Modelling a Distributed Railway Control System

Morten Skjoldborg Madsen & Martin Møller Bæk

Master of Science Thesis
Kongens Lyngby 2005
IMM-DTU

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

This thesis concerns the development of a distributed control system for a simple railway line. Control systems exist to ensure safety of trains by preventing events like derailments and collisions.

Formal development methods and specification languages can increase the correctness of software systems. These methods are essential to the development of safety critical systems where human lives are at stake. Therefore a formal method is applied to the development in this thesis.

A formal model, using the RAISE specification language (RSL), of a distributed control system for railway lines is developed. The formal specification language is used to ensure correctness and safety of the system. The model is separated in modules so a clear separation of the static, dynamics, and control properties is obtained.

The model is constructed with provability of safety in mind. Proof obligations are sketched and the theory of how to prove safety properties in the model is briefly described. A single informal proof of one proof obligation is performed.

The model is refined through a number of steps. This is done by first specifying an abstract applicative model which then is refined to a concrete version. The concrete model is transformed to an imperative version.

The imperative model is implemented in the JAVA programming language. The result is a generic simulator which can take a configuration (a railway line structure) as input and simulate trains operating on this line. A configuration editor is developed to ease the construction of new railway configurations.

The developed model is fairly complex compared to other formally developed models since it also concerns time issues. These complicate the model by adding a considerably larger state space to the model. Events like collisions and braking distances become major issues in the development.

Keywords: formal specification, railway lines, control systems, JAVA, XML, simulation, safety, RAISE.

Resumé

Denne rapport omhandler udviklingen af et distribueret styresystem til en simpel jernbane. Styresystemers opgave er at sørge for togenes sikkerhed ved at forhindre visse situationer såsom afsporing og kollisioner.

Formelle metoder til udvikling og specifikation kan forøge korrektheden af software systemer. Disse metoder er essentielle i udviklingen af systemer, hvor sikkerheden er i højsædet, fordi menneskeliv er involveret. Derfor er formelle metoder brugt i dette projekt.

En formel model af et sådant styresystem er udviklet ved at bruge RSL (RAISE specification language). Det formelle specifikationssprog er brugt til at sikre at systemet er sikkert og korrekt implementeret. Modellen er opdelt i moduler, der adskiller de statiske, dynamiske og sikkerhedsmæssige egenskaber af systemet.

Modellen er konstrueret på en måde, så det er muligt at bevise sikkerheden af systemet. Bevisforpligtelser er skitseret, og teorien for, hvordan sikkerheden bevises, er beskrevet kort. En enkel bevisforpligtelse er bevist uformelt.

Modellen er trinvist forfinet. Først er en abstrakt applikativ model specificeret. Dernæst er modellen gjort konkret, og tilsidst er den tranformeret til en imperativ model.

Den imperative model er implementeret i programmeringssproget JAVA. Resultatet er en generisk simulator, der tager en konfiguration (strukturen af en jernbane) som input og simulerer togene på jernbanen. Desuden er der udviklet et værktøj til at lave nye konfigurationer.

Den udviklede model er ret kompleks i forhold til andre formelle modeller, da den også behandler tidsaspektet. Dette komplicerer modellen, da tilstandsrummet bliver forholdsvis stort. Situationer såsom kollisioner og bremselængder er væsentlige emner i den udviklede model.

Nøgleord: formel specifikation, jernbaner, styresystemer, JAVA, XML, simulation, sikkerhed, RAISE.

Preface

This paper is written to document the master thesis project *Modelling a Distributed Railway Control System - Formal Methods for Software Development*. The master thesis starts the third of January 2005 and is handed in the first of August 2005. The project is performed at the department *Computer Science and Engineering (CSE)* at the institute *Informatics and Mathematical Modelling (IMM)* at *Technical University of Denmark (DTU)* in Lyngby.

The project is supervised by Associate Professor, Ph.D. Anne E. Haxthausen.

Contents

1 Introduction **21**

1.1 Background . . . 21

1.2 Project motivation . . . 21

1.3 Formal methods and safety . . . 22

1.3.1 In general . . . 22

1.3.2 RSL and levels of formality . . . 23

1.4 Thesis objectives . . . 24

1.4.1 Thesis concept diagram . . . 25

1.5 The railway domain . . . 26

1.5.1 Railways in general . . . 26

1.5.2 Control / safety Systems . . . 26

1.5.3 Railway accidents . . . 27

1.5.4 Railways today . . . 27

2 Thesis overview **29**

3 Informal domain description **33**

3.1 Railway line . . . 34

3.2 Segment . . . 34

3.2.1 Static properties . . . 34

3.3 End station area (ESA) . . . 34

3.3.1 Static properties . . . 35

3.3.2 Dynamic properties . . . 35

3.4 Train . . . 35
3.4.1 Static properties . . . 35
3.4.2 Dynamic properties . . . 36
3.5 Junction / point . . . 36
3.5.1 Static properties . . . 36
3.5.2 Dynamic properties . . . 37
3.6 Crossing . . . 37
3.6.1 Dynamic properties . . . 37
3.7 Switch Box . . . 38
3.7.1 Static properties . . . 38
3.8 Sensor . . . 39
3.8.1 Dynamic properties . . . 39
3.9 Stations . . . 39
3.10 Signals . . . 40
3.11 An example of a railway line . . . 40
3.12 Single lines . . . 41

4 Main idea and concept **43**
4.1 A simple railway line . . . 43
4.2 Control system . . . 44
4.2.1 The reservation concept . . . 44
4.2.2 Reservations and braking . . . 45

5 Engineering concepts **47**
5.1 Sensors . . . 47
5.2 Reservation and brake points . . . 48
5.3 Time modelling . . . 49
5.4 Safety . . . 50
5.5 Deadlock . . . 51
5.6 Livelock . . . 52

6 Control system requirements **53**

6.1 Safety requirements 53
6.2 Functional requirements 54

7 Simulator requirements 57
7.1 Train simulator requirements 57
7.1.1 Formal model requirements 57
7.1.2 Visual parts 57
7.2 Switch off the control system 58
7.3 Railway line configurations 58
7.3.1 Create configurations 58
7.3.2 Export/import configurations 59
7.3.3 Load configuration 59

8 Model structure 61

9 Physical design 63
9.1 Static network modelling 64
9.2 Train positions 64
9.3 ESAs 65
9.4 Crossings 65
9.5 Points 66

10 Train dynamics analysis 67
10.1 Train / segment length 67
10.2 Collision modelling 67
10.3 Time and speed considerations 69
10.3.1 Collision detection 69
10.3.2 Brake point requirements 70
10.3.3 ESA requirements 71
10.3.4 Train max speed checking 71
10.3.5 Segment max speed checking 72
10.3.6 Acceleration considerations 72

11 Control system design **75**

11.1 Control system algorithm 75

11.1.1 Obtaining reservations 75

11.2 Train Control Computer algorithm 78

11.2.1 TCC - Speed checking 79

11.2.2 Clearing reservations 80

11.2.3 Reservations handling 80

11.3 Switch box algorithm 81

11.3.1 Sensor process 82

11.3.2 Message process 83

11.3.3 Handle TCC request message 84

11.3.4 Handling a line-branch request 85

11.3.5 Handle line-branch response 85

11.3.6 Handle SB dereservation message 86

11.3.7 Prepare process 87

12 Glossary **89**

13 GUI design **93**

13.1 Train simulator 93

13.2 Configuration editor 94

14 Assumptions and invariants **95**

15 RSL modelling method summary **97**

15.1 Initial specification 97

15.1.1 Initial model overview 98

15.1.2 Types 98

15.1.3 Statics 98

15.1.4 Dynamics 99

15.1.5 Control 101

15.2 Type decomposition 103

15.2.1 Decomposed model overview 103

15.2.2 Statics 104
15.2.3 Dynamics 105
15.2.4 Control 105
15.3 Concrete refinement 106
15.4 Imperative transformation 106
15.4.1 Statics 106
15.4.2 Dynamics 107
15.4.3 Control 107
15.5 Concurrent transformation 108

16 Initial Model 109

16.1 Initial model structure 109
16.2 From design to model 110
16.3 Types 111
16.3.1 Tick 111
16.3.2 Ends 111
16.3.3 Entity IDs 111
16.3.4 SB types 112
16.3.5 Crossing, point and sensor 112
16.3.6 Train position 112
16.3.7 Reservation 113
16.3.8 Messages 113
16.4 Statics 113
16.4.1 Type of interest 114
16.4.2 Observers 114
16.4.3 Derived observer 116
16.4.4 Wellformedness 116
16.5 Dynamics 122
16.5.1 Type of interest 122
16.5.2 Observers and generators 123
16.5.3 Updating the physical system 124

16.5.4 Derived observer and generators . . . 129
16.5.5 Wellformedness . . . 129
16.5.6 The safe predicate . . . 131
16.5.7 Initial requirement . . . 133
16.5.8 Observer/generator axioms . . . 135
16.5.9 Generator preserving wellformedness . . . 135
16.6 Control . . . 136
16.6.1 Type of interest . . . 136
16.6.2 Observers and generators . . . 136
16.6.3 Updating the control system . . . 138
16.6.4 Wellformedness . . . 141
16.6.5 The *consistent* predicate . . . 142
16.6.6 Initial requirement . . . 145
16.6.7 Observer/generator axioms . . . 147
16.6.8 Generator preserving wellformedness . . . 148

17 Decomposed model 149
17.1 Decomposed model structure . . . 149
17.2 Types . . . 149
17.3 Statics . . . 149
17.3.1 SBs . . . 151
17.3.2 Segs . . . 151
17.3.3 ESAs . . . 151
17.3.4 Trains . . . 151
17.4 Dynamics . . . 152
17.4.1 TrainDyn . . . 153
17.4.2 SBDyn . . . 153
17.5 Control . . . 153
17.5.1 TCC . . . 154
17.5.2 SBCC . . . 154
17.5.3 ComService . . . 154

17.6 Implementation relation 155
17.6.1 Types 155
17.6.2 Statics 155
17.6.3 Dynamics 156
17.6.4 Control 157

18 Concrete model 161
18.1 Types 161
18.2 Statics 161
18.2.1 SBs 161
18.2.2 Segs 162
18.2.3 ESAs 162
18.2.4 Trains 163
18.3 Dynamics 163
18.3.1 TrainDyn 163
18.3.2 SBDyn 164
18.4 Control 164
18.4.1 TCC 164
18.4.2 SBCC 164
18.5 Implementation relation 165
18.5.1 Types 165
18.5.2 Statics 165
18.5.3 Dynamics 165
18.5.4 Control 166

19 Imperative model 167
19.1 Types 167
19.2 Statics 167
19.2.1 SBs 168
19.2.2 Segs 168
19.2.3 ESAs 168
19.2.4 Trains 168

19.3 Dynamics 169
19.3.1 TrainDyn 169
19.3.2 SBDyn 169
19.4 Control 169
19.4.1 TCC 170
19.4.2 SBCC 170
19.5 Implementation relation 170

20 Implementing the simulator 171
20.1 Translating the model to JAVA 171
20.1.1 Schemes and objects 171
20.1.2 Basic types 172
20.1.3 Cartesian product types 172
20.1.4 Map types 173
20.1.5 Variant types 174
20.1.6 Case expressions 179
20.1.7 Preconditions 179
20.1.8 Axioms and predicates 180
20.1.9 Concurrency 183
20.1.10 Example: Model translation 184
20.2 JAVA program structure 186
20.2.1 The types package 187
20.2.2 The statics package 188
20.2.3 The dynamics package 189
20.2.4 The control package 190
20.2.5 The gui package 191
20.2.6 The editor package 193
20.2.7 The exceptions package 194
20.3 Translating configuration to XML 195
20.3.1 SBs 196
20.3.2 Segs 197

20.3.3 ESAs . . . 197
20.3.4 Trains . . . 198
20.4 Differences from RSL model . . . 199
20.4.1 Train re-request timing . . . 199

21 Using the simulator 201
21.1 Starting the simulator . . . 201
21.2 Playing train driver . . . 201
21.2.1 The railway line . . . 201
21.2.2 Buttons . . . 203
21.2.3 Control system . . . 203
21.2.4 Autodrive . . . 204
21.3 Creating new configurations . . . 204
21.3.1 Add new configuration . . . 204
21.3.2 Add a segment . . . 204
21.3.3 Delete a segment . . . 204
21.3.4 Add a train . . . 205
21.3.5 Delete a train . . . 205
21.3.6 Update properties . . . 205
21.3.7 Save a configuration . . . 206
21.3.8 Checking wellformedness . . . 206
21.3.9 Loading a configuration . . . 206
21.4 XML importing/export . . . 206

22 Test 207
22.1 Test configuration . . . 207
22.2 Test strategy . . . 207
22.2.1 Basic tests . . . 207
22.2.2 Performance tests . . . 208
22.3 Test listings . . . 208
22.3.1 Basic test listings . . . 208
22.3.2 Performance test listings . . . 211

23 Verification **213**

23.1 The idea of provability 213

23.1.1 The *safe* predicate 213

23.1.2 The *consistent* predicate 214

23.1.3 Preconditions (guards) 215

23.1.4 Wellformedness 216

23.1.5 The *init_req* predicate 217

23.1.6 Verifying control algorithm 217

23.2 Proof obligations 217

23.2.1 *[gen_wf_pres]* 218

23.2.2 *[gen_safe_pres]* 218

23.2.3 *[gen_consistent_pres]* 218

23.2.4 *[init_is_safe]* 218

23.2.5 *[init_is_consistent]* 218

23.3 Proof: *[init_is_safe]* 219

24 Ideas & future work **221**

24.1 Improved exceptions 221

24.2 Pipelining of trains 221

24.3 Complex networks 221

24.4 Automatic reservation- and brake points 222

24.5 Speed reduction before entering segment 222

24.6 Time tables and stations 223

24.7 Automatic train behavior 223

24.7.1 Time table based behavior 223

24.8 Ideas for concurrency 223

25 Related work **225**

25.1 Automatic translation from RSL to JAVA 225

25.2 Formal Development and Verification of a Distributed Railway Control System 225

25.3 Domain Specific Languages 226

25.3.1 Domain Specific language 227
25.3.2 Verifying safety 227
25.4 Modelling interlocking systems 227
25.4.1 Train dynamics 227
25.4.2 Verifying safety 227

26 Discussion 229
26.1 Predicates and preconditions 229
26.1.1 Predicates 229
26.1.2 Preconditions 229
26.2 A safe algorithm 230
26.2.1 Two trains collide 230
26.2.2 Collisions at a crossing 230
26.2.3 Derailing at a junction 231
26.2.4 External events 231

27 Conclusion 233
27.1 Summary of results 233
27.1.1 RSL model 233
27.1.2 Control system / algorithm 233
27.1.3 XML configuration language DTD 234
27.1.4 JAVA train simulator 234
27.1.5 JAVA configuration editor 234
27.2 Evaluation of results 235
27.2.1 Design method 235
27.2.2 Train dynamics analysis 235
27.2.3 Model 235
27.2.4 Verification 236
27.2.5 Modelling method 236
27.2.6 JAVA translation method 237

28 Tools used in this project 239

29 Bibliography **241**

A Design of GUI **245**

A.1 TrainSimulator . . . 245

A.2 Configuration builder . . . 246

B RSL method description **249**

B.1 Abstract applicative . . . 249

B.2 Type decomposition (optional) . . . 251

B.3 Concrete applicative . . . 252

B.4 Concrete imperative . . . 252

C XML DTD **255**

D Test images **257**

D.1 Collisions . . . 257

D.2 Derailings . . . 257

E Concurrency **261**

E.1 Concurrency in RSL . . . 261

E.2 Concurrency in JAVA . . . 261

E.3 Shared variables in RSL . . . 262

E.4 Shared variables in JAVA . . . 263

E.5 Channel communication in JAVA . . . 264

E.5.1 Socket communication . . . 264

E.5.2 Shared varibles . . . 264

E.5.3 Direct function call . . . 265

F RSL modules **267**

F.1 Initial model . . . 267

F.1.1 Types . . . 267

F.1.2 Statics . . . 270

F.1.3 Dynamics . . . 280

F.1.4 Control . . . 311

F.2 Decomposed model 340
F.2.1 Types 340
F.2.2 Statics 340
F.2.3 Dynamics 352
F.2.4 Control 378
F.3 Concrete model 403
F.3.1 Types 403
F.3.2 Statics 407
F.3.3 Dynamics 422
F.3.4 Control 445
F.4 Imperative model 465
F.4.1 Types 465
F.4.2 Statics 469
F.4.3 Dynamics 485
F.4.4 Control 508

Chapter 1

Introduction

This report is written as documentation for the master thesis project mentioned in *Preface* at page 5. The main goal of this project is to develop a model of a distributed control system for a railway line. The model is specified using the formal specification language RSL (RAISE[1] Specification Language). The model is later implemented as a graphical simulator written in the JAVA programming language.

It is a primary concern of this project to use formal specification and refinement in the development process.

1.1 Background

At *CSE, DTU* much work has been done in the field of modelling railways and different control systems. The creation of methods to ease and generalize this work has also been a goal. Associate Professor Anne E. Haxthausen has played a major role in this in cooperation with Professor Jan Peleska from the University of Bremen. Together they have published several papers on this topic. A number of students have done special projects or master thesis' in the field of modelling railways or control systems which were supervised by Anne. This project follows in the foot steps of this work and in particular of [1].

1.2 Project motivation

This project builds on the idea of a distributed control system for simple railway networks. A such system was modelled in [5] but never implemented in any way.

[1] Rigorous Approach to Industrial Software Engineering

In [6] a simple simulator for a basic interlocking system was modelled and implemented as a discrete event based simulator.

Both projects did not concern the issue of time and only targeted the actual changes in the state of the control system. The issue of time was not modelled in these projects because safety should be provable and time would only complicate matters further.

When a model puts aside the concept of time, issues, such as braking distances and collision detection, do not arise.

The idea of this project is to construct a system which deals with all of the issues mentioned above thus combine modelling physical and control aspects of the system. This also means that proving the system to be absolutely safe is out of the scope of this project, but the ideas of proof techniques (chapter 23) and some informal argumentation is done (section 26.2).

The model in this project has a fully independent physical module which models the behavior of actual trains driving on a track. Therefore a lot of other aspects comes into scope of this project. Instead of just considering IF a train may enter a track segment we also have to consider if the train can be able to brake BEFORE entering the segment.

These physical aspects make the elicitation of requirements to the system much more realistic.

1.3 Formal methods and safety

This section contains a general discussion of the benefits of formal methods in software development and where it can be beneficial. It is also explained how RSL is used in this project.

1.3.1 In general

A formal method is a mathematical systematic approach to software specification and development. A good software development process utilizing different forms of (semi formal) notation is sufficient for most software development.

But what if human lives are at stake? An example of this could be a control program for an air craft or some medical equipment. How can we ensure that this software is absolutely fail safe?

In the experience of the authors of this report, there is no such thing as being 100% certain in the real world. But formal methods can get us some of the way.

The question is how much time and effort / resources we want to spend to ensure correctness of a software system. There are many different levels of formality that can be applied to the development of software which increasingly

ensure confidence in the software system. But what is the use of spending many resources on verifying the software if the underlying hardware is subject to failure and instability?

These are questions which must be considered before any development is initiated.

