A Railway System
Coordination '97
Case Study Workshop Example

Dines Bjørner, Chris W. George, Bo Stig Hansen, Hans Laustrup and Søren Prehn — and with contributions from Jakob Braad, Jon Fuglsang, Karin S. Mogensen and Ulrik Svanlund

Spring 1997 — Fall 1998

Draft

UNU/IIST Report No. 93

UNU/IIST

UNU/IIST enables developing countries to attain self-reliance in software technology by: (i) their own development of high integrity computing systems, (ii) highest level post-graduate university teaching, (iii) international level research, and, through the above, (iv) use of as sophisticated software as reasonable.

UNU/IIST contributes through: (a) advanced, joint industry–university advanced development projects in which rigorous techniques supported by semantics-based tools are applied in case studies to software systems development, (b) own and joint university and academy institute research in which new techniques for *(1) application domain* and computing platform modelling, *(2) requirements capture,* and *(3) software design & programming* are being investigated, (c) advanced, post-graduate and post-doctoral level courses which typically teach Design Calculi oriented software development techniques, (d) events [panels, task forces, workshops and symposia], and (e) dissemination.

Application-wise, the advanced development projects presently focus on software to support large-scale infrastructure systems such as transport systems (railways, airlines, air traffic, etc.), manufacturing industries, public administration, telecommunications, etc., and are thus aligned with UN and International Aid System concerns. UNU/IIST is a leading software technology centre in the area of infrastructure software development.

UNU/IIST is also a leading research centre in the area of Duration Calculi, i.e. techniques applicable to *real-time, reactive, hybrid & safety critical systems.* The research projects parallel and support the advanced development projects.

At present, the technical focus of UNU/IIST in all of the above is on applying, teaching, researching, and disseminating Design Calculi oriented techniques and tools for trustworthy software development. UNU/IIST currently emphasises techniques that permit proper development steps and interfaces. UNU/IIST also endeavours to promulgate sound project and product management principles.

UNU/IIST's primary dissemination strategy is to act as a clearing house for reports from research and technology centres in industrial countries to industries and academic institutions in developing countries. At present more than 200 institutions worldwide contribute to UNU/IIST's report collection while UNU/IIST at the same time subscribes to more than 125 international scientific and technical journals. Information on reports received (and produced) and on journal articles is to be disseminated regularly to developing country centres — which are then free to order a reasonable number of report and article copies from UNU/IIST.

Dines Bjørner, Director — 2.7.1992–1.7.1997

UNU/IIST Reports are either $\mathcal{R}$esearch, $\mathcal{T}$echnical, $\mathcal{C}$ompendia or $\mathcal{A}$dministrative reports:

$\boxed{\mathcal{R}}$ Research Report • $\boxed{\mathcal{T}}$ Technical Report • $\boxed{\mathcal{C}}$ Compendium • $\boxed{\mathcal{A}}$ Administrative Report

P.O. Box 3058
Macau

A Railway System
Coordination '97
Case Study Workshop Example

Dines Bjørner, Chris W. George, Bo Stig Hansen, Hans Laustrup and Søren Prehn — and with contributions from Jakob Braad, Jon Fuglsang, Karin S. Mogensen and Ulrik Svanlund

Abstract

This report presents a solution to the problem presented by the organisers of a Case Study Workshop in connection with Coordination'97: the second international conference on Coordination Models and Languages, Berlin, September 4, 1997.

The solution covers three stages of development: a domain theory of railway systems, a set of requirements definitions, a computing systems architecture and program organisation. All stages are both informally and formally described in The RAISE Specification Language, and correctness relations between stages are formally expressed using The RAISE Method Implementation Relation.

Draft

Dines Bjørner: Prof., Dept. of IT, Techn.Univ.of Denmark; formerly Director of UNU/IIST, 1992–1997;

Chris George: Senior Research Fellow, UNU/IIST;

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

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

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

Jakob Braad: M.Sc. Student, Dept. of IT, Techn.Univ.of Denmark;

Jon Fuglsang: M.Sc. Student, Dept. of IT, Techn.Univ.of Denmark;

Karin S. Mogensen: M.Sc. Student, Dept. of IT, Techn.Univ.of Denmark;

Ulrik Svanlund: M.Sc. Student, Dept. of IT, Techn.Univ.of Denmark.

Contents

1 Introduction

1.1 Aims & Objectives

This report presents a solution to the problem presented by the organisers of a Case Study Workshop in connection with Coordination'97: the *Second International Conference on Coordination Models and Languages*, Berlin, Germany, 4 September, 1997.

The solution decomposes the problem into the solution of a number of subsidiary problems and is presented in the form of a set of four pairs of descriptions:

1. A Comprehensive Domain Theory
   (a) Informal Descriptions: Synopsis, Terminology, Narrative
   (b) A Formal Description

   The domain theory addresses the issue: *What is a Railway?*

   Our domain theory provides a partial answer to this question without any reference to computing. We say that the domain theory is comprehensive since it covers a wide spectrum of the concept of being a railway.

2. A set of n Requirements Definitions

   Each consisting of:

   (a) Informal Descriptions: Synopsis, Terminology, Narrative
   (b) A Formal Description

   The requirements definitions address the issues: *What kind of Computing (& Communications) Support could be provided for the Railway?* for a variety of distinct applications within a railway organisation.

   "Exercising" several rather distinct requirements allow us to identify overlapping sub-systems: that is potential software that can be shared amongst rather different applications.

3. A Computing Systems, in particular Software Architecture
   (a) Informal Descriptions: Synopsis, Terminology, Narrative
   (b) A Formal Description

   The software architecture identifies major "engines" that interface to the application users.

4. Program Organisation

(a) Informal Descriptions: Synopsis, Terminology, Narrative

(b) A Formal Description

The program organisation identifies a number of internal processes.

Our distinction between Software Architecture and Program Organisation is quite simple: whatever the user "sees" (i.e. experiences) of the software: its commands, data structures, etc., that is: the external interfaces, belongs to architecture; whatever internal interfaces that cannot be ascertained by the user, for example process decompositions, process-to-process messages, etc., belongs to program organisation. We therefore call 'program organisation' what our colleagues at Carnegie-Mellon University, Profs. David Garland, Mary Shaw, etc., call "software architecture"! And we like what they are doing!

The formal descriptions are expressed in RSL: The RAISE Specification Language. Relations between the three stages of development (1.–2.–3.) are expressed in terms of the RAISE Implementation Relation.

1.2 Background

The work of this report has been carried out by many people. The authors are at or come from either IT/TUD: the Institute of Information Technology, the Technical University of Denmark (DB, BSH, HL), CRI Inc. (CWG, SP), or UNU/IIST (DB, CWG, BSH, HL). Therefore we briefly state the four background components: RAISE, IT/TUD, CRI Inc., and UNU/IIST.

1.2.1 RAISE

RAISE stands for Rigorous Approach to Industrial Software Engineering. RAISE contains a Method: The RAISE Method, a specification and design language RSL: The RAISE Specification Language, and The RAISE Tool Set. RAISE features both functional, axiomatic, imperative, algebraic and process algebra constructs, an implementation relation, and a proof system and thus permits formal verification of properties and steps of refinement, model as well as property oriented specifications, abstract as well as concrete designs of applicative as well as concurrent systems. RAISE thus embodies logic and features of VDM, OBJ, ccs, CSP, and Standard ML. , indexMethod!VDM

1.2.2 The IT/TUD Background

Three of the authors (DB, BSH, HL) come from IT/TUD, but were at UNU/IIST during the major development of this report.

The research and education profile of IT/TUD's Programming Methodology Group focuses very much on the issues of this report.

1.2.3 The CRI Inc. Background

Two of the co-authors (CWG, SP) of this report come from or are at CRI Inc.

CRI Inc. has been a major developer and researcher of The RAISE Method, Specification Language (RSL) and Tool Set. It took over this effort from Dansk Datamatik Centre (DDC) when its programming methodology staff and projects were transferred from DDC to CRI Inc. in late 1988. The RAISE project was joint with, amongst others, STL Harlow (now BNR [Nortel]). After the 1985–1989 RAISE R&D project CRI Inc. conducted the LaCoS (Large-scale Construction of Software using formal methods) project together with several industrial partners around Europe. Here RAISE was applied to a number of diverse applications: *tethered satellites, global shipping transactions, marine engineering control, railways,* etc. The railway sub-project was conducted primarily by Matra Transportation.

1.2.4 The UNU/IIST Background

The first and founding Director of UNU/IIST (DB) was a co-developer and researcher of both VDM and RAISE and helped instigate the RAISE and LaCoS projects. He was also a co-founder of DDC — while at TUD.

Two CRI Inc. staff (CWG, SP; co-instigators and main researchers and developers of RAISE) have together spent the first five years of UNU/IIST's existence at UNU/IIST.

During the first five years of UNU/IIST, that is: since July 1992, software development has been researched and practiced on the background of first developing domain theory (enterprise model) descriptions, from domain theories requirements descriptions were refined; and finally software was developed from requirements.

UNU/IIST has in its many advanced development consultancy projects, together with fellows who visit UNU/IIST for 8–12 month periods, researched and developed many domain theories: *railways, manufacturing, multi-script document systems, air traffic, airlines, radio telecommunications systems, toll-ways,* etc. With the Ministry of Railways of the People's Republic of China and of the Russian Federation, UNU/IIST has studied or are studying such models and, with fellows, jointly developed advanced control software for railway sub-systems. The models given in this report derives from varieties of similar railway system models.

2 Coordinated Models & Languages

A main term of the conference to one of whose workshops this report — or, more likely, an extended summary of this report — is intended to be submitted is that of 'coordination'. We do not know of the conference organisers definition of this term. We will anyway suggest our own, albeit implicit definition. This section serves that rôle.

2.1 Understanding Large–scale Application Domains

The problem that this paper addresses is that of understanding application domains. That is: of constructing descriptions, both informal and formal, of application domains "devoid" of any reference to computing. We wish to refer to such descriptions as 'domain theories' and to their development and investigation as 'domain analysis'. Other names for such descriptions are 'enterprise models' or 'business engineering'.

We wish in particular to focus our techniques on such application domains which we could call 'infrastructures'.

The next two subsections therefore will spend a few more words on these two concepts: 'infrastructures' and 'domain analyses & theories'.

The concept of 'coordination models and languages' perhaps derives its basic justification exactly from the need to provide software support for infrastructures.

2.2 Infrastructures

2.2.1 Delineation

A country's infrastructure is normally thought of as one thing composed from such other things as: transportation systems ((i) roads, (ii) railways, (iii) shipping, (iv) air transport, etc.), utilities ((v) telecommunications, (vi) electricity supply, (vii) gas supply, (viii) water supply, (ix) sewage disposal, etc.), (x) manufacturing markets (suppliers, consumers, traders and producers), etc.

We will, in this paper, take the view that an infrastructure is one of these "other things". Such an infrastructure is "large enough" to warrant a separate investigation: a theory (or set of theories as it may well turn out to become) and its analysis (respectively their analyses).

A definition of infrastructure could either emphasise the socio-economic aspects,[1], or could

[1] As in The World Bank definition: *infrastructure is an umbrella term for many activities referred to as "social overhead capital" by some development economists, and encompasses activities that share technical and economic*

emphasise the operational aspects[2], or — as we will do it in this paper — will emphasise the linguistic facets:

Definition: *An infrastructure is seen as a set of distinct, professional languages sharing a common set, a base, or a core of concepts.*

Let us illustrate the above:

The Railway System Example: *We look at the various kinds of professionals that populate any typical railway system: (1) There are the system strategy planners, and they talk about increased or decreased traffic, of lines and stations, and of specific train or other services; (2) then there are the 'plant' development (tactical) planners and developers, and they talk about implementations of the abstracted strategy concepts: specific lines, stations and services so as to realize specific traffic (etc.) goals, they also talk about acquiring new or disposing old resources: monies (i.e. capital), personnel, main and auxiliary 'plant' equipment (rails, buildings, switching & signaling "gear", locomotives, wagons (cars), etc.); (3) then there are the operations planning & development staff, and they talk about timetables, resource schedules & allocations, quality assurance, etc.; then there are the ground (or field) staff (the previous 'classes' of staff can be thought of as sitting in office buildings not directly related to the main rail plant), and they talk about (4) accepting seat reservations and selling passenger tickets, (5) of day-to-day freight handling, (6) train dispatch, monitoring & control, (7) signaling, (8) line management, (9) station management (including shunting and marshalling yard management), etc.; and (10) then there is the staff which gather statistics for various purposes: preventive maintenance, local scheduling & allocation adjustments, and more global, medium to long term planning (see items (1)–(2)–(3)).*[3]

The point of the above example is to try illustrate that although wide in spectrum (in terms of job profiles etc.), all these people share a sufficiently interesting (large) number of concepts, viewed, however, with properly "intersecting" but not "identical meanings".

The intersecting parts form the 'base' (or 'core'), and the professional languages spoken by the various classes of staff (viz.: (1)–(10)) form 'extensions'.

2.2.2 Discussion

Classically software has been provided piecemeal to enterprises within each of these sectors — with little (other than conventional operating and database system) support for these diverse

features (such as economies of scale and spill-overs from users to non-users).

[2] An operational definition of 'infrastructure' could be: an infrastructure is a set of distributed, concurrent main processes that share, change and exchange state information, that start, delay, halt and stop (or "kill') subsidiary processes, etc.

[3] Reminder to DB: Must check that this 'decomposition' is reasonably realistic with for example DSB.

software packages to invoke each other and share data. In our approach to software for infrastructures we provide frameworks of application specific software that tie existing and (especially) future application packages together.

Software for infrastructures is typically distributed and is concerned with the flow of operations and communication of information across geographically wide areas. Issues of openness, timeliness, security, lack of corruption and resilience are often important.

Modern techniques now allow developers to model very large scale infrastructures and thus to develop software for their support.

2.2.3 Laws of Man–made Systems

Just like physical systems — like mechanics (including celestial mechanics), electricity, thermodynamics, etc. — satisfy laws such as Newton's (Copernicus', Gallileo's, Kepler's), Ohm's, the first, second, etc. laws of thermodynamics, etc., so we may expect that man-made systems — like railways, etc. — satisfy certain laws. We shall attempt to identify such laws in connection with sub-parts of railway systems.

Our descriptions, whether informal or formal, must imply these laws.

2.3 Domain Analyses & Theories

The goal of establishing a domain theory is to give semantics to the core terms as well as to terms of the extended languages.

We emphasise that we are not, we repeat: not, giving semantics to the professional languages of the various infrastructure parts, but "only" to their crucial terms.

We further emphasise that our descriptions are both informal and formal. This paper will only illustrate these descriptions but will not present the varieties of techniques used by their constructors. We usually "divide" the informal descriptions into three parts: A brief Synopsis, a longer Narrative, and an accompanying Terminology.

Terminology: We do not illustrate the concept of Terminology, but stress that in any project the terminology become a central object of development and management control: at any stage in the overall development process the Terminology must be up-to-date or "ahead": any discrepancy ("lagging behind") is usually a clear sign of project mis-management. So: A Terminology document must be established, maintained and adhered to: all the terms used in other description documents must "follow" the definitions given in the Terminology.

2.3.1 Core + Extensions vs. Intrinsic *&c.* Views

— Core + Extensions

In **The Railway System Example** of subsection 2.2.1 we illustrated 10 sub-languages. They all shared the core concepts of rail net, trains and traffic. Each sub-language description extends this core by 'own' extensions. Examples of extensions are the terms, i.e. the concepts and facilities of: (1) strategic planning & planners, (2) tactical resource allocation & scheduling planners, etc.

How will we know whether a term belongs to the core or some extension? We may only know so after some experimentation — where the experimentation consists of writing descriptions of the individual sub-languages (i.e. of their universes of discourse, that is: subdomains). Only then might we, through careful examination (i.e. analysis) discover the common (that is, the core) terms, and, by exclusion, thus also the extension terms. More realistically there is a whole lattice of cores and extensions, sub-cores and sub-extensions. When we present our full formal models (in appendix A then the reader will see these sub-cores and sub-extensions.

— Intrinsic *&c.* Views

We suggest to build up the description of any of the sub-languages — that is: of any one of the above mentioned parts (1)–(10) of a railway system — by the stepwise development and composition of a set of views of the sub-language domain. A view is a selection of individuals and phenomena of the domain at the exclusion of others — so a view is a partial description of a domain.

Major views can be given names:

- Intrinsics
- Support Technologies
- Rules & Regulations
- Staff and Client Behaviours
- Environment
- Economics

It is a part of the method of domain description that the developer (the scribe) analyse the domain to ascertain which of these and other views are relevant, and in which order of composition they occur.

Description of what each of these views cover (i.e. focus on) are given in the next subsections.

2.3.2 Intrinsics

What separates a view of the intrinsics of a domain from non-intrinsic views is that the former is invariant w.r.t. time, w.r.t. ever changing supporting technologies, w.r.t. ever changing rules & regulations, etc. There will always be: *lines and stations, tracks, switches and cross-overs, time tables, and traffic.* Thus intrinsic notions are often abstract in the sense that their current concretization also reflects a choice of supporting technology, a seemingly arbitrary collection of rules & regulations — which may change with technology, etc.

Both the informal synopsis, narrative and terminology, and the formal description of the intrinsics must capture all components and their properties. In the intrinsics descriptions the emphasis is on abstract properties.

