yasuo Sato: "User services ofCyberRail - toward system architecture of future railway-", Proc. of the World Congress on Railway Research 2001 (2001-11)
19. Takahiko Ogino, Ryuji Tsuchiya, Kiyotaka Seki, Yasuo Sato: "CyberRail - information infrastructure for intermodal passengers-", Proc. of the World Congress on Railway Research 2001 (2001-11)
20. Kiyotaka Seki, Ryuji Tsuchiya, Takahiko Ogino, Yasuo Sato: "Construction of future railway system utilizing information and telecommunication technologies", Proc. of the World Congress on Railway Research 2001 (2001-11)

21. Ryuji Tsuchiya, Akihiko Matsuoka, Takahiko Ogino, Kouich Goto, Toshiro Nakao, Hajime Takebayashi: "Experimental system for CyberRail passenger information providing and guidance", 40th Railway-Cybernetics Symposium (2003-11) (in Japanese)
22. Akihiko Matsuoka, Ryuji Tsuchiya, Takahiko Ogino, Toshio Hirota: "CyberRail System Architecture", 40th Railway-Cybernetics Symposium (2003-11) (in Japanese)
23. Ryuji Tsuchiya, Akihiko Matsuoka, Takahiko Ogino, Kouich Goto, Toshiro Nakao, Hajime Takebayashi: "Location-sensitive Itinerary-based Passenger Information System", Applying to IEE Journal (in Japanese)

Part VI

Closing

10 Conclusion

Dines Bjørner

- In addition to the chapters of this compendium, we can refer to published papers which cover additional aspects of the railway domain:
  1. J.U.Skakkebæk, A.P.Ravn, H.Rischel, and Zhou Chaochen. *Specification of embedded, real-time systems.* Proceedings of 1992 Euromicro Workshop on Real-Time Systems, pages 116–121. IEEE Computer Society Press, 1992.
  2. C.W. George. *A Theory of Distributing Train Rescheduling.* In FME'96: Industrial Benefits and Advances in Formal Methods, proceedings, LNCS 1051,
  3. Anne Haxthausen and Jan Peleska, *Formal Development and Verification of a Distributed Railway Control System,* IEEE Transaction on Software Engineering, 26(8), 687–701, 2000
  4. Morten Peter Lindegaard and Peter Viuf and Anne Haxthausen, *Modelling Railway Interlocking Systems,* Eds.: E. Schnieder and U. Becker, Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15, 2000, Braunschweig, Germany, 211-217, 2000
  5. A. E. Haxthausen and J. Peleska, *A Domain Specific Language for Railway Control Systems,* Sixth Biennial World Conference on Integrated Design and Process Technology, (IDPT2002), Pasadena, California, Society for Design and Process Science, P.O.Box 1299, Grand View, Texas 76050-1299, USA, June 23-28, 2002
  6. A. Haxthausen and T. Gjaldbæk, *Modelling and Verification of Interlocking Systems for Railway Lines,* 10th IFAC Symposium on Control in Transportation Systems, Tokyo, Japan", August 4–6, 2003
- And we can refer to the papers which are to be/were presented August 26, 2004, as Topic 11: TRain: The Railway Domain – A Grand Challenge for Computing Science: Towards a Domain Theory for Transportation, during the IFIP World Computer Congress, 2004, at Toulouse:
  1. Dines Bjrner: TRain: *The Railway Domain — A "Grand Challenge" for Computing Science and Transportation Engineering.*
     Sets the stage for the TRain effort.
  2. Denis Sabatier: *Reusing Formal Models: Domain capitalization via formalization.*
     A careful discussion is presented of the benefits of developing, studying and using formal models. After a careful analysis of two kinds of uses, a discussion follows of how to reuse and (thus) capitalize on formal models.
  3. Alistair A. McEwan and J.C.P.Woodcock: *A calculated, refinement-based approach to building fault-tolerance into a railway signaling device.*
     Exemplifies the concept of integrating formal techniques in the provably correct development of software for a railway real–time embedded system.
  4. Martin Penicka: *From Railway Resource Planning to Train Operation.*
     Illustrates, in survey fashion, a number of railway models: From nets, via scheduling and allocation of resources (net development, time tables, rolling stock deployment, staff rostering, rail car maintenance planning, to station interlocking, line direction monitoring & control, automatic line signaling.
  5. Wolfgang Reif: *Integrated Formal Methods for Safety Analysis of Train Systems.*
     An approach is shown in which correct functioning, analysis of failures and their effects, and quantitative analyses of the risks of systems and subsystems, all based on formal techniques, are applied, in a coherent fashion, to a railway example.
  6. Theo C. Giras and Zhongli Lin: *Stochastic Train Domain Theory Framework.*
     The axiomatic safety–critical assessment process (ASCAP) is briefly analysed as a stochastic, Monte Carlo simulation model. The rail line taxonomy is thus characterised as a stochastic domain that provides for either a design–for–safety, or a risk–assessment framework — and these are seen as dual. The need for formal validation, verification and certification is presented.