1.3.2 RSL and levels of formality

As described in [3] the RAISE method and RSL can be used in many and flexible ways. In section 1.3.1 it is mentioned that different levels of formality can be applied to the development of a software system. Therefore development using RSL and the RAISE method is not bound to any static procedure or required actions.

In the most simple case RSL can be used as a very abstract requirement specification which then serves as a programming guideline and a contract for the developers of the system.

To take the formal development one step further, the abstract specification could be refined into a concrete specification and an implementation relation could be specified. An implementation relation specifies that a specification is an implementation of another specification.

A specification is only an implementation if all axioms from the other specification is true in this new specification. This relation could be reviewed without any actual formal justification or verification.

To further increase the level of formality the implementation relation between the different specifications could be justified starting with the most important ones leaving out all the trivial and well known modules.

As a last resort all modules could be justified to increase trust in the software reliability.

One critical step in the development though is the translation step. A translation involves translating the formal specification to a specific programming language. This translation process does not include any formalization and guarantee of correctness, unless the code is automatically generated from the specification using a previously proved translator.

It is essential for the RSL development, that strict procedures are enforced when translating the RSL into a specific programming language. Else all proofs and formality would seem pointless.

1.4 Thesis objectives

Control system Design a control system and an associated algorithm which ensures safety on a simple railway line. The entities and nature of the control system should be distributed. This means that no global state of the entire control system is stored in any one control entity.

RSL model Develop a mathematical model of the physical domain of a railway line and a control system which ensures safety. This model should be refined through a number of steps from an abstract applicative to a concrete imperative specification.

XML domain specific language Convert the data type of the physical railway configuration into a XML syntax definition (DTD). This DTD defines the syntax for XML structures which describes physical railway configuration in the considered domain. These XML files should be input to a generic simulator.

Implement simulator A generic simulator of the physical system and the control system is implemented in JAVA. By generic is meant that the simulator should be able to take a configuration as input in form of the XML structures mentioned above and therefore not have a static layout or entity set. The simulator should visualize the layout of the railway line, show the trains on the railway line, make it possible to change some of the dynamic physical parameters and make it possible to switch off the control system.

Configuration editor Construct a *configuration editor* that makes it possible to create and edit configurations of railway lines. These configurations should be used as input to the simulator.

1.4.1 Thesis concept diagram

Figure 1.1 shows a diagram of the entire system to be developed:

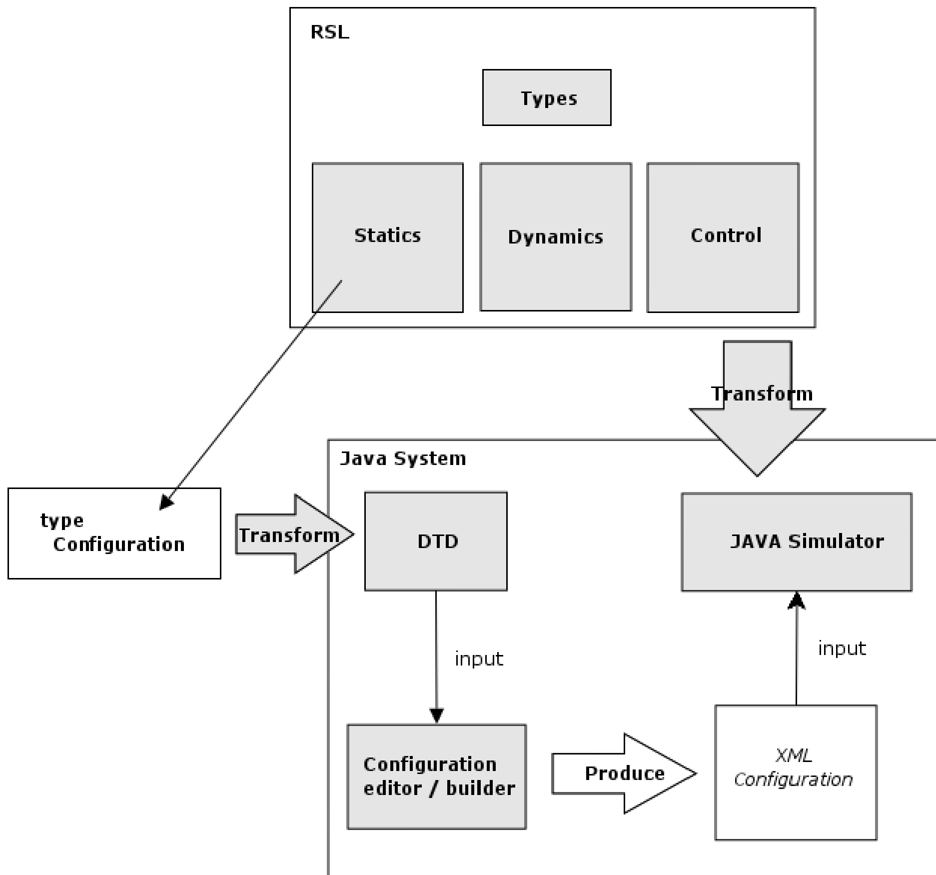

Figure 1.1: The entire project diagram

The thesis objectives described above are marked with gray background. These are:

1. RSL model.
2. XML configuration DTD (syntax definition).
3. Configuration editor / builder.
4. Java Simulator.

1.5 The railway domain

Here follows a brief overview of the railway domain considered in this thesis. For detailed description see chapter 3: "*Informal domain description*".

1.5.1 Railways in general

Railways (and trains) have since their invention been used for both transporting cargo / goods and passengers. Long railway lines (tracks) have been constructed throughout Europe for these purposes.

In Denmark exists fairly complex railway networks with lines coming from many destinations merging into one track using junctions and points that can switch between the incoming tracks. One example of this could be the S-train (Danish: S-tog) that covers most of the capital region of Zealand (Danish: Sjælland).

Simple railway networks

In this thesis we focus on small simple railway lines between two destinations. The railway line may be branching into two tracks but this is only for trains to pass each other in opposite directions (at intermediate stations). These small lines are characterized by having low intensity traffic and that the trains do not overtake each other.

A good example is the train running from Hillerød to Tisvildeleje. Another is the Nærum line in Lyngby. These are both private lines with low density traffic.

1.5.2 Control / safety Systems

In the following, the terms stated below will be used:

Safety system Does not directly control an entity but feeds the controller of an entity with information that helps to safely control the entity.

Control system Is a system which actively controls an entity and its actions / movement. This system also implements a safety system upon which the control system gets its data. Without safety such an automatic control system would be useless because it would require constant human observation.

Interlocking systems

The interlocking system is a very basic safety system which purpose is to prevent trains from colliding and derailing. A basic interlocking system operates on the

signals which guards each end of a *segment*. A segment is a partition of the railway track. By setting these signals appropriately, based on the locations of the involved trains the interlocking system ensures that only one train has exclusive access to a segment.

This system only sets the signals. Therefore it is essential that these signals are obeyed. If a train driver ignores the signals the system is practically useless and thereby only ensures safety if the associated procedure concerning the signals is followed. [4]

Automatic Train Control (ATC)

One of the most popular control systems for trains is the ATC system. This system automatically monitors and manages maximum velocity, stop at red signals and other safety related activities. Without getting into details the ATC system kicks in if the train driver violates the safety procedures of the railway.

Many variations of this system has been implemented utilizing only subsets of the ATC system due to the massive cost of implementing full ATC. [4] [11].

1.5.3 Railway accidents

Though train accidents are fairly rare when compared to car accidents a great deal of effort is put into development of train safety and control systems. One can argument that the cost of development is too high compared to the minimal effect in form of saved lives.

Research has shown that people are willing to accept greater risk when in control themselves (as in a car). Much higher demands are put on trains where people feel that they put their lives in the hands of another person or authority. Therefore to ensure popularity among passengers trains has to offer a high level of safety even if it really is unnecessary in some way.

On the other hand, when a train accident do occur, the loss of life and equipment damage is much higher than in a normal car accident. The cost to the society in the form of delayed traffic and repairs is also much greater and is therefore center of attention to the media which again makes the public very much aware of every single accident that occur. [11]

1.5.4 Railways today

Railways in Europe today have a large variety of control and safety systems. Until a few years ago there had not been introduced any form of standardization of automatic train control in the European countries. Due to this lack of standard nearly all countries have invented their own form of train control systems.

Therefore there would be problems if an international high intensity traffic line should be introduced on the existing railway networks.

ETCS - The European initiative

Therefore an European initiative has begun to standardize the concept of train control. The result is a European standard ETCS[2] which defines the requirements for the system.

The idea is that this system should be implemented without re-designing every interlocking system in Europe. Therefore the ETCS has been designed so it interfaces with the current interlocking / occupation management system. These interface requirements has been stated in so called TSI[3] documents.

All vendors that decide to produce ETCS equipment have to fulfill these specifications. [12]

Safety in Denmark

In Denmark several systems are implemented on the railways throughout the country. Some of these are ATC, ATP (a simplified ATC system), HKT[4], and a simplified HKT system.

The latest train accident in Denmark involved a collision between two trains at Lyngby station the 14th of Feb. 2005. A train approaching the station missed stopping at a red light and collided with a train that was waiting to depart from the station.

The interesting part in this, is that at Lyngby and northwards along the S-train rail only a simplified version of the HKT system is installed. Technical documents specify that the simplified HKT system does not ensure in all cases that a train is stopped in time if missing a red light.

As a consequence of this accident, full HKT is implemented from Hellerup and all the way to Holte station. This implementation should be finished summer 2005.

As of this date there is still implemented simplified HKT from Holte to Hillerød station. [11].

---

[2] European Train Control System
[3] Technical Specifications of Interoperability
[4] (Danish: Hastigheds Kontrol og Togstop) a Danish train velocity control and braking system

Chapter 2

Thesis overview

This chapter presents an overview of all chapters in this thesis. The chapters in this document are organized as the phases in this project. This means that the chapters in this report are ordered chronologically as the phases where executed.

Chapter 3: Informal domain description This chapter describes the domain of which we are to develop a control system - this domain is a simple railway line. Therefore this domain is described in lack of any form for control. All physical entities of the domain is described along with their physical state and possible events (state changes).

Chapter 4: Main idea and concept This chapter explains the main ideas of how to solve the problems which can arise on a simple railway line. The main concept of a control system and how this will work is sketched.

Chapter 5: Engineering concepts This chapter describes the different technologies and terms which are basic to this project.

Chapter 6: Control system requirements This chapter lays out the requirements to the control system. Solutions to these requirements are sketched. These are the ideas of the basic functionality of the control system. These solutions are later transformed to a control algorithm. This algorithm is described in chapter 11.

Chapter 7: Simulator requirements This chapter lays out the requirements to the actual graphical interface which is designed to interact with the model converted to JAVA.

Chapter 8: Model structure This chapter displays an illustration of the overall structure of the model. This figure only lists what information is grouped in the different modules of the model. The model mentions all entities that has been mentioned so far.

Chapter 9: Physical design This chapter describes the design decisions the physical domain in lack of control (the domain described in chapter 3). All design decisions and considerations are listed here, but no data type or model specific information is listed.

Chapter 10: Train dynamics analysis In this chapter the dynamics of trains and the derived physical requirements to the system are elicitated and calculated. Many requirements are necessary for the system to stay consistent. This chapter reflects the many physical aspects that the project has adopted compare to using a highly discrete physical model.

Chapter 11: Control system design This chapter describes the design of the control system. All the algorithms of the control entities are shown as state flow diagrams, and the overall control algorithm and a sample scenario is presented.

Chapter 12: Glossary This is a glossary of all domain specific terms used in this report.

Chapter 13: GUI design This chapter lays out the design of the GUI to the model implemented in JAVA. Some prototype screens are shown.

Chapter 14: Invariants and assumptions This chapter lists all the invariants and assumptions that have been identified and found necessary in the design and analysis chapters. All these invariants should be implemented in the model in such a way that they are enforced in the implementation.

Chapter 15: RSL modelling method summary This chapter summarizes the method used to construct the RSL model, how it is refined and decomposed, how the requirements and safety is enforced, and how the structure is prepared for proving safety in the model.

Chapter 16: Initial model An illustration of the initial model schemes is shown. A short description of the initial model which is abstract is presented. All major data types are therefore also abstract (sorts).

Chapter 17: Decomposed model A short description of the decomposed model where related data has been grouped into modules (objects).

Chapter 18: Concrete model A short description of the concrete model where all major data types has been explicitly defined.

Chapter 19: Imperative model A short description of the imperative model. Variables have been added to the model and all functions are converted from applicative to imperative to prepare the model for translation into JAVA.

Chapter 20: Implementing a simulator The method of translation from RSL to JAVA is discussed. The structure of the actual program which is converted from RSL to JAVA is described.

Chapter 21: Using the simulator A small users guide to the simulator and an introduction to simulator functionality.

Chapter 22: Test This chapter lays out the test strategy of the simulator and presents some test results.

Chapter 23: Verifying safety This chapter presents a discussion of provability of safety in the system. Some methodology is presented of how to do this, and some informal argumentation is done on some selected areas following this method to prove safety.

Chapter 24: Ideas & future work Presents all ideas for further development of the system. All ideas and considerations discussed when this project was executed are included.

Chapter 25: Related work Related work in the form of papers and projects are discussed and related to this project.

Chapter 26: Discussion Selected subjects are discussed

Chapter 27: Conclusion The results of this thesis are evaluated and commented.

Appendix A: Design of the GUI Images of the GUI design prototypes

Appendix B: RSL method description Describes the method used and steps taken developing the RSL model without being specific to the developed model.

Appendix C: XML DTD The DTD describing the syntax of the XML containing a configuration

Appendix D: Test images Images from the test

Appendix E: Concurrency Describes the ideas for using concurrency in the RSL model and the translation to JAVA.

Appendix F: RSL modules All the modules of the model in the different refinement steps

Chapter 3

Informal domain description

This chapter describes the physical parts of a railway line.

This description is based on a simple railway line with low intensity traffic. Compared to a real railway line many domain specific details have been left out to simplify the model.

A physical railway line consists of a number of entities which are described in the following sections. The static and dynamic properties of each entity are listed.

The following entities constitute a physical railway line:

- Railway line
- Segments
- End station areas
- Trains
- Junctions / points
- Crossings
- Switch boxes
- Sensors

3.1 Railway line

A railway line is a *single track* between two ends that are called *high* and *low*. When presented on illustrations the low end is always to the left, and high is to the right.

By “single track” is meant, that though the track may branch into two separate tracks (for trains to pass each other), it immediately joins again. The direction from low to high is called *up* and the opposite direction is called *down*.

See section 3.11 for an example of a railway line.

3.2 Segment

The railway line is divided into a number of coherent *segments*. A segment is basically just a division of the railway line. At each end of a segment is a switch box (see section 3.7).

The segments are used by the control system which, through the switch boxes, controls access to each segment (see section 11.1).

At the border between two segments (at a *switch box*, see section 3.7) there is often an entity like a junction (see section 3.5) or a crossing (see section 3.6) which needs special handling (controlled by a switch box). These entities are modelled as point shaped, i.e. there is not free space between two segments. Being located on a junction or on a crossing then means to be located on two segments - one on each side of the entity.

3.2.1 Static properties

The static properties of a segment are:

- *Length* in meters: the length of the segment
- *Max Speed* in m/s: the max allowed speed for a train on that segment

3.3 End station area (ESA)

Before the first and after the last segment of the railway line is an end station area (ESA). These are i.a. used to park the trains when they are not used. It is not further specified how the trains are parked.

Initially all trains are located in one of the ESAs. From an ESA a train can enter the first (or last) segment of the line. During the day a train drives between the two ends of the railway line. When it reaches one of the ends it enters an ESA.

After a while, located in an ESA, the train might change direction and drive to the opposite end.

Access to the ESAs are controlled by the switch boxes (see section 3.7) at the ends of the railway line.

An ESA is assumed to have an infinite capacity of parked trains. In the real world it would of course have a finite capacity, but since the number of trains is supposed to be very small in this railway line, the assumption should not cause any problems.

The ESA in the low end is called the *low ESA* and the ESA in the high end is called the *high ESA*.

3.3.1 Static properties

The static property of an ESA is:

- *End*: the end of the railway line at which the ESA is.

3.3.2 Dynamic properties

The dynamic property of an ESA is:

- *Parked trains*: a list of trains that are in the ESA

3.4 Train

A train is a number of connected vehicles that moves on a railway line. While moving along the line the train enters and exits segments along the route. It is assumed that the length of a train is always less than the length of any segment. Therefore a train is either on one or two segments at a time. It is on two segments when it passes from one segment to another.

Each train has a train driver that manually controls the train and a Train Control Computer (TCC) that is used in the control system to reserve segments before entering, and stop the train when it tries to enter a segment it has not reserved. The TCC also gives relevant information to the train driver.

3.4.1 Static properties

The static properties of a train are:

- *Length* in meters: the length of the train.

- *Max speed*: the max obtainable speed of the train.
- *Max acceleration*: the max obtainable acceleration of the train.
- *Max deceleration (braking capacity)*: the max braking capacity of the train.

3.4.2 Dynamic properties

- *Acceleration*: the current acceleration of the train.
- *Speed*: the current speed of the train.
- *Position*: the position of the train in the railway line
- *Direction*: the driving direction of the train (up / down).

3.5 Junction / point

A junction is a construction enabling three segments to be joined. These segments are the stem, the *left branch* and the *right branch*. The left branch is the segment to the left when you look at the branches of the junction from the stem. The left- and right branch are also called *branch segments*.

As described in section 3.2, a junction is considered to be a point-shaped entity. Therefore if a train is located on a junction, it means that it is located on the stem and on one of the branches.

Every segment that is either left- or right branch of a junction has a junction in both of its ends. This means that whenever the railway line branches into two segments it gathers again at the end of the branches (segments). Each junction has a point that can connect one of the branches to the stem.

A point is a movable device attached to a junction. The point connects either the left- or the right branch with the stem or it is in an intermediate position between the branches moving towards one of the branches. The point is never statically positioned in an intermediate position between the branches.

In figure 3.1 the terms *up branch* and *down branch* are used. This definition is related to the control algorithm and uniquely identifies the branches. This definition is explained in chapter 6.

3.5.1 Static properties

The static properties of a junction/point

- *Stem*: The stem segment

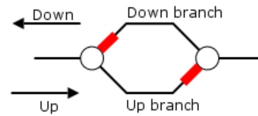

Figure 3.1: A railway line with four segments, two junctions and their points

- *Up branch*: The up-branch segment
- *Down branch*: The down branch segment

3.5.2 Dynamic properties

The dynamic property of a point is:

- *Position*: whether the point is in *up*, *down*, *moving-up* or *moving-down* position.

3.6 Crossing

A crossing is a construction enabling a railway line and a road to cross each other in the same level. Crossings are equipped with signals with both visual and audio warnings and with barriers. These are used to preserve safety by stopping cars etc. from entering a crossing when a train is about to cross. Figure 3.2 and 3.3 shows crossings.

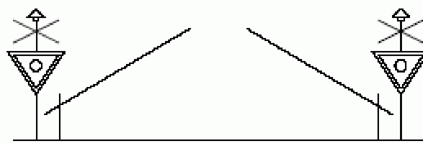

Figure 3.2: A crossing seen from the road with signals and barriers

3.6.1 Dynamic properties

The dynamic properties of a crossing are:

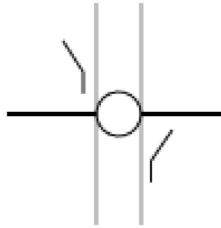

Figure 3.3: A crossing with open barriers seen from above

- *Position of barriers*: whether the barriers are *open*, *closed*, *opening* or *closing*.
- *Signal status*: whether the signals are turned on or off.