2.3.3 Support Technologies

By support technologies we understand concretizations of intrinsic notions. *A switch, a hundred years ago, was typically 'thrown' manually by a railway worker positioned at the switch, while maybe still today it may be electro-mechanically operated from a seemingly remote cabin tower, or even part of a solid state (i.e. electronic + communications) interlocking arrangement.*

Both the informal synopsis, narrative and terminology, and the formal description of the support technologies must capture all support technology components and their properties. In the support technology descriptions the emphasis is on concrete behaviours and must include dependability issues. Typically we find that temporal descriptions of reactive (including hybrid) systems are required.

2.3.4 Rules & Regulations

By rules & regulations we understand prescriptions of staff and client behaviour vis-a-vis the intrinsics, the supporting technologies and other staff and clients of the domain. *In China, at most railway stations, at most one train is allowed to enter or leave a station (boundary) in any two minute interval. A single or return railway ticket is not allowed to imply travel that passes the same station more than once. Etc.*

Both the informal synopsis, narrative and terminology, and the formal description of the rules & regulations must capture as much of the logics of these procedures, including their implications. In the rules & regulation descriptions the emphasis is on logic properties — and usually a variety of logics are need to adequately cover a domain.

2.3.5 Staff & Client Behaviours

By staff and client behaviours we mean the sequences of those events (actions and reactions) that involve, i.e. interact with, intrinsic or supporting technology components of the domain and/or with other staff or clients of the domain. Staff and client behaviours are not 'programmable', at most 'biddable'!

Both the informal synopsis, narrative and terminology, and the formal description of the human behaviours must capture as much of these behaviours, including their fallibility. In the human behaviour descriptions the emphasis is on reactiveness, temporality and logic — and usually a variety of [modal] logics are need to adequately cover a domain.

2.3.6 Environment

By environment we here mean the temporal state of environmental (biological, mineralogical, historical, cultural, etc.) indicators of the domain. Again: Intrinsic components, the support technologies, and the staff and client behaviours influence environment indicators.

Both the informal synopsis, narrative and terminology, and the formal description of the [desired] environment of the domain must capture as many equities and indicators. Usually the formal descriptions are couched in classical mathematical, cum biological etc. formalisms.

2.3.7 Economics

By economy of a domain we understand the temporal state of economic indicators of the domain. Intrinsic component, the support technology, and the staff and client behaviours are expected to achieve some optimality functions, typically wrt. economics, resulting in profits, etc.

Both the informal synopsis, narrative and terminology, and the formal description of the possible economics of the domain must capture as much of the economic indicators, including risk factors. Usually the models are "borrowed" from the fields of economics and enterprise management.

3 The Given Narrative

The following text is "lifted" directly from the workshop announcement referred to in the Abstract and Introduction.

3.1 Problem Description

Consider a railway network consisting of a railway tracks, junctions and stations. A number of trains is expected to traverse the network in accordance to a global schedule. For each train the schedule determines the route that is to be traversed and a timetable, that specifies the expected times of arrival and departure for each station along the route.

It is required that a control system be developed for a given railway network. The primary objective of the system is to avoid collisions between trains. The secondary objective of the system is to respect the timetables as much as possible. A third objective is to make travelling as comfortable as possible, which means that trains should try to avoid abrupt changes in speed.

Below, the problem is explained in more detail. Note, however, that the description is by no means precise or complete. If required, the reader is free to make additional assumptions, as long as these assumptions can be justified as being reasonable and realistic.

3.1.1 The Railway Network

The railway network consists of a two-dimensional map of railway tracks. Wherever two or more tracks meet, there is a junction. A railway station is situated along one or more parallel tracks. At each of these tracks a platform allows passengers and cargo to enter or leave the train. Tracks, platforms, as well as trains have a certain length. Consequently, platforms and railway tracks can hold a limited number of trains.

The tracks are bidirectional, i.e. trains may move in either way. However, trains cannot pass each other on a single track, and two trains that keep moving in opposite directions towards each other on the same track will eventually collide.

3.1.2 Schedules

Each train travels according to a – possible infinite – schedule that lists the stations it subsequently has to arrive at (these are not necessarily neighbours). In addition, the schedule specifies for each of the listed stations, the platform the train should stop at, and the time of arrival and departure. Trains should adhere to this schedule as much as possible. They may arrive earlier,

and depart later that specified, but one should take into account that this might delay other trains. The exact route a train has to follow is left unspecified, so given the current situation, the railway control system may choose any suitable path for a train to reach its next destination.

3.1.3 Trains

Trains have both a maximum speed and a maximum acceleration/deceleration. In addition, they have a certain length. Trains are fragile objects: if at any time they fail to keep a safe distance, they collide and break down. The latter means that both trains instantly stop and block the track (possibly causing more collisions). Then a collision has occurred, the track will be cleared after some period of time.

3.1.4 Infrastructure

Communication between the trains and the railway control system is established by means of a mobile wide area network (MWAN). The network supports at least broadcasting of messages, but also multicasting and point-to-point addressing can be used, if so desired. The communication network has limited bandwidth and messages are subject to some maximum latency. The MWAN is typically used to transfer information from the trains to the control system, and conversely, to relay commands from the control system to the trains.

3.2 System Requirements

Repeatedly the railway control system has to perform the following tasks.

- Monitor the position of each of the trains (speed, acceleration and direction should at least be derivable from this). At all times, a train is capable of determining its own position.
- Assess the current situation, predict future developments, and if necessary, prepare corrective action in order to meet the three objectives of the system: avoiding collisions, respecting the timetables as much as possible, and arranging a comfortable journey. Basically there are two possible actions. The speed of one or more of the trains can be adjusted, either by acceleration or deceleration. Alternatively a train can be rerouted, for instance along a faster route, or make way for another train to pass by.
- Execute the planned actions. Using the communication network, the planned actions are made available to the trains. A certain amount of time has to be reserved for the execution of actions, since neither communication nor the actions themselves occur instantaneously.

Besides the functional requirements of the system, a number of additional aspects must be taken into account.

3.2.1 Timing

Deadlines of different importance have to be taken into account. Firstly, potential collisions must be resolved before they become unavoidable, and secondly, actual rerouting should take place before arrival at the corresponding station or junction.

3.2.2 Scalability

The control system must be scalable. Ideally, a control system should work without modifications for a railway network of any size, and for any (feasible) schedule. In practice this will not be possible, due to real-time constraints. With respect to this, one should be able to assess the ability of a certain configuration to handle particular networks and schedules.

3.2.3 Extendibility

The railway control system should support anticipated changes in its environment. Since railway services must continue to be available at all times, the system must allow upgrading on-line, i.e. during operational use.

In reality, nothing is ever fixed. In our world, merely three things may change:

- The railway network topology may be altered. This means either that new tracks and/or stations get inserted in the network, or that some get removed. Removal of a station implies modifications of all schedules that contain that station.
- The parameters of a train, i.e. its maximum speed and acceleration/deceleration, and its length, may change. For instance, this may occur when a train is replaced by a new model. These changes only occur at the railway stations.
- The schedule may get altered. This includes the addition of extra trains to the cup final, or removal of trains that are considered too expensive because very few passengers use it (as soon as a train has no schedule, it will be removed). Trains may also be delayed at a station, for example because of some sort of unexpected maintenance. Trains will only be removed when they are at a station (unless accidents occur and trains are removed at the location of the accident). Likewise, new trains will only be inserted at a station, after having made sure that this will not lead to *inevitable* collisions.

3.2.4 Fault Tolerance

In reality both the railway control system and its environment may exhibit unexpected behaviour due to failures. Despite these failures, however, the system should be able to continue operating, at least in a degraded mode where the safety-critical functions, related to collision avoidance, are retained.

We distinguish between three different types of failures.

- Trains may fail for numerous reasons at unpredictable times (e.g. someone pulling the emergency brake). In our world, a broken train typically decelerates as quickly as possible to a full stop, effectively blocking the railway track for other trains. A broken train will get fixed after some period of time and will proceed on its journey.
- The communication network that is used for information exchange between the control system and the trains, is unreliable. Messages will not be corrupted, but they tend to get lost at unpredictable times. So, there is no guarantee that messages will arrive. This is one of the most fundamental problems that the system has to deal with. (Solutions that employ an abundant duplication of messages to increase reliability will have to make some realistic assertions concerning the necessary bandwidth.)
- The processor, or processors, of the computing system that hosts the railway control system, are prone to failure. Whenever a processor fails, it immediately stops executing without further notice. By incorporating redundant hardware the control system should be able to continue operating, either fully or in a degraded mode.

4 The Domain Model

We rephrase and significantly expand on the domain aspects as presented in section 3.1.

4.1 Synopsis

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

4.2 Narrative

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

4.2.1 Rail Net

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

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

— Units & Connectors

The rail net consists of connected units. Units are the blocks, from which the rail net is build.

Units have connectors. Connectors are further undefined quantities.

If two units have a connector in common then they are said to be connected at that connector.

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

Units are either linear units (pairs of straight or curved rails, having two distinct connectors, one at either "end" of the unit), or are junctions (switches, one connector at one end, two at the other end, all distinct), or are cross-overs (two connectors at either end, all distinct, two at either end), etc.

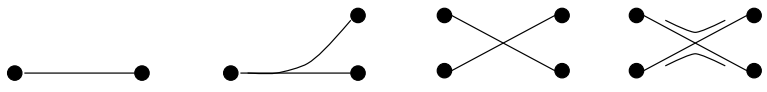

Figure 1: Different kinds of units

```
value
  is_Linear_U: U → Bool,
  is_Junction_U: U → Bool,
  is_Crossover_U: U → Bool
axiom
```

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

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

— Unit Paths & States

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

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

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

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

should therefore be included in the intrinsic model. The physical state of a unit will allways be one of the possible states of that unit.

type
  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,

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

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

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

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

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

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

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

The unit gives rise to potentially four different managed states:

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

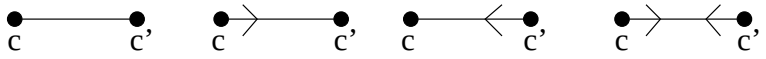

Figure 2: States of a linear unit

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

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

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

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

— Auxiliary Unit Attributes

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

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

— **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 3 will not be legal in any network.

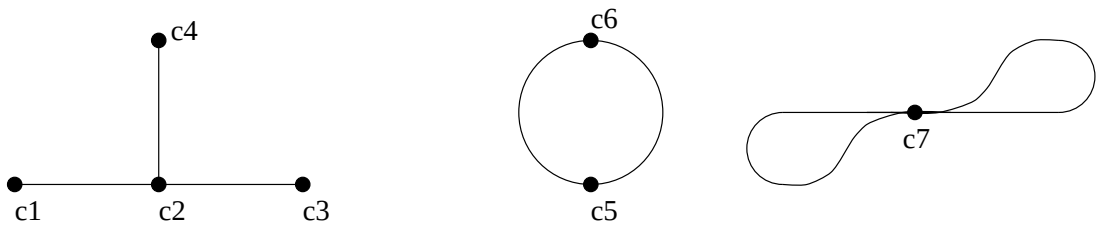

Figure 3: Illegal compositions of units

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

  /∗ In a network, two units do not contain the same path ∗/
  ∀ n:N, u,u′:U •
    {u,u′} ⊆ obs_N_Us(n) ∧ u≠u′ ⇒ U_Ps(u) ∩ U_Ps(u′) = {}

— Routes: Open and Closed

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 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.

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.

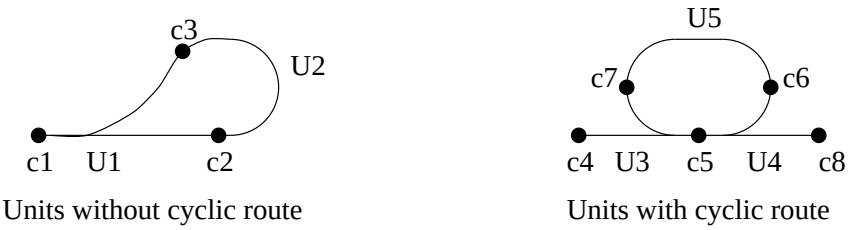

Figure 4: Cyclic and acyclic routes

This means that in figure 4, the route

⟨ c1,c2,c3,c1 ⟩

is an acyclic route, while the route

⟨ c4,c5,c6,c7,c5,c8 ⟩

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 ~→ U*
  Rt_Ul(rt,n) as ul
  post
    len ul = (len rt)−1 ∧
    elems ul ⊆ obs_N_Us(n) ∧
    ∀ i:Nat • {i,i+1} ⊆ inds rt ⇒ (rt(i),rt(i+1)) ∈ U_Ps(ul(i))
  pre Rt_possible_paths(rt,n) ∧ len rt ≥ 2,

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

  /* All units of a route */
```

```
Rt_Us: Rt × N ~→ U-set
Rt_Us(rt,n) ≡ elems Rt_Ul(rt,n)
pre feasible_Rt(rt,n),

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

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

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

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

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

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

/* All feasible routes of a 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 ~→ Bool
Rt_DisjUs(rt,n) ≡ card Rt_Us(rt,n) = len Rt_Ul(rt,n)
pre feasible_Rt(rt,n),
```

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

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

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

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

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

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

— 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, those belonging to lines and the rest. A line is a routable sequence of linear units. That is, a connectable sequence of linear units, such that there is a route through the units (in one or two directions). A station is any set of units, including linear, junctions (switches), crossovers, etc. Two lines meeting in a junction thus gives rise to a station. This station may just consist of that one junction though. The sets of units of a station can be decomposed into those belonging to tracks, that is routable sequences of linear units, and the rest. Among the rest there may still be identifiable routable sequences of linear units — be that as it may. Part of tracks form platforms, sidings, etc. A line always connects exactly two distinct stations. That is, a line always forms a route between two stations.

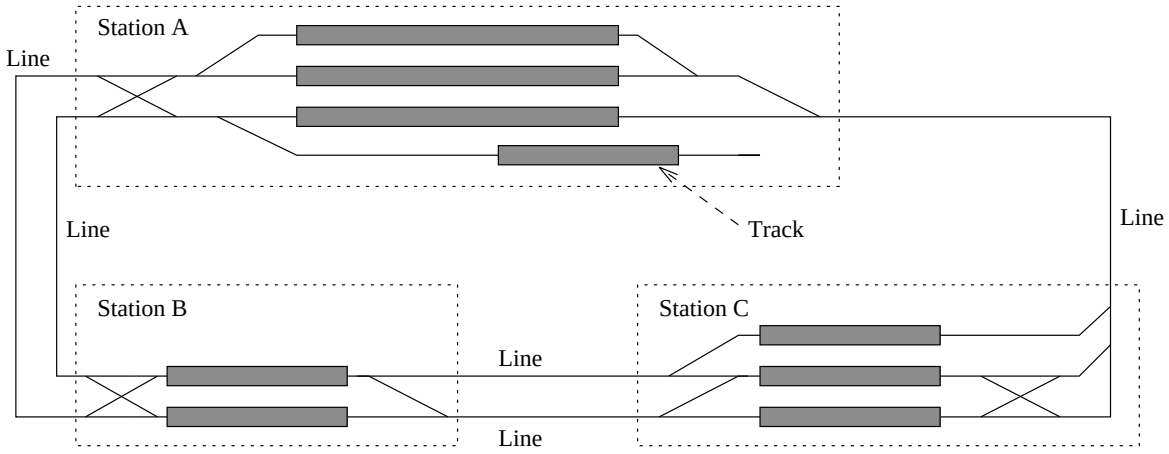

Figure 5: A network of lines and stations

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
  L, S, Trk

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

obs_Trk_Us: Trk → U-**set**,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  /* Stations are part of some network */
  net_Us(obs_S_Us(s)),
```

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

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

  /* Lines connect stations */
  l ∈ obs_N_Ls(n) ⇒
    ∃ s,s′:S •
      s≠s′ ∧ {s,s′} ⊆ obs_N_Ss(n) ∧
      SLSConnection(s,l,s′),

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

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

Under requirements, see section 5, we shall see that the setting of routes that connect lines and tracks is a typical station management function (*station route setting*) — and, given appropriate technologies, is supported by *solid state interlocking*.

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_N_SnSm: N → (Sn ~m~> S),

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

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

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

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

— **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)

— 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.

```
type
  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′)
```

More to be written

— **Laws of Rail Nets**

TO BE WRITTEN

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

4.2.2 Traffic

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

Thus, a traffic gives at any point in time the entire state of the rail net and all trains.

Traffic within stations may be due to shunting and marshalling.

A traffic may also record other matters not covered here.

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

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

type
  TF′ = T → RS,
  TF = {| tf:TF′ • wf_TF(tf) |},
  RS,
  TP = Tn $\overrightarrow{m}$ TS,
  Tn, TS
value
  net: RS → N,
  trns: RS → TP,

  obs_TS_TR: TS → TR,
  obs_TS_Velocity: TS → ...
  obs_TS_Acc: TS → ...