7. Takahiko Ogino: *CyberRail: Information Infrastructure for New Intermodal Transport Business Model.* Outlines dramatic new paradigms for passenger transport.
   Chapter 9 is based on the above paper.

- Together: With the above–referenced papers, with the papers referenced in those (above referenced) papers, and with the chaptyers of this compendium, we can claim that there exists a beginning of what could evolve into contributions to a Domain Theory of Railways.
- Much remains to be done. We mention but a few, and obvious:
  - **Other Railway Domain Facets:** The papers presented at IFIP WCC'2004, and listed above, as well as an additional presentation, not yet documented, by Eckehard Schnieder, points to several further aspects of the railway domain — some in need of precise description, some in need of further research, and some already yilding quite exciting results.
    We have, independently, worked out material for some, and we would like to see (some, perhaps, mundane) research & development of further railway facets:
    - *Net Planning:* Given a map of a region (a metropolitan area, a province, a country, a sub–continent, etc.), and given seasonal statistics (obtained by inquiry or otherwise), say hour–by–hour, of how many passengers would like to travel from some point to some other point, describe the planning of optimal nets to serve such transportation. From, and intertwined with that:
    - *Time Tabling:* Develop, probably jointly with *Net Planning* describe the planning of optimal time tables to serve the traffic implied by the statistics. From, and intertwined with that:
    - *Train Composition & Decomposition:* Develop, probably jointly with *Net Planning* and *Time Tabling,* describe the planning of optimal ways of composing and decomposing trains from and into carriages.
    - $\mathcal{E}$tecetera, etcetera, etc. !
  - **Integrating Formal Techniques:** As was evident from Chaps. 6, 7, and 8, combining one form of formal specification (viz.: RSL) with other forms (viz.: Petri Nets, Live Sequence Charts, Statecharts, Duration Calculus, etc.), is "a must". Integrating Formal Techniques is currently the focus of many research groups worldwide.
  - **Models of Agent Behaviours:** So much (of what is going on) in the railway domain is governed by human behaviours. Studies into modelling human behaviours, agents, their speech acts, their knowledge & belief, their promise & commitment, is needed.
- $\mathcal{E}$tecetera, etcetera, etc. !

[illegible]

∅

August 5, 2004

Part VII

Appendices

A

An RSL Primer

Dines Bjørner

Contents

This is an ultra–short introduction to The RAISE Specification Language.

A.1 Types

The reader is kindly asked to study first the decomposition of this section into its subparts and sub-subparts.

A.1.1 Type Expressions

RSL has a number of *built–in* types.

There are the Booleans, integers, natural numbers, reals, characters, and texts.

From these one can form type expressions: Finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

type
[1] **Bool**
[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

[7] A-**set**
[8] A-**infset**
[9] $A \times B \times \ldots \times C$
[10] A^*
[11] A^ω
[12] $A \underset{m}{\rightarrow} B$
[13] $A \rightarrow B$
[14] $A \overset{\sim}{\rightarrow} B$
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

(save the [i] line numbers) are generic type expressions:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., -2, -1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values "a", "b", ...
6. The text type of character string values "aa", "aaa", ..., "abc", ...
7. The set type of finite set values, see below.
8. The set type of infinite set values.
9. The Cartesian type of Cartesian values, see below.
10. The list type of finite list values, see below.
11. The list type of infinite list values.
12. The map type of finite map values, see below.
13. The function type of total function values, see below.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian $B \times C \times \ldots \times D$, in which case it is identical to type expression kind 9,
    - or not to be the name of a built–in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, eg: ($A \underset{m}{\rightarrow} B$), or ($A^*$)-**set**, or (A-**set**)list, or (A|B) $\underset{m}{\rightarrow}$ (C|D|($E \underset{m}{\rightarrow} F$)), etc.
16. The (postulated disjoint) union of types A, B, . . . , and C.
17. The record type of mk_id–named record values mk_id(av,...,bv), where av, . . . , and bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18. The record type of unnamed record values (av,...,bv), where av, . . . , and bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

A.1.2 Type Definitions

Concrete Types:

Types can be concrete in which case the structure of the type is specified by type expressions:

```
type
   A = Type_expr
```

Some schematic type definitions are:

```
[1] Type_name = Type_expr /* without |s or sub−types */
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
         mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
         ... |
         mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • 𝒫(v) |}
```

where a form of [2–3] is provided by combining the types:

```
   Type_name = A | B | ... | Z
   A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
   B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
   ...
   Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which has type B and which satisfy the predicate $\mathcal{P}$, constitute the sub–type A:

```
type
   A = {| b:B • 𝒫(b) |}
```

Sorts (Abstract Types)

Types can be sorts (abstract) in which case their structure is not specified:

```
type
   A, B, ..., C
```

A.2 The RSL Predicate Calculus

A.2.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values. Then:

```
   false, true
   a, b, ..., c
   ∼a, a∧b, a∨b, a⇒b, a=b, a≠b
```

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, and = are Boolean connectives (i.e., operators). They are read: *not*, *and*, *or*, *if-then* (or *implies*), *equal* and *not-equal*.

A.2.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non–Boolean values, and let i, j, . . ., k designate number values, then:

```
false, true
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i>j, ...
```

are simple predicate expressions.

A.2.3 Quantified Expressions

Let X, Y, . . ., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which z, y, and z are free. Then:

```
∀ x:X • 𝒫(x)
∃ y:Y • 𝒬(y)
∃ ! z:Z • ℛ(z)
```

are quantified expressions — also being predicate expressions. They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and: there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

A.3 Concrete RSL Types

A.3.1 Set Enumerations

Let the below as denote values of type A, then the below designate simple set enumerations:

```
{{}, {a}, {a₁,a₂,...,aₘ}, ...} ∈ A-set
{{}, {a}, {a₁,a₂,...,aₘ}, ..., {a₁,a₂,...}} ∈ A-infset
```

The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is highly abstract in the sense that it does not do so by following a concrete algorithm.

```
type
   A, B
   P = A → Bool
   Q = A ~→ B
value
   comprehend: A-infset × P × Q → B-infset
   comprehend(s,𝒫,𝒬) ≡ { 𝒬(a) | a:A • a ∈ s ∧ 𝒫(a) }
```

A.3.2 Cartesian Enumerations

Let e range over values of Cartesian types involving A, B, . . ., C (allowing indexing for solving ambiguity), then the below expressions are simple Cartesian enumerations:

```
type
   A, B, ..., C
   A × B × ... × C
value
   ... (e1,e2,...,en) ...
```

A.3.3 List Enumerations

Let a range over values of type A (allowing indexing for solving ambiguity), then the below expressions are simple list enumerations:

$\{\langle\rangle, \langle a\rangle, ..., \langle a1,a2,...,am\rangle, ...\} \in A^*$
$\{\langle\rangle, \langle a\rangle, ..., \langle a1,a2,...,am\rangle, ..., \langle a1,a2,...,am,... \rangle, ...\} \in A^\omega$

$\langle$ ei .. ej $\rangle$

The last line above assumes e_i and e_j to be integer valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former then the list is empty.

The last line below expresses list comprehension.

type
  A, B, P = A $\to$ **Bool**, Q = A $\xrightarrow{\sim}$ B
value
  comprehend: $A^\omega \times P \times Q \xrightarrow{\sim} B^\omega$
  comprehend(lst,$\mathcal{P}$,$\mathcal{Q}$) $\equiv$
    $\langle$ $\mathcal{Q}$(lst(i)) | i **in** $\langle$1..**len** lst$\rangle$ • $\mathcal{P}$(lst(i)) $\rangle$

A.3.4 Map Enumerations

Let a and b range over values of type A and B, respectively (allowing indexing for solving ambiguity); then the below expressions are simple map enumerations:

type
  A, B
  M = A $\overrightarrow{m}$ B
value
  a,a1,a2,...,a3:A, b,b1,b2,...,b3:B

  [ ], [a$\mapsto$b], ..., [a1$\mapsto$b1,a2$\mapsto$b2,...,a3$\mapsto$b3] $\forall \in$ M

The last line below expresses map comprehension:

type
  A, B, C, D
  M = A $\overrightarrow{m}$ B
  F = A $\xrightarrow{\sim}$ C
  G = B $\xrightarrow{\sim}$ D
  P = A $\to$ **Bool**
value
  comprehend: M$\times$F$\times$G$\times$P $\to$ (C $\overrightarrow{m}$ D)
  comprehend(m,$\mathcal{F}$,$\mathcal{G}$,$\mathcal{P}$) $\equiv$
    [$\mathcal{F}$(a) $\mapsto$ $\mathcal{G}$(m(a)) | a:A • a $\in$ **dom** m $\wedge$ $\mathcal{P}$(a)]

A.3.5 Set Operations

value
  $\in$: A $\times$ A-**infset** $\to$ **Bool**
  $\notin$: A $\times$ A-**infset** $\to$ **Bool**
  $\cup$: A-**infset** $\times$ A-**infset** $\to$ A-**infset**
  $\cup$: (A-**infset**)-**infset** $\to$ A-**infset**
  $\cap$: A-**infset** $\times$ A-**infset** $\to$ A-**infset**
  $\cap$: (A-**infset**)-**infset** $\to$ A-**infset**

$\backslash$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
$\subset$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
$\subseteq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
$=$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
$\neq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
card: A-**infset** $\overset{\sim}{\rightarrow}$ **Nat**

examples
a $\in$ {a,b,c}
a $\notin$ {}, a $\notin$ {b,c}
{a,b,c} $\cup$ {a,b,d,e} = {a,b,c,d,e}
$\cup${{a},{a,b},{a,d}} = {a,b,d}
{a,b,c} $\cap$ {c,d,e} = {c}
$\cap${{a},{a,b},{a,d}} = {a}
{a,b,c} $\backslash$ {c,d} = {a,b}
{a,b} $\subset$ {a,b,c}
{a,b,c} $\subseteq$ {a,b,c}
{a,b,c} = {a,b,c}
{a,b,c} $\neq$ {a,b}
card {} = 0, **card** {a,b,c} = 3

- $\in$ The membership operator expresses that an element is member of a set.
- $\notin$: The non-membership operator expresses that an element is not member of a set.
- $\cup$ The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets
- $\cap$ The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
- $\backslash$ The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
- $\subseteq$ The proper subset operator expresses that all members of the left operand set are also in the right operand set.
- $\subset$ The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
- $=$ The equal operator expresses that the two operand sets are identical.
- $\neq$ The non–equal operator expresses that the two operand sets are *not* identical.
- **card** The cardinality operator gives the number of elements in a (finite) set.

The operations can be defined as follows:

value
s′ $\cup$ s″ $\equiv$ { a | a:A • a $\in$ s′ $\vee$ a $\in$ s″ }
s′ $\cap$ s″ $\equiv$ { a | a:A • a $\in$ s′ $\wedge$ a $\in$ s″ }
s′ $\backslash$ s″ $\equiv$ { a | a:A • a $\in$ s′ $\wedge$ a $\notin$ s″ }
s′ $\subseteq$ s″ $\equiv$ $\forall$ a:A • a $\in$ s′ $\Rightarrow$ a $\in$ s″
s′ $\subset$ s″ $\equiv$ s′ $\subseteq$ s″ $\wedge$ $\exists$ a:A • a $\in$ s″ $\wedge$ a $\notin$ s′
s′ = s″ $\equiv$ $\forall$ a:A • a $\in$ s′ $\equiv$ a $\in$ s″ $\equiv$ s$\subseteq$s′ $\wedge$ s′$\subseteq$s
s′ $\neq$ s″ $\equiv$ s′ $\cap$ s″ $\neq$ {}
card s $\equiv$
    if s = {} **then** 0 **else**
    let a:A • a $\in$ s **in** 1 + **card** (s $\backslash$ {a}) **end end**
    pre s /∗ is a finite set ∗/
card s $\equiv$ **chaos** /∗ tests for infinity of s ∗/

A.3.6 Cartesian Operations

type
  A, B, C
  g0: G0 = A $\times$ B $\times$ C
  g1: G1 = (A $\times$ B $\times$ C)
  g2: G2 = (A $\times$ B) $\times$ C
  g3: G3 = A $\times$ (B $\times$ C)

value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decomposition expressions
  let (a1,b1,c1) = g0,
    (a1′,b1′,c1′) = g1 **in** .. **end**
  let ((a2,b2),c2) = g2 **in** .. **end**
  let (a3,(b3,c3)) = g3 **in** .. **end**

A.3.7 List Operations

value
  hd: $\mathsf{A}^\omega \xrightarrow{\sim} \mathsf{A}$
  tl: $\mathsf{A}^\omega \xrightarrow{\sim} \mathsf{A}^\omega$
  len: $\mathsf{A}^\omega \xrightarrow{\sim}$ **Nat**
  inds: $\mathsf{A}^\omega \rightarrow$ **Nat-infset**
  elems: $\mathsf{A}^\omega \rightarrow$ A-**infset**
  .(.): $\mathsf{A}^\omega \times$ **Nat** $\xrightarrow{\sim} \mathsf{A}$
  $\widehat{\ }$: $\mathsf{A}^* \times \mathsf{A}^\omega \rightarrow \mathsf{A}^\omega$
  =: $\mathsf{A}^\omega \times \mathsf{A}^\omega \rightarrow$ **Bool**
  $\neq$: $\mathsf{A}^\omega \times \mathsf{A}^\omega \rightarrow$ **Bool**

examples
  hd⟨a1,a2,...,am⟩=a1
  tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
  len⟨a1,a2,...,am⟩=m
  inds⟨a1,a2,...,am⟩={1,2,...,m}
  elems⟨a1,a2,...,am⟩={a1,a2,...,am}
  ⟨a1,a2,...,am⟩(i)=ai
  ⟨a,b,c⟩$\widehat{\ }$⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
  ⟨a,b,c⟩=⟨a,b,c⟩
  ⟨a,b,c⟩ $\neq$ ⟨a,b,d⟩

- **hd** Head gives the first element in a non–empty list.
- **tl** Tail gives the remaining list of a non–empty list when Head is removed.
- **len** Length gives the number of elements in a finite list.
- **inds** Indices gives the set of indices from 1 to the length of a non–empty list. For empty lists, this set is the empty set as well.
- **elems** Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$ Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the i'th element of the list.
- $\widehat{\ }$ Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.

- = The equal operator expresses that the two operand lists are identical.
- ≠ The non–equal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