3.7 Switch Box

A switch box (SB) is a device used by the control system and is purely a virtual / logical entity in the way that is has no mechanical functionality whatsoever. Therefore it has no physical state (dynamic physical properties).

Switch boxes guard segments so that two trains do not enter / reside in the same segment. There are four different types of switch boxes. Each are different in the way they function. Some SBs has the responsibility of a special entity which it controls (points and crossings):

- *End SBs* with one bordering segment at one of the ends of the railway line and an ESA at the other end.
- *Point SBs* with three bordering segments at a junction/point (stem, up branch and down branch).
- *Crossing SBs* with two bordering segments at a crossing - one segment at each side of the crossing
- *Plain SBs* with two bordering segments - one at each side of the SB. It is primarily used if it is preferred to split up a large segment in several smaller segments.

3.7.1 Static properties

The static properties of a SB depends on the type of the SB:

End SB :

- *End*: the end (high or low) of the railway line

Point SB :

- *Stem*: the stem segment of the junction.
- *Up branch*: the up branch segment of the junction.
- *Down branch*: the down branch segment of the junction.
- *point*: the point it controls.

Crossing SB / Plain SB :

- *Up segment*: the segment in the direction up.
- *Down segment*: the segment in the direction down.
- *Crossing*: the crossing it controls.

Plain SB :

- *Up segment*: the segment in the direction up
- *Down segment*: the segment in the direction down

3.8 Sensor

A sensor is located at the boundary between two segments (at a SB). It senses if a train is located on it, i.e. if a train passes from one segment to another. The switch box at the sensor can read the state of the sensor.

3.8.1 Dynamic properties

The dynamic property of a sensor is:

- *Active status*: whether the sensor is active or not, i.e. whether a train is located on it or not.

3.9 Stations

In the railway line we have already introduced the main stations, which are the two ESAs. Besides these, minor stations can freely be placed along any segment. Typically they are placed between two branch segments (figure 3.4) but it does not have to be the case.

The minor stations are not included in the formal model, since they have no impact on the control system. Trains are not statically located (parked) at minor

stations. A train stops at minor stations to enable passengers to get on or off the train. Afterwards the train continues to the next station. It will therefore be possible to introduce new minor stations without changing the configuration of the railway line or the control system.

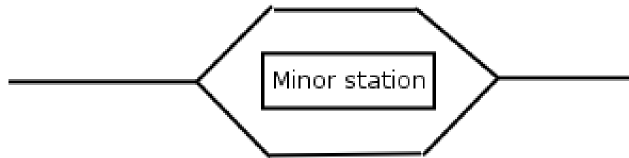

Figure 3.4: Typical location of a minor station

3.10 Signals

Some railway networks use signals as a part of the control system to show the train driver if the train is allowed to proceed. This is not the case in the system considered here. The control system is formed by the switch boxes and the on board train control computers, that control the trains and give *go/no-go* information to the driver. Therefore signals are not necessary.

3.11 An example of a railway line

In the previous sections all the entities of a railway line were described. Figure 3.5 shows an example of a railway line.

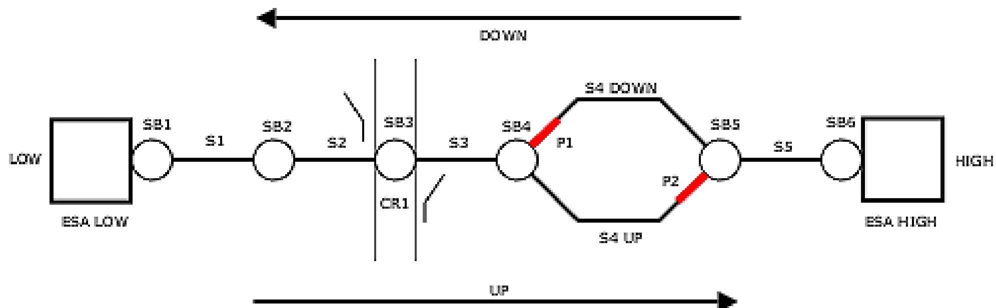

Figure 3.5: An example of the considered railway line

The figure shows the different entities and concepts:

End station areas: `ESA LOW`, `ESA HIGH`

Switch boxes: `SB1`, `SB2`, `SB3`, `SB4`, `SB5`, `SB6`

Segments: `S1`, `S2`, `S3`, `S4 DOWN`, `S4 UP`, `S5`

Crossing: `CR1`

Points (red): `P1`, `P2`

Ends: `LOW`, `HIGH`

Directions: `UP`, `DOWN`

Sensors: not shown - same places as the SBs

Trains: not shown

3.12 Single lines

A single line is a number of coherent segments on which trains are allowed to move in both directions (not at the same time, beacuse this would cause a collision).

The line connects either the stem of two junctions, the stem of a junction and an end station area or two end station areas if no junctions exist in the railway line.

The segments `S1`, `S2` and `S3` in figure 3.5 form a single line. So does segment `S5`.

Chapter 4

Main idea and concept

This chapter presents the problems which are solved by this thesis. The main idea and concept behind the solution to these problems is presented. Some terms concerning the domain are also explained.

4.1 A simple railway line

This section briefly explains the concept of a simple railway line. For a detailed description of all entities please refer to chapter 3, Informal domain description.

Figure 4.1 shows an example of a simple railway line.

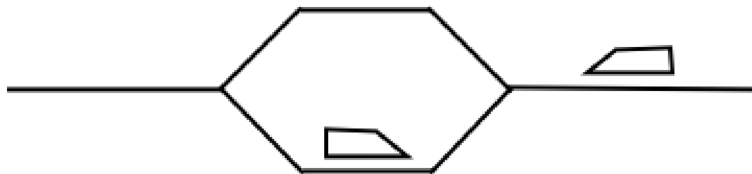

Figure 4.1: A simple railway line

The line is simple in the way that it only runs from one end to another. When the line branches, it immediately joins again. Trains driving in opposite directions can pass each other at these branches.

Three basic problems arise on this simple line:

Collisions occur when two trains collide.

Derailings occur when a train drives through a point which either is switching or is set to the opposite branch of the one the train is coming from.

Deadlocks occur when two trains are on the same line and face each other. Then one train has to reverse to a branch and let the other train pass.

All these problems should be solved by a control system.

4.2 Control system

A control system exists to ensure that trains do not collide or derail. Furthermore the control system considered here does also actively enforce certain behavior on the trains which ensures safety. On top of this, the control system also ensures that trains do not end up in a position blocking each others access to the rest of the line. This means the trains should stop at a branch and let another train, coming in the opposite direction, pass before leaving the branch.

Figure 4.2 shows an illustration of the interaction between the control system and the physical domain.

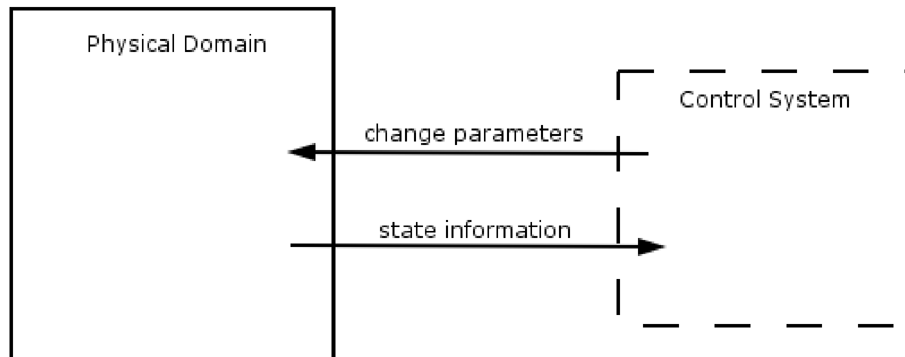

Figure 4.2: Interaction with control system

The control system adjusts parameters in the physical system to uphold safety. The control system obtains knowledge of state information by using equipment as track sensors, GPS and other well-known technologies.

4.2.1 The reservation concept

For the control system to be able to control train access to different parts of the line, the line is separated into several segments. The idea of the control system is to allow only one train at a time at a given segment.

At each end of a segment a switch box is placed to guard the entrance (like a signal). A train on-board control computer (TCC) communicates with a switch

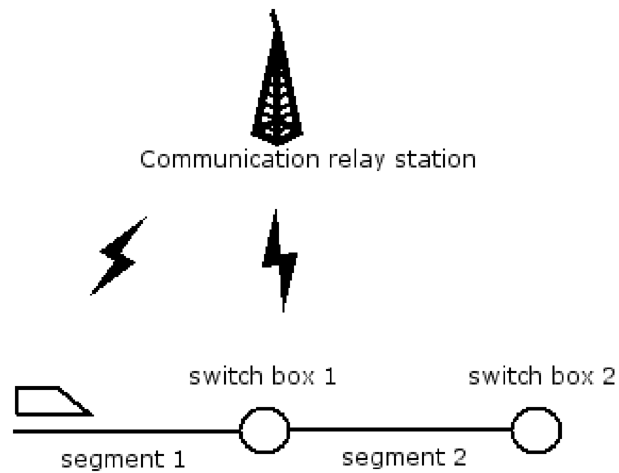

Figure 4.3: Train communicating with switch box

box (SB) control computer (SBCC) (through a GSM communication service or the like) to request / gain access to a segment. A switch box is normally placed where control is needed. For example at a point where the switch box also controls the movement of the point.

The switch box then decides whether or not the train should get the permission to enter the specified segment. A primary concern for the switch box is to ask the switch box at the other end of the segment also. This is necessary to ensure that two different trains are not allowed access to the same segment, at the same time, from two opposite ends.

4.2.2 Reservations and braking

As mentioned in section 4.2.1 a train has to reserve the next segment through a switch box to gain access to the segment. Figure 4.4 illustrates two fixed points on a segment:

Figure 4.4: The reservation and brake point on a segment

This illustration shows the reservation point (*res*) and the brake point (*brk*). These dictate respectively that when a point on the segment is passed then the train has to:

- Ask for a reservation for next segment (when passed reservation point).
- Brake the train if no reservation has been obtained (when passed brake point).

For calculations concerning acceleration and braking we assume that trains have a constant acceleration / braking coefficient. This is not the case in the real world though. In the real world the braking coefficient is based on friction which is heavily dependent of the speed of the train.

Chapter 5

Engineering concepts

This chapter explains some technologies, assumptions about these, and terms, which are basic to this thesis.

5.1 Sensors

In the model developed in this thesis a sensor is located at a switch box, which is situated at the border between two segments.

When a train passes from one segment to the next, the sensor becomes *active* at the time the front end of the train enters the new segment. The sensor becomes *inactive* again when the rear end of the train enters the new segment. This is an abstraction from the real world in which several types of sensory technology is used:

Track isolations (circuits) the track is physically separated in isolations which each carries a low electrical current. When a train is located on an isolation it will short circuit this with its wheels. In this way the presence of a train can be detected.

If the actual track is equipped with isolations we assume that the SB has some kind of interface to this sensory input. The SB has to use this input to determine if a train is passing by and thereby mapping the input to correspond to a single point sensor placed at the SB. We can consider the following possibilities:

- A SB is placed between two segments each with its own isolation. We assume that if any spacing (without sensors) exists between the two isolations it is smaller than any train length.

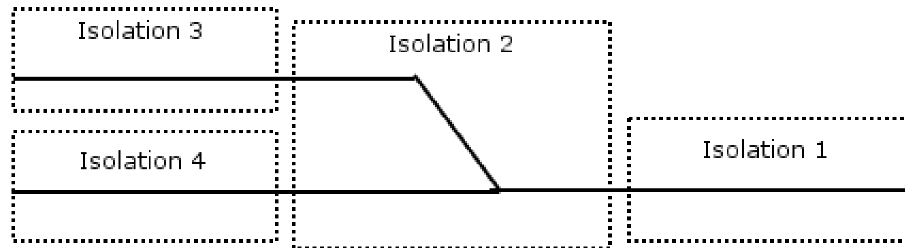

Figure 5.1: Track isolations

Because of the system design structure the SB knows that a train passes if both isolations are active at the same time. This is true because the SB has to give permission before a train can pass it.

So if both isolations are active at the same time without a permission is given, two different trains occupy the two segments at each side. This also means that if an error occurs in the system, the SB (in this case) cannot be certain how to interpret the sensory input.

- A point SB is placed at a junction between 3 segments. If each segment has its own isolation - like in the example above - the result is exactly the same. If the SB cannot trust its local reservation information it has no way of knowing if two active isolations is one train passing or two different trains occupying one segment each.

  But if the junction itself has an isolation (like in figure 5.1) the functionality can be mapped directly to the behavior of a sensor. The SB knows if a train is positioned on the junction or not.

Axle counters some units that counts the axles on a train are placed on the track. When a train passes over this unit it counts how many axles that passes. The system knows that a train exists between two axle counter units until an equal number of axles have passed them both.

Apart from the difference in technology the sensory information is the same. The system knows if a train exists on a specific length of track.

5.2 Reservation and brake points

Some trivial assumption about the reservation point and brake point should hold:

$$reservation_point < segment.length$$

$$brake_point < reservation_point$$

Also, for the system to work, a train must not acquire a new reservation while on the border between two segments. Therefore the following requirement:

$$reservation_point < segment.length - train.length$$

5.3 Time modelling

The method chosen to update time in the simulators is to use discrete time steps. This means that time is not updated continuously but in blocks of a certain interval. Some process on the top level of the simulator, which contains all layers of the model (statics, dynamics and control), initiates the time updates.

One basic requirement needed to be fulfilled by the time update method is that all entities have to have a fair distribution of time. This means that it should not be possible for some entities to progress in time faster just because they have a simpler calculation at each time update. Therefore in the end all entities have to wait for the slowest entity to update before progressing in time.

The time updates are executed by calling a *tick* method in each entity that is time dependent. The value passed by the tick function represents the amount of seconds passed since last tick.

Each entity then calculates its new state (e.g. a train updates its position, velocity and acceleration) depending on the previous state and the time passed since last tick.

Figure 5.2 illustrates the time update principle.

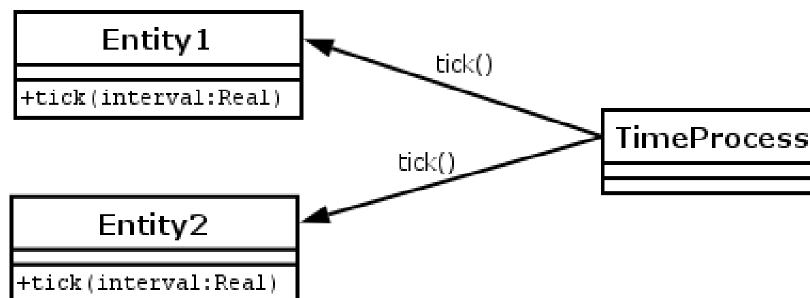

Figure 5.2: Time update principle

Using the time structure mentioned above, at least 2 different implementations can be chosen:

Parallel updates all entity time updates are handled concurrently. This gives rise to the need of synchronizing all variables which are shared between the entities of the system.

Sequential updates all entities are called one after another. The processing of time update in one entity is terminated before another is executed.

Beside the major difference in the structure of the two implementations there is essentially no effectual difference in functionality. Because of the discretized time updates all time processing in all entities should have terminated before the next tick. If this is not ensured then some entities could be further ahead in time than others.

This effectually eliminates the advantages of concurrency since some kinds of barrier synchronization is needed after each tick anyway.

The messages between switch boxes and trains are modelled to arrive instantly. Some communication delay could be modelled but the system should be designed so the system is not dependent on fast message delivery. Each control entity have a message buffer that receives messages independently of a tick. At each tick the control entity processes a message from the buffer.

5.4 Safety

The railway line is required to be safe in all possible situations when using the system. The railway line is considered safe (for trains) when certain events are avoided. This definition of safety does not concern exceeding max speed.

The railway line is safe when:

- No accidents involving trains occur.

There are a number of different situations with accidents involving trains:

- Two trains *collide*
- A road vehicle and a train *collide* at a crossing
- A train is *derailed* at a junction if it approaches from the stem and the point is in an intermediate position, i.e. the point is not connected to either the right- or left branch.
- A train is *derailed* at a junction if it approaches from one of the branches and the point is not connected to the branch the train is located on.
- A person or an animal walks on the railway line and is *hit* by a train.
- Any *physical defects* in the railway system that causes the system to fail like cracks in the railway line and broken electrical wires.

The last two situations are caused by external circumstances and are not considered in the model. Although they should at least be considered for the individual implementations of the system. An example could be the danish S-train system which quite often (several times a month) experiences electrical failures in both trains and railway power supply wires.

5.5 Deadlock

It is preferable to avoid deadlock of trains in the railway line.

A situation with deadlock is in this system defined as:

- Two or more trains are waiting for each other to move before they are able to move themselves. In this situation they will never be able to do so without either violating the rules of the control system because they are all just passively waiting for each other to move. Without some deadlock resolving algorithm the system will never leave this state.

The figures 5.3 and 5.4 show examples of deadlock:

Figure 5.3: Example of deadlock 1

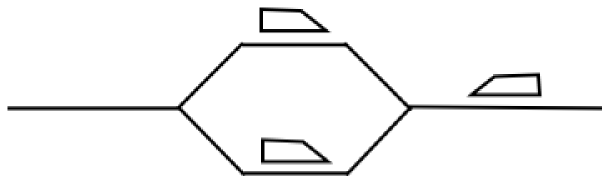

Figure 5.4: Example of deadlock 2

In both examples the safety requirements (as described in section 6.1) prohibit the trains from entering the same segment. The trains will therefore not collide, but to make progress one of the trains has to change direction. This is not desired since all trains are supposed to drive continuously between the two ends of the railway line without changing direction half way (if ever this is possible).

5.6 Livelock

Livelock of trains can be very hard to avoid. A situation with livelock is defined as:

- One or more trains are currently not able to move to another segment but eventually it may be able to do so without either violating the rules of the control system or performing an undesired action.

Livelock is in many ways like deadlock. The difference is that at deadlock you are stuck with no future possibility to proceed. At livelocks you often have this possibility.

Figure 5.5 shows an example of livelock.

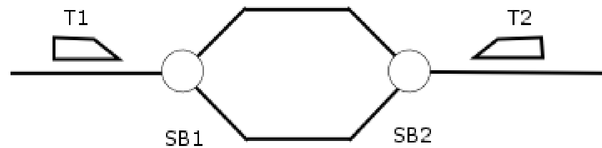

Figure 5.5: Example of livelock

In figure 5.5 livelock can arise if both trains continually ask for a reservation for the same branch at the same time.

Chapter 6

Control system requirements

This chapter concerns the requirements to the distributed control system. These are the safety requirements and some functional requirements.

The requirements are listed together with sketched solutions to the requirements. These solutions will be part of the control system design. To see how this is reflected in the design please refer to chapter 11.

A glossary is provided in chapter 12 to explain the terms used in the requirements.

6.1 Safety requirements

The safety requirements are derived from the definition of *safety* in section 5.4.