  /∗ Trains do not jump ∗/
  wf_TF: TF′ → **Bool**
  wf_TF(tf) ≡ continous_movement(tf),

  /∗ The trains of a TP are on units of the network ∗/
  TP_on_physical_open_routes: N × TP → **Bool**
  TP_on_physical_open_routes(n,tp) ≡
    ∀ tn:Tn • tn ∈ **dom** tp ⇒
      obs_TS_TR(tp(tn)) ∈ Physical_Open_N_Rts(n),

/∗ Trains do not jump from one spot to another ∗/
continous_movement: TF′ → **Bool**
continous_movement(tf) ≡
∀ t:T, tn:Tn • tn ∈ TF_Tns(tf,t) ⇒
train_removed(tf,tn,t) ∨
train_wf_move(tf,tn,t),

/∗ In the traffic tf the train tn is removed at time t ∗/
train_removed: TF′ × Tn × T → **Bool**
train_removed(tf,tn,t) ≡
tn ∈ TF_Tns(tf,t) ∧
∃ t′:T • t′>t ∧
∀ t″:T • t<t″≤t′ ⇒ tn ∉ TF_Tns(tf,t″),

/∗ In the traffic tf the train tn is performing a wellformed
(continuous) move at time t ∗/
train_wf_move: TF′ × Tn × T → **Bool**
train_wf_move(tf,tn,t) ≡
∃ t′:T • t′>t ∧
∀ t″:T • t≤t″≤t′ ⇒
tn ∈ TF_Tns(tf,t″) ∧
wf_TR_move(TF_TR(tf,tn,t),TF_TR(tf,tn,t″)),

TF_Tns: TF′ × T → Tn-**set**
TF_Tns(tf,t) ≡ RS_Tns(tf(t)),

RS_Tns: RS → Tn-**set**
RS_Tns(rs) ≡ **dom** trns(rs),

TF_TR: TF′ × Tn × T → TR
TF_TR(tf,tn,t) ≡ RS_TR(tf(t),tn)
pre tn ∈ TF_Tns(tf,t),

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

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

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

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

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

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

— Traffic Quality

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

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

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

```
value
  /* Trains are on open routes */
  TF_Managed_Open_Rts: TF → Bool
  TF_Managed_Open_Rts(tf) ≡
    ∀ t:T, tn:Tn • tn ∈ TF_Tns(tf,t) ⇒
      TF_TR(tf,tn,t) ∈ Managed_Open_N_Rts(TF_N(tf,t)),

  /* Trains are on lines or within stations of the network */
  TF_on_S_or_L: TF → Bool
  TF_on_S_or_L(tf) ≡
    ∀ t:T, tn:Tn • tn ∈ TF_Tns(tf,t) ⇒
      TF_TR(tf,tn,t) ∈ Us_Rts(N_L_Us(TF_N(tf,t)) ∪ N_S_Us(TF_N(tf,t)))
```

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

```
value
  TF_no_collisions: TF → Bool
  TF_no_collisions(tf) ≡ TF_disj_TR(tf) ∧ TF_trdisj(tf),

  /∗ No two trains share units ∗/
  TF_disj_TR: TF → Bool
  TF_disj_TR(tf) ≡ ∀ t:T • RS_disj_TR(tf(t)),

  RS_disj_TR: RS → Bool
  RS_disj_TR(rs) ≡
    ∀ tn,tn′:Tn • {tn,tn′} ⊆ RS_Tns(rs) ∧ tn≠tn′ ⇒
      Rt_Disj(RS_TR(rs,tn),RS_TR(rs,tn′),net(rs)),

  /∗ A trainroute does not run through the same unit twice ∗/
  TF_trdisj: TF → Bool
  TF_trdisj(tf) ≡ ∀ t:T • RS_trdisj(rs),

  RS_trdisj: RS → Bool
  RS_trdisj(rs) ≡
    ∀ tn:Tn • tn ∈ RS_Tns(rs) ⇒
      Rt_DisjUs(RS_TR(rs,tn),net(rs))
```

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

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

— **Laws of Traffic**

TO BE WRITTEN

1. **Law of :**

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

4.2.3 Schedules

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

type
SC = TF-**infset**

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

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

— Schedule Quality

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

— Laws of Schedules

TO BE WRITTEN

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

4.2.4 Passenger & Freight Train Timetables

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

type
TT

— **Schedules and Timetables**

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

value
TT_SC: TT → SC

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

value
SC_sat_TT: SC × TT → **Bool**
SC_sat_TT(sc,tt) ≡ sc ⊆ TT_SC(tt)

— **Timetable descriptions**

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

We shall in this section only deal with the timetables of passenger and freight trains: plans for shunting and marshalling will be covered later.

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

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

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

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

  J′ = SV-**set**,
  J = {| j:J′ • wf_J(j) |},

  SV′ :: sta : Sn arrival : T depart : T,
  SV = {| sv:SV′ • wf_SV(sv) |}
value
  /* Arrival before departure */
  wf_SV: SV′ → **Bool**
  wf_SV(sv) ≡ arrival(sv) ≤ depart(sv),

  /* Station visits are disjoint */
  wf_J: J′ → **Bool**
  wf_J(j) ≡
    ∀ sv,sv′:SV • {sv,sv′} ⊆ j ∧ sv≠sv′ ⇒ disj_SV(sv,sv′),

  disj_SV: SV × SV → **Bool**
  disj_SV(sv,sv′) ≡ arrival(sv)>depart(sv′) ∨ arrival(sv′)>depart(sv)

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

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

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

```
value
  TT_SC: TT → SC
  TT_SC(tt) ≡ { tf | tf:TF • TF_sat_TT(tf,tt) },

  TF_sat_TT: TF × TT → Bool
  TF_sat_TT(tf,tt) ≡
    ∀ tn:Tn, sv:SV •
      tn ∈ dom tt ∧ sv ∈ tt(tn) ⇒
      TF_sat_SV(tf,tn,sv),

  /∗ For the duration of the stationvisit, the station
      is part of the network and the train is at a track
      within the station ∗/
  TF_sat_SV: TF × Tn × SV → Bool
  TF_sat_SV(tf,tn,sv) ≡
    ∀ t:T • arrival(sv)≤t≤depart(sv) ⇒
      sta(sv) ∈ obs_N_Sns(TF_N(tf,t)) ∧
      TR_at_StaTrk(TF_TR(tf,tn,t),obs_N_SnSm(TF_N(tf,t))(sta(sv)))
```

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

— Laws of Timetables

TO BE WRITTEN

4.2.5 Rescheduling

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

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

value
  TF_on_SC: TF × SC → **Bool**
  TF_on_SC(tf,sc) ≡ tf ∈ sc

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

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

value
  /∗ Examine if all possible traffics are on schedule ∗/
  TFs_on_SC: TF-**set** × SC → **Bool**
  TFs_on_SC(tfs,sc) ≡
    ∀ tf:TF • tf ∈ tfs ⇒ TF_on_SC(tf,sc),

  /∗ Examine if no possible traffics are on schedule ∗/
  TFs_not_on_SC: TF-**set** × SC → **Bool**
  TFs_not_on_SC(tfs,sc) ≡
    ∀ tf:TF • tf ∈ tfs ⇒ ∼TF_on_SC(tf,sc)

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

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

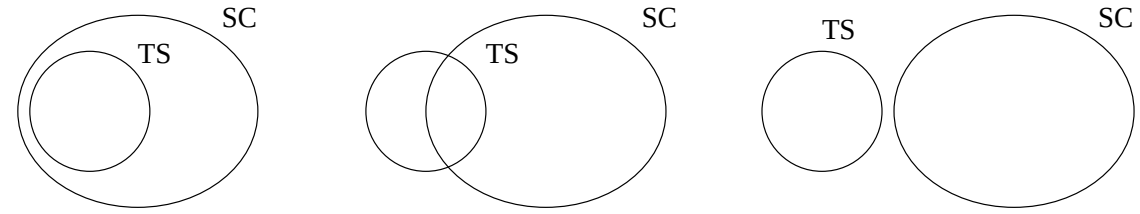

Figure 6: Schedules and sets of possible traffics

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

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

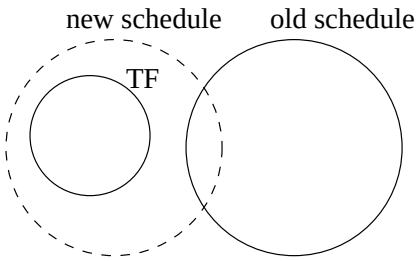

Figure 7: Rescheduling

value
/∗ Traffic should adhere to the new schedule ∗/
new_SC: TF-**set** × SC → **Bool**
new_SC(tfs,sc) ≡ ∼disruption(tfs,sc)

The new schedule should satisfy the new timetable.

value
new_TT: TT × SC → **Bool**
new_TT(tt,sc) ≡ SC_sat_TT(sc,tt)

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

— **Laws of Rescheduling**

TO BE WRITTEN

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

4.2.6 Shunting and Marshalling

By a train body we shall here mean a sequence of freight and specialty cars and passenger wagons.

To allocate, i.e. compose train bodies, one uses shunting and marshalling.

— **Shunting**

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

— **Marshalling**

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

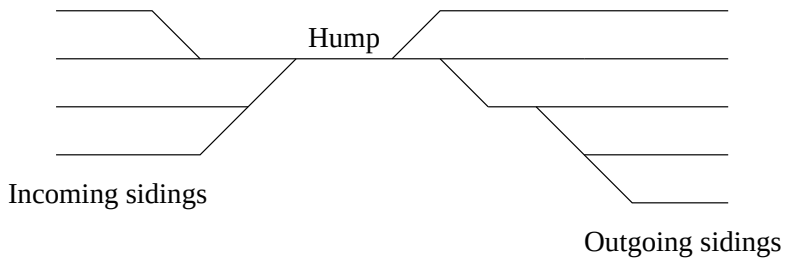

Figure 8: A marshalling yard

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

type
MY

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

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

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

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

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

    /* There is a route from any incoming connector
       to any outgoing connector */
    c ∈ obs_MY_incoming(my) ∧ c′ ∈ obs_MY_outgoing(my) ⇒
      ∃ rt:Rt • rt ∈ MY_inout_Rts(my) ∧
        (c,c′) = (Rt_firstC(rt),Rt_lastC(rt)),

    /* Between any two connectors in a marshalling yard
       there is no more than one route */
    ~∃ rt,rt′:Rt • rt ≠ rt′ ∧ {rt,rt′} ⊆ MY_Rts(my) ∧
      (Rt_firstC(rt),Rt_lastC(rt)) = (Rt_firstC(rt′),Rt_lastC(rt′)),

    /* All connectors in a marshalling yard are part of a route
       from an incoming connector to an outgoing connector
       which goes through the hump */
    c ∈ Us_Cs(obs_MY_Us(my)) ⇒
      ∃ rt:Rt • rt ∈ MY_Rts(my) ∧
      {c} ∪ obs_U_Cs(obs_MY_hump(my)) ⊆ elems rt,
```

```
  /* Units are never managed open in direction from the outgoing side
       towards the incoming side */
  ∀ rt:Rt •
    (rt ∈ MY_inout_Rts(my) ∧
    u ∈ obs_MY_Us(my) ∧
    (c,c′) ∈ elems Rt_Pl(rt)) ⇒ (c′,c) ∉ obs_U_Managed_Σ(u)
```

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

```
type
  TB = W*,
  W
value
  obs_TS_TB: TS → TB,
  obs_W_kind: W → ...
```

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

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

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

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

```
type
  MD′ = MS*,
  MD = {| md:MD′ • wf_MD(md) |},
  MS :: incoming: TB-set  outgoing: TB-set
value
  wf_MD: MD′ → Bool
  wf_MD(md) ≡
    ∀ i:Nat • {i,i+1} ⊆ inds md ⇒
```

```
        wf_state_shift(md(i),md(i+1)),

    wf_state_shift: MS × MS → Bool
    wf_state_shift(ms,ms′) ≡
        ∃ w:W, tb,tb′:TB •
        let ims=incoming(ms)\{tb} ∪ {tl tb},
            oms=outgoing(ms)\{tb′} ∪ {tb′^⟨hd tb⟩}
        in ms′ = mk_MS(ims,oms) end
```

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

```
value
    feasible_MP: MP → Bool
    feasible_MP(mp) ≡
        ∃ md:MD •
            incoming(hd md) = obs_MP_incoming(mp) ∧
            outgoing(md(len md)) = obs_MP_outgoing(mp)
```

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

```
value
    feasible_MP_wrt_MY: MP × MY → Bool
    feasible_MP_wrt_MY(mp,my) ≡
        feasible_MP(mp) ∧
        card obs_MY_incoming(my) ≥ card obs_MP_incoming(mp) ∧
        card obs_MY_outgoing(my) ≥ card obs_MP_outgoing(mp)
```

— **Laws of Marshalling**

Laws of marshalling could be:

1. **Law of Preservation:**

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

2. **Law of No–Reversal:**

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

3. **Law of Ordering:**

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

4.2.7 Passengers

Trains of the rail net may carry passengers. In this section, the journies and states of passengers will be described.

— **Passenger Traffic**

At any time there is a set of passengers in trains of the rail net system. Each passenger is identified by for instance a name. The names and states of passengers can be extracted from a traffic.

type
  Pn, PS,
  PP = Pn $\overrightarrow{m}$ PS
value
  obs_RS_PP: RS → PP

  /* The passengers in a traffic at a given point in time */
  TF_Pns: TF × T → Pn-**set**
  TF_Pns(tf,t) ≡ **dom** obs_RS_PP(tf(t))

  /* The state of a passenger at a given point in time */
  TF_PS: TF × T × Pn $\xrightarrow{\sim}$ PS
  TF_PS(tf,t,pn) ≡ obs_RS_PP(tf(t))(pn)
  pre pn ∈ TF_Pns(tf,t)

The passenger information may for instance describe the position of the passenger (train and seat), the ticket carried by the passenger etc.

value
  obs_PS_Tn: PS → Tn

For any Rail State, passengers must always be on trains of the network.

axiom
  /* All passengers are on trains of the rail net */
  ∀ rs:RS, ps:PS •
    ps ∈ **rng** obs_RS_PP(rs) ⇒ obs_PS_Tn(ps) ∈ **dom** trns(rs)

Other axioms could be included as well. For instance, in a traffic it should not be possible for passengers to jump from one train to another.

A passenger history gives the state of a passenger at any point in time where the passenger is in the rail net.

type
  PH = T $\overrightarrow{m}$ PS
value
  /* Find the passenger history of a passenger in given traffic */
  TF_PH: TF × Pn → PH
  TF_PH(tf,pn) ≡
    [t ↦ TF_PS(tf,t,pn) | t:T • pn ∈ TF_Pns(tf,t)]

— **Laws of Passengers**

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

4.2.8 Resources & Allocations

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

— **Rolling Stock**

type
RollingStock,
Wtype

value
/∗ Observer functions ∗/
obs_Wagons : RollingStock → W-**set**,

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

obs_RollingStock_from_Net : N → RollingStock,
obs_Wtype : W → Wtype,

During our work with the specification of plans for composing a certain train body from the rolling stock, we found that we needed a bag-type. Instead of specifying a specific bag for our purpose, we decided to make the bag specification parameterized. By doing so we will be able to use the specification for other purposes if necessary.
In the following we give the general specification of bags.

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

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

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

end
```

We use the bag specification to specify test functions. These test if a certain train body can be composed by the wagons in the rolling stock. As a special case, we also test if the desired train body already exists as a train body somewhere in the rolling stock.

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

value
  /* Test if a certain train can be composed */
  Train_Exists : N × Wtype* → Bool
  Train_Exists(n, wtypelist) ≡
  let
    r = obs_RollingStock_from_Net(n),
    rs_ws = obs_Wagons(r),
    rs_ls = Convert(rs_ws),
    rs_wtypebag = WTYPE_BAG.List_to_Bag(rs_ls),
    wtypebag = WTYPE_BAG.List_to_Bag(wtypelist)
  in
    ∀ wtype : Wtype •
    wtype ∈ dom wtypebag ⇒
```

```
    wtype ∈ dom rs_wtypebag ∧ (wtypebag(wtype) ≤ rs_wtypebag(wtype))
  end,

  /∗ Converts a W-set to a Wtype-list. Needed to convert a W-set to a Wtype-bag ∗/
  Convert: W-set → Wtype*
  Convert(ws) as wtypelist
    post
      card ws = len wtypelist ∧
      (∀ w:W • w ∈ ws ≡ obs_Wtype(w) ∈ elems wtypelist),

  /∗ Test if a certain train already exists in RollingStock ∗/
  Train_Already_Exists : N × Wtype* → Bool
```

We also need to be able to compose train bodies. In order to do so, we need to know where wagons of a certain type are located. We also need to be able to find the locations of a certain train body if such exists.