```
value
   is_finite_list: $A^\omega \to$ Bool

   len q ≡
      case is_finite_list(q) of
         true → if q = ⟨⟩ then 0 else 1 + len tl q end,
         false → chaos end

   inds q ≡
      case is_finite_list(q) of
         true → { i | i:Nat • $1 \le i \le$ len q },
         false → { i | i:Nat • i≠0 } end

   elems q ≡ { q(i) | i:Nat • i ∈ inds q }

   q(i) ≡
      if i=1
         then
            if q≠⟨⟩
               then let a:A,q′:Q • q=⟨a⟩⁀q′ in a end
               else chaos end
         else q(i−1) end

   fq ⁀ iq ≡
         ⟨ if $1 \le i \le$ len fq then fq(i) else iq(i − len fq) end
          | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
      pre is_finite_list(fq)

   iq′ = iq″ ≡
      inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

   iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

A.3.8 Map Operations

```
value
   m(a): M → A $\overset{\sim}{\to}$ B, m(a) = b

   dom: M → A-infset [domain of map]
        dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

   rng: M → B-infset [range of map]
        rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

   †: M × M → M [override extension]
        [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]

   ∪: M × M → M [merge ∪]
        [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]


   \: M × A-infset → M [restriction by]
        [a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]
```

```
/: M × A-infset → M [restriction to]
    [a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]

=,≠: M × M → Bool

°: (A →m B) × (B →m C) → (A →m C) [composition]
    [a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]
```

- $m(a)$ Application gives the element of which a maps to in the map m
- **dom** Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- † Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map,
- ∪ Merge. When applied to two operand maps, it gives it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set
- / Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- = The equal operator expresses that the two operand maps are identical.
- ≠ The non–equal operator expresses that the two operand maps are *not* identical.
- ° Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a, in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

The map operations can also be defined as follows:

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
     [ a↦b | a:A,b:B •
        a ∈ dom m1 \ dom m2 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
               a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]


  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
     dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
     [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
     pre rng m ⊆ dom n
```

A.4 Lambda–Calculus + Functions

RSL supports function expressions for λ–abstraction.

A.4.1 The Lambda–Calculus Syntax

```
type /* A BNF Syntax: */
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /* variables, i.e. identifiers */
    ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
value /* Examples */
    ⟨L⟩: e, f, a, ...
    ⟨V⟩: x, ...
    ⟨F⟩: λ x • e, ...
    ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

A.4.2 Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

A.4.3 Substitution

In RSL, the following rules for substitution apply:

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,
  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λ x•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λ y• **subst**([N/x]P),
  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),
  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

A.4.4 α–Renaming and β–Reduction

- α–renaming: λx•M
  If x y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M): We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β–reduction: (λx•M)(N)
  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result.
  (λx•M)(N) ≡ **subst**([N/x]M)

A.4.5 Function Signatures

For some functions, we want to abstract from the function body:

```
value
  obs_Pos_Aircraft: Aircraft → Pos,
  move: Aircraft × Dir → Aircraft,
```

A.4.6 Function Definitions

Functions — with body — can be defined explicitly:

value
  f: A $\times$ B $\times$ C $\to$ D
  f(a,b,c) $\equiv$ Value_Expr

  g: B-**infset** $\times$ (D $\xrightarrow[m]{}$ C-**set**) $\overset{\sim}{\to}$ A*
  g(bs,dm) $\equiv$ Value_Expr
  pre $\mathcal{P}$(dm)

or implicitly:

value
  f: A $\times$ B $\times$ C $\to$ D
  f(a,b,c) **as** d
  post $\mathcal{P}_1$(d)

  g: B-**infset** $\times$ (D $\xrightarrow[m]{}$ C-**set**) $\overset{\sim}{\to}$ A*
  g(bs,dm) **as** al
  pre $\mathcal{P}_2$(dm)
  post $\mathcal{P}_3$(al)

The symbol $\overset{\sim}{\to}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by pre–conditions stating the criteria for arguments to be meaningful to the function.

A.5 Other Applicative Expressions

A.5.1 Let Expressions

Simple (i.e., non–recursive) **let** expressions:

  let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

  (λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

Recursive **let** expressions are written as:

  let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

  let f = **Y**F **in** B(f,a) **end**

where:

  F $\equiv$ λg•λa•(E(g)) and YF = F(YF)

Predicative **let** expressions:

  let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

Patterns and *Wild Cards* can be used:

```
let {a} ∪ s = set in ... end
let {a,_} ∪ s = set in ... end

let (a,b,...,c) = cart in ... end
let (a,_,...,c) = cart in ... end

let ⟨a⟩^ℓ = list in ... end
let ⟨a,_,b⟩^ℓ = list in ... end

let [a↦b] ∪ m = map in ... end
let [a↦b,_] ∪ m = map in ... end
```