Requirements

- Two trains are not allowed to be positioned on the same segment at the same time.

Solutions:

- To enter a segment, a train must have a *segment reservation* for that segment.
- Only one train may have a segment reservation for a particular segment at a time.

- *Reservations* must be managed by the SBs
- It is the job of the TCCs to make requests to the proper SBs to obtain the necessary segment reservations.
- If a train does not have the necessary segment reservation when it is about to enter a segment, the TCC should stop the train in a safe distance of the segment.
- Before a train enters a crossing the barriers must be closed and the signals turned on. It is the job of the SB at the crossing to take care of this. If this is not done, the TCC should stop the train in a safe distance of the crossing.
- Before a train enters a junction the point must be connected with one of the branches (see also section 6.2). If the train approaches from a branch segment, the point must be connected with that branch. It is the job of the SB at the point to take care of this. If this is not done, the TCC should stop the train in a safe distance of the junction.
- When a train is on a junction the point must not be moving.

6.2 Functional requirements

The functional requirements concerns railway line efficiency. These functional requirements are derived from the identified cases of deadlock and livelock in sections 5.5 and 5.6.

Requirements:

- At any time deadlock must be avoided. To make sure that deadlock does not occur, any situation that can lead to deadlock must be avoided. (Two examples of deadlock are illustrated in section 5.5).
- Livelocks where train requests could conflict with each other should be avoided. (An example of a possible livelock is illustrated in section 5.6).

Solution:

- A train must always use the right branch of a junction (seen from its driving direction). This makes the branches at a junction unidirectional so that trains always pass each other to the right. Since the trains are not supposed to overtake each other this does not constitute a problem in the workings of the railway line.

  This solution avoids the second deadlock example and the livelock example.

Using the convention of always using the right branch, the right branch seen from the up direction is called the *up branch.* Likewise the right branch seen from the down direction is called the *down branch.* Using the terms up- and down branch we have uniquely named the branches of a junction/point. These terms are therefore used in the following sections.

- Before a train enters a line where traffic in opposite direction is possible, the whole bi-directional line must be reserved until a segment which is only uni-directional.

  This should prevent the first example of deadlock in section 5.5.

To see how these solutions are implemented in the control algorithm please refer to chapter 11, *Control system design.*

Chapter 7

Simulator requirements

In the following the requirements to the simulator system are described.

The simulator should be developed in the programming language JAVA with a graphical user interface (simulator GUI). The GUI should contain a menu in which it should be possible to:

- Show the train simulator (the main window shown at startup)
- Switch off the control system.
- Create, select etc. configurations of a railway line with trains to be used in the train simulator

This functionality is described in the following sections.

7.1 Train simulator requirements

7.1.1 Formal model requirements

The basis for the train simulator is the concrete imperative model of the physical railway line and its control system. The model should systematically be transformed into JAVA code as described in section 20.1. The JAVA code transformed from the RSL model is called the *simulator core*.

7.1.2 Visual parts

The train simulator should visualize the following:

- The layout of the railway line
- The trains at their location on the railway line
- The configuration and state details of every entity in the railway line

The GUI should also make it possible to change the state of the different dynamic entities in the railway line. The accelerate, brake and change direction commands are available to act as the train driver. These can be used in conjunction with the control system which will handle undesired actions initiated by the train driver

The actions to switch points and crossings are on the other hand not handled by the system because this should not be a possibility in an actual system implementation. These actions are only for experimental purposes:

- Change the acceleration of a train thereby starting, stopping or changing the speed of a train, i.e. act as train driver.
- Change the direction of a train when it is located in an ESA. The train must be stopped not facing the line.
- Change the position of a point.
- Open and close the barriers at a crossing and thereby turning the signals on and off (when the barriers close, the signals must be turned on and otherwise turned off).

7.2 Switch off the control system

When the control system is switched off the railway line is not safe for trains. Therefore the accidents involving trains described in section 5.4 might occur. By having the facility to switch the control system on and off, it becomes clearer how the control system avoids accidents.

7.3 Railway line configurations

The train simulator should be generic. This means that it should be possible to use different railway line configurations as input.

7.3.1 Create configurations

It should be possible to create new configurations of a railway line. The way to do this is:

- Start with the smallest possible railway line
- Add SB/segment pairs with standard properties.
- Add trains.
- Change properties of entities.
- Save the configuration.

7.3.2 Export/import configurations

It should be possible to export a configuration of a railway line to a file using the XML format. Likewise it should be possible to import a configuration from a XML file

7.3.3 Load configuration

All the configurations in the simulator should be listed, so it is possible to select a configuration to be loaded into the train simulator.

Chapter 8

Model structure

Figure 8.1 illustrates the concept model structure.

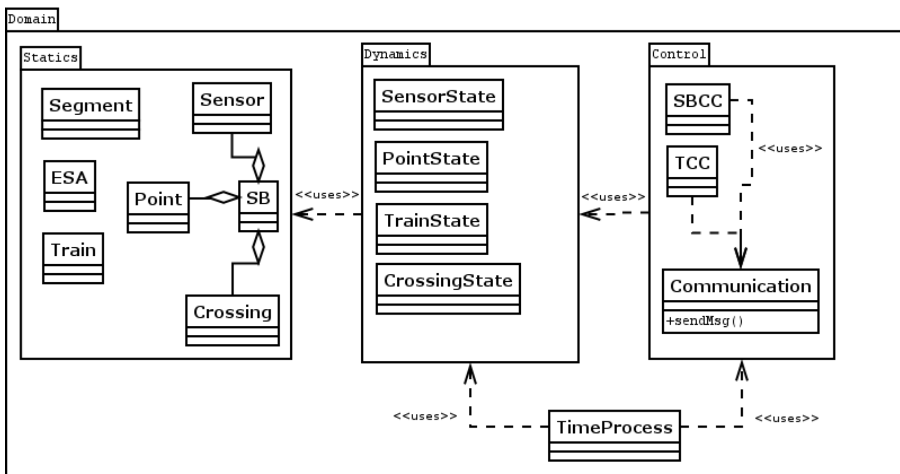

Figure 8.1: Concept model

By concept is meant that this UML diagram only illustrates **what** information should be reflected in the different model modules and not **how** they should be structured.

For example one can see that both crossings and points are parts of the composite entity SB. This is not necessarily how the module structure will be implemented in the model, but we know that a SB should have the information of a crossing and a point represented somewhere.

Chapter 9

Physical design

This chapter concerns modelling the physical part of the railway system and discusses how to model the physical domain in lack of any kind of control or safety aspects.

This chapter do not mention any type- or function specific details about the actual model design but only a high level strategy of what is to be modelled.

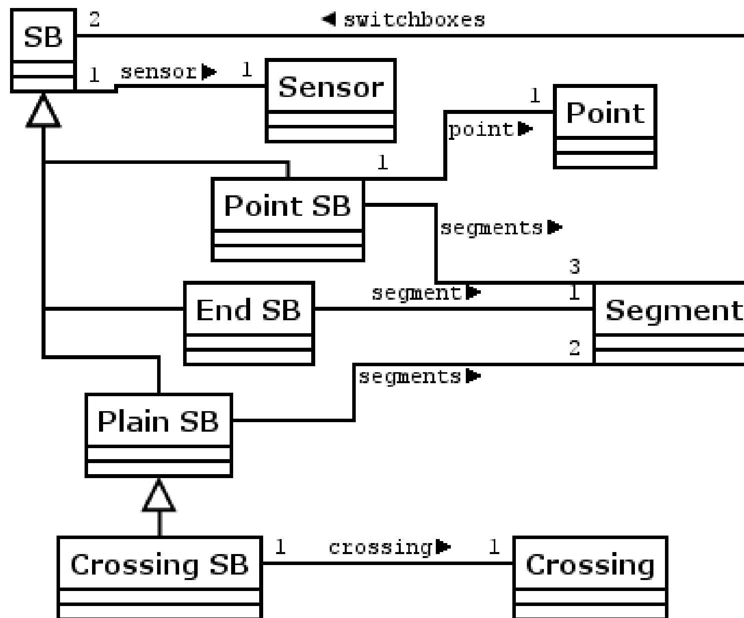

Figure 9.1: Network entity relationship

9.1 Static network modelling

The network model defines the relationship between the static entities in the domain. The network model defines the relationships shown in the figure 9.1.

As shown in figure 9.1, all SBs, no matter the type, have an associated sensor. Depending on the type of SB it can be related to a point, a crossing or none of those. All SBs are related to at least one segment. All segments are related to exactly two SBs no matter the type.

How this should be reflected in the actual RSL model is addressed in chapter 16.

9.2 Train positions

This section discusses the choice of how to model train positions.

Both the front- and rear position of a train are defined by [*location*, *length*]:

location the segment or ESA it is currently positioned at.

length a length that represents the distance to the low end of the segment.

The idea is sketched in figure 9.2.

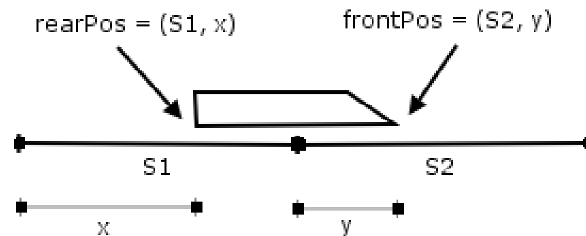

Figure 9.2: Train position modelling

To optimize the performance of the system, the position of a train is modelled as both the position of the front and rear end of the train. This eliminates the need of constantly calculating the rear position from the front position. One drawback of this is that a wellformedness requirement has to state that the front and rear positions always should be separated by the length *train.length*.

When a train moves, both its positions are updated with the $\Delta length$, which the train has moved in the given time interval. If the new length goes beyond the border of the current segment, the next segment is calculated from its current position and direction.

9.3 ESAs

An ESA represents a station or storage area capable of holding several trains. This is modelled as an undefined space with a certain length. It is assumed that trains (or parts of trains) existing in an ESA cannot collide with each other.

Trains driving over the edge of an ESA away from the line is by definition derailed.

9.4 Crossings

Figure 9.3 sketches a state chart for a crossing. Be aware that the state graph is constructed to perform one transition per tick (see section 5.3). The variables `signals_only` and `bars_moving` represent the time needed to perform the associated action.

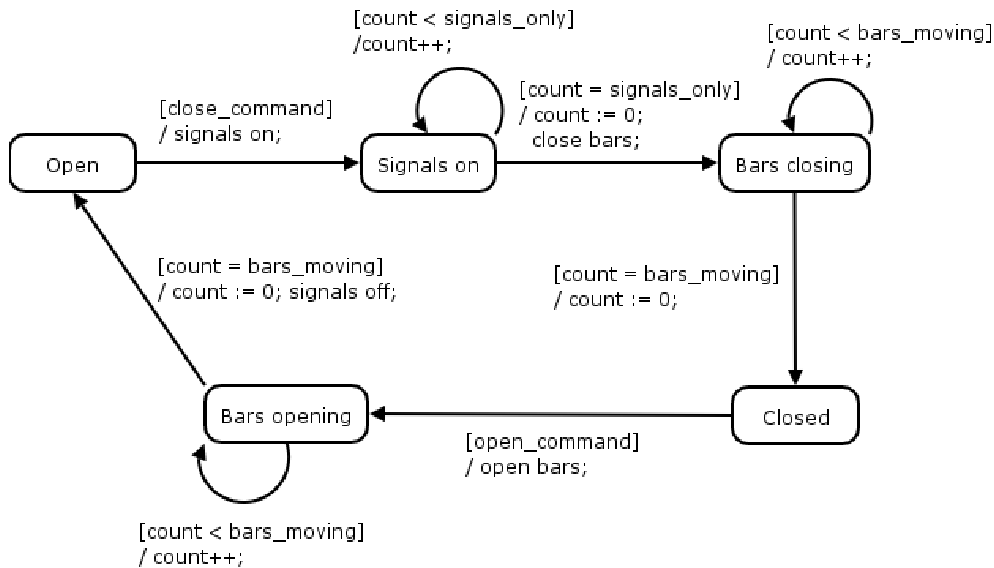

Figure 9.3: Crossing state chart

9.5 Points

Figure 9.4 sketches the state chart of a point. The state graph is constructed to perform one transition per tick (see section 5.3). The variable `point_switching` represents the time needed to switch a point.

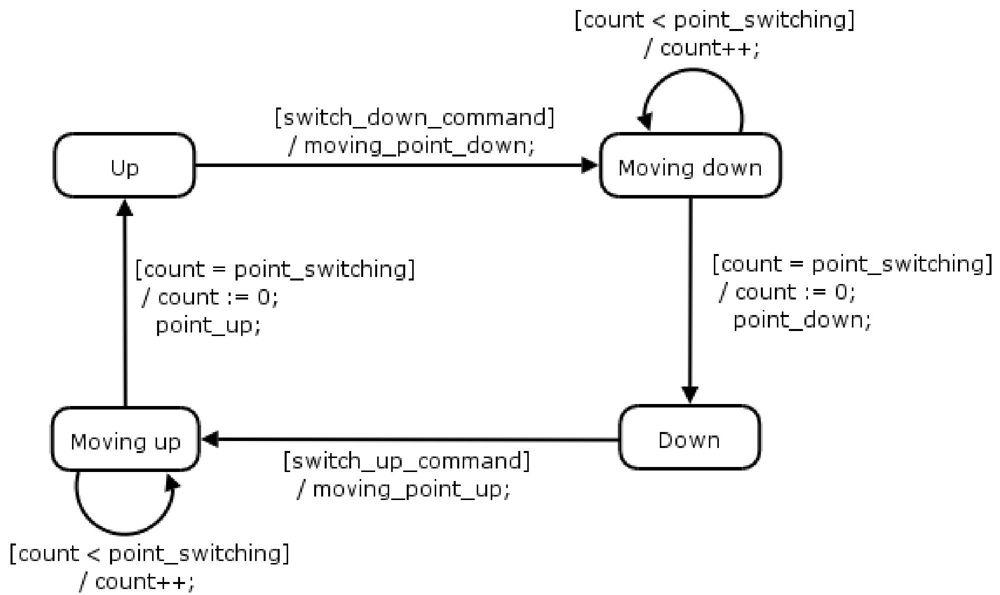

Figure 9.4: Point state chart

Chapter 10

Train dynamics analysis

At each tick with time interval t the train physics module should calculate the following:

Position: $pos_1 = pos_0 + v * t + \frac{1}{2} a * t^2$

Velocity: $v_1 = v_0 + a * t$

Acceleration: Set by control system and external events (train driver and control system).

10.1 Train / segment length

To simplify the modelling of train positions in this system, it is required that $train.length < segment.length$. This implies that a train at most can be on two segments at a time, i.e. when passing the border from one segment to another.

10.2 Collision modelling

This section describes how collisions are defined and modelled in this system.

A typical frontal collision is showed in figure 10.1.

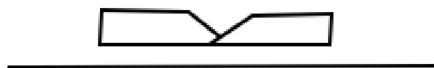

Figure 10.1: Typical frontal collision

As the figure shows, the front of the two trains overlaps just a tiny bit. One could argue that, seen strictly from a mathematical point of view, a collision is when any part of two trains overlaps. This means that the situation showed in figure 10.2 mathematically also is a collision.

Figure 10.2: Mathematically also a collision

We have decided only to handle the initial situation of a collision, i.e when the first overlap of train positions is detected. That is when a train tries to move to a location which is already occupied by another train. This means that we do not intend to check collisions on the entire state at each update, but only check on a train which is about to move.

To make this approach work, the time update interval has to be sufficiently small to detect a collision before the trains have passed the initial part of the collision, which is the only part that is detected. This is discussed further in section 10.3.

This leads to the following two collision types:

In figure 10.1 we have a classic frontal collision. In this case a collision is defined as:

- Two trains have at least 1 segment in common.
- They are driving in opposite directions.
- Their front positions are on the same segment.
- The train driving *UP* has a front position which is further UP than the train driving down.

Figure 10.3 illustrates the second type of collision.

Figure 10.3: Collision from the rear

In this figure a train has collided with the rear end of another train. This collision type can be defined like this:

- Two trains have at least one segment in common.
- They are driving in the same direction.

- One train has a front position which is higher UP than the other trains rear position but lower than the other's front position.

The opposite case, where both trains are driving DOWN, has the exact same definition when switching UP / DOWN and higher / lower in the statements above.

In this definition it is reflected that we only check on collisions from the point of view of the train which finally moves into another train. Therefore the above definition only concerns the train driving into the other's rear and not the train being hit.

10.3 Time and speed considerations

This section discusses the requirements and assumptions necessary for the system to work.

10.3.1 Collision detection

Collision detection concerns the method used to detect if two trains have collided in a given state. Collision detection can easily be very expensive in time critical computations, because the calculations have to be done in every state of the system. Of course the calculations in this case are only executed if two trains are located on the same segment.

The actual collision detection is done by carrying out the algorithm sketched in section 10.2. But for this to work we have to make sure that the time between two updates is so small, that all collisions are detected. This is not possible if the time update interval is set to a large value (e.g 10 seconds or so). Then the collision detection (and all other processing in the system) is only executed every 10th second. 10 seconds is more than enough for two trains to pass through each other in opposite directions. Therefore it is not enough to ensure that all collisions are detected.

A frontal collision is the fastest way for two trains to close in on each other. Therefore it represents the worst case scenario of what the system needs to detect. Furthermore we would like the collision to be detected before the trains have driven too far through each other. It is therefore relevant to investigate how far two trains could drive through each other using an example interval.

Two trains t_1 and t_2 are positioned toward each other and are both driving with the highest possible speed v_{max}. The time is updated each Δt seconds. The distance s_{err} that they both travel in Δt seconds is then:

$$s_{col} = 2 * s_{err} = 2 * v_{max} * \Delta t$$

If we assume that the max speed of the train is 120 km/h ≈ 33 m/s, and we assume that $\Delta t = 0.05s$ (system updated 20 times/s) then we get:

$$s_{col} = 2 * 33m/s * 0.05s = 3.3m$$

We can also calculate how an increase by 5 updates pr. second (a decrease in Δt by 0.01s) would affect s_{col}:

$$\Delta s_{col} = 2 * v_{max} * 0.01 = 0.02 * v_{max}$$

which in this case is $0.66m$.

The requirement for this system to work is then:

$$s_{col} < tlength_{min}$$

where $tlength_{min}$ is the length of the shortest train in the system. If this is fulfilled then no front of another train can manage to go all the way through the shortest train without being detected.