```
value
  /∗ The locations of wagons of a certain type ∗/
  Wagons_Locations : N × Wtype → (U*)-set
  Wagons_Locations(n, wtype) ≡
    { u | u : U* •
      let r = obs_RollingStock_from_Net(n), rs_ws = obs_Wagons(r)
      in ∀ w : W • w ∈ rs_ws ∧ u = obs_Units_from_Wagon(r,w)
           ∧ obs_Wtype(w) = wtype
      end
    },

  /∗ The locations of an entire trainbody of a certain type ∗/
  Trainbody_Location : N × Wtype* → (U*)-set,

  /∗ The locations of a trainbody of a certain type ∗/
  Train_Location : N × Wtype* → ((U*)*)-set

axiom
  ∀ n : N, wtypelist : Wtype* •
    Train_Already_Exists(n, wtypelist) ⇒ Train_Exists(n, wtypelist),

  ∀ n : N, wtypelist : Wtype* •
    Train_Already_Exists(n, wtypelist) ≡ (Trainbody_Location(n, wtypelist) ≠ {}),

  ∀ n : N, wtypelist : Wtype* •
```

Train_Exists(n, wtypelist) ≡ (Train_Location(n, wtypelist) ≠ {})

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

type
  Plan,
  PlanId

value
  /∗ Observer functions ∗/
  obs_destinationUnit_from_Plan : Plan → U,
  obs_Wagons_from_Plan : Plan → (W $\overrightarrow{m}$ U*),

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

Often, it is possible to make several plans for a certain train body. In practice one often want to choose the plan which is considered best wrt. some criteria, e.g. cost, environmental issues, time, distance etc. In the following we have chosen to model this by a predicate *Better_Plan* which tests if one plan is better than another. We do not give any specification of what makes one plan better than another.

value
  /∗ A plan is better than another ∗/
  Better_Plan : N × Plan × Plan → **Bool**,

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

$\equiv \sim$Better_Plan(n, p′, p))
},

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

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

We could also model some predicates which test whether a plan is better than another plan e.g. wrt. cost, distance or time:

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

After deciding which plan to use, we want to register this plan. This can be done by inserting it in the plan for the rolling stock. This plan, called rsPlan, contains information about all the chosen plans.
We also have to consider what to do with the wagons in the inserted plan. For example, what if we make a plan for composing another train body and that plan uses some of the wagons in the inserted plan. This would result in a conflict.
We have chosen to resolve this by having a reservoir which contains information about the wagons already committed to a plan. Whenever a plan is inserted in rsPlan, the wagons committed to that plan is removed from the wagon pool in the rolling stock and inserted in the reservoir. When a plan is executed, it is removed from the rsPlan.

type
rsPlan,
Reservoir

```
value
  /* Observer functions */
  obs_rsPlan : RollingStock → rsPlan,

  /* obs_Plan_from_rsPlan is partial since PlanId might not exist in rsPlan */
  obs_Plan_from_rsPlan : rsPlan × PlanId ~→ Plan,

  obs_PlanId_from_rsPlan : rsPlan → PlanId-set,
  obs_Reservoir_from_Net : N → Reservoir,
  obs_Wagons_from_Reservoir : Reservoir → W-set,

  /* When a Plan is inserted in rsPlan, the wagons in the Plan is moved to Reservoir */
  Insert_Plan : N × Plan ~→ N × PlanId
  Insert_Plan(n,p) as (n′,pid′)
    post
      let
        wm = obs_Wagons_from_Plan(p),
        ws = dom wm,
        r = obs_RollingStock_from_Net(n′),
        rs_ws = obs_Wagons(r),
        res = obs_Reservoir_from_Net(n′),
        res_ws = obs_Wagons_from_Reservoir(res),
        rsplan = obs_rsPlan(r)
      in
        ∼(ws ⊆ rs_ws) ∧ (ws ⊆ res_ws) ∧
        obs_Plan_from_rsPlan(rsplan, pid′) = p
      end
    pre Test_Plan(n,p),

  /* When a Plan is cancelled, it's removed from rsPlan */
  Cancel_Plan: N × PlanId ~→ N
  Cancel_Plan(n,pid) as n′
    post
      let
        r = obs_RollingStock_from_Net(n),
        rsplan = obs_rsPlan(r),
        r′ = obs_RollingStock_from_Net(n′),
        rsplan′ = obs_rsPlan(r′)
      in
        ∼ ( ∃ pid′ : PlanId •
        obs_Plan_from_rsPlan(rsplan′, pid′) = obs_Plan_from_rsPlan(rsplan,pid)
        )
      end
    pre pid ∈ obs_PlanId_from_rsPlan(obs_rsPlan(obs_RollingStock_from_Net(n))),
```

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

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

/* A Plan is removed from rsPlan when it is executed */
Exec

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

/* A wagon is inserted in RollingStock */
Insert_Wagon: N × W × U* $\xrightarrow{\sim}$ N
Insert_Wagon(n,w,ulist) **as** n′
  post
    let r = obs_RollingStock_from_Net(n′)
    in w ∈ obs_Wagons(r) ∧ ulist = obs_Units_from_Wagon(r,w)
    end
  pre w ∉ obs_Wagons(obs_RollingStock_from_Net(n))

— **Laws of Resources & Allocation**

Laws of resources & allocation could be:

1. **Law of Best Plan:**

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

2. **Law of Preservation:**

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

4.2.9 Customer Services

1. **Passenger Ticketing:**

   Any passenger or an agent acting on behalf of any group of one or more passengers may inquire about, reserve, cancel, re-book and actually buy tickets for simple or composite journies on scheduled trains.

   In the domain model, however, we only speak about the booking and occupancy status of passenger trains.

   Requirements, see section 5, will detail specific functionalities.

   More to be written

2. **Client Freighting:**

   In the domain model we similarly only speak about the booking and occupancy status of freight trains. We have chosen to model the stake holder perspective as seen from the customers. That is, we concentrate on the activities which is needed to fulfil the customers needs and wishes.
   In the following we refer to the system which handles the booking etc. as Freight Handling.

   type
   FTn = Tn,
   Freight,
   FreightId,
   FreightSc,
   FreightInfo,
   FreightSystem,
   FreightISI = FreightId × FreightSc × FreightInfo

   value
   /* Observer functions */
   obs_FreightId : FreightSystem → FreightId-**set**,

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

   obs_FreightSystem_from_Net : N → FreightSystem

axiom
/* For each FreightId in a FreightSystem exists exactly one Freight and one FreightSc */
∀ f:FreightSystem, fid, fid′ : FreightId •
{fid,fid′} ⊆ obs_FreightId(f) ⇒
fid = fid′ ≡
let
freight = obs_Freight(f,fid),
freight′ = obs_Freight(f,fid′),
finfo = obs_FreightInfo(f,fid),
finfo′ = obs_FreightInfo(f,fid′),
fsc = obs_Schedule(f,fid),
fsc′ = obs_Schedule(f,fid′)
in
freight = freight′ ∧
finfo = finfo′ ∧
fsc = fsc′
end

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

value
/* Test if Freight is delivered. False if FreightId doesn't exist in FreightSystem */
Is_Delivered : N × FreightId → **Bool**,

/* Test if Freight is at destination. False if FreightId doesn't exist in FreightSystem */
At_Destination : N × FreightId → **Bool**,

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

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

/* Freight request - Do not make a reservation */
Request : N × TT × Sn × Sn × T × Freight → FreightISI,

/* Confirm (reserve) a request if possible */
Confirm : N × FreightISI → N × **Bool**,

/* Freight reservation */

Reservation : N × TT × Sn × Sn × T × Freight → N × FreightISI,

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

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

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

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

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

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

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

From the FreightSystem we can now extract information about the pieces of freight. This information is needed to decide which wagons are needed to fulfil the agreement with the customers.

value
/* Find which wagons are needed for the freight */
GetTrain : N → Wtype*

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

type
FreightTraffic,
Location,
Place,
Pos

value
/* Observer functions */
obs_FreightSet : FreightTraffic → FreightId-**set**,

/* obs_Location is partial since we might not know the location of a certain freight, e.g. if it is lost */
obs_Location : FreightTraffic × T × FreightId $\xrightarrow{\sim}$ Location,

obs_FreightTraffic_from_Net : N → FreightTraffic,

/* Test if Place is a Station Name */
Is_Sn : Place → **Bool**,

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

/* Test if Location is a Station Name */
Is_Sn : Location → **Bool**,

/* Test if Location is a Train Name */
Is_FTn : Location → **Bool**,

Can_obs_Location : FreightTraffic × T × FreightId → **Bool**,

/* Freight Tracing */

```
  Freight_Trace : N × TF × T × FreightId ~→ Place

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

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

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

— **Scheduling of Freight**

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

— **Lost & Found**

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

We can model this by using the following functions:

```
type
  Freights,
  FT

value
```

obs_Freights_from_Train: FT → Freights,
Load_Freights: FT × Freights → FT,
/* Merge takes two Freights and combines them */
Merge: Freights × Freights → Freights

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

We now have to consider three cases:

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

Formally, this can be formulated as:

axiom
∀ ft : FT, frs, frs′ : Freights •
let frs′ = obs_Freights_from_Train(ft) **in**
obs_Freights_from_Train(Load_Freights(ft,frs)) = merge(frs,frs′)
end

(b) **Freight is lost :** In this case we cannot observe all the freight which were loaded on the train.

Formally, this can be formulated as:

axiom
∀ ft : FT, frs, frs′ : Freights •
let frs′ = obs_Freights_from_Train(ft) **in**
obs_Freights_from_Train(Load_Freights(ft,frs)) ⊂ merge(frs,frs′)
end

(c) **Freight is found:** In this case we can observe more freight on the train than the freight we actually loaded on the train.

Formally, this can be formulated as:

axiom
∀ ft : FT, frs, frs′ : Freights •
let frs′ = obs_Freights_from_Train(ft) **in**
obs_Freights_from_Train(Load_Freights(ft,frs)) ⊃ merge(frs,frs′)
end

When freight is lost, we want to be able to store information about it, its FreightInfo and FreightSc, etc. We also need to be able to check if freight is lost etc.

In the following we assume that we can obtain FreightId on freight found. We use this to remove the recovered freight from LostFreight.

Formally, this can be formulated as:

```
type
  LostFreight

value
  /* Observer function */
  obs_LostFreight: FreightSystem → LostFreight,
  obs_Freight_from_LostFreight: LostFreight → FreightId-set,

  /* obs_FreightInfo_from_LostFreight and obs_FreightSc_from_LostFreight */
  /* are partial since FreightId might not exist in LostFreight */
  obs_FreightInfo_from_LostFreight: LostFreight × FreightId ~→ FreightInfo,
  obs_FreightSc_from_LostFreight: LostFreight × FreightId ~→ FreightSc,

  Is_Freight_Lost: N × FreightId → Bool
  Is_Freight_Lost(n,fid) ≡
    let
      fs = obs_FreightSystem_from_Net(n),
      lf = obs_LostFreight(fs),
      lostfid = obs_Freight_from_LostFreight(lf)
    in fid ∈ lostfid
    end,

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

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

end
pre Is_Freight_Lost(n,fid)

— **Laws of Freight Customers**

A law of Freight Customers could be:

(a) **Law of Preservation:**
In any timeinterval such that no freight is delivered or received, the sum of traceable freight and lost freight is constant.

4.2.10 Station and Line Management

Station management consists of:

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

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

   Train dispatch also involves determination of train positions.

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

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

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

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

4. Other Resource Allocation & Scheduling:

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

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

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

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

   Line management also involves determination of train positions.

More to be written

— **Laws of Management**

TO BE WRITTEN

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

5 Requirements

Again, as for the railway application domain theory we paraphrase the requirements as originally given, see section 3.2.

5.1 Synopsis

A computerised system shall be developed for the monitoring and control of train traffic according and schedules as the latter are determined by the rail net and timetables. Scheduling involves choosing routes. The system shall be safety critical (no train crashes), shall afford punctual traffic (trains arrive and depart according to timetables), and comfortable (no abrupt changes in train speed, reasonable station stop intervals, reasonable train speeds, etc.).

We divide the requirements into a number of co-ordinated requirements:

1. Rail Net Development

   A software package is required which supports railway system planners division in designing changes to the rail net.

2. Train Dispatch

   A software package is required which supports station cabin men in the monitoring of trains arriving at a station as well as in the dispatch of trains from that station. The software packages shall support the eventual rescheduling of traffic and timetables in case of certain delays.

   This software package is different from the next one in that it primarily works with timetables and schedules, the latter in the form of running maps.

3. Signalling

   A software package is required which supports station cabin men in the setting of junctions (routes), blocking of lines (hence setting of line signals), and setting of station signals.

4. Train Control

   A software package is required which supports train engine men (the drivers) in the automatic to semi-automatic running of passenger and freight trains from station to station.

   This particular software packages seems to be the one most directly aimed at in the Coordination'97 Railway System Workshop Case Study.

5. Rolling Stock Monitoring & Control

   A software package is required which supports train planners in the monitoring of the whereabouts of all rolling stock: those cars and wagons which are part of scheduled trains, or of train bodies in marshalling yards, or are otherwise 'parked' on sidings anywhere on

the rail net. The package shall also support the choice of cars and wagons to be composed into new train bodies and trains.

This software package is different from , but is expected to interface to the next one:

6. Shunting & Marshalling

   A software package is required which supports train planners in the actual, physical shunting and marshalling of train bodies into new train bodies as per specifications emanating from the previous (the rolling stock monitoring & control) software package.

7. Passenger Reservation & Ticketing

   A software package is required which supports ticket reservation and ticketing staff in handling passenger inquiries, requests for train tickets, and in the actual station platform and on board train control of tickets.

8. Freight Handling

   A software package is required which supports freight reservation and handling staff in handling customer inquiries, requests for freight space, and in the actual station handling, including tracing of freight.

5.2 Example Requirements Narratives

Many different computing systems could be constructed to support the running and scheduling of rail net systems. In this section we will give examples and outlines of the requirements for such systems.

5.2.1 Scheduling Systems

The scheduling of the rail net and traffic on the net will be done by four systems: The Rail Net Development System, The Train Dispatch System, The Signalling System and The Train Control System. Before describing these individual systems, we will in this section examine the general properties of this kind of scheduling systems.

— Communication between systems

The Rail Net Development System takes care of the rail net scheduling. That is for instance insertion or removal of lines and stations.

The Train Dispatch System handles scheduling of train traffic. This includes the arrival and departure of trains from stations and may also include information on which lines to use when travelling between stations. To do this kind of scheduling, information from the Rail Net Development System will be needed. For instance future changes to the rail net may be of concern when planning train traffic.

The Signalling System schedules the setting of station routes, including setting of light signals and switch points. The Signalling system will need information on scheduled train arrivals and departures. This information comes from the Train Dispatch System.

The Train Control System handles the actual control of trains on the net. To do this, schedules from the earlier mentioned systems must be known.

The connection between systems can be done by communicating scheduling plans from one system to another. For instance, the Rail Net Development System produces a Rail Development Plan, *rndPlan*, describing the scheduled history of the rail net. This plan will be passed to the Train Dispatch System. This system will need the Development Plan, for instance when rescheduling of train traffic is needed. The rescheduling process will construct a new plan, the Train Dispatch Plan, *tdPlan*. This plan describes all decisions made by the Train Dispatch System as well as by the Rail Net Development System. This will be passed to the Signalling System. This system constructs the Signalling Plan, *sigPlan*, which is passed to the Train Control System.

Figure 9: System Connections

For any plan, it is possible to find the schedule containing exactly those traffics that are allowed by the plan.

type
  rndPlan, tdPlan, sigPlan
value
  rndPlan_SC: rnd_Plan → SC,
  tdPlan_SC: rdPlan → SC,
  sigPlan_SC: sigPlan → SC

Note that any traffic allowed by the Train Dispatch Plan should always be allowed by the Rail Net Development Plan. That is, the Train Dispatch System must not make plans that allow traffics which the Rail Net Development System has excluded. This means that the schedule of the Train Dispatch Plan will always be a subset of the schedule of the Rail Net Development Plan. For the same reasons, the schedule of the Signalling Plan must always be a subset of the schedule of the Train Dispatch Plan.

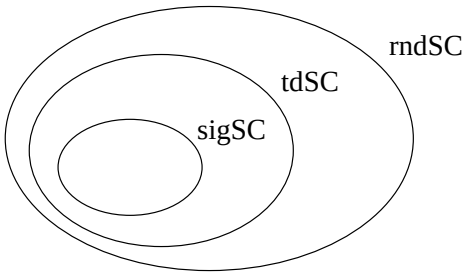

Figure 10: Schedules of the Rail Net Development Plans, Train Dispatch Plans and Signalling Plans

— Distributed Scheduling

The communication described assumes that there is only one instance of each system. This may not be the case however. For instance, one will usually have a number of Train Dispatch Centres, each handling train scheduling in a specific area of the rail net. Each of the centres will have their own Train Dispatch System.

Likewise, there will usually be a Signalling System at each station of the rail net, and a Train Control System on each train.

This distribution of scheduling gives rise to some problems, concerning communication between the systems. It will be necessary to make clear which systems are in charge of which areas, and what information is needed by the systems for the scheduling task.

— Control Areas

Any system that takes part in the scheduling controls a given area. For instance, a Train Dispatch Centre will control the overall traffic of trains within a specific area of the rail network, a Signalling System may control the routing and setting of signals within a particular station and a Train Control System may control the setting of speed and acceleration for a specific train.

The area of control for a given system can be described as a binary relation between traffics. This relation indicates for any pair of traffics if these traffics are equal within the area controlled. That is, the projections of the traffics onto the controlled area are equal. For instance, for a signalling system, the setting of a signal located at another station would be outside the control area of the system. Two traffics, which are equal, except for the state of this signal, would have the same projection onto the control area of this signalling system.

type
  /∗ Control Area ∗/
  CA = TF × TF → **Bool**
axiom
  ∀ ca:CA, tf,tf′,tf″:TF •
    [reflexive] ca(tf,tf) ∧
    [symmetric] ca(tf,tf′) ⇒ ca(tf′,tf) ∧
    [transitive] (ca(tf,tf′) ∧ ca(tf′,tf″)) ⇒ ca(tf,tf″)

A projection onto a control area can be described by the sets of traffics having that projection. From a control area it is possible to find all projections of that area.

type
  /∗ Projection ∗/
  Proj = TF-**set**
value
  CA_Projs: CA → Proj-**set**
  CA_Projs(ca) ≡ { TF_Proj(ca,tf) | tf:TF },

  TF_Proj: CA × TF → Proj
  TF_Proj(ca,tf) ≡ { tf′ | tf′:TF • ca(tf,tf′) }

Note that the projections of any control area are disjoint.

— Scheduling Systems

A scheduling system receives from a main scheduling system a plan, *inPlan*. For instance, a Train Dispatch System receives a Rail Net Development Plan from a Rail Net Development System. The plan, *inPlan*, describes the decisions made on the traffics by the main system. From *inPlan*, the scheduling system must construct another plan, *outPlan*, for the sub scheduling systems. For instance, a Train Dispatch System makes a Train Dispatch Plan for its Signalling Systems.

The semantics, *inSC* and *outSC*, of the received and generated plans can be found.

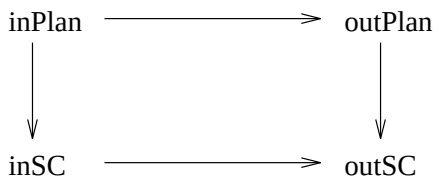

Figure 11: Scheduling Plans and their semantics

Not any plan can be created by a system. For instance, the train dispatch centre generates a plan to be used by a signalling system. This plan contains information on how to run the train traffic. This may include information on traffic within as well as outside the station controlled by the signalling system. The plan generated by the signalling system should contain not only the decisions made by the signalling system, but also the overall schedule information from the dispatch centre. Note that the signalling system can never make decisions of parts of traffics which are outside its control area.

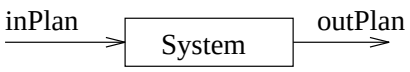

Figure 12: A Scheduling System

Therefore, any scheduling system must choose a set of projections. The plan generated by the system must now describe which projections were selected as well as the information contained in the received plan. The schedule of the generated plan will be the intersection of the schedule of the received plan and the set of all traffics of the selected projections.

type