A.5.2 Conditionals

Various kinds of conditional expressions are offered by RSL:

```
if b_expr then c_expr else a_expr end

if b_expr then c_expr end ≡ /* same as: */
    if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_exprt_n then c_expr_n end

case expr of
    choice_pattern_1 → expr_1,
    choice_pattern_2 → expr_2,
    ...
    choice_pattern_n_or_wild_card → expr_n
end
```

A.5.3 Operator/Operand Expressions

```
⟨Expr⟩ ::=
          ⟨Prefix_Op⟩ ⟨Expr⟩
        | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
        | ⟨Expr⟩ ⟨Suffix_Op⟩
        | ...
⟨Prefix_Op⟩ ::=
        − | ∼ | ∪ | ∩ | card | len | inds | elems | hd | tl | dom | rng
⟨Infix_Op⟩ ::=
          = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
        | ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ^ | † | °
⟨Suffix_Op⟩ ::= !
```

A.6 Imperative Constructs

Often, following the RAISE method, software development starts with highly abstract–applicative which, through stages of refinements, are turned into concrete and imperative. Imperative constructs are thus inevitable in RSL.

A.6.1 Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

A.6.2 Statement Sequences and skip

Sequencing is done using the ';' operator. **skip** is the empty statement having no value or side–effect.

2. **skip**
3. stm_1;stm_2;...;stm_n

A.6.3 Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

A.6.4 Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

A.6.5 Iterative Sequencing

8. **for** b **in** list_expr • P(b) **do** S(b) **end**

A.7 Process Constructs

A.7.1 Process Channels

Let A, B and KIdx stand for a type of (channel) messages, respectively; then:

channel c:A
channel { k[i]:B • i:KIdx }

declare a channel, c, and a set of channels, k[i], able of communicating values of the designated types.

A.7.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels.

Let P() and Q(i) stand for process expressions, then:

P() ∥ Q(i)	Parallel composition
P() ⌷ Q(i)	Non−deterministic External Choice (either/or)
P() ⌈⌉ Q(i)	Non−deterministic Internal Choice (either/or)
P() ╫ Q()	Interlock Parallel composition

express the parallel (∥) of two processes, the non–deterministic choice between two processes: Either external (⌷) or internal (⌈⌉). The interlock (╫) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

A.7.3 Input/Output Events

Let c, k[i] and e designate a channels of type A and B, respectively; then:

c ?, k[i] ?	Input
c ! e, k[i] ! e	Output

expresses the willingness of a process to engage in an event that "reads" an input, and respectively "writes" an output.

A.7.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature via which channels they wish to engage in input and output events.

value
  P: **Unit** $\rightarrow$ **in** c **out** k[i] **Unit**
  Q: i:KIdx $\rightarrow$ **out** c **in** k[i] **Unit**

  P() $\equiv$ ... c ? ... k[i] ! e ...
  Q(i) $\equiv$ ... k[i] ? ... c ! e ...

The process function definitions (i.e., their bodies) express possible events.

A.8 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects; as often done in RSL. an RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

type
  ...
variable
  ...
channel
  ...
value
  ...
axiom
  ...

References

1. Dines Bjrner, Dong Yu Lin, and S. Prehn. Domain Analyses: A Case Study of Station Management. In KICS'94: *Kunming International CASE Symposium, Yunnan Province, P.R.of China*. Software Engineering Association of Japan, 16–20 November 1994.
2. Dines Bjrner, C.W. George, and S. Prehn. *Scheduling and Rescheduling of Trains*, chapter 8, pages 157–184. *Industrial Strength Formal Methods in Practice,* Eds.: Michael G. Hinchey and Jonathan P. Bowen. FACIT, Springer–Verlag, London, England, 1999.
3. Dines Bjrner, Sren Prehn, and Chris W. George. Formal Models of Railway Systems: Domains. FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, France.
4. Dines Bjrner, Sren Prehn, and Chris W. George. Formal Models of Railway Systems: Requirements. FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, France.
5. Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk.
6. Dines Bjrner. New Results and Trends in Formal Techniques for the Development of Software for Transportation Systems. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems.* 2003. Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.
7. Dines Bjrner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Oxford, UK, August 4-6 2003. Elsevier Science Ltd.
8. Dines Bjrner, Chris W. George, and Sren Prehn. Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In *Integrated Design and Process Technology. Editors: Bernd Kraemer and John C. Petterson*, 24–28 June 2002. Society for Design and Process Science.
9. Dines Bjørner, Chris George, Anne E. Haxthausen, Christian Krog Madsen, Steffen Holmslykke, and Martin Pěnička. "UML"–ising Formal Techniques. In *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering*. Institut für Softwaretechnik und Theoretische Informatik, Sekr. FR 6-1, Techn.Univ. of Berlin, Franklinstrasse 28/29, D–10587 Berlin, Germany, 28 March 2004, ETAPS, Barcelona, Spain. To be published in INT–2004 Proceedings, Springer–Verlag.
10. Dines Bjørner. *Software Engineering*, volume Vol. 1: Abstraction and Modelling, Vol. 2: Specification of Systems and Languages, Vol. 3: Domains, Requirements and Software Design, Vol. 4: Management. Springer–Verlag, 2005. Volumes 1–3 to be published early 2005; Volume 4 planned.
11. Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001. Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems* (FMOODS'99), Kluwer, 1999, pp. 293–312While message sequence charts (MSCs) are widely used in industry to document the interworking of processes or objects, they are expressively quite weak, being based on the modest semantic notion of a partial ordering of events as defined, e.g., in the CCITT standard. A highly expressive and rigorously defined MSC language is a must for serious, semantically meaningful tool support for use-cases and scenarios. It is also a prerequisite to addressing what we regard as one of the central problems in behavioral specification of systems: relating scenario-based inter-object specification to state-machine intra-object specification. This paper proposes