10.3.2 Brake point requirements

As seen in the last section, s_{err} is the greatest distance a train can travel before the system is updated and the control system checks and reacts upon the state of the physical world. This leads to the following requirement of the system:

$$bp > s_{brk} + s_{err} \tag{10.1}$$

where bp is the length from the brake point to the end of the segment and s_{brk} is the length used to brake if the train is running at maximum velocity v_{max} and braking with the acceleration a_{brk}.

s_{brk} is calculated by the following:

First we need to calculate the time used to brake t_{brk}:

$$\begin{aligned} 0 &= v_{max} + a_{brk} * t_{brk} \Rightarrow \\ t_{brk} &= \frac{-v_{max}}{a_{brk}} \end{aligned} \tag{10.2}$$

We then calculate the distance used to brake:

$$\begin{aligned} s_{brk} &= v_{max} * t_{brk} + \frac{1}{2} * a_{brk} * t_{brk}^2 \Leftrightarrow \\ s_{brk} &= -(v_{max}^2) * \frac{1}{a_{brk}} + \frac{1}{2} * \frac{1}{a_{brk}} * v_{max}^2 \Leftrightarrow \\ s_{brk} &= -\frac{1}{2} * \frac{v_{max}^2}{a_{brk}} \end{aligned} \tag{10.3}$$

Using the numbers from the example above and assume a braking capacity of $(-1.3m/s^2)$[1] we get:

$$\begin{aligned} bp &> s_{brk} + s_{err} \Leftrightarrow \\ bp &> -\frac{1}{2} * \frac{v_{max}^2}{a_{brk}} + v_{max} * \Delta t \Leftrightarrow \\ bp &> -\frac{1}{2} * \frac{(33m/s)^2}{1.3m/s^2} + 33m/s * 0.05s \Leftrightarrow \\ bp &> 22.59m \end{aligned}$$

10.3.3 ESA requirements

To make sure that an ESA is long enough for a train to be able to brake entirely the following requirement must hold:

$$ESA.length > bp \tag{10.4}$$

10.3.4 Train max speed checking

The error of using discrete time updates are also reflected in the speed checking system of the TCC.

Due to the delay of Δt seconds in the simulation the train could theoretically spend this time accelerating at max acceleration even though it has passed the max velocity limit.

Using the current set of requirements, the speed checking system only checks if $train.speed > train.maxSpeed$. In the following it is assumed that $train.speed = train.maxSpeed$. Δt seconds will pass before the control system again checks the speed of the train. The speed has then exceeded the max speed by:

[1] The average braking capacity of passenger trains found by searching the web. This could easily vary from one train system to another.

$$v_{err} = a * \Delta t$$

Therefore the control system must check if the following property is true:

$$train.speed + v_{err} > train.maxSpeed$$

10.3.5 Segment max speed checking

As described in the *Domain description* (see chapter 3) a segment defines its own max allowed speed.

In this system a train is able to exceed the max speed on a segment but only temporarily. This happens when a train passes from one segment to another where the new segment has a lower max speed than the previous one. This situation is not specifically handled, but could easily be handled by checking the speed of the next segment, and calculate the point to begin deceleration.

In this system the speed of the train is eventually reduced to the allowed limit but not instantly. It is ensured though, that any train can brake at any brake point at the max speed of the train (see section 10.3.2).

One could argue that this puts an unnecessary restriction in creating new railway scenarios in the system editor, but, keeping in mind that safety is the issue in this report, we have chosen this approach.

10.3.6 Acceleration considerations

Acceleration is the only property that the control system can change in the physical train (by applying speeder / brakes). This means that the acceleration is not time dependent and thereby not automatically calculated from the value in last time update (like in the case with velocity).

A problem in using a discrete time model arises if the acceleration is not constant in the time interval between the updates (this is possible if the control system is updated concurrently and independently of the physical system). The error caused by this can of course be minimized by choosing a low tick interval.

The error in position is calculated by the following scenario where a is set to a new value during a tick $t = t_0 + t_1$. The tick is separated in t_0 (where the acceleration is a_0) and t_1 (where acceleration is a_1):

$$errpos = \frac{1}{2} * \Delta a * t_0 + a_1 * t_0 * t_1$$

Δa is the change in acceleration.

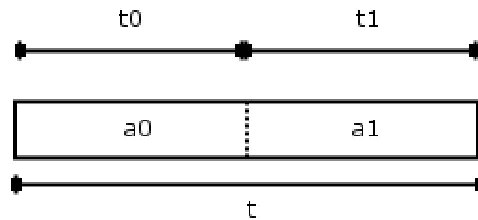

Figure 10.4: Acceleration change in time interval t

t_0 is the time from the beginning of the interval to the time where the acceleration is changed. So if the acceleration is changed during a tick, the entire tick interval is handled as if it had the same acceleration as in the end of the interval. t_0 is thereby the time spent with the wrong acceleration value.

In any case this update interval must be very small compared to the speed of the train. This is a consequence of the fact that the control system is based on the input from the physical model. If this system is only updated, e.g. once every minute, the control system will probably never notice that the train is nearing the segment brake point. This corresponds to if a train driver was only allowed to open his eyes once a minute.

Another approach to eliminate the calculation error is to update the control system at each tick after the physical system is updated. This synchronizes the physical system and control system updates.

The latter approach is chosen in this project. It does not make any sense that the control system is checking several times on a state that does not change.

Chapter 11

Control system design

In this chapter the control system algorithm is explained. A glossary is provided in chapter 12 to explain the terms that are used in the algorithm.

11.1 Control system algorithm

This and the following sections describe the algorithm of the distributed control system. The algorithm is derived from the requirements of the control system (solutions to these) described in chapter 6.

Since the control system is formed by the train control computers (TCCs) and the switch boxes (SBs), the algorithm for the control system is formed by the algorithm for the TCCs and the SBs. These algorithms are described in the following sections. First it is explained how reservations are obtained.

11.1.1 Obtaining reservations

This section describes how reservations are obtained in the control system.

Safety aspects in TCC

In general, when a train (`T1`) (figure 11.1) needs to enter the segment in front of it (`S1`), the TCC sends a segment request to the first SB in its driving direction (`SB1`). The SB then finds out if it may give the train a segment reservation for the segment and communicates the result back to the train.

The TCC only needs to know whether or not it has obtained a segment reservation for the segment it is about to enter. It does not know any other details about how reservations are managed by the SBs.

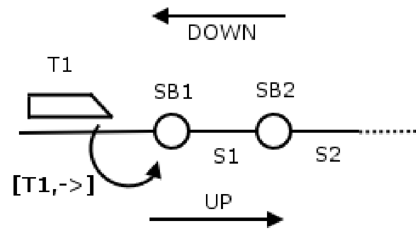

Figure 11.1: Basic TCC functionality

Safety aspects in SB

The SBs have another notion of reservations, since they must make sure that deadlock (see section 5.5) is avoided. This is done by having three different kinds of reservations: *line reservation*, *branch reservation* and *line-segment reservation*. These are described in the glossary in chapter 12 and in the scenario below.

A sample scenario

In the following a small scenario is explained where a train drives from one branch on a railway line to another. The point is to show what happens in the SBs along the route when the train progresses and continuously asks the SBs for permission to enter the next segment.

Figure 11.2, 11.3, 11.4 and 11.5 show a part of a railway line. These figures are used in this sample scenario.

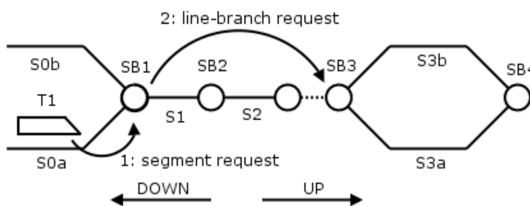

Figure 11.2: Line-branch request

First request Before the train can enter segment `S1` it must obtain a segment

reservation for S1. Therefore the TCC requests the SB in front of it for permission to enter the segment.

Check reservation SB1 checks its local state for any existing line reservation. There may only be one line reservation at a time in a SB.

SB1 **requests line and branch** SB1 sends a message to SB3 to obtain a reservation for the line between them and the branch S3a guarded by SB3.

SB3 **replies** Line reservations are saved in the two single line guards (SB1 and SB3). The branch reservation of S3a is only saved in SB3. This is because each branch is unidirectional and therefore only need a guard in one end. Line-segment reservations are not saved in a SB since they only exist to ensure that the line segment is prepared for a train to pass. Besides, only one train is allowed to be at the single line at a time, so it is not necessary to save the line-segment reservations in all intermediate SBs.

SB1 **prepares first segment** If S1 needs preparation (i.e. the point needs to be switched to the appropriate branch) then this is handled before a positive reservation message is returned to the train.

If the single line ends at an ESA there is no branch segment. Therefore only the line reservation should be obtained in SB3.

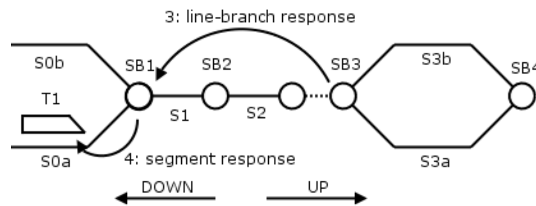

Figure 11.3: Line-branch response

Train enters S1 If the reservations are obtained, SB1 returns a segment reservation for S1 to the TCC of the train. Then the train can proceed to S1. Otherwise SB1 returns negative response to the TCC indicating that the segment reservation could not be obtained.

Train progresses along the line Each time the train is about to enter a new segment, it requests a reservation for that segment at the next SB. If anything requires preparation (like closing a crossing) before the train can pass, this is done before a positive response is sent from the SB to the train.

Train enters S3a When the train is located at the segment just before SB3 and wants to enter S3a, it request SB3 for a segment reservation for S3a. This is given when S3a is prepared (as with line-segment reservations). Note that the branch reservation is already obtained. Therefore the S3a only needs to be prepared.

Line reservation cleared When the train enters S3a and thereby exits the single line, SB3 sends a dereservation message to SB1 to clear the line reservation.

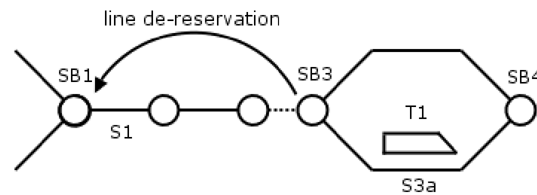

Figure 11.4: Line dereservation

Branch reservation cleared When the train exits the S3a branch segment and passes the sensor of SB4, SB4 sends a dereservation message for the branch to SB3.

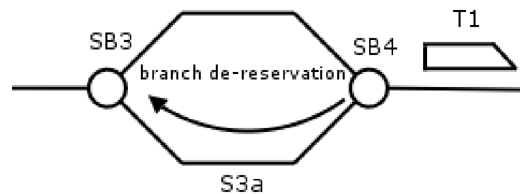

Figure 11.5: Branch dereservation

The next section explains in detail the algorithms used in the TCC and the SB.

11.2 Train Control Computer algorithm

The TCC algorithm consists of three parts besides an *idle state* as shown in figure 11.6. This algorithm is executed each time the TCC is ticked.

At each tick the TCC checks the speed of the train (section 11.2.1), clears old segment reservations (section 11.2.2), and handles segment reservations (section 11.2.3). These three parts are executed sequentially.

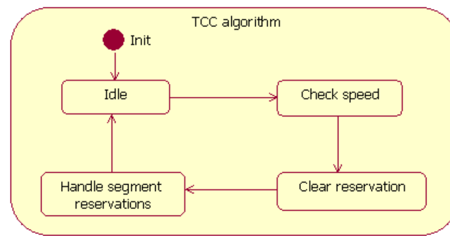

Figure 11.6: The TCC algorithm

11.2.1 TCC - Speed checking

It is the job of the TCC to make sure that the train does not drive too fast. The speed checking algorithm is shown in figure 11.7.

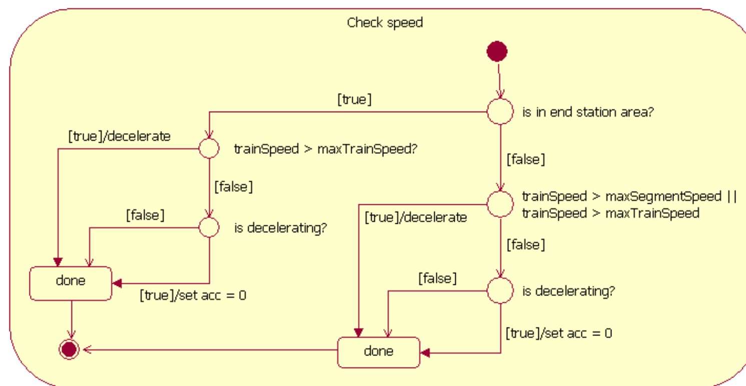

Figure 11.7: TCC speed checking

The speed checking algorithm consists of the following steps:

- Check if train drives too fast
- If so, then decelerate
- Else check if the train is decelerating
- If so, then set acceleration to zero
- Else do nothing

When the algorithm is executed, it first checks if the train is entirely in an ESA or not. The reason for that is that the rules for determining whether a train

drives too fast is different for ESAs and segments. In both cases a train is not allowed to drive faster than the max allowed speed of the train. A train located on a segment also drives to fast if it exceeds the max allowed speed for the segment.

11.2.2 Clearing reservations

When a train leaves a segment and enters the next, it must have a segment reservation for the next segment. This reservation is removed when the train enters the segment.

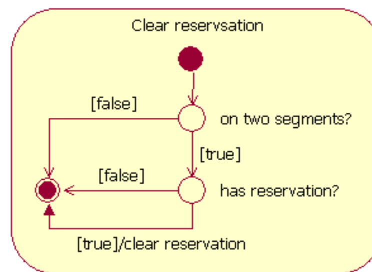

Figure 11.8: TCC clear reservations

Figure 11.8 shows how this is done. Whenever a train is located on two segments (or a segment and an ESA), the segment reservation is removed if it exists, i.e. if it has not already been removed.

11.2.3 Reservations handling

It is the job of the TCC to make sure that the train has a segment reservation for the segment it is about to enter. Therefore the TCC makes requests for segment reservations and brakes the train if it is about to enter a segment it does not have a segment reservation for.

The terms reservation point and brake point (see chapter 12) are used in the following, see figure 11.9.

When a train (`T1`) passes the reservation point(R.P.) the TCC requests `SB2` for a segment reservation of `S2`. If the train has not obtained the segment reservation when it passes the brake point(B.P.), the TCC brakes the train. It may proceed when it has obtained a segment reservation for `S2`.

The algorithm performed at every tick is shown in figure 11.10.

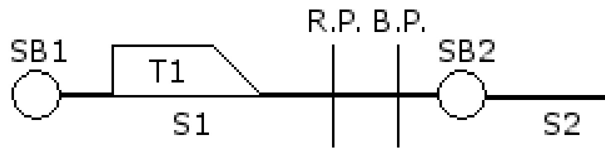

Figure 11.9: Reservation- and brake point

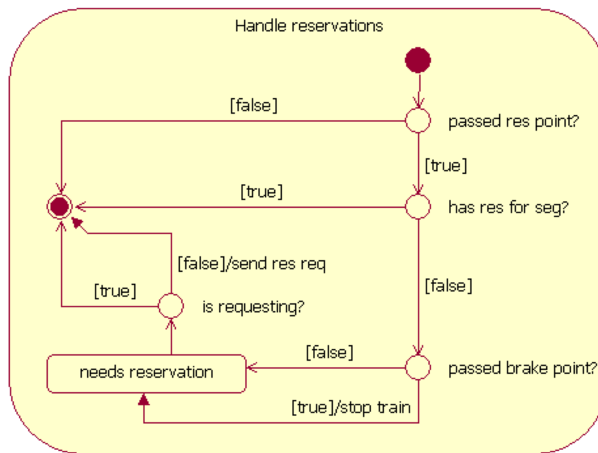

Figure 11.10: TCC reservations handling

11.3 Switch box algorithm

The job of a SB is to manage reservations, shift points and operate the barriers at a crossing. All this is included in the switch box algorithm.

The algorithm is shown in figure 11.11. At every tick the sensor process is performed. After that the prepare process is performed if the SB currently is preparing a segment. Otherwise the message process is performed.

The three parts (processes) are shortly described as:

Sensor process Check sensor state

Message process Receive and handle messages

Prepare process Preparing the next segment

The sensor-, message- and prepare-processes are explained in the following.

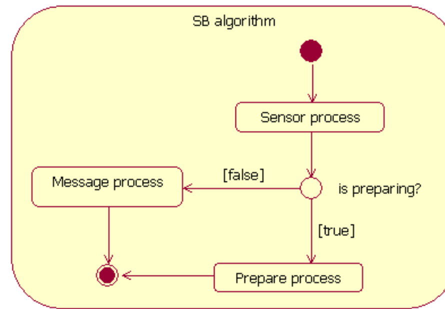

Figure 11.11: The switch box algorithm

11.3.1 Sensor process

The sensor process is shown in figure 11.12.

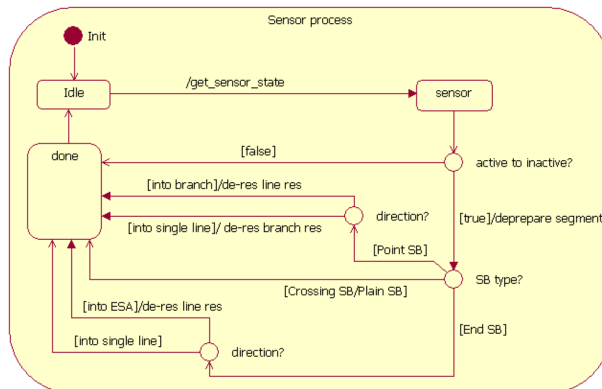

Figure 11.12: The sensor process

The sensor process starts in state `idle`. First the state of the sensor (active/inactive) is fetched. This value is compared to the last state. If it has turned from active to inactive, a train has just passed it.

If a train has just passed, the following will happen:

- The segment the train left is deprepared
- If the SB at the sensor is a point SB and the train has entered a branch segment, the SB sends a line dereservation message to the single line guard at the opposite end of the single line.
- If the SB at the sensor is a point SB and the train has left a branch segment

(entered a single line), the SB sends a branch dereservation message to the single line guard at the opposite end of the branch segment.

- If the SB at the sensor is an end SB and the train has driven into the ESA, the SB sends a line dereservation message to the single line guard at the opposite end of the single line.
- If the SB at the sensor is a crossing SB or a plain SB, nothing is dereserved.

To see illustrations of all these events please refer to section 11.1.1: "*A sample scenario*".

11.3.2 Message process

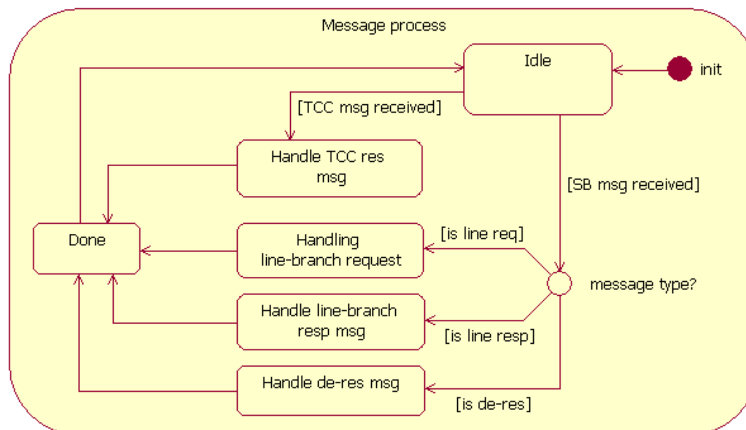

Figure 11.13: The switch box message algorithm

The switch box algorithm is explained through the state chart diagram in figure 11.13. Four of the states (all except `Idle` and `done`) are decomposed in separate state chart diagrams and are described in the following.

- A SB is `idle` on start up and ready to receive messages. A received message comes from either a TCC or another SB.
- If a TCC is the sender, the message is handled as described in section 11.3.3.
- If a SB is the sender, the message is either a line-branch request, a line-branch response or a dereservation message.

The individual handling of different message types is described in the following sections.