```
  /* Scheduling System */
  Sys
value
  /* Semantics of the received plan */
  Sys_inSC: Sys → SC,
  /* Semantics of the generated plan */
  Sys_outSC: Sys → SC,
  /* Control area of the system */
  Sys_CA: Sys → CA,

  /* All schedules that adhere to the received plan of a system */
  Sys_plannedSCs: Sys → SC-set
  Sys_plannedSCs(s) ≡ CA_plannedSCs(Sys_CA(s),Sys_inSC(s)),

  /* All schedules that adhere to a received plan, given a control
     area */
  CA_plannedSCs: CA × SC → SC-set
  CA_plannedSCs(ca,mainsc) ≡
    { mainsc ∩ Projs_TFs(ps) | ps:Proj-set •
      ps ⊆ CA_Projs(ca) } \ { empty_SC },

  /* All traffics in a set of projections */
  Projs_TFs: Proj-set → TF-set
  Projs_TFs(ps) ≡
    { tf | tf:TF • ∃ p:Proj • p ∈ ps ∧ tf ∈ p },

  /* The empty schedule */
  empty_SC : SC = {}
```

The system should only generate plans that have allowed schedules. This could be formulated as a requirement for scheduling systems:

```
value
  /* The generated plan of a system adheres to the received plan */
  Sys_planned_outSC: Sys → Bool
  Sys_planned_outSC(s) ≡ Sys_outSC(s) ∈ Sys_plannedSCs(s)
```

— Distributors

For the distribution of information between scheduling systems, a special kind of system will be needed. These systems will be called Distributors. Any scheduling system has got a distributor, that handles the distribution of the generated plan to a number of subsystems.

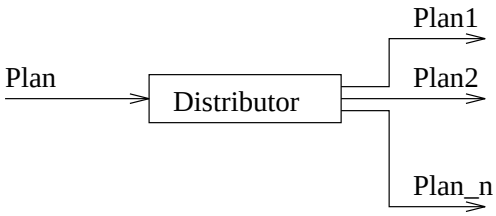

Figure 13: The Distributor

type
  /∗ Distributor ∗/
  Dist
value
  /∗ The distributor of a system ∗/
  Sys_Dist: Sys → Dist,
  /∗ The semantics of plans to be sent to the subsystems ∗/
  Dist_SysSCs: Dist → (Sys $\overrightarrow{m}$ SC),
  /∗ The semantics of the received plan ∗/
  Dist_inSC: Dist → SC
axiom
  ∀ s:Sys • Dist_inSC(Sys_Dist(s)) = Sys_outSC(s)

Note that the schedules of the plans distributed may not be identical. For instance a signalling system may not need much information about the other stations on the net. That is, each signalling system will receive only information about its own station.

It is possible to find the control area for a system and all its subsystems. For instance, a signalling system that sends information to a number of train control systems will have a total control area that includes the trains and the setting of routes and signals within the station.

value
  /∗ Find the total control area for a system and all its subsystems ∗/
  Sys_TotalCA: Sys → CA
  Sys_TotalCA(s) ≡
    CA_combine(
      { Sys_CA(s) } ∪
      { Sys_TotalCA(s′) | s′:Sys • s′ ∈ **dom** Dist_SysSCs(Sys_Dist(s)) }
    ),

  /∗ Combine a set of control areas ∗/
  CA_combine: CA-**set** → CA

CA_combine(cas) **as** ca
post ∀ tf,tf′:TF • ca(tf,tf′) = ∀ ca′:CA • ca′ ∈ cas ⇒ ca′(tf,tf′)

The distribution of plans can be done in many different ways. Usually the subsystems would want as much information on the scheduling of their own control area as possible, but would not need much information on other control areas.

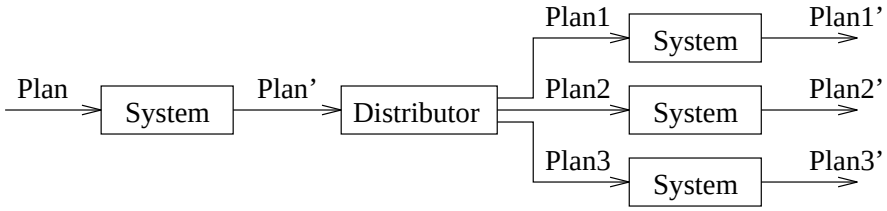

Figure 14: The scheduling hierarchy

Not any distribution should be allowed. The distributor must always make sure that no matter what schedules are chosen by the subsystems, their intersection is non-empty and is a subset of the schedule of the received plan.

value
/∗ Allowed total schedules for a system and all its subsystems ∗/
possible_TotalSC: Sys × SC → SC-**set**
possible_TotalSC(s,mainsc) ≡ CA_allowed_SCs(Sys_TotalCA(s),mainsc),

/∗ Possible intersections of generated subsystem schedules, given
the schedules distributed ∗/
inter_SubSCs: (Sys $\overrightarrow{m}$ SC) → SC-**set**
inter_SubSCs(sm) ≡
{ SCs_inter(**rng** sm′) | sm′:Sys $\overrightarrow{m}$ SC •
dom sm = **dom** sm′ ∧
∀ s:Sys • s ∈ **dom** sm ⇒ sm′(s) ∈ possible_TotalSC(s,sm(s))
},

/∗ Intersection of a set of schedules ∗/
SCs_inter: SC-**set** → SC
SCs_inter(scs) ≡
{ tf | tf:TF • ∃ sc:SC • sc ∈ scs ∧ tf ∈ sc }
axiom
/∗ Intersections of schedules generated by subsystems are non−empty
and are subsets of the main schedule ∗/
∀ d:Dist, sc:SC •

sc ∈ inter_SubSCs(Dist_SysSCs(d)) ⇒
(sc≠empty_SC ∧ sc ⊆ Dist_inSC(d))

— **A scheduling example**

An example of a simple scheduling task may help understanding how the distributed scheduling works.

A simple scheduling task could be selection of a specific point in two dimensions.

type
Point :: x:**Real** y:**Real**
SC = Point-**set**

Here, a schedule is a set of points, containing exactly those points which adhere to the schedule.

The problem of selecting points may be distributed. One system may take care of choosing the x-coordinate while another chooses the y-coordinate.

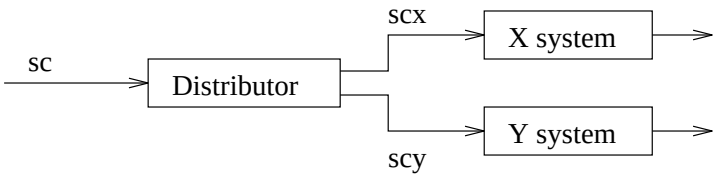

Figure 15: Scheduling example

The control areas of each system must be defined.

value
sysx,sysy : Sys

cax: Point × Point → **Bool**
cax(p,p′) ≡ x(p) = x(p′)

cay: Point × Point → **Bool**
cay(p,p′) ≡ y(p) = y(p′)
axiom
Sys_TotalCA(sysx) ≡ cax,
Sys_TotalCA(sysy) ≡ cay

The task may for instance be to select a point within an area limited by maximum and minimum values for x and y. The main schedule will be:

value
  mainsc : SC = { p | p:Point • min≤x(p)≤max ∧ min≤y(p)≤max }

This schedule may be distributed. For instance the following distribution could be made:

value
  scx : SC = { p | p:Point • min≤x(p)≤max }
  scy : SC = { p | p:Point • min≤y(p)≤max }

It can be proved that the distributor axioms hold. Slightly changed, this can be expressed as:

∀ sc:SC •
  sc ∈ { sx ∩ sy | sx,sy:SC •
    sx ∈ {{p | p:Point • x(p) = xs} | xs:**Real-set** • ok_set(xs)} ∧
    sy ∈ {{p | p:Point • y(p) = ys} | ys:**Real-set** • ok_set(ys)}
  } ⇒ (sc ≠ {} ∧ sc ⊆ mainsc)

where ok_set is defined as:

value
  ok_set: **Real-set** → **Bool**
  ok_set(rs) ≡ rs≠{} ∧ ∀ r:**Real** • r ∈ rs ⇒ min≤r≤max

This can be simplified as:

∀ sc:SC •
  (∃ xs,ys:**Real-set** • ok_set(xs) ∧ ok_set(ys) ∧
    sc = {p | p:Point • (x(p),y(p))=(xs,ys) }) ⇒
    (sc ≠ {} ∧ sc ⊆ mainsc)

Which can be seen to hold.

— Possible traffics

When a scheduling system decides on a plan, the information from the received plan is usually not enough. Information will also be needed on what is actually happening on the real rail net. Schedules should always be adjusted, so that it is possible for the traffic on the real rail net to be on schedule.

The scheduling systems get information from sensors on the net and communication with the workers, engine men, station men etc. on the rail net. The exact state of the rail net will probably not be known to any scheduling system, but it is possible to describe a set of traffics that contain the real traffic of the rail net. This set of possible traffics is a part of the state of a scheduling system. The set is used when selecting schedules.

value
  Sys_posTFs: Sys → TF-**set**

  Sys_posSCs: Sys → SC-**set**
  Sys_posSCs(s) ≡
    { sc | sc:SC • ∼disruption(Sys_posTFs(s),sc) }
axiom
  /∗ It must be possible for the real traffic to be on schedule ∗/
  ∀ s:Sys • Sys_outSC(s) ∈ Sys_posSCs(s)

— Control area borders

As described, a number of scheduling systems take part in the scheduling of traffic. The network must be split into control areas for these systems. For instance, there will usually be a number of train dispatch centres. Each centre takes care of a specific area of the network.

Problems may arise when scheduling is done for areas which are on the border between two scheduling systems. For instance, there will be areas on the border between two signalling centres. In these areas, it should be clear which signalling centre manages which areas. That is, the control areas of the signalling centres should not intersect.

An area may be needed between two such systems. When, for instance, a train travels from the control of one signalling system to the control of another, it may be necessary to have an area between these systems, in which the train may adjust from one schedule to another. In this area, that movement of the train may only be limited by the restrictions made on the train from the two control areas. For instance, the train should leave one control area at a certain time, and enter the other at another point in time.

Inside the mentioned border areas, trains may for instance be under the control of a train dispatch centre, that handles both control areas. You could instead just have a simple static plan describing how travel should occur when not inside the control area of any signalling centre.

The same kind of problems may occur when splitting the network into control areas for the train dispatch centres.

5.2.2 Requirements for Scheduling Systems

There are several requirements that should hold for all scheduling systems. These include some of the main decisions on how to run the train traffic. In this section, we will propose such requirements.

— Schedule Quality

We postulate a function for comparing the quality of two schedules:

value
  better_SC: SC × SC × Sys → **Bool**
axiom
  ∀ sc,sc′,sc″:SC, s:Sys •
    ∼better_SC(sc,sc,s) ∧
    better_SC(sc,sc′,s) ⇒ ∼better_SC(sc′,sc,s) ∧
    (better_SC(sc,sc′,s) ∧ better_SC(sc′,sc″,s)) ⇒ better_SC(sc,sc″,s)

This function gives a (partial) ordering of schedules for any scheduling system. This ordering will reflect some of the main decisions made by the company running traffic on the rail net.

It may not be easy to specify the quality function. It depends on very many aspects, and in some cases it may actually change over time. That is, new decisions are made, prices on fuel, trains or tickets change and so on. We will here consider the function static, though.

There are many criteria to be considered when running trains: Avoiding collisions, respecting timetables, making journies comfortable, minimising fuel consumption etc. The priority between these factors may not be easy to formalise. Avoiding train collisions will usually be more important than making the journey comfortable. That is, if the train could avoid a collision by violating the conditions for making the journey comfortable, this would be preferred. However, it may for instance not be easy to decide if fuel consumption or comfortable journies is more important. This may depend very much on the exact situation as well as the general policy of the company running the trains. If this kind of decisions were to be made by the scheduling systems, we would have to formalise the problem (and the decided requirements and priorities).

A simple solution could be based on the decision that avoiding train collisions is always the most important factor, then respecting timetables/plans and then all other qualities. These priorities will be used here.

— Train Collisions

Avoiding train collisions is considered a very important quality of a schedule. At least, we should say that any schedule containing only traffics with collisions is worse than schedules that contain traffics without collisions.

value
  better_SC_wrt_collisions: SC × SC → **Bool**

  SC_collision: SC → **Bool**
  SC_collision(sc) ≡ ∀ tf:TF • tf ∈ sc ⇒ ∼TF_no_collisions(tf)
axiom
  ∀ sc,sc′:SC •
    SC_collision(sc) ∧ ∼SC_collision(sc′) ⇒ better_SC_wrt_collisions(sc,sc′)

However, we cannot for sure say that one collision is better than two. This may depend on the nature of these collisions.

We could also express quality relations between schedules that do not include collisions. For instance, being close to a collision is not desirable.

— Quality wrt. Plans

In the last section, we said that a scheduling system must always select an allowed schedule. That is, it should always obey the rules of the received plan. Considering schedule quality, this may not be a good decision however. For instance, avoiding train collisions should be regarded as more important than adhering to plans and timetables.

We will decide, that an allowed plan is better than a not allowed plan. That is, if possible, we should obey the rules of the incoming plan.

value
  better_SC_wrt_plans: SC × SC × S → **Bool**
axiom
  ∀ sc,sc′:SC, s:Sys •
    sc ∈ Sys_plannedSCs(s) ∧ sc′ ∉ Sys_plannedSCs(s) ⇒
      better_SC_wrt_plans(sc,sc′,s)

Two schedules that are not allowed may also be compared. It would be preferred that such schedules are “close to” an allowed schedule.

— Comfortable Journies

One would want train journies to be as comfortable as possible for the passengers. For instance, sudden breaks or accelerations should be avoided.

```
value
  TF_acceptable_Acc: TF → Bool
  TF_acceptable_Acc(tf) ≡ ∀ t:T • RS_acceptable_Acc(tf(t)),

  RS_acceptable_Acc: RS → Bool
  RS_acceptable_Acc(rs) ≡
    ∀ tn:Tn • tn ∈ RS_Tns(rs) ⇒
      min_Acc ≤ obs_TS_Acc(RS_TS(rs,tn)) ≤ max_Acc,

  min_Acc, max_Acc: Real
```

Here, *min_Acc* and *max_Acc* are the minimum and maximum accelerations that should be used for trains. Another rule for making train journies comfortable could be the introduction of a maximum speed when passing curves.

We postulate a quality relation between schedules wrt. the comfort of the journey.