an extension of MSCs, which we call live sequence charts (or LSCs), since one of our main extensions deals with specifying "liveness", i.e., things that must occur. In fact, LSCs allow the distinction between possible and necessary behavior both globally, on the level of an entire chart and locally, when specifying events, conditions and progress over time within a chart. We also deal with subcharts, synchronization, branching and iteration. .

12. Caprara, A., M. Fischetti, P. Toth, D. Vigo and P.L. Guida, "Algorithms for Railway Crew Management". Publication in Mathematical Programming 79 (1997) 125-141.
13. Caprara, A., M. Fischetti, P.L. Guida, P. Toth and D. Vigo. *Solution of Large-Scale railway Crew Planning Problems: the Italian Experience,* in N.H.M. Wilson (ed.) Computer-Aided Transit Scheduling, Lecture Notes in Economics and Mathematical Systems 471, Springer-Verlag (1999) 1-18.
14. Caprara, A., M. Monaci and P. Toth. *A Global Method for Crew Planning in Railway Applications,* in J. Daduna, S. Voss (eds.) Computer-Aided Transit Scheduling, Lecture Notes in Economics and Mathematical Systems 505, Springer-Verlag (2001) 17-36.
15. Ernst, A., H. Jiang, M. Krishnamoorthy, H. Nott and D. Sier. *Rail Crew Scheduling and Rostering: Optimisation Algorithms,* CSIRO Mathematical and Information Sciences, Australia.
16. Chris George, Peter Haff, Klaus Havelund, Anne Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.
17. Chris George, Anne Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
18. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
19. David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.
20. David Harel. *The Science of Computing — Exploring the Nature and Power of Algorithms.* Addison-Wesley, April 1989.
21. David Harel and Eran Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
22. David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990.
23. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
24. David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.
25. David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.
26. David Harel and Michael Politi. *Modelling Reactive Systems with Statecharts: The Statemate Approach*. McGraw Hill, October 8 1998. 258 pages.
27. Anne Haxthausen and Xia Yong. Linking DC together with TRSL. In *Proceedings of 2nd International Conference on Integrated Formal Methods (IFM'2000), Schloss Dagstuhl, Germany, November 2000*, number 1945 in Lecture Notes in Computer Science, pages 25–44. Springer-Verlag, 2000.
28. C.A.R. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
29. Ian Horrocks and Peter F. Patel-Schneider. A proposal for an owl rules language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 723–731, New York, USA, May 2004. ACM. http://www.cs.man.ac.uk/˜horrocks/DAML/Rules/.
30. Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.add-wes.co.uk, 1995. ISBN 0-201-87712-0; xiv + 228 pages.
31. Kurt Jensen. *Coloured Petri Nets*, volume 1: Basic Concepts (234 pages + xii), Vol. 2: Analysis Methods (174 pages + x), Vol. 3: Practical Use (265 pages + xi) of *EATCS Monographs in Theoretical Computer Science*. Springer–Verlag, Heidelberg, 1985, revised and corrected second version: 1997.
32. Karras, P. and Bjørner, D. (2002). *Train composition and decomposition: From passenger statistics to schedules.* Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark.
33. Kroon, L. and Fischetti, M (2000). *Crew Scheduling for Netherlands Railways Destination: Customer.* ERIM Report Series: Research In Management, Netherlands.
34. Kroon, L. (2001). *Models for rolling stock planning.* Research report, Univ. of Utrecht, Netherlands.