In all of the mentioned situations the SB ends up in the state `done` and go to the state `idle`. These two states do not differ but are created for visual reasons.

11.3.3 Handle TCC request message

When a SB receives a message from a TCC it is handled as shown in the state chart diagram in figure 11.14. This message is always a segment request. The SB handles this request and sends the response back to the TCC.

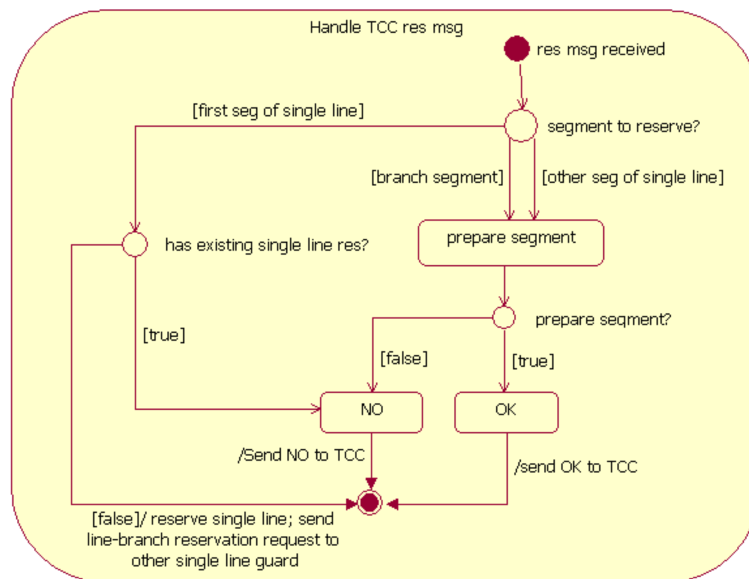

Figure 11.14: Handling a message from a train

- The SB first checks what kind of segment that is to be reserved. It can decide this by its location in the network.
- If it is a **branch segment** or **a line segment** (except the first segment in the single line), the SB only has to prepare the segment.
- If the preparation fails, the SB should respond with a denial of the segment reservation (`NO`). Otherwise it should respond with a segment reservation (`OK`).
- If the segment is the first in a **single line**, the SB needs to obtain a line reservation and a branch reservation. This is done by first checking if it has an existing line reservation.

- If so the SB should respond with a denial of the segment reservation (`NO`). Otherwise the SB should reserve the single line locally and send a line-branch request to the opposite single line guard.

The handling of a line-branch request in the opposite SB is covered in section 11.3.4 below. The handling of the associated response is covered in section 11.3.5.

11.3.4 Handling a line-branch request

When a SB receives a line-branch request it reacts as shown in figure 11.15. If it receives this kind of request it is a single line guard, which is either a **point SB** or an **end SB**.

- First it check for an existing line reservation.
- If it has, then `NO`.
- Else if it is an end SB, then `YES`.
- If it is a point SB, it check if it has an existing branch reservation. If it has, then `NO`, else `YES`.

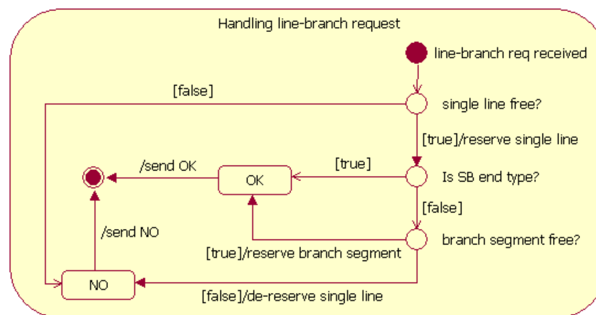

Figure 11.15: Handling a line-branch request

11.3.5 Handle line-branch response

When a SB receives a response message it reacts as shown in figure 11.16.

- If the response is negative(`NO`) the SB cancels (dereserves) the local line reservation and sends a denial of the segment reservation (`NO`) to the TCC.

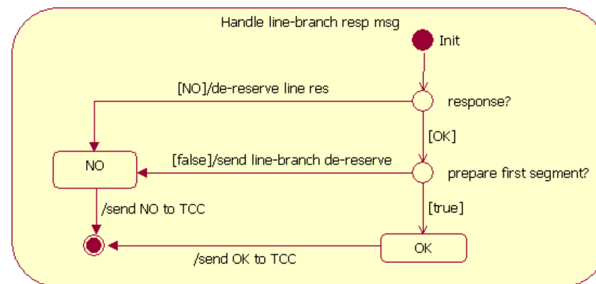

Figure 11.16: Handling a response

- If the response is positive (`OK`) the SB prepares the first segment of the single line (see section 11.3.7).
- If the preparation fails, the SB first sends a line-branch de-reservation message to the opposite single line guard, dereserve its local reservation and then sends a denial of the segment reservation (`NO`) to the TCC.
- Otherwise it sends a segment reservation (`OK`) to the TCC.

11.3.6 Handle SB dereservation message

When a SB receives a dereservation message it reacts as shown in figure 11.17.

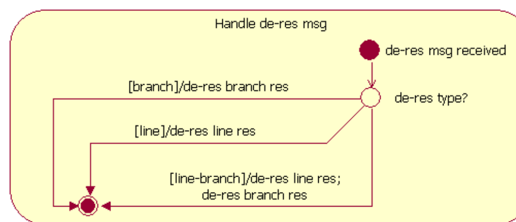

Figure 11.17: Handle dereservation msg

- If the dereservation message is a branch dereservation message, the SB cancels its branch reservation.
- If it is a line dereservation message, the SB cancels its line reservation.
- If it is a line-branch dereservation message, the SB cancels both its line reservation and its branch reservation.

11.3.7 Prepare process

Figure 11.18 shows the prepare process.

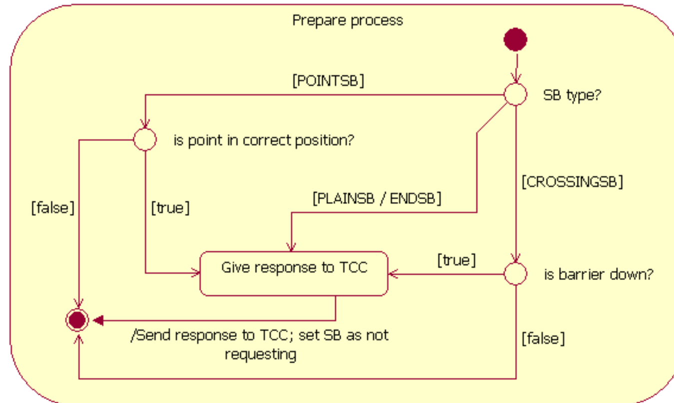

Figure 11.18: The prepare process.

- First the type of the SB is found. If it is a point SB it is tested if the point is in the correct position. If so, response is returned to the TCC. Otherwise nothing further is done.
- If the SB is a crossing SB it is tested if the barriers are down. If so, `YES` response is returned to the TCC. Otherwise nothing further is done.
- If the SB is a plain SB or an end SB, `YES` is immediately returned to the TCC.

Chapter 12

Glossary

This chapter presents a glossary that uses figure 12.1 as basis for the explaination of terms. The figure shows a part of a railway line. The circles denotes *switch boxes (SBs)* and the lines denotes *segments*. The dots denotes that more *segments* could be placed here (but no junctions).

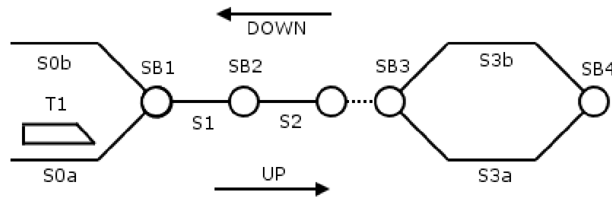

Figure 12.1: A part of a railway line

SB A *switch box* is a controlling unit located at the boundary between *segments* (described in section 3.7). `SB1`, `SB2`, `SB3` and `SB4` are *switch boxes*.

TCC A *Train Control Computer* is a computer in a train. It is used in the control system to reserve *segments* before entering them, stopping the train when it tries to enter a segment it has not reserved etc. (see also section 3.4). `T1` is a train. The *TCC* in `T1` is also denoted `T1`.

Line segment A *line segment* is a segment in a *single line*. `S1` and `S2` are *line segments*.

Branch segment A *branch segment* is a segment that connects two junctions and is a branch of both junction (remember a junction always has two branches), i.e. the stem of a junction is not a *branch segment*. `S0a`, `S0b`, `S3a` and `S3b` are *branch segments*.

Single line A *single line* is a number of coherent *segments* on which trains are allowed to move in both directions. The line connects either the stem of two junctions, the stem of a junction and an end station area, or two end station areas if no junctions exist in the railway line. The *segments* between `SB1` to `SB3` form a *single line*.

Single line guard A *single line guard* is a *SB* that is located at the end of a *single line*. I.e. the *SB* is either a *point SB* or an *end SB*. `SB1`, `SB3` and `SB4` are *single line guards*.

Reservation A *reservation* in general is a permission to enter a part of the railway line. In the *SBs* there are three different kinds of *reservations*, which are *line-segment reservations*, *branch reservations*, and *line reservations*. The *TCCs* only knows one kind of reservations called *segment reservations*.

Line-segment reservation A *line-segment reservation* is a *reservation* of a certain *line segment*. The *reservation* is given when the segment has been *prepared*. The only situation for a denial of a *line-segment reservation* is, if the segment for some reason could not be *prepared*.

Branch reservation A *branch reservation* is a *reservation* of a certain *branch segment*.

Line reservation A *line reservation* is a *reservation* of a certain *single line* in a certain direction.

Segment reservation A *segment reservation* is a *reservation* of a certain segment in a certain direction. This notion is only used in the TCCs.

Request message A *request message* is either a *line-branch request* or a *segment request*.

Line-branch request A *line-branch request* is a combined request for a *line reservation* and a *branch reservation* send from one *single line guard* to the opposite *single line guard* of the *single line*. If the *single line* ends at an *end station area* there is no *branch segment*. Therefore only the *line reservation* is requested.

Segment request A *segment request* is a request for a *segment reservation* send from a *TCC* to a *SB*.

Response message A *response message* is either a *line-branch response* or a *segment response*.

Line-branch response A *line-branch response* is a response to a *line-branch request* telling whether both a *line reservation* and a *branch reservation* could be obtained or not. If the *single line* ends at an *end station area* there is no *branch segment*. Therefore the *line-branch response* only tells if a *line reservation* could be obtained or not.

Segment response A *segment response* is the response to a *segment request* telling whether a *segment reservation* could be obtained or not.

Dereservation message A *dereservation message* is either a *branch dereservation message*, a *line dereservation message* or a *line-branch dereservation message*.

Branch dereservation message A *branch dereservation message* is a message telling a *SB* to cancel its *branch reservation*.

Line dereservation message A *line dereservation message* is a message telling a *SB* to cancel its *line reservation*.

Line-branch dereservation message A *line-branch dereservation message* is a message telling a *SB* to cancel its *branch reservation* and its *line reservation*.

Prepare a segment When a train is to enter a new segment the segment needs to be prepared. If the *SB* between the two *segments* is a *point SB*, preparing the new segment means to switch the point so that the point is connected to the correct branch. If the *SB* between the two *segments* is a *crossing SB*, preparing the new segment means to close the barriers and turn on the signals at the *crossing* etc. according to the algorithm (see section 9.4). If the *SB* between the two *segments* is a *plain SB* or an *end SB* nothing is done when the segment is prepared.

Deprepare a segment When a train has left a segment, the segment must be deprepared. The only time something happens at a depreparation is when the passed SB is a crossing SB. Then the crossing should open again. If a point is switched when a segment is prepared, the point is shifted back when the segment is deprepared.

Reservation point A *reservation point* is a point at a segment (do not confuse this with the point at a junction). The point is not physically visible and is expressed as a distance to the end of the segment. When the train passes this point (and not before), the *TCC* sends a *reservation request* for the next segment to the *SB* between the segments. A segment has a *reservation point* in both directions. A train passes the *reservation point* before it passes the *brake point*.

Brake point A *brake point* is a point at a segment (do not confuse this with the point at a junction) at which the *TCC* should brake the train if it does not have a *reservation* for the next segment. The point is not physically visible and represents a distance to the end of the segment. A segment has a *brake point* in both directions. A train passes the *reservation point* before it passes the *brake point*.

Point ticks *Point ticks* denotes the number of seconds it takes for a point to switch from one branch to the other.

Signal ticks *Signal ticks* denotes the number of seconds the signal at a crossing is turned on before the barriers begin to close. In section 9.4 signal ticks are shown in the state diagram as *signals_only*.

Barrier ticks *Barrier ticks* denotes the number of seconds it takes for the barriers at a crossing to either close or open. In section 9.4 barrier ticks are shown in the state diagram as *bars_moving*.

Chapter 13

GUI design

This chapter briefly describes the overall view of graphical user interface (GUI) without going into details. The design conforms with the simulator requirements described in chapter 7.

All the figures showing the GUI designs can be found in appendix A.

13.1 Train simulator

Figure A.1 shows the design of the train simulator. The two menus are shown as they would appear when they are expanded.

The figure shows how the layout of the railway line are placed in the top. The ESAs are shown as green rectangles. SBs are shown as circles, segments as lines, and trains as colored pentagons placed on segments.

At a junction both branch segments are shown and the position of the coherent points are shown using small colored rectangles above and below it. A green rectangle denotes that the point is positioned at the branch segment near the rectangle. Red means that the point is at the opposite branch and orange means that the point is in an intermediate position.

The crossings are shown as two parallel vertical lines enclosing a SB (circle). The colored rectangles (green, orange or red) above and below the crossing show whether the barriers are closed or not. Red means that they are open, orange that they are moving, and red that they are closed.

Below the layout of the railway line, rows of buttons are showed. These corresponds to the entities of the railway line. When one of the buttons is pressed the static, dynamic and control properties of the selected entity are showed below the buttons.

13.2 Configuration editor

Figure A.2 shows the GUI when the configuration editor is started and a train is selected.

The list in the left side shows all the configurations available in the system.

Importing and exporting configurations between the configuration editor and a XML file is done using the `Import` and `Export` buttons.

When a configuration is selected, pressing the `load` button shows the train simulator GUI with the selected configuration loaded into the simulator. A selected configuration can be deleted by pressing the `Delete` button.

When a configuration is selected, pressing the `Is wellformed` button checks whether the selected configuration is wellformed.

New configurations are created in the configuration editor by pressing the `New` button. By doing that the smallest possible railway line is shown with the two ESAs, two SBs and one segment. Each of these entities are shown as buttons containing their name. By pressing an entity button its properties are shown above the configuration and these properties can now be changed. By pressing the `update` button the changed properties are saved.

A SB/segment pair is added by selecting the type of the SB and then pressing the `Add SB/segment` button. In doing so both a segment and a SB are added at the right just before the rightmost SB. Segments can only be deleted from the right. It is done by pressing the `delete SB/segment` button.

A train is added by pressing the `Add train` button. The properties of a train can be changed by pressing the proper train button.

When a configuration is complete and its name is entered along with the reservation point and brake point, it can be saved by pressing the `Save configuration` button.

Chapter 14

Assumptions and invariants

This chapter lists the assumptions and invariants identified for the system to work. This list of invariants and assumptions is a summary of the decisions made in chapters 10 and 9.

1. It is assumed that TCCs and SBCCs can communicate using some existing communication service like a GSM network.
2. It is assumed that a TCC knows it's current position by either measuring length from last station or using GPS.
3. It is assumed that trains cannot collide when inside and ESA.
4. It is assumed that SBs have some interface to existing sensory equipment on the track.
5. The distance from a segment border to a reservation point must be less than the length of the segment ($rp < segment.length$)
6. The brake point must be closer to the segment border than the reservation point ($rp > bp$).
7. The train must not be on two segments when crossing a reservation point ($rp < segment.length - train.length$)
8. At the most, a train can be on two segments at a time. This means that the train must be shorter than any segment ($train.length < segment.length$).
9. The length of the shortest train must be longer than the largest possible collision detection error ($s_{col} < train.length_{min}$)
10. A train must be able to stop before entering the next segment when it starts braking at the brake point. Therefore this length has to be longer than the max brake length + the max simulation error made by discrete time updates ($bp > s_{brk} + s_{err}$).

11. A train must be able to brake entirely in an ESA ($ESA.length > bp$).
12. The TCC must handle the error made by discrete time updates ($v_{train} + v_{err} < v_{max}$)

Chapter 15

RSL modelling method summary

This chapter briefly lists the method used to develop the model in the following chapters. This is not a general description of transformation of RSL models so all trivial steps are left out and only changes specific to this system / model are listed.

For a model-specific description of the modules, their function, and the development of the model in this project please refer to chapter 16.

For a more detailed and general description of RSL model transformations / refinements please refer to *appendix B*.

The model is constructed to satisfy two conditions:

1. Suitable for easy translation into JAVA. Therefore the use of RSL specialties like subtypes - which are not directly implementable in JAVA - is minimized as much as possible.

2. Structured for proving safety. Therefore axioms and invariant predicates are added indicating what should hold for this system. To see the theory of how to use these predicates to prove safety please refer to chapter 23.

15.1 Initial specification

This section describes the development of the initial abstract specification.

In this phase all modules (schemes) are flat specifications. This means that no object oriented structure is used yet. All modules define a main type on which all functions in the module are based. One exception though, is the *Types*

module which is only a utility module.

15.1.1 Initial model overview

Figure 15.1 shows an illustration of the schemes of the initial model. The arrows indicate which schemes are parameterized by other schemes.

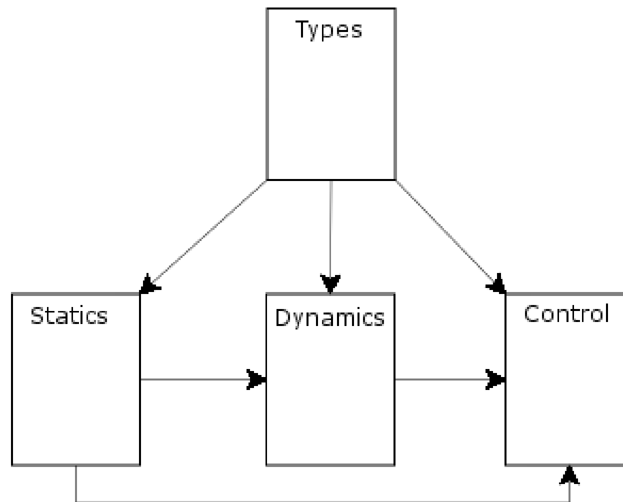

Figure 15.1: Initial model structure

15.1.2 Types

A *Types* module is defined which contains all common types for all modules. The types module also contains utility functions that only operates on types defined in the *Types* module itself. All other modules is parameterized by this module.

15.1.3 Statics

This scheme defines the static physical aspects of the system.