```
value
  better_SC_wrt_comfort: SC × SC → Bool
```

— Other Qualities

Lots of other qualities could be expressed as well. We have now defined quality as a relation on schedules, depending on the scheduling system. This is probably not enough however. To really describe the quality relation, a lot of other information would be needed. This may for instance include political and economic considerations. We will not in these requirements include these aspects.

— Priorities

As mentioned earlier, we may postulate a priory between the different kinds of qualities of schedules. We may for instance use these priorities:

1. Avoiding train collisions
2. Adhering to plans/timetables
3. Making journies comfortable

This decision will effect the definition of the quality relations.

axiom
```
  /* 1. Priority: Collisions */
  ∀ sc,sc′:SC, s:Sys •
  better_SC_wrt_collision(sc,sc′)
    ⇒ better_SC(sc,sc′,s),

  /* 2. Priority: Plans */
  ∀ sc,sc′:SC, s:Sys •
  (∼better_SC_wrt_collision(sc′,sc) ∧
   better_SC_wrt_plan(sc,sc′,s)
  ) ⇒ better_SC(sc,sc′,s),

  /* 3. Priority: Comfort */
  ∀ sc,sc′:SC, s:Sys •
  (∼better_SC_wrt_collision(sc′,sc) ∧
   ∼better_SC_wrt_plan(sc′,sc,s) ∧
   better_SC_wrt_comfort(sc,sc′)
  ) ⇒ better_SC(sc,sc′,s)
```

— **Best Schedules**

The specified partial ordering on schedules reflect the management decisions to consider when scheduling train traffic. A requirement for scheduling systems could for instance be that from the possible and allowed schedules, a "best" schedule should always be selected by the system. Note that this requirement does not tell us exactly which schedule to select, as the quality ordering is only partial. That is, there may be several "best" schedules.

value
```
  /* Find the set of best schedules for a system */
  Sys_bestSCs: Sys → SC-set
  Sys_bestSCs(s) ≡
    let ps = Sys_possibleselectedSCs(s) in
      { sc | sc:SC • sc ∈ ps ∧
```

```
      ~∃ sc':SC • sc' ∈ ps \ {sc} ∧ better_SC(sc',sc,s)
      }
  end

/* The set of all schedules that may be selected by a system */
Sys_possibleselectedSCs: Sys → SC-set
Sys_possibleselectedSCs(s) ≡
    Sys_disruptionfreeSCs(s) ∩ Sys_describableSCs(s),

/* The set of all disruptionfree schedules */
Sys_disruptionfreeSCs : Sys → SC-set
Sys_disruptionfreeSCs(s) ≡
    { sc | sc:SC • ~disruption(Sys_posTFs(s),sc) },

/* The set of all schedules that can be described by a generated
    plan of a given system */
Sys_describableSCs: Sys → SC-set
```

We introduce the requirement, that a scheduling system must always choose a "best" schedule.

axiom

$\forall$ s:Sys • Sys_outSC(s) $\in$ Sys_bestSCs(s)

In the following sections, we will describe the individual scheduling systems and their requirements.

5.2.3 Rail Net Development & Maintenance

We describe the requirements for the computerised support for Rail Net Development & Maintenance.

— Synopsis

By rail net development we mean the design, simulation, building and installation (or removal) of new (respectively old) station and line facilities. By maintenance we mean the temporary suspension of rail units and their subsequent resumption.

The Rail Net Development & Maintenance System shall allow the simultaneous development and maintenance of an indefinite number of rail net fragments. Included in this is the ability to experiment with (i.e. simulate) various rail net design alternatives.

— Rail Net Scheduling

The Rail Net Development & Maintenance System is a scheduling system. That is, all the axioms stated for scheduling systems should hold for Rail Net Development Systems as well.

type
  rndSys = {| s:Sys • is_rndSys(s) |}
value
  is_rndSys: Sys → **Bool**

Note however, that the Rail Net Development & Maintenance System does not control the entire rail net system. The control area of a Rail Net Development System must only control the possible rail nets and not the movement of trains on the net.

type
  /* Net History */
  NH = T → N
value
  is_rndCA: CA → **Bool**
  is_rndCA(ca) ≡
    ∀ tf,tf′:TF • TF_proj_NH(tf) = TF_proj_NH(tf′) ⇒ ca(tf,tf′),

  TF_proj_NH: TF → NH
  TF_proj_NH(tf) **as** nh

post ∀ t:T • nh(t) = TF_N(tf,t)
axiom
∀ s:Sys •
is_rndSys(s) ⇒ is_rndCA(Sys_CA(s))

— State Identification

We speak of two kinds of states: the real state and the image state. The real state is that of the status of the real world components: the rail net, the timetables, the schedules, the traffic, bookings, reservations, etc. The image state is an abstraction of the real state and may not necessarily contain all the sub-components of the real state.

System state identification is an art! It is also a very important step in both domain analysis and in requirements capture.

The Rail Net Development & Maintenance state should contain all state components of a scheduling system. Besides the task of scheduling, the Rail Net Development & Maintenance System must also support the simulation of rail net systems.

The state of the Rail Net Development & Maintenance System shall therefore have a number of components:

1. inPlan: The received scheduling plan:

   The scheduling plan describing general rules that should be obeyed by the scheduled traffics.

2. outPlan: The plan generated by the Rail Net Development System:

   The plan for rail net developments. This includes the intended future changes to the rail net.

3. simPlan: The plan for simulated rail nets:

   A virtual plan used for simulating and experimenting with the rail net and possible future changes to the net.

4. TFs: The possible traffics:

   The existing, current possible traffics of the rail net. This part of the state will be updated each time information is received about the real rail net.

type
mainPlan, rndPlan

value
rndSys_inPlan: rndSys → mainPlan,
rndSys_outPlan: rndSys → rndPlan,
rndSys_simPlan: rndSys → rndPlan

Appropriate state invariants must be regularly checked by the Rail Net Development & Maintenance System.

— Rail Net Development Plans

A specification of Rail Net Development Plans can be made. For instance the following description could be used:

type
rndPlan = rndTFdesc-**set**,
rndTFdesc = T $\overrightarrow{m}$ N
value
/* The semantics of a Rail Net Development Plan */
rndPlan_SC: rndPlan → SC
rndPlan_SC(rndplan) ≡
{ tf | tf:TF • ∃ rndtf : rndTFdesc •
rndtf ∈ rndplan ∧ TF_sat_rndTFdesc(tf,rndtf)
},

/* A traffic satisfies the conditions of a rndTFdesc */
TF_sat_rndTFdesc: TF × rndTFdesc → **Bool**
TF_sat_rndTFdesc(tf,rndtf) ≡
∀ t,t′:T •
(t≤t′ ∧ t ∈ **dom** rndtf ∧
∼∃ t″:T • t<t″≤t ∧ t″ ∈ **dom** rndtf) ⇒
TF_N(tf,t′) = rndtf(t)

— State Input and Update

Once the relevant state has been identified — that is: that part of the real state which is 'also' to be represented [by the image state] inside the computer — the development questions arise: how to input that state, and how often (and how) to update it? When state components are input and updated we need validate ('vet') that appropriate data has been obtained, and (verification) that the data values obtained satisfy appropriate constraints.

The Rail Net Development & Maintenance System assumes an initialisation and delineation subsystem whose purpose it is to record (express) any form of rail net: that is, the input, vetting and validation of data that sufficiently describes aspects of appropriate rail net, sufficient for the purposes of Rail Net Development & Maintenance. The input rail nets may be that of the real-time rail net, of any subset thereof, of rail nets that are new wrt. the real rail net, etc.

The Rail Net Development & Maintenance System, as part of this, also assumes that the current state, that is the state for suitably chosen time points, of all recorded real rail net units properly reflects the physical state of these units.

Functions can be made for updating the state. These functions may be used when observations are made.

```
value
  /* Update the possible traffics when observing a train route */
  TFs_update_TR: TF-set × T × Tn × TR → TF-set
  TFs_update_TR(tfs,t,tn,tr) ≡
    { tf | tf:TF • tf ∈ tfs ∧
      tn ∈ TF_Tns(tf,t) ∧ TF_TR(tf,tn,t) = tr
    },

  /* Update the possible traffics when observing the network */
  TFs_update_N: TF-set × T × N → TF-set
  TFs_update_N(tfs,t,n) ≡
    { tf | tf:TF • tf ∈ tfs ∧ TF_N(tf,t) = n }
```

More to be written

— State Output

Some parts of the information stated in the generated plan should be used for making changes to the real rail net.

More to be written

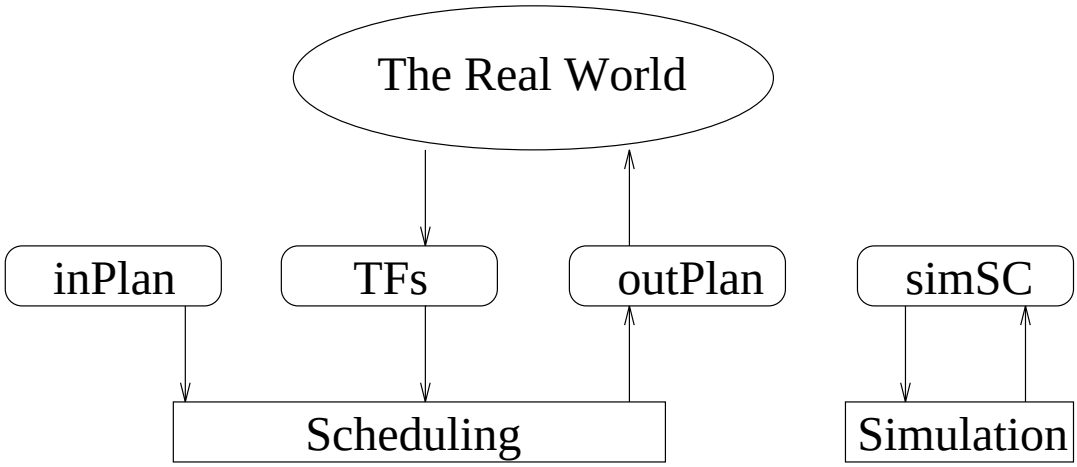

Figure 16: The Rail Net Development and Maintenance System

— **Design & Simulation Support**

Several functions are useful when designing and simulating networks.

A set of units can be added to a network.

value
  N_add_Us: N × U-**set** $\xrightarrow{\sim}$ N
  N_add_Us(n,us) **as** n′
  post N_added_Us(n,n′,us)
  pre N_wf_add_Us(n,us),

  N_wf_add_Us: N × U-**set** → **Bool**
  N_wf_add_Us(n,us) ≡
    obs_N_Us(n) ∩ us = {} ∧
    ∃ n′:N • N_added_Us(n,n′,us),

  N_added_Us: N × N × U-**set** → **Bool**
  N_added_Us(n,n′,us) ≡
    N_unchanged_Ls(n,n′) ∧ N_unchanged_Ss(n,n′) ∧
    obs_N_Us(n′) = obs_N_Us(n) ∪ us,

  N_unchanged_Us: N × N → **Bool**
  N_unchanged_Us(n,n′) ≡ obs_N_Us(n) = obs_N_Us(n′),

N_unchanged_Ss: N × N → **Bool**
N_unchanged_Ss(n,n′) ≡ obs_N_Ss(n) = obs_N_Ss(n′),

N_unchanged_Ls: N × N → **Bool**
N_unchanged_Ls(n,n′) ≡ obs_N_Ls(n) = obs_N_Ls(n′)

A new line can be added to a network. The line may only consist of units that are already in the network.

value
N_add_L: N × L ~→ N
N_add_L(n,l) **as** n′
post N_added_L(n,n′,l)
pre N_wf_add_L(n,l),

N_wf_add_L: N × L → **Bool**
N_wf_add_L(n,l) ≡
  l ∉ obs_N_Ls(n) ∧
  ∃ n′:N • N_added_L(n,n′,l),

N_added_L: N × N × L → **Bool**
N_added_L(n,n′,l) ≡
  N_unchanged_Us(n,n′) ∧ N_unchanged_Ss(n,n′) ∧
  obs_N_Ls(n′) = obs_N_Ls(n) ∪ {l}

A station can also be added.

value
N_add_S: N × S ~→ N
N_add_S(n,s) **as** n′
post N_added_S(n,n′,s)
pre N_wf_add_S(n,s),

N_wf_add_S: N × S → **Bool**
N_wf_add_S(n,s) ≡
  s ∉ obs_N_Ss(n) ∧
  ∃ n′:N • N_added_S(n,n′,s),

N_added_S: N × N × S → **Bool**
N_added_S(n,n′,s) ≡

N_unchanged_Us(n,n′) ∧ N_unchanged_Ls(n,n′) ∧
obs_N_Ss(n′) = obs_N_Ss(n) ∪ {s}

Also, it is possible to remove units, lines and stations.

value
N_remove_Us: N × U-**set** → N
N_remove_Us(n,us) **as** n′
post N_removed_Us(n,n′,us)
pre N_wf_removed_Us(n,us),

N_wf_removed_Us: N × U-**set** → **Bool**
N_wf_removedUs(n,us) ≡
us ⊆ obs_N_Us(n) ∧
∃ n′:N • N_removed_Us(n,n′,us),

N_removed_Us: N × N × U-**set** → **Bool**
N_removed_Us(n,n′,us) ≡
N_unchanged_Ls(n,n′) ∧ N_unchanged_Ss(n,n′) ∧
obs_N_Us(n′) = obs_N_Us(n)\us,

N_remove_L: N × L → N
N_remove_L(n,l) **as** n′
post N_removed_L(n,n′,l)
pre N_wf_removed_L(n,l),

N_wf_removed_L: N × L → **Bool**
N_wf_removed_L(n,l) ≡
l ∈ obs_N_Ls(n) ∧
∃ n′:N • N_removed_L(n,n′,l),

N_removed_L: N × N × L → **Bool**
N_removed_L(n,n′,l) ≡
N_unchanged_Us(n,n′) ∧ N_unchanged_Ss(n,n′) ∧
obs_N_Ls(n′) = obs_N_Ls(n)\{l},

N_remove_S: N × S → N
N_remove_S(n,s) **as** n′
post N_removed_S(n,n′,s)
pre N_wf_removed_S(n,s),

N_wf_removed_S: N × S → **Bool**
N_wf_removed_S(n,s) ≡
s ∈ obs_N_Ss(n) ∧

$\exists$ n′:N • N_removed_S(n,n′,s),

N_removed_S: N × N × S → **Bool**
N_removed_S(n,n′,s) ≡
  N_unchanged_Us(n,n′) ∧ N_unchanged_Ls(n,n′) ∧
  obs_N_Ss(n′) = obs_N_Ss(n)\{s}

More to be written

— Installation & Maintenance Support

The Rail Net Development & Maintenance generates a plan, *outPlan*. This plan describes future changes to the rail net due to maintenance and development. It will be distributed to for instance the train dispatch centres. For constructing this plan, the specified functions for manipulating networks can be used.

Several qualities could be checked when generating the development schedule. For instance, at any time, units which are not part of lines or stations should be managed closed.

value
  N_unused_Us_managed_closed: SC × T → **Bool**
  N_unused_Us_managed_closed(sc,t) ≡
    ∀ tf:TF • tf ∈ sc ⇒
    let n = TF_N(tf,t) **in**
      managed_closed_Us(obs_N_Us(n)\N_L_Us(n)\N_S_Us(n))
    end

Note that this condition also ensures that units removed from the net are always managed closed.

Many other conditions on the constructed Development Schedules could be checked as well.

— Miscellaneous

TO BE WRITTEN

5.2.4 Train Dispatch

— Synopsis

The Train Dispatch System shall support the dispatch of trains according to schedules that have been derived from timetables and the rail net, and as constrained by the current stations shunting and marshalling plans. The dispatch includes (i) sending trains off from station platforms onto lines (but not the station management of routing within stations), (ii) monitoring train arrivals, and, in case train arrivals are late wrt. the operative schedule, (iii) the scheduling and rescheduling of trains.

— Train Dispatch System Scheduling

The Train Dispatch System is a scheduling system. Train Dispatch Systems control the running of train on the net. The systems should not however control rail net routes or states.

```
type
  tdSys = {| s:Sys • is_tdSys(s) |},
  /* Train History */
  TH = T → TP
value
  is_tdSys: Sys → Bool,

  is_tdCA: CA → Bool
  is_tdCA(ca) ≡
    ∀ tf,tf′:TF • TF_proj_TH(tf) = TF_proj_TH(tf′) ⇒ ca(tf,tf′),

  TF_proj_TH: TF → TH
  TF_proj_TH(tf) as th
  post ∀ t:T • th(t) = trns(tf(t))
axiom
  ∀ s:Sys •
    is_tdSys(s) ⇒ is_tdCA(Sys_CA(s))
```

— State Identification

After some analysis we arrive at the following required state components:

1. inPlan: The received scheduling plan

The plan, for instance received from a Rail Net Maintenance Centre.

2. outPlan: The generated plan

   The scheduling plan made be the Train Dispatch Centre.

3. TT: The Current Timetable

4. TFs: The Set of Possible Traffics

type
  tdPlan
value
  tdSys_inPlan: tdSys → rndPlan,
  tdSys_outPlan: tdSys → tdPlan,
  tdSys_TT: tdSys → TT

— State Input & Update

We refer to the previous subsection's (subsection 5.2.3) similarly titled paragraphs.

More to be written

— State Output

TO BE WRITTEN

— Support of Train Dispatch Operations

The following are some of the many train dispatch operations:

1. Running Map Construction & Display:

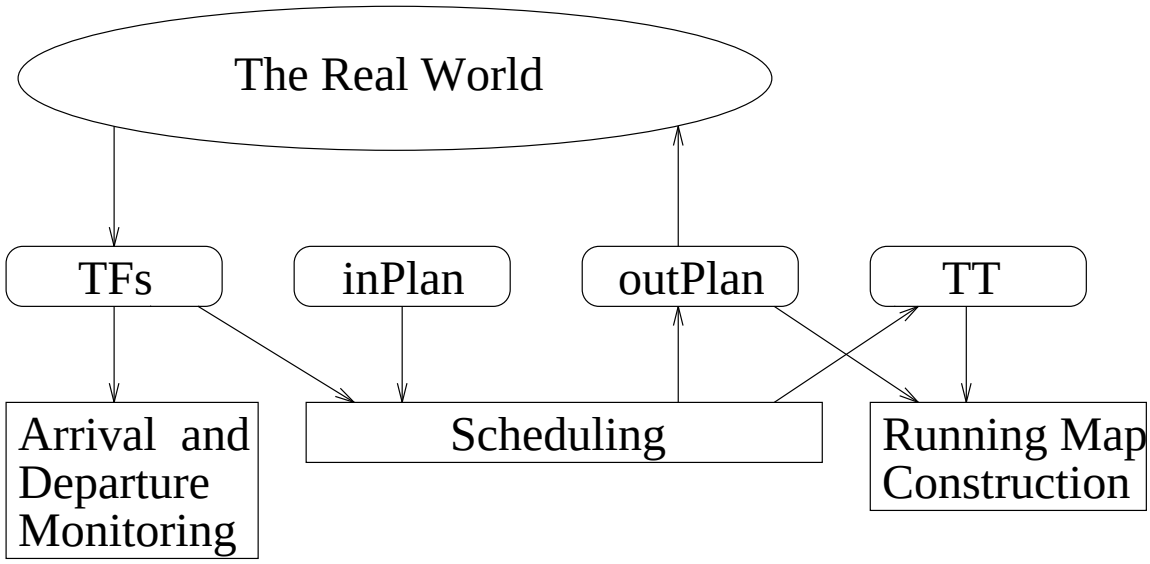

Figure 17: The Train Dispatch System

More to be written

2. Train Arrival and Departure Monitoring:

   It is possible to determine if the arrival or departure of a train at a station has been observed at a given point in time.

   A train has been observed arriving at a station at time t, if the train is known to be at the station at this time, and the last known position of the train is outside the station. Likewise for train departures.