35. Lentink, R., M. Odijk and E.Rijn. *Crew Rostering for the High Speed Train,* ERIM Report Series Research In Management, Netherlands, February 2002.
36. Christian Krog Madsen. Integration of Specification Techniques. Msc thesis report, Institute of Informatics and Mathematical Modelling, Technical University of Denmark. Bldg.322, DK–2800 Kgs.Lyngby, Denmark, November 30, 2003. (Statecharts and Live Sequence Charts: Their Glueing and Relation to the RAISE Specification Language.) The goal of the project is to create an integrated formal method for software engineering combining the strong sides of formal specification languages with those of graphical notations. Formal specification languages provide precise descriptions of domains and requirements and allow properties of such descriptions to be verified by formal proofs. Graphical notations are intuitively understandable and typically offer a hierarchical view of a system that allows major system parts to be easily identified. As a preparation for the thesis I wrote a report, [37] in which the syntax and well–formedness conditions of Message Sequence Charts, Statecharts and Petri Nets are formalised in RSL. The report also presents three examples where a system is modelled in RSL and one of the graphical notations in parallel.
37. Christian Krog Madsen. Study of Graphical and Temporal Specification Techniques. Pre–msc thesis report, Institute of Informatics and Mathematical Modelling, Technical University of Denmark. Bldg.322, DK–2800 Kgs.Lyngby, Denmark, 2 June, 2003. The syntax and well–formedness conditions of Message Sequence Charts, Statecharts and Petri Nets are formalised in RSL. The report also presents three examples where a system is modelled in RSL and one of the graphical notations in parallel. See follow–on report[36].
38. Maróti, G. (2001). *Maintenance Routing.* Research report, CWI, Amsterdam and NS Reizigers, Utrecht, Netherlands. The EU IST Research Training Network AMORE: Algorithmic Models for Optimising Railways in Europe: `www.inf.uni-konstanz.de/algo/amore/.` Contract no. HPRN-CT-1999-00104, Proposal no. RTN1-1999-00446
39. Takahiko Ogino. Aiming for Passenger Interoperability. Technical report, Railway Technical Research Institute, Transport Information Technology Division, Railway Technical Research Institute, 2-8-38 Hikari-cho, Kokubunji-shi, Tokyo, 185-8540 Japan, 2003.
40. Takahiko Ogino. CyberRail: For Urban Mobility Tomorrow. Technical report, Railway Technical Research Institute, Transport Information Technology Division, Railway Technical Research Institute, 2-8-38 Hikari-cho, Kokubunji-shi, Tokyo, 185-8540 Japan, 2004.
41. Takahiko Ogino. CyberRail: Information Infrastructure for New Intermodal Transport Business Model. In *Topical Days @ IFIP World Computer Congress 2004*, IFIP Series. IFIP, Kluwer Academic Press, August 2004.
42. Takahiko Ogino, Koivhi Goto, Ryuji Tsuchiya, Kiyotaka Seki, and Akihiko Matsuoka. CyberRail and its significance in the coming ubiquitous society. In , 2004.
43. Pěnička, M., Strupchanska, A. K., and Bjørner, D. (2003). *Train maintenance routing.* In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.
44. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, May 1985.
45. Wolfgang Reisig. *A Primer in Petri Net Design*. Springer-Verlag, 1992.
46. Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, December 1998. xi + 302 pages.
47. Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, December 1998. 400 pages, Amazon price:US $ 42.95.
48. Wolfgang Reisig. The Expressive Power of Abstract–Sate Machines. *Computing and Informatics*, 22(1–2), 2003.
49. Strupchanska, A. K., Pěnička, M., and Bjørner, D. (2003). *Railway staff rostering.* In FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems. L'Harmattan Hongrie. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.
50. Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors. *FM'99 — Formal Methods*, volume 1709 of *LNCS: Lecture notes in computer science*. Springer–Verlag, 1999. This is volume II of a two volume proceedings from the first World Congress on Formal Methods in the Development of Computing Systems. Organised jointly by FME (Formal Methods Europe) and ONERA (The French Government's Space Research Centre), Toulouse, France, Sept. 20–24, 1999.
51. Xia Yong and Chris W. George. An Operational Semantics for Timed RAISE. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, pages 1008–1027. FME, Springer–Verlag, 1999. Cf. [50].
52. Zhou Chaochen and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems.* Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.

53. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.
54. Zhou Chaochen. Duration Calculi: An Overview. Research Report 10, UNU/IIST, P.O.Box 3058, Macau, June 1993. Published in: *Formal Methods in Programming and Their Applications*, Conference Proceedings, June 28 – July 2, 1993, Novosibirsk, Russia; (Eds.: D. Bjørner, M. Broy and I. Pottosin) LNCS 736, Springer-Verlag, 1993, pp 36–59.
55. Zhou Chaochen, Anders P. Ravn, and Michael R. Hansen. An Extended Duration Calculus for Real-time Systems. Research Report 9, UNU/IIST, P.O.Box 3058, Macau, January 1993. Published in: *Hybrid Systems*, LNCS 736, 1993.
56. Zhou Chaochen and Li Xiaoshan. A Mean Value Duration Calculus. Research Report 5, UNU/IIST, P.O.Box 3058, Macau, March 1993. Published as Chapter 25 in *A Classical Mind*, Festschrift for C.A.R. Hoare, Prentice-Hall International, 1994, pp 432–451.