- This scheme is **parameterized** by the *Types* module.

  scheme Statics(T : Types)

- The **type of interest** contains the entire static system configuration.

type
Configuration

- A **constant** containing the actual configuration instance is defined. This is necessary to be able to express some properties of the actual configuration instance.

value
conf : Configuration

- *Functions* that extract information (**observers**) from the main configuration type are added.

value
obs_i : $\text{T}_j \times .. \times$ Configuration $\rightarrow \text{T}_n$

- **Wellformedness requirements** are defined in terms of a boolean function (predicate). The wellformedness predicate defines all the constraints of a wellformed **Configuration** type. This predicate is based on properties observed by observer functions defined above. The predicate defines the relationship between the observer functions.

value
is_wf : Configuration $\rightarrow$ **Bool**
is_wf(con) $\equiv$ p(..,obs_i(..,con))

- An **axiom** is added stating that the configuration instance `conf` must be **wellformed**. This represents that a configuration loaded from an external supplier - such as a XML file - must be checked for wellformedness before use.

axiom
[conf_is_wf]
is_wf(conf)

15.1.4 Dynamics

This scheme defines the dynamic physical aspects of the system.

- The scheme is **parameterized** with the *Statics* and *Types* schemes.

scheme Dynamics(T : Types, S : Statics(T))

- The **type of interest** contains the composite state of the entire physical system.

  type
  State

- A **constant** symbolizing the **initial state** of the system is defined. This is necessary to be able to express some properties of this state.

  value
  initState : State

- **Observer** and **generator functions** are defined to be able to extract / change information in the main type.

  value
  obs_i : $\text{T}_k \times .. \times \text{State} \rightarrow \text{T}_n$,
  gen_i : $\text{T}_k \times .. \times \text{T}_n \times \text{State} \rightarrow \text{State}$

- **Wellformedness requirements** is defined in terms of a predicate based on the observer functions. This predicate also needs a configuration as input because the dynamic properties are based on the static domain. Therefore a state is only wellformed if the underlying static configuration is wellformed.

  value
  is_wf : State × S.Configuration → **Bool**
  is_wf(s,con) ≡ S.is_wf(con) ∧ p(..,obs_i(..,s))

- Some requirements for the **initial state** of the system are - like the wellformedness predicate - defined as a predicate based on the observer functions. This predicate includes the wellformedness requirements. This also needs a configuration as input.

  value
  init_req : State × S.Configuration → **Bool**
  init_req(s,con) ≡ is_wf(s,con) ∧ p(..,obs_i(..,s))

- A **predicate** *safe* is also defined. This predicate defines what a safe physical state is. This is usually the fact that no entities are **colliding** or **derailing** (in the railway domain). This predicate also includes the *is_wf* predicate.

  value
  safe : State × S.Configuration → **Bool**
  safe(s,con) ≡ is_wf(s,con) ∧ p(..,obs_i(..,s))

- **Observer/generator axioms** are added to define the relationship between these. **Preconditions** for the observers and generators are added to the observer_generator axioms.

```
axiom
  [ obs_i_gen_j ]
    obs_i(..,gen_j(..,s)) ≡ val_expr
  pre precond_j(...)
```

- Axioms are added stating that **generators preserves wellformedness** if the preconditions are satisfied.

```
axiom
  [ wf_pres_gen_i ]
    ∀ s : State,
          con : S.Configuration •
      is_wf(s,con) ∧ precond_i(..) ⇒ is_wf(gen_i(..,s)))
```

- An axiom is added requiring the **initial state** to satisfy the initial state requirements. The constant `S.conf` is used as parameter.

```
axiom
  [ init_state_req ]
    init_req(initStat,S.conf)
```

15.1.5 Control

This scheme defines the state of the entire control system.

- The scheme is **parameterized** by the *Types* module, the *Statics* module, and the *Dynamics* module.

```
scheme Control(T : Types, S : Statics(T), D : Dynamics(T,S))
```

- A **type of interest** is defined to contain the entire control system state.

```
type
  ControlState
```

- A **constant** is defined to represent the initial state of the control system.

```
value
  initControlState : ControlState
```

- **Observer** and **generator functions** are defined to extract / change information in the main type.

value
obs$_i$: T$_k$ × .. × ControlState → T$_n$,
gen$_i$: T$_k$ × .. × T$_n$ × ControlState → ControlState

- A **wellformedness predicate** is defined for the control state. For a control state to be wellformed both configuration and dynamic state must also be wellformed.

value
is_wf : ControlState × D.State × S.Configuration → **Bool**
is_wf(cs,ds,con) ≡ D.is_wf(ds,con) ∧ p(..,obs$_i$(..,cs))

- A predicate *consistent* is defined. It defines the relationship between the physical state and the control state, and perhaps some safety measures which only concern the physical state.

value
consistent : ControlState × D.State × S.Configuration → **Bool**
consistent(cs,ds,con) ≡ is_wf(cs,ds,con) ∧ p(..,obs$_i$(..,cs))

- A predicate *init_req* defines the requirements for the initial control state.

value
init_req : ControlState × D.State × S.Configuration → **Bool**
init_req(cs,ds,con) ≡ is_wf(cs,ds,con) ∧ p(..,obs$_i$(..,cs))

- An axiom is added stating that the **initial state** must satisfy the `init_req` predicate.

axiom
init_req(initControlState,D.initState,S.conf)

- **Observer / generator axioms** are added defining the relationship between observers and generators. Preconditions are also added to these axioms.

axiom
[obs$_i$_gen$_j$]
∀ cs : ControlState •
    obs$_i$(..,gen$_j$(..,cs)) ≡ val_expr
pre precond$_j$(..)

- An **axiom** is added for each generator stating that all generators must **preserve wellformedness** if the preconditions are satisfied.

axiom
  [wf_pres_gen$_i$]
  ∀ cs : ControlState,
    ds : D.State,
    con : S.Configuration •
      is_wf(cs,ds,con) ∧ precond$_i$(..,cs) ⇒
        is_wf(gen$_i$(..,cs),ds,con))

15.2 Type decomposition

This section explains the changes applied to the schemes during decomposition. The schemes are decomposed to obtain a more object oriented structure and better overview by grouping related observers and generators in separated schemes.

15.2.1 Decomposed model overview

Figure 15.2 shows an illustration of the decomposed scheme structure.

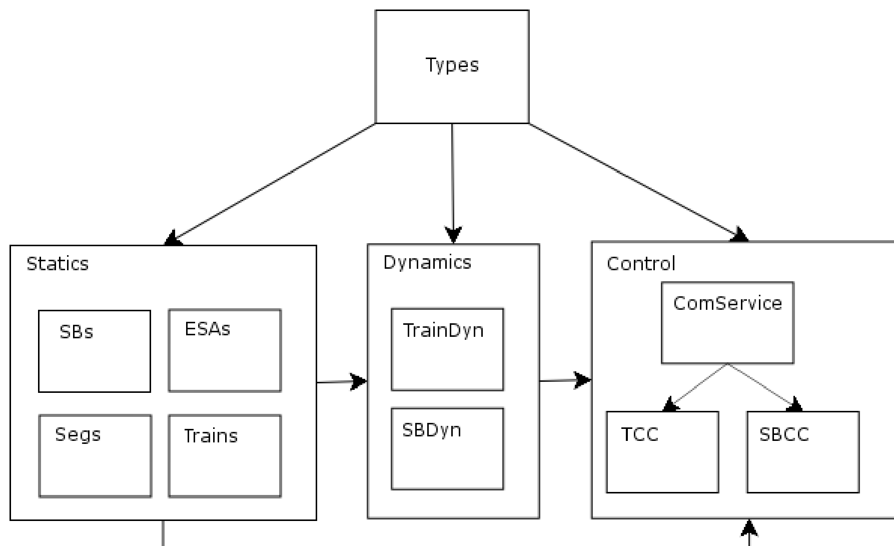

Figure 15.2: Decomposed model structure

15.2.2 Statics

The statics module is decomposed by performing the detailed steps in appendix B. This is a short summary of the decomposition steps:

- The **Configuration** is changed to a product of several smaller types. A new module is created for each type and the observers and generators which cover the area of this type, are copied to the new modules. The new sub modules are given their own *is_wf* predicate and also a constant symbolizing the major type instance.
- For each of the new modules an **object** is created in the statics module so the types and functions in the sub modules are accessible.
- The type of interest in statics, which before was a sort, is now a **product of the types** of interest in the new modules (that all are sorts).

```
object
  O₁ : Sub₁,
  O₂ : Sub₂
type
  Configuration = O₁.T₁ × O₂.T₂
```

- The actual configuration value in *Statics*, which did not have a value before, is now given a concrete value being the product of the configuration instance values in the new modules.

```
value
  conf : Configuration = (O₁.confT₁, O₂.confT₂)
```

- All functions in *Statics*, that also exist in one of the new modules, are changed so that they call the similar functions in the new modules. All other functions in the decomposed statics module are left unchanged. This rule does apply to *is_wf* and *init_req* which are moved to the appropriate sub schemes.

```
value
  obs₁ : .. × Configuration → ..
  obs₁(..,(t₁,t₂)) ≡
    O₁.obs₁(..,t₁)
```

- The wellformedness predicate *is_wf* is changed so it calls the *is_wf* functions in the new modules along with those wellformedness functions that are not copied to one of the new modules (because they use observers from more than one of the new modules). The wellformedness functions, that have been copied to one of the new modules, are deleted from the statics module.

```
value
    is_wf : Configuration → Bool
    is_wf((t1,t2)) ≡
        O1.is_wf(t1) ∧
        O2.is_wf(t2) ∧
        p1(t1,t2) ∧
        ...
```

15.2.3 Dynamics

The dynamics module is decomposed like the statics in above section.

- One difference though is that dynamics also has a *init_req* predicate. It is handled as the *is_wf* predicate in statics, so it becomes a conjunction of the corresponding functions in the sub modules.

```
value
    init_req : State × S.Configuration → Bool
    init_req((t1,t2),con) ≡
        S.is_wf(con) ∧
        O1.is_wf(t1) ∧
        O2.is_wf(t2)
```

15.2.4 Control

Control is in this case decomposed a little differently than the statics and dynamics schemes. In these schemes each sub module represented a collection of entities contributing to the entire state or configuration. Since the control system is distributed, each controlling entity is modelled by having its own module.

- To model the many different entity states a map is created for each type of controlling entity (TCC, SB) to contain all the entity states of that type. The **ControlState** is then a product of these state maps.

```
object
    CE1 : ControlEntity1,
    CE2 : ControlEntity2

type
    ControlState = CE1Map × CE2Map

    CE1Map = CE1ID →m CE1.State,
    CE2Map = CE2ID →m CE2.State
```

- Like in the *Statics* scheme the sub modules have their own initial value which must satisfy the *init_req* predicate. But another predicate is needed ensuring that all states in the maps are initially the initial values from the sub schemes.

```
value
is_ce1_init : ControlState → Bool
is_ce1_init((ce1map,ce2map)) ≡
(
    ∀ state : CE1.State •
        state ∈ rng(ce1map) ⇒ state = CE1.initState
)
```

- And then the predicate ensuring that all states satisfy the `init_req()` predicate.

```
value
    all_ce1_initReq : ControlState → Bool
    all_ce1_initReq((ce1map,ce2map)) ≡
    (
        ∀ state : CE1.State •
            state ∈ rng(ce1map) ⇒ CE1.init_req(state)
    )
```

15.3 Concrete refinement

No changes specific to this model are applied during refinement to concrete data types. Only standard changes are performed. To see these please refer to Appendix B.

15.4 Imperative transformation

This section explains the changes made to the model during transformation to imperative notation.

15.4.1 Statics

The statics module is made imperative. This means that variables are introduced in all sub modules containing the type of interest.

- The variables are initialized with the constant which where defined to represent the actual configuration of the module.

type
T_1

value
$confT_1$

variable
$v_T_1 := confT_1$

- An axiom `[initial]` is added to the parent module `Statics` expressing that the predicate `is_wf()` must hold after initialization of the module.

axiom
[initial]
initialise post is_wf()

15.4.2 Dynamics

Dynamics is transformed exactly like `Statics` with the exception that it is the `init_req()` that is used in the axiom. This is because the dynamics module also has some requirements for the initial state and these include the *is_wf* predicate. Beside that, the variables are initialized with the initial state.

15.4.3 Control

Variables are also introduced in the sub modules of the *Control* module. One difference is though that the sub modules only contain a variable with one single state each.

- Object arrays are created to represent the many control entities in the system. Each control entity has their own state stored in variables. The object arrays are now used instead of the maps which where necessary before.

object
$CE_1[n : CE_1Index] : ControlEntity_1,$
$CE_2[n : CE_2Index] : ControlEntity_2$

- The types CE_1`Index` and CE_2`Index` are created together with two maps to be able to map from an array index to an entity ID.

type
  $\mathrm{CE_1Index} = \{| \; \mathrm{n} : \mathbf{Nat} \bullet \mathrm{n} > 0 \wedge \mathrm{n} \leq \mathbf{card}\ \mathrm{T.ce_1IDSet} \; |\}$,
  $\mathrm{CE_2Index} = \{| \; \mathrm{n} : \mathbf{Nat} \bullet \mathrm{n} > 0 \wedge \mathrm{n} \leq \mathbf{card}\ \mathrm{T.ce_2IDSet} \; |\}$

value
  $\mathrm{ce_1Index} : \mathrm{T.CE_1ID} \underset{m}{\overrightarrow{}} \mathbf{Nat}$,
  $\mathrm{ce_2Index} : \mathrm{T.CE_2ID} \underset{m}{\overrightarrow{}} \mathbf{Nat}$

- Like in the *Statics* and *Dynamics* modules an *[initial]* axiom is added ensuring consistency of the initial state. One difference is that this axiom is added in the sub modules of the control entities parent module.

axiom
  [initial]
    initialise post initReq()

15.5 Concurrent transformation

No concurrent transformation is necessary because we do not implement this system as parallel processes but use a sequential approach. The ideas for implementing this system as a concurrent system are discussed in appendix E.

Chapter 16

Initial Model

This chapter concerns the development of the initial abstract RSL model. First a section gives an overview of the model module structure. Then a section briefly describes how material produced in the analysis and design sections is used in creating the model.

The following sections describe the detailed step by step development of the initial model. This development follows the modelling method described in chapter 15. The entire model can be found in appendix F.

It should be noted that basic observers at this level are left unspecified because the main data structures are abstract sorts at this level of development.

16.1 Initial model structure

The structure of the model is illustrated in figure 16.1.

The figure illustrates the most important information of the modules that are to be developed. This model shows that:

Types a common types module which enables all modules to use same types.

Statics defines the physical parts of the system as segment, switch box, end station area, sensor, point, train, crossing and the physical relationship between these.

Dynamics defines the dynamic part (physical states) of the entities defined in **Statics**.

Control defines entities for controlling the physical domain. The entities defined are switch box control computer (SBCC) and train control computer

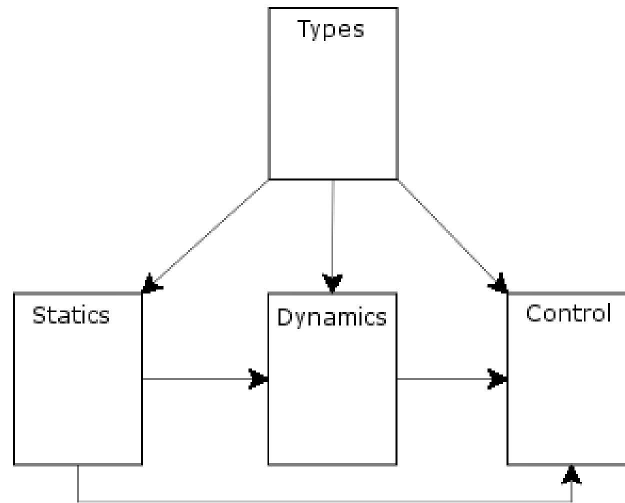

Figure 16.1: Initial model structure

(TCC). A communication service is also defined which enables these control entities to communicate.

16.2 From design to model

This section briefly lays out what was produced in the analysis chapters and how this is implemented in the model.

Assumptions and invariants These are summarized in chapter 14 and are typically in the form of a mathematical inequality or equation. They are mostly implemented as sub predicates for the wellformedness predicates *is_wf* in the model, and as preconditions to the generator functions to ensure that wellformedness is maintained.

Algorithms / UML state charts These are used in the main processes or functions for the entities in the system. They are the processes which are called when the system is updated by the *tick* function (The tick principle is briefly described in section 5.3).

Safety requirements These are implemented as sub predicates of the safety predicate *safe* in the model, and as preconditions for the generator functions to ensure that safety is maintained.

16.3 Types

The *Types* module contains all common types for all modules and some utility functions.

In the following the most important types is described. None of the utility functions is described. The entire Types module can be found in appendix F.1.1.

16.3.1 Tick

A type **Tick** and a constant *tick_interval* of this type is defined to specify the interval of seconds between (and the size of) each time update. This value is used in some predicates which ensure that the interval is small enough for collisions and brake point exceedings to be detected.

```
type
    Tick = real

value
    tick_interval : Tick
```

16.3.2 Ends

The railway line is defined to have two ends, that is the *high* and the *low* end. The direction from low to high is called *up* and the opposite direction is called *down*:

```
type
    End == HIGH | LOW,
    Direction == UP | DOWN
```

16.3.3 Entity IDs

The four entities ESA, SB, segment and train are all represented by their ID. For the two ESAs the ID is just the end at which the ESA is located. The three other entities are defined as an *ID*, which is a sort, but limited through the use of a subtype. The functions *sbIDLimit*, *segIDLimit* and *trainIDLimit* checks if an *ID* is a valid ID for a SB, segment or train.

```
type
    ID,
    ESAID = End,
```

```
    SBID = {| sb : ID • sbIDLimit(sb) |},
    SegmentID = {| seg : ID • segIDLimit(seg) |},
    TrainID = {| t : ID • trainIDLimit(t) |}
value
    sbIDLimit : ID → Bool,
    segIDLimit : ID → Bool,
    trainIDLimit : ID → Bool,
```

16.3.4 SB types

The *SBSegment* type is defined to hold the "place(s)" in the railway line after a SB. This can be a plain segment (*seg)*, the two branch segments at a point (*point)* or an ESA (esa).

The *SBType* defines the type of a SB.

```
type
    SBSegment == seg(getSeg : SegmentID) |
                 point(getUpSeg : SegmentID, getDownSeg : SegmentID) |
                 esa(getESA : ESAID),
    SBType == POINTSB | ENDSB | CROSSINGSB | PLAINSB
```

16.3.5 Crossing, point and sensor

A number of types define the position or status of points, barriers, signals and sensors.

```
type
    PointPosition == UP | DOWN | MOVINGUP | MOVINGDOWN,
    BarrierPosition == UP | DOWN | MOVINGUP | MOVINGDOWN,
    SignalStatus == ON | OFF,
    SensorStatus == ACTIVE | INACTIVE
```

16.3.6 Train position

The *Location* type defines the location of a train, which is either in an ESA or on a Segment. The *SegmentPosition* type defines the precise position in the railway line. The *TrainPosition* type defines both the front- and rear position of a train.

```
type
    Location == isESA(getESA : ESAID) | isSeg(getSeg : SegmentID),
    TrainPosition :: frontPos : SegmentPosition ↔ setFrontPos
                     rearPos : SegmentPosition ↔ setRearPos,
```

```
SegmentPosition :: getLoc : Location
                   getLength : Length
```

16.3.7 Reservation

The *Reservation* type defines a reservation for a certain train in a certain direction. The *HasRes* type defines either the existence of a particular reservation or the absence of a reservation.

```
type
    HasRes == res(Reservation) | noRes,
    Reservation == mk_res(getTrain : TrainID, getDir : Direction)
```

16.3.8 Messages

A number of message types etc. are defined to be used in the control system, so that SBs and trains can communicate with each other.

```
type
    Message = TCCMsg | SBCCMsg,
    TCCMsg == segReq(Reservation),
    SBCCMsg = SBCCResMsg | SBCCDeResMsg | SBCCRespMsg,
    SBCCResMsg == lineBranchReq(Reservation),
    SBCCDeResMsg == lineBranchDeRes | lineDeRes
                  | branchDeRes,
    SBCCRespMsg = LineBranchResp | SegmentResp,
    LineBranchResp == lineBranchResp(getRes : Reservation, isPos : Bool),
    SegmentResp == segResp(isPos : Bool),

    ReturnSBCCMsg == hasMsg(SBCCMsg) | noSBCCMsg,

    ComID == isSB(SBID) | isTrain(TrainID),
    ComMsg == mk_comMsg(getSender : ComID,
                        getReceiver : ComID,
                        getMsg : Message),
    HasComMsg == comMsg(ComMsg) | noComMsg
```

16.4 Statics

This section describes the *Statics* module.