```
value
  /* The arrival of a train at a station has been observed */
  Tn_Sn_arrival: Tn × Sn × TF-set × T → Bool
  Tn_Sn_arrival(tn,sn,tfs,t) ≡
    ∃ t′:T •
    Tn_at_Sn(tn,sn,tfs,t) ∧
    Tn_not_at_Sn(tn,sn,tfs,t′) ∧
      ∀ t″:T • t′<t″<t ⇒
        ~Tn_at_Sn(tn,sn,tfs,t″) ∧ ~Tn_not_at_Sn(tn,sn,tfs,t″),

  /* The departure of a train from a station has been observed */
  Tn_Sn_departure: Tn × Sn × TF-set × T → Bool
  Tn_Sn_departure(tn,sn,tfs,t) ≡
    ∃ t′:T •
```

```
      Tn_not_at_Sn(tn,sn,tfs,t) ∧
      Tn_at_Sn(tn,sn,tfs,t′) ∧
      ∀ t″:T • t′<t″<t ⇒
        ~Tn_at_Sn(tn,sn,tfs,t″) ∧ ~Tn_not_at_Sn(tn,sn,tfs,t″),

  /* Examine if a train is known to be at a station */
  Tn_at_Sn: Tn × Sn × TF-set × T → Bool
  Tn_at_Sn(tn,sn,tfs,t) ≡
    ∀ tf:TF • tf ∈ tfs ⇒ TF_Tn_at_Sn(tf,tn,sn,t),

  /* Examine if a train is known not to be at a station */
  Tn_not_at_Sn: Tn × Sn × TF-set × T → Bool
  Tn_not_at_Sn(tn,sn,tfs,t) ≡
    ∀ tf:TF • tf ∈ tfs ⇒ ~TF_Tn_at_Sn(tf,tn,sn,t),

  /* In a traffic a train is at a station at a given point in time */
  TF_Tn_at_Sn: TF × Tn × Sn × T → Bool
  TF_Tn_at_Sn(tf,tn,sn,t) ≡
    sn ∈ TF_Sns(tf,t) ∧ tn ∈ TF_Tns(tf,t) ∧
    TR_at_S(TF_TR(tf,tn,t),TF_S(tf,sn,t))
```

3. Train Dispatch:

   For a train to be dispatched from a station, the train must be known to be at the station.

```
  value
    Tn_Sn_dispatch_allowed: Tn × Sn × TF-set × T → Bool
    Tn_Sn_dispatch_allowed(tn,sn,tfs,t) ≡ Tn_at_Sn(tn,sn,tfs,t)
```

   It is possible to determine when a train should be dispatched, according to the current Dispatch Schedule.

```
  value
    /* For a train at a station at a given time, determine all departure
       times for the train, that makes the traffic on schedule, assuming the
       train does not leave the station before this departure time */
    SC_departure_Ts: Tn × Sn × TF-set × SC × T → T-set
    SC_departure_Ts(tn,sn,tfs,sc,t) ≡
      { t′ | t′:T •
        ∃ tf:TF • tf ∈ tfs ∩ sc ∧
          ~TF_Tn_at_Sn(tf,tn,sn,t′) ∧
          ∀ t″:T • t≤t″<t′ ⇒ TF_Tn_at_Sn(tf,tn,sn,t″)
      }
```

The actual dispatch of trains may be done by the Train Dispatch System alone or by the station managers.

4. Delay Detection & Analysis:

   At any time it is possible to check whether traffic is known to be on schedule or known not to be on schedule.

   One can also examine, if a given train is on schedule or not.

```
type
  TSH = T → TS
value
  /* Train is known not to be on schedule */
  Tn_not_on_SC: Tn × TF-set × SC → Bool
  Tn_not_on_SC(tn,tfs,sc) ≡
    ~∃ tf, tf′:TF • tf ∈ tfs ∧ tf′ ∈ sc ∧
      TF_TSH(tf,tn) = TF_TSH(tf′,tn),

  TF_TSH: TF × Tn → TSH
  TF_TSH(tf,tn) as tsh
  post ∀ t:T • tsh(t) = trns(tf(t))(tn)
```

   The observation that traffic is not on schedule may give rise to rescheduling. It is however a management decision if rescheduling is needed or not. The Train Dispatch System should therefore be able to observe and display delays in train traffic.

5. Rescheduling & Running Map Re–Construction:

   Scheduling and rescheduling of train traffic is handled by the Train Dispatch System.

   Sometimes it may also be necessary to create a new timetable, TT. Note that the schedule of the generated plan, outPlan, should always be a subset of the schedule of the timetable. That is, any traffic on schedule will satisfy the timetable.

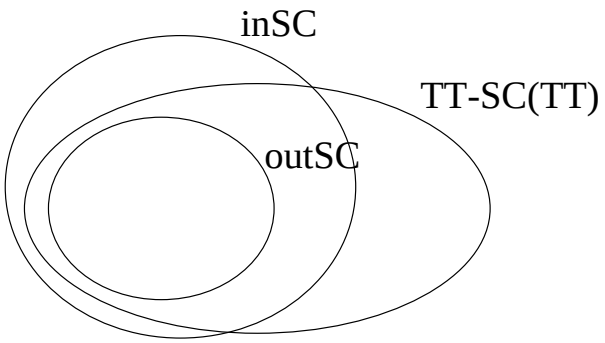

Figure 18: The Train Dispatch Schedule

axiom
∀ tds: tdSys •
Sys_outSC(tds) ⊆ TT_SC(tdSys_TT(tds))

Functions are needed for rescheduling support.

When it is decided that a rescheduling is needed, a possibly infinite number of reschedulings could be made.

value
possible_tdSCs: TF-**set** × SC → SC-**infset**
possible_tdSCs(tfs,rndsc) ≡
{ sc | sc:SC • sc ⊆ rndsc ∧ tfs ∩ sc ≠ {} }

A problem is choosing between this large number of possible new schedules. This decision will probably not be carried out be the Dispatch System alone. The System could be of help in deciding an appropriate new schedule through.

One could for instance want a new schedule that from some time, t, was identical to the old schedule. That is, for any traffic of the new schedule, there was a traffic of the old schedule such that the two traffics are identical from time t and visa versa.

value
SC_identical_from_T: SC × SC × T → **Bool**
SC_identical_from_T(sc,sc′,t) ≡
∀ tf:TF •
tf ∈ sc ⇒
∃ tf′:TF • tf′ ∈ sc′ ∧ TF_identical_from_T(tf,tf′,t) ∧
tf ∈ sc′⇒
∃ tf′:TF • tf′ ∈ sc ∧ TF_identical_from_T(tf,tf′,t),

TF_identical_from_T: TF × TF × T → **Bool**
TF_identical_from_T(tf,tf′,t) ≡
∀ t′:T • t′≥t ⇒ tf(t)=tf(t′)

6. Statistics Gathering:
7. *&c.*

More to be written

— New Functionalities

TO BE WRITTEN

— Miscellaneous

TO BE WRITTEN

5.2.5 Signalling

— **Synopsis**

The Signalling System supports the setting of routes and unit states. This includes opening and closing of units and routes of the rail net. The system should also make sure that certain safety conditions are met to avoid train collisions.

— **Signalling System Scheduling**

The Signalling System is a scheduling system. Signalling systems may only control the setting of physical and managed states of units.

```
type
  sigSys = {| s:Sys • is_sigSys(s) |},
  /* Path set history */
  PsH = T → P-set
value
  is_sigSys: Sys → Bool,

  is_sigCA: CA → Bool
  is_sigCA(ca) ≡
    ∀ tf,tf′:TF •
      (TF_proj_phy_PsH(tf),TF_proj_man_PsH(tf)) =
      (TF_proj_phy_PsH(tf′),TF_proj_man_PsH(tf′)) ⇒ ca(tf,tf′),

  TF_proj_phy_PsH: TF → PsH
  TF_proj_phy_PsH(tf) as psh
  post ∀ t:T • psh(t) =
    { p | p:P •
      ∃ u:U • u ∈ obs_N_Us(TF_N(tf,t)) ∧ p ∈ obs_U_Physical_Σ(u)
    },

  TF_proj_man_PsH: TF → PsH
  TF_proj_man_PsH(tf) as psh
  post ∀ t:T • psh(t) =
    { p | p:P •
      ∃ u:U • u ∈ obs_N_Us(TF_N(tf,t)) ∧ p ∈ obs_U_Managed_Σ(u)
    }
axiom
  ∀ s:Sys •
    is_sigSys(s) ⇒ is_sigCA(Sys_CA(s))
```

— State Identification

The following state components will be needed:

1. inPlan: The received scheduling plan

   The plan, for instance received from a Train Dispatch Center.

2. outPlan: The generated plan

   The scheduling plan made by the Signalling Centre.

3. TFs: The Set of Possible Traffics

type
  sigPlan
value
  sigSys_inPlan: sigSys → tdPlan,
  sigSys_outPlan: sigSys → sigPlan

— State Input & Update

TO BE WRITTEN

— State Output

TO BE WRITTEN

— Supported Signalling Functions

1. Line Blocking:

2. Station Route Setting:

   The setting of station routes should be done according to the current Train Dispatch Schedule and the current possible traffics of the rail net. That is, at any time, the Signalling Schedule should be a subset of the Dispatch Schedule and should include at least one possible traffic.

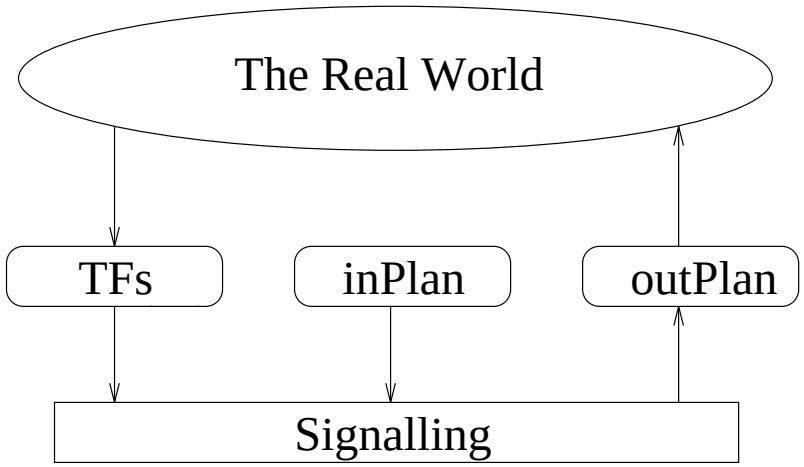

Figure 19: The Signalling System

value
  possible_sigSCs: TF-**set** × SC → SC-**set**
  possible_sigSCs(tfs,tdsc) ≡
    { sc | sc:SC • sc ⊆ tdsc ∧ tfs ∩ sc ≠ {} }

For any station name, one can find the possible histories of that station. This is a partial function from time to the state of the station. The function is only defined for times when the station is part of the network.

type
  SH = T $\xrightarrow{\sim}$ S
value
  TF_SH: TF × Sn → SH
  TF_SH(tf,sn) **as** sh
  post ∀ t:T • sn ∈ TF_Sns(tf,t) ⇒ sh(t) = TF_S(tf,sn,t)

The setting of station routes should be done according to the intended station history. The intended history will be one that is allowed by the current schedule and possible, according to the set of possible traffics of the net.

value
  SC_TFs_SHs: SC × TF-**set** × Sn → SH-**set**
  SC_TFs_SHs(sc,tfs,sn) ≡ TFs_SHs(sc ∩ tfs,sn),

  TFs_SHs: TF-**set** × Sn → SH-**set**
  TFs_SHs(tfs,sn) ≡ { TF_SH(tf,sn) | tf:TF • tf ∈ tfs }

The actual used station history is a decision of the station management.

3. Safety control:

   The Signalling System should support the checking of safety conditions for the rail net.

   The units of a managed open route of a network should never contain more than one train.

```
value
  TF_safe_Open_Rts: TF → Bool
  TF_safe_Open_Rts(tf) ≡
    ∀ t:T, rt:Rt •
      rt ∈ Managed_Open_N_Rts(TF_N(tf,t)) ⇒
        safe_Open_Rt(rt,tf,t),

  safe_Open_Rt: Rt × TF × T → Bool
  safe_Open_Rt(rt,tf,t) ≡
    ~∃ tn,tn′:Tn • tn≠tn′ ∧ {tn,tn′} ⊆ TF_Tns(tf,t) ∧
      Rt_Disj(TF_TR(tf,tn,t),TF_TR(tf,tn′,t),TF_N(tf,t))
```

   This ensures that if trains stay within the open routes of a network, train crashes will not occur.

   The Signalling System should not make it impossible to violate the safety conditions, but it should warn the net managers if the rail net is at any time not safe.

More to be written

— **New Functionalities**

TO BE WRITTEN

— **Miscellaneous**

TO BE WRITTEN

5.2.6 Train Control

— Synopsis

Each train, when riding from one station to a next station is running according to a train plan. The plan indicates train speed, possible line stretches of acceleration or deceleration, expected or possible stops (at signals along lines, or within larger station areas), etc. The Train Control System shall support a number of on board functions that range from fully automatic to semi-automatic train control — the latter in support of the train engine man. The Train Control System will receive guidance from neighbouring station management, line equipment and possibly the engine man.

— Train Control System Scheduling

The Train Control System is a scheduling system. Train Control Systems control the trains of the rail net.

type
  tcSys = {| s:Sys • is_tcSys(s) |}
value
  is_tcSys: Sys → **Bool**,

  is_tcCA: CA → **Bool**
  is_tcCA(ca) ≡
    ∀ tf,tf′:TF • TF_proj_TH(tf) = TF_proj_TH(tf′) ⇒ ca(tf,tf′)
axiom
  ∀ s:Sys •
    is_tcSys(s) ⇒ is_tcCA(Sys_CA(s))

— State Identification

The state consists of:

1. inPlan: The received scheduling plan

   The scheduling plan, for instance received from a Signalling System.

2. outPlan: The generated plan

   The scheduling plan made by the Train Control System.

3. TFs: The set of Possible Traffics

type
  tcPlan
value
  tcSys_inPlan: tcSys → sigPlan,
  tcSys_outPlan: tcSys → tcPlan

— **State Input & Update**

TO BE WRITTEN

— **State Output**

TO BE WRITTEN

— **Support of Train Functions**

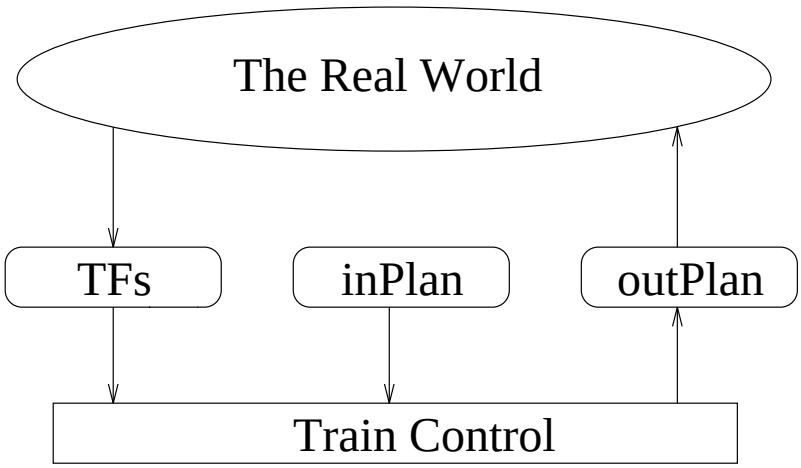

Figure 20: The Train Control System

Not all train schedules are equally "good". Many different quality criteria should be taken into account. Trains may use different quantities of fuel or power depending on how it is run. Also, one would want the journey to be as comfortable as possible for the passengers. For instance, sudden breaks or accelerations should be avoided.

1. :
2. :
3. *&c.*

5.2.7 Rolling Stock: Monitoring & Control

— Synopsis

TO BE WRITTEN

— State Identification

TO BE WRITTEN

— State Input & Update

TO BE WRITTEN

— State Output

TO BE WRITTEN

— Support of Rolling Stock M&C Functions

1. Rolling Stock Inventory:
2. Train Body Composition:
3. *&c.*

More to be written

— New Functionalities

TO BE WRITTEN

— Miscellaneous

TO BE WRITTEN

5.2.8 Marshalling

— Synopsis

The Marshalling System shall support the creation of marshalling plans and their execution: the monitoring and control of the marshalling yard.

— State Identification

TO BE WRITTEN

— State Input & Update

TO BE WRITTEN

— State Output

TO BE WRITTEN

— Supported Marshalling Operations

1. Marshalling Plan Construction:
2. Incoming Train: Monitoring & Control:
3. Marshalling – Monitoring & Control:
4. Outgoing Train Monitoring & Control:
5. *&c.*

More to be written

— **New Functionalities**

TO BE WRITTEN

— **Miscellaneous**

TO BE WRITTEN

5.2.9 Passenger Reservation & Ticketing

— Synopsis

The Passenger Reservation & Ticketing System will handle passenger queries, tickets, reservations etc.

— State Identification

The state of the Passenger Reservation & Ticketing System will consist of these components:

1. TT: The Current Time Table
2. Ress: The set of all reservations made

From the state of the Passenger Reservation & Ticketing System the state components can be extracted.

type
  pratsState, Res
value
  obs_pratsState_TT: pratsState → TT,
  obs_pratsState_Ress: pratsState → Res-**set**

— Tickets and Reservations

A ticket can be described as the set of passenger histories allowed by the ticket.

type
  Ticket = PH-**infset**

A passenger holding a number of tickets is allowed to be in any state allowed by the tickets. A set of tickets can be descriped as a single ticket. That is, the ticket descriping passenger histories allowed by the set of tickets.

value

```
combine_Tickets: Ticket-set → Ticket
combine_Tickets(ts) ≡
  { ph | ph:PH • ∀ t:T • ∃ tk:Ticket •
    tk ∈ ts ∧ Ticket_PS_allowed(tk,ph(t),t) },

Ticket_PS_allowed: Ticket × PS × T → Bool
Ticket_PS_allowed(tk,ps,t) ≡
  ∃ ph:PH • ph ∈ tk ∧ ph(t) = ps
```

Some of the passenger histories allowed by a set of tickets may be reserved by other passengers holding seat reservations. A reservation contains information of the histories reserved. No passenger history can be reserved by two distinct reservations though.

```
value
  obs_Res_PHs: Res → PH-set
axiom
  /* No passenger history is reserved more than once */
  ∀ ps:pratsState, r1,r2:Res •
    r1≠r2 ∧ {r1,r2} ⊆ obs_pratsState_Ress(ps) ⇒
      obs_Res_PHs(r1) ∩ obs_Res_PHs(r2) = {}
```

From the state of the Passenger Reservation & Ticketing System, one can find the set of all passenger histories reserved.

```
value
  pratsState_resPHs: pratsState → PH-set
  pratsState_resPHs(ps) ≡
    { ph | ph:PH • ∃ r:Res •
      r ∈ obs_pratsState_Ress(ps) ∧ ph ∈ obs_Res_PHs(r)
    }
```

It is possible to find all allowed passenger histories of a passenger holding a set of tickets and a set of reservations. A passenger is allowed to have any history that is allowed by the tickets, except those that are reserved by other passengers.

```
value
  /* Find the allowed histories of a passenger carrying a given set
      of tickets and reservations */
  allowed_PHs: Ticket-set × Res-set × pratsState → PH-set
  allowed_PHs(ts,rs,ps) ≡
```

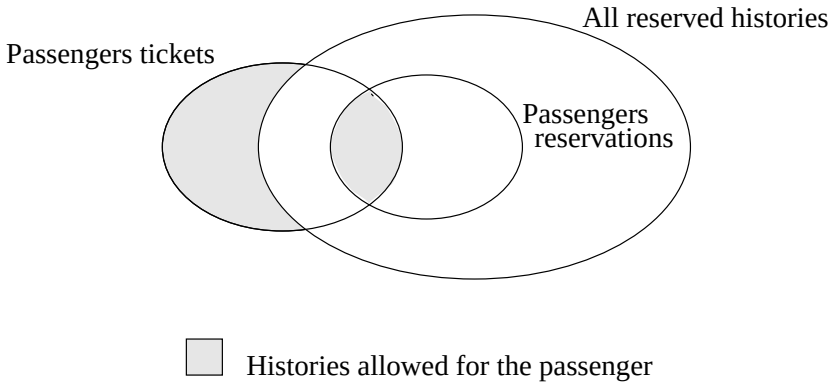

Figure 21: Allowed passenger histories

```
    { ph | ph:PH • ph ∈ combine_Tickets(ts) ∧
        (ph ∉ pratsState_resPHs(ps) ∨
         ∃ r:Res • r ∈ rs ∧ ph ∈ obs_Res_PHs(r))
    }
pre rs ⊆ obs_pratsState_Ress(ps)
```

— State Input & Update

TO BE WRITTEN

— State Output

TO BE WRITTEN

— Supported Reservation & Ticketing Functions

1. Reservations:

   Passengers can buy seat reservations. Reservations can only be made for not already reserved passenger histories.

   ```
   value
     possible_Res: Res × pratsState → Bool
     possible_Res(r,ps) ≡
       obs_Res_PHs(r) ∩ pratsState_resPHs(ps) = {}
   ```

When a reservation is bought, the state of the Passenger Reservation & Ticketing System should be updated accordingly.

value
new_Res: Res × pratsState $\xrightarrow{\sim}$ pratsState

It will probably not be possible to buy any reservation. For instance, the history of not being in the rail net at all can of course not be bought. The decision of which reservations are sold is a decision of the rail net system management.

2. Reservation Change & Cancellation:

Reservations may be cancelled by removing them from the state of the Passenger Reservation & Ticketing System.

value
cancel_Res: Res × pratsState $\xrightarrow{\sim}$ pratsState

3. Passenger Ticketing:

4. Journey Queries:

The system must be able to answer questions from passengers. A query could be: Does a train run from a given station to another, departing at a given point in time and arriving at another given time? (according to the timetable)

type
Tn_Journey′ :: trn:Tn fromSn:Sn depT:T toSn:Sn arrT:T,
Tn_Journey = {| tj:Tn_Journey′ • arrT(tj)≥depT(tj) |}
value
/∗ Train runs from a station to another, departing and arriving at
given points in time ∗/
Tn_journey_query: TT × Tn_Journey → **Bool**
Tn_journey_query(tt,tj) ≡
let mk_Tn_Journey′(tn,sn,t,sn′,t′) = tj **in**
Tn_Sn_departure(tn,sn,tt,t) ∧ Tn_Sn_arrival(tn,sn′,tt,t′) ∧
{ t″ | t″:T • t≤t″≤t′ } ⊆ Tn_Ts(tn,tt)
end,

/∗ The set of times where a trains is known to be in the rail net ∗/
Tn_Ts: Tn × TF-**set** → T-**set**
Tn_Ts(tn,tfs) ≡
{ t | t:T • ∀ tf:TF • tf ∈ tfs ⇒ tn ∈ TF_Tns(tf,t) }

Another query could be: Is it possible to travel from a station to another, departing and arriving at certain points in time, possibly using several trains for the journey?

```
type
  Journey′ = Tn_Journey*,
  Journey = {| j:Journey′ • wf_Journey(j) |}
value
  wf_Journey: Journey′ → Bool
  wf_Journey(j) ≡
    j ≠ ⟨⟩ ∧
    ∀ i:Nat • {i,i+1} ⊆ inds j ⇒
      arrT(j(i))<depT(j(i+1)) ∧ toSn(j(i))=fromSn(j(i+1)),

  dep_arr_query: TT × Sn × T × Sn × T → Bool
  dep_arr_query(tt,sn,t,sn′,t′) ≡
    dep_arr_possible_journies(tt,sn,t,sn′,t′) ≠ {},

  dep_arr_possible_journies: TT × Sn × T × Sn × T → Journey-set
  dep_arr_possible_journies(tt,sn,t,sn′,t′) ≡
  { j | j:Journey •
      fromSn(hd j)=sn ∧ depT(hd j)=t ∧
      toSn(j(len j))=sn′ ∧ arrT(j(len j))=t′ ∧
      journey_query(tt,j)
  },

  /* Journey is possible according to the timetable */
  journey_query: TT × Journey → Bool
  journey_query(tt,j) ≡
    ∀ i:Nat •
      ({i,i+1} ⊆ inds j ⇒
        { t | t:T • arrT(j(i))≤t≤depT(j(i+1)) } ⊆ Sn_Ts(toSn(j(i)),tt)
      ) ∧
    i ∈ inds j ⇒ Tn_journey_query(tt,j(i)),

  /* The set of times where a station is known to be in the rail net */
  Sn_Ts: Sn × TF-set → T-set
  Sn_Ts(sn,tfs) ≡
    { t | t:T • ∀ tf:TF • tf ∈ tfs ⇒ sn ∈ TF_Sns(tf,t) }
```

Other queries could be: What is the fastest journey from one station to another, or what is the cheapest journey?

5. *&c.*

More to be written

— **New Functionalities**

TO BE WRITTEN

— **Miscellaneous**

TO BE WRITTEN

6 Computing Systems Architecture

From any one of the set of requirements definition, given in section 5, we can now identify alternative computing systems architectures. Each of these consists of hardware and software functionalities, that is: hard and soft concepts & facilities. Co-ordinating across the set of requirements and their architectural alternatives we might choose among overall architectures that can all support the full set of required computing systems:

1. Conventional, Centralised Main Frame Computing:

   TO BE WRITTEN

2. Conventional, Decentralised Client/Server Computing:

   TO BE WRITTEN

3. Combinations of Above + Massively Parallel Computing + Fault Tolerance:

   TO BE WRITTEN

Which of the choices to select will be the result of individual and comparative throughput & dependability performance modelling.

In the following we assume choice number 2: Conventional, Decentralised Client/Server Computing.

6.1 Hardware Systems Architecture

TO BE WRITTEN

6.2 Software Systems Architecture

TO BE WRITTEN

6.2.1 The State Repositories

TO BE WRITTEN

6.2.2 The Engines

TO BE WRITTEN

The Command Interface & HCI Engine[s]

TO BE WRITTEN

The Visualisation Engine[s]

TO BE WRITTEN

The State Engine[s]

TO BE WRITTEN

The Communication Engine[s]

TO BE WRITTEN

The Compute Engine[s]

TO BE WRITTEN

Other Engines

TO BE WRITTEN

6.2.3 Miscellaneous Issues

TO BE WRITTEN

7 Program Organisation

To be written

7.1 Dependability Measures

To be written

7.2 Process Decompositions

To be written

7.2.1 Processes

To be written

7.2.2 Connectors

To be written

7.2.3 Glues & Ports

To be written

7.2.4 Miscellaneous Process Issues

To be written

7.3 Internal Data Structures

To be written

8 Conclusion

A Full Formal Models

A.1 Formal Domain Model

A.1.1 Rail Net

A.1.2 Timetables

A.1.3 Schedules

A.1.4 Traffic

A.1.5 Rescheduling

A.1.6 Resources & Allocations

A.1.7 Shunting and Marshalling

A.1.8 Station Management

A.1.9 Customer Services

A.2 Requirements Models

A.2.1 Rail Net Development & Maintenance

A.2.2 Train Dispatch

A.2.3 Train Control

A.2.4 Signaling

A.2.5 Rolling Stock: Monitoring & Control

A.2.6 Marshalling

A.2.7 Passenger Reservation & Ticketing

A.2.8 Freight Handling

A.3 Systems Architecture

A.3.1 Hardware Architecture

A.3.2 Software Architecture

A.4 Program Organisation

B Terminology

Connector: A connector is a further undefined entity.

Pragmatics: *Connectors delineate units and serve to identify (label etc.) points where units may be joined. At most two units may share a connector, and they are then said to be connected at that point.*

Used by: *Path, unit.*

Line: A line is a sequence of linear units.

Pragmatics: *Lines connect stations.*

Used by: *Network.*

Uses: *Linear unit.*

Linear unit: A linear unit is identified by a pair of ('opposite end') connectors.

Pragmatics: *The linear unit allows four states: closed, open in one direction, open in the opposite direction, and open in both directions.*

Used by: *Line.*

Uses: *Unit, connector.*

Marshalling: The decomposition and composition of train bodies within a marshalling yard.

Pragmatics: .

Used by: .

Uses: *Marshalling yard, train body.*

Marshalling plan: A plan describing a marshalling.

Pragmatics: .

Used by: .

Uses: *Marshalling.*

Marshalling yard: A special part of a station used for marshalling.

Pragmatics: .

Used by: *Marshalling.*

Uses: *Marshalling, station.*

Network: A network consists of sets of stations, lines and other units.

Pragmatics: *A network designates the physical layout of lines, stations and units.*

Used by: .

Uses: *Unit, station, line.*

Passenger: .

Pragmatics: .

Used by: *Rail state.*

Uses: .

Path: A path is a pair of connectors.

Pragmatics: *Paths are defined by units. Paths designate a direction of travel through a unit.*

Used by: *State, unit.*

Uses: *Connector, unit.*

Rail state: The state of all trains and passengers at some point in time.

Pragmatics: .

Used by: *Traffic.*

Uses: *Train, passenger.*

Real traffic: A traffic.

Pragmatics: *The actual traffic happening in the real world.*

Used by: .

Uses: *Traffic.*

Rescheduling: The action of changing the current schedule for train traffic.

Pragmatics: .

Used by: .

Uses: *Schedule.*

Route: A sequence of connectors.

Pragmatics: *A route designates a possible sequence of paths through the units of a network.*

Used by: .

Uses: *Connector.*

Schedule: A set of traffics.

Pragmatics: *A schedule designates the traffics that are considered on schedule.*

Used by: *Scheduled traffic, rescheduling.*

Uses: .

Scheduled traffic: A traffic.

Pragmatics: *A scheduled traffic is any traffic on schedule wrt. a schedule.*

Used by: .

Uses: *Schedule.*

Station: A station contains a set of tracks and a set of other units.

Pragmatics: .

Used by: *Network.*

Uses: *Unit, track.*

Timetable: A description of a schedule.

Pragmatics: *A timetable may for instance descible points in time where given trains are*

to be at given points within the network.

Used by: .

Uses: *Train.*

Traffic: The rail state at any point in time.

Pragmatics: *The position and other information about all trains, passengers etc. at any point in time.*

Used by: .

Uses: *Rail state.*

Track: A set of connected linear units.

Pragmatics: *Connected linear units within a station, where trains may stop and load/unload goods and passengers.*

Used by: *Station.*

Uses: *Linear unit.*

Train: .

Pragmatics: .

Used by: *Timetable, rail state.*

Uses: .

Train body: A sequence of train cars.

Pragmatics: *A train body is a collection of cars, put together in a particular order.*

Used by: .

Uses: *Train car.*

Train car: .

Pragmatics: .

Used by: *Train body.*

Uses: .

Unit: A unit is a "smallest" piece of rail net. Units are either linear units, switch (junction or point 'machine') units, crossover units, or other.

Pragmatics: .

Used by: *Connector, linear unit, network, station, line.*

Uses: .

C Type Name Explanations

Typename	Page	Explanation
C	15	Connector
CA	73	Control Area (TF $\times$ TF $\to$ **Bool**)
Dist	75	Distributor
Freight	59	Measurements and other info needed to make a schedule
FreightId	59	Freight Identifier
FreightInfo	59	Prices, minimum/maximum deliverytime, ...
Freights	63	Pieces of Freight
FreightSc	59	Freight Schedule
FreightSystem	59	FreightId, Freigt, FreightSc and FreightInfo
FT	63	Freight Train
FTn	59	Freight Train Name (Tn)
FreightTraffic	59	Where is a specific piece of freight at a certain time
Journey	116	Travelling from one place to another (Tn_Journey*)
L	25	Line
Location	62	A location is either at a station or in a FreightTrain
LostFreight	65	Contains the lost freight
mainPlan	88	
MD	47	Marshalling Description (MS-**set**)
MP	47	Marshalling Plan
MR	31	Managed Rail Net (T $\xrightarrow[m]{}$ N)
MS	47	Marshalling State
MY	45	Marshalling Yard
N	20	Net
NH	87	Net History (T $\to$ N)
Ω	17	State Space (Σ-**set**)
P	17	Path
PH	51	Passenger History (T $\xrightarrow[m]{}$ PS)
Place	62	Either Sn or FTn
Plan	55	The plan for composing a certain train from the RollingStock
PlanId	55	Identifier for a Plan
Pn	50	Passenger Identifier
Pos	62	Position - e.g. GPS
PP	50	Passenger Info (Pn $\xrightarrow[m]{}$ PS)
pratsState	113	Passenger Reservation & Tickening System State
Proj	73	Projection onto a Control Area (TF-**set**)
PS	50	Passenger State
PsH	102	Path-set History (T $\to$ P-**set**)

Typename	**Page**	**Explanation**
Res	113	Set of Reservations made
Reservoir	56	Reservoir containing the wagons extracted from RS and not yet transferred to another system
rndPlan	72	Rail Net Development Plan
rndSys	87	Rail Net Development System
rndTFdesc	89	Rail Net Development Traffic description (T $\overrightarrow{m}$ N)
RollingStock	52	Rolling Stock
RS	33	Rail Net State
rsPlan	56	Rolling Stock Plan
Rt	22	Route (C-**set**)
S	25	Station
Σ	17	State (P-**set**)
SC	38	Schedule (TF-**set**)
SH	104	Station History (T $\rightarrow$ S)
sigPlan	72	Signalling Plan
sigSys	102	Signalling System
Sn	28	Station Name
Sys	74	Scheduling System
T	31	Time
TB	47	Train Body (W-**set**)
tcPlan	107	Train Control Plan
tcSys	106	Train Control System
tdPlan	72	Train Dispatch Plan
tdSys	95	Train Dispatch System
TF	33	Traffic (T $\overrightarrow{m}$ RS)
TH	95	Train History (T $\rightarrow$ TP)
Ticket	113	Ticket (PH-**infset**)
Tn	33	Train Name
Tn_Journey	116	Train Journey
TP	33	Train Position (Tn $\overrightarrow{m}$ TS)
TR	29	Train Route (Route)
Trk	25	Track
TS	33	Train State
TSH	99	Train State History (T $\rightarrow$ TS)
TT	39	Time Table
U	15	Unit
W	47	Wagon
Wtype	52	WagonType - fx. Oil-, Car-, Passenger-Wagon, locomotive ...

D Bibliographical Notes

Index