16.4.1 Type of interest

The *Configuration* type is defined as the type of interest. In this initial specification it is a sort. A constant denoting the actual configuration is also defined.

type
  Configuration

value
  conf : Configuration

16.4.2 Observers

A number of basic observers are defined and shown below grouped after which entity they concern.

End station areas

value
  getESASB : T.ESAID × Configuration $\stackrel{\sim}{\to}$ T.SBID,
  getESALength : T.ESAID × Configuration $\stackrel{\sim}{\to}$ T.Length,
  esaExistsInConf : T.ESAID × Configuration → **Bool**

getESASB returns the end SB at the ESA

getESALength returns the length of the ESA

esaExistsInConf tells if the configuration contains data describing the ESA

Switch boxes

value
  getSBSeg : T.SBID × T.Direction × Configuration $\stackrel{\sim}{\to}$ T.SBSegment,
  getSBType : T.SBID × Configuration $\stackrel{\sim}{\to}$ T.SBType,
  sbExistsInConf : T.SBID × Configuration → **Bool**

getSBSeg returns the *SBSegment* next to the SB in a certain direction. SBSegment is defined in the Types module in section 16.3.

getSBType returns the type of the SB, see section 3.7.

sbExistsInConf tells if the configuration contains data describing the SB

Segments

getSegSB : T.SegmentID × T.Direction × Configuration $\stackrel{\sim}{\to}$ T.SBID,
getSegLength : T.SegmentID × Configuration $\stackrel{\sim}{\to}$ T.Length,
getSegMaxSpeed : T.SegmentID × Configuration $\stackrel{\sim}{\to}$ T.Speed,
segExistsInConf : T.SegmentID × Configuration → **Bool**

getSegSB returns the SB next to the segment in a certain direction.

getSegLength returns the length of the segment

getSegMaxSpeed returns the max allowed speed on the segment

segExistsInConf tells if the configuration contains data describing the segment

Trains

value
getTrainLength : T.TrainID × Configuration $\stackrel{\sim}{\to}$ T.Length,
getTrainMaxSpeed : T.TrainID × Configuration $\stackrel{\sim}{\to}$ T.Speed,
getTrainMaxAcc : T.TrainID × Configuration $\stackrel{\sim}{\to}$ T.Acceleration,
getTrainMaxDeAcc : T.TrainID × Configuration $\stackrel{\sim}{\to}$ T.Acceleration,
trainExistsInConf : T.TrainID × Configuration → **Bool**

getTrainLength returns the length of the train

getMaxSpeed returns the max allowed speed of the train

getMaxAcc returns the max possible acceleration

getMaxDeAcc returns the max possible deceleration

trainExistsInConf tells if the configuration contains data describing the train

Reservation- and brake point

value
getResPoint : Configuration $\stackrel{\sim}{\to}$ T.Length,
getBrakePoint : Configuration $\stackrel{\sim}{\to}$ T.Length

getResPoint returns the reservation point which is common for all segments

getBrakePoint returns the brake point which is common for all segments

16.4.3 Derived observer

Some derived observers are defined in terms of the basic observers but they are not shown here. The entire module can be found in appendix F.1.2.

16.4.4 Wellformedness

The wellformedness predicate is defined as:

```
value
    is_wf : Configuration → Bool
    is_wf(con) ≡
        sbs_is_wf(con) ∧
        segs_is_wf(con) ∧
        esas_is_wf(con) ∧
        trains_is_wf(con) ∧
        composed_is_wf(con)

    sbs_is_wf : Configuration → Bool
    sbs_is_wf(con) ≡
        sbsHaveConf(con) ∧
        getSBSeg_diff(con) ∧
        getSBSeg_point_wf(con) ∧
        getSBSeg_injective(con) ∧
        getSBSegType_wf(con),

    segs_is_wf : Configuration → Bool
    segs_is_wf(con) ≡
        segsHaveConf(con) ∧
        getSegSB_injective(con) ∧
        brakeResPoint_wf(con),

    esas_is_wf : Configuration → Bool
    esas_is_wf(con) ≡
        esasHaveConf(con),

    trains_is_wf : Configuration → Bool
    trains_is_wf(con) ≡
        trainsHaveConf(con),

    composed_is_wf : Configuration → Bool
    composed_is_wf(con) ≡
        getESASBSeg_wf(con) ∧
        getSBSeg_getSegSB_wf(con)∧
        seg_train_length_wf(con) ∧
        esa_train_length_wf(con) ∧
        brakePoint_wf(con) ∧
        resPoint_wf(con) ∧
        collisions_detectable(con)
```

The sub predicates constituting the wellformedness predicate is explained in the following:

sbsHaveConf

Each SB must have a configuration and the reservation- and brake point must be greater than zero:

```
value
    sbsHaveConf : Configuration → Bool
    sbsHaveConf(con) ≡
    (
        (∀ seg : T.SegmentID •
            sbExistsInConf(seg,con)) ∧
         getResPoint(con) > 0.0 ∧
         getBrakePoint(con) > 0.0
    )
```

getSBSeg_diff

The segments next to a SB are different in both directions (UP and DOWN). I.e. the line is not circular

```
value
    getSBSeg_diff : Configuration → Bool
    getSBSeg_diff(con) ≡
    (
        ∀ sb : T.SBID •
            getSBSeg(sb,T.UP,con) ≠ getSBSeg(sb,T.DOWN,con)
    )
```

getSBSeg_point_wf

The two branches of a junction are different:

```
value
    getSBSeg_point_wf : Configuration → Bool
    getSBSeg_point_wf(con) ≡
    (
        ∀ sb : T.SBID,
                seg1,seg2 : T.SegmentID,
                dir : T.Direction •
                    T.point(seg1,seg2) = getSBSeg(sb,dir,con) ⇒
                        seg1 ≠ seg2
    )
```

getSBSeg_injective

Two different SBs have different SBSegments in the same direction:

```
value
    getSBSeg_injective : Configuration → Bool
    getSBSeg_injective(con) ≡
    (
        ∀ sb1, sb2 : T.SBID,
                dir : T.Direction •
                    sb1 ≠ sb2 ⇒
                        getSBSeg(sb1,dir,con) ≠ getSBSeg(sb2,dir,con)
    )
```

getSBSegType_wf

The type of a SB must conform with the result of getSBSeg:

```
value
    getSBSegType_wf : Configuration → Bool
    getSBSegType_wf(con) ≡
    (
        ∀ sb : T.SBID •
            case getSBType(sb,con) of
                T.ENDSB →      (∃! dir : T.Direction, esa : T.ESAID •
                                        esa = T.dir2End(dir) ∧
                                        getSBSeg(sb,dir,con) = T.esa(esa)),
                T.POINTSB →    (∃! dir : T.Direction,
                                               seg1,seg2 : T.SegmentID •
                                         getSBSeg(sb,dir,con) = T.point(seg1,seg2)),
                T.CROSSINGSB → (∀ dir : T.Direction •
                                        ∃ seg : T.SegmentID •
                                          getSBSeg(sb,dir,con) = T.seg(seg)),
                T.PLAINSB →    (∃! dir : T.Direction •
                                        ∃ seg : T.SegmentID •
                                          getSBSeg(sb,dir,con) = T.seg(seg))
            end
    )
```

segsHaveConf

A configuration for each Segment must exist:

```
value
    segsHaveConf : Configuration → Bool
    segsHaveConf(con) ≡
    (
        ∀ seg : T.SegmentID •
            segExistsInConf(seg,con)
    )
```

getSegSB_injective

The SB at the end of a segment is different for two different segments or they are the same in both directions (being branches):

```
value
  getSegSB_injective : Configuration → Bool
  getSegSB_injective(con) ≡
  (
    ∀ seg1, seg2 : T.SegmentID,
          dir : T.Direction •
            seg1 ≠ seg2 ⇒
            (
              getSegSB(seg1,dir,con) ≠ getSegSB(seg2,dir,con)
            )
              ∨
            (
              getSegSB(seg1,T.UP,con) = getSegSB(seg2,T.UP,con) ∧
              getSegSB(seg1,T.DOWN,con) = getSegSB(seg2,T.DOWN,con)
            )
  )
```

brakeResPoint_wf

The reservation-point must be placed before the brake-point, i.e. there is a greater distance from the end of a segment to the reservation-point than to the brake-point:

```
value
  brakeResPoint_wf : Configuration → Bool
  brakeResPoint_wf(con) ≡
    getResPoint(con) > getBrakePoint(con)
```

esasHaveConf

A configuration for each ESA must exist:

```
value
  esasHaveConf : Configuration → Bool
  esasHaveConf(con) ≡
  (
    ∀ esa : T.ESAID •
      esaExistsInConf(esa,con)
  )
```

trainsHaveConf

A configuration for each train must exist:

```
value
  trainsHaveConf : Configuration → Bool
  trainsHaveConf(con) ≡
  (
    ∀ t : T.TrainID •
      trainExistsInConf(t,con)
  )
```

getESASBSeg_wf

Given an ESA, from the coherent END SB the next SBSegment directed against the ESA must be the ESA itself:

```
value
  getESASBSeg_wf : Configuration → Bool
  getESASBSeg_wf(con) ≡
  (
    ∀ esa : T.ESAID •
      getSBSeg(getESASB(esa,con),T.end2Dir(esa),con) = T.esa(esa)
  )
```

getSBSeg_getSegSB_wf

Calculating the SB in a direction from each segment in the SBSegment calculated from a SB in the opposite direction must give the original SB:

```
value
  getSBSeg_getSegSB_wf : Configuration → Bool
  getSBSeg_getSegSB_wf(con) ≡
  (
    ∀ sb : T.SBID, dir : T.Direction, seg : T.SegmentID •
      seg ∈ T.sbSegToSet(getSBSeg(sb,dir,con)) ⇒
        getSegSB(seg,T.inverseDir(dir),con) = sb
  )
```

seg_train_length_wf

All segments must be longer than any train:

```
value
  seg_train_length_wf : Configuration → Bool
```

```
    seg_train_length_wf(con) ≡
    (
        ∀ seg : T.SegmentID, t : T.TrainID •
            getSegLength(seg,con) > getTrainLength(t,con)
    )
```

esa_train_length_wf

All ESAs must be longer than any train:

```
value
    esa_train_length_wf : Configuration → Bool
    esa_train_length_wf(con) ≡
    (
        ∀ esa : T.ESAID, t : T.TrainID •
            getESALength(esa,con) > getBrakePoint(con) + getTrainLength(t,con)
    )
```

brakePoint_wf

If a train starts to brake at the brakepoint it must be able to stop entirely before entering the next segment

```
brakePoint_wf : Configuration → Bool
brakePoint_wf(con) ≡
(
    ∀ t : T.TrainID, tAcc : T.Acceleration,
            brakeP, brakeL, s_err : T.Length,
            tSpeed : T.Speed •
                tAcc = getTrainMaxDec(t,con) ∧
                brakeP = getBrakePoint(con) ∧
                tSpeed = getTrainMaxSpeed(t,con) ∧
                s_err = tSpeed ∗ T.tick_interval ∧
                brakeL = −0.5 ∗ tSpeed ∗ tSpeed / tAcc
                    ⇒
                        brakeP > brakeL + s_err
),
```

resPoint_wf

When a train reach the break point it must be entirely on a single segment and the brake point must be smaller than the length of any segment:

```
value
    resPoint_wf : Configuration → Bool
    resPoint_wf(con) ≡
```

```
(
    ∀ t : T.TrainID, seg : T.SegmentID,
            tlen, slen, resPoint, brakePoint : T.Length •
                tlen = getTrainLength(t,con) ∧
                slen = getSegLength(seg,con) ∧
                resPoint = getResPoint(con) ∧
                brakePoint = getBrakePoint(con)
                    ⇒
                        slen > (resPoint + tlen) ∧
                        brakePoint < slen
)
```

collisions_detectable

This predicate ensures that the time update interval (*tick*) in the simulator is sufficiently small so that frontal collisions between two trains moving at top speed is detected.

For the calculations associated with this predicate please refer to section 10.3.1.

```
collisions_detectable : Configuration → Bool
collisions_detectable(con) ≡
(
    ∀ t1, t2 : T.TrainID, sp1, sp2 : T.Speed,
            s_err1, s_err2, s_col : T.Length •
                sp1 = getTrainMaxSpeed(t1,con) ∧
                sp2 = getTrainMaxSpeed(t2,con) ∧
                s_err1 = sp1 ∗ T.tick_interval ∧
                s_err2 = sp2 ∗ T.tick_interval ∧
                s_col = s_err1 + s_err2
                    ⇒
                        s_col < getTrainLength(t1,con)
)
```

16.5 Dynamics

This section describes the *Dynamics* module.

16.5.1 Type of interest

The *State* type is defined as the type of interest. In this initial specification it is a sort. A value containing the initial state is also defined.

```
type
    State
value
    initState : State
```

16.5.2 Observers and generators

A number of basic observers and generators are defined and shown below grouped after which entity they concern.

Point

value
getPointPosition : T.SBID × State × S.Configuration $\stackrel{\sim}{\to}$ T.PointPosition,
setPointPosition : T.SBID × T.PointPosition × State × S.Configuration $\stackrel{\sim}{\to}$ State,

getPointPosition returns the position of a point

setPointPosition changes the position of a point

Crossing

value
getBarrierPosition : T.SBID × State × S.Configuration $\stackrel{\sim}{\to}$ T.BarrierPosition,
getSignalStatus : T.SBID × State × S.Configuration $\stackrel{\sim}{\to}$ T.SignalStatus,
setBarrierPosition : T.SBID × T.BarrierPosition × State × S.Configuration $\stackrel{\sim}{\to}$ State,
setSignalStatus : T.SBID × T.SignalStatus × State × S.Configuration $\stackrel{\sim}{\to}$ State

getBarrierPosition returns the position of the barriers at a crossing

getSignalStatus returns the status (on/off) of the signals at a crossing

setBarrierPosition changes the position of the barriers at a crossing

setSignalStatus changes the status of the signals at a crossing

Sensor

getSensorStatus : T.SBID × State → T.SensorStatus,
setSensorStatus : T.SBID × T.SensorStatus × State × S.Configuration $\stackrel{\sim}{\to}$ State

getSensorStatus returns the status(active/inactive) of a sensor

setSensorStatus changes the status of a sensor

Train

```
value
  getTrainAcc : T.TrainID × State → T.Acceleration,
  getTrainSpeed : T.TrainID × State → T.Speed,
  getTrainPosition : T.TrainID × State → T.TrainPosition,
  getTrainDirection : T.TrainID × State → T.Direction,

  setTrainAcc : T.TrainID × T.Acceleration × State × S.Configuration ~→ State,
  setTrainSpeed : T.TrainID × T.Speed × State × S.Configuration ~→ State,
  setTrainPosition : T.TrainID × T.TrainPosition × State × S.Configuration ~→ State,
  setTrainDirection : T.TrainID × T.Direction × State ~→ State
```

getTrainAcc returns the current acceleration of a train

getTrainSpeed returns the current speed of a train

getTrainPosition returns the current position of a train

getTrainDirection returns the current direction of a train

setTrainAcc changes the current acceleration of a train

setTrainSpeed changes the current speed of a train

setTrainPosition changes the current position of a train

setTrainDirection changes the current direction of a train

16.5.3 Updating the physical system

The physical system(the *Dynamics* module) periodically receives a notification to update its state. We say that the physical system is ticked. This is done by calling the *tick* function stating how much time has passed since last update. Using this *tick value* the physical system calculates the new state of points, crossings and trains according to the algorithm and some physical laws concerning movement of the train. The specification of the tick function looks like:

```
value
  tick : T.Tick × S.Configuration × State ~→ State
  tick(tick,con,s) ≡
    let
      s = tickPoints(tick,con,s),
      s = tickCrossings(tick,con,s),
      s = tickTrains(tick,con,s)
    in
      s
    end
```

The tick function just ticks every point, crossing and train after each other. Below these three functions are described.

Ticking points

The *tickPoints* finds all the point IDs (SB IDs for point SBs) and then calls the *pointProcess* function with the set of the point IDs as parameter.

```
value
  tickPoints : T.Tick × S.Configuration × State ~→ State
  tickPoints(tick,con,s) ≡
    let
      points = { p | p : T.SBID • S.getSBType(p,con) = T.POINTSB }
    in
      pointProcess(points,tick,con,s)
    end
```

The *pointProcess* method handles one point at the time. It takes a point ID from the set of point IDs and uses this as argument to the *updatePoint* function. After *updatePoint* has been executed *pointProcess* calls itself recursively after removing the mentioned point ID from the set of point IDs. The function terminates when all point IDs have been processed, i.e. when the set of point IDs is empty.

updatePoint finds the new position of the point. If the point is moving either up or down respectively the point either remains the same or switches up or down respectively. This simulates that it takes some amount of time to switch a point. When the model is made concrete *point ticks* are introduced which specifies how many seconds it takes to switch the point.

The states of a point are modelled as a state machine in figure 9.4 in section 9.5.

```
value
  pointProcess : T.SBID-set × T.Tick × S.Configuration × State ~→ State
  pointProcess(points,tick,con,s) ≡
    if(points = {})
    then
      s
    else
      let
        p : T.SBID • p ∈ points,
        points = points \ {p},
        s = updatePoint(p,tick,con,s)
      in
        pointProcess(points,tick,con,s)
      end
    end
  pre S.sbsArePoints(points,con),
```

```
updatePoint : T.SBID × T.Tick × S.Configuration × State ~→ State
updatePoint(p,tick,con,s) ≡
    let
        pp = getPointPosition(p,s,con)
    in
        case pp of
            T.MOVINGDOWN → s ⌈⌉ setPointPosition(p,T.DOWN,s,con),
            T.MOVINGUP → s ⌈⌉ setPointPosition(p,T.UP,s,con),
            _ → s
        end
    end
pre S.getSBType(p,con) = T.POINTSB
```

Ticking crossings

The *tickCrossings* finds all the crossing IDs (SB IDs for crossing SBs) and then calls the *crossingProcess* function with the set of the crossing IDs as parameter.

```
value
    tickCrossings : T.Tick × S.Configuration × State ~→ State
    tickCrossings(tick,con,s) ≡
        let
            crossings = { c | c : T.SBID • S.getSBType(c,con) = T.CROSSINGSB }
        in
            crossingProcess(crossings,tick,con,s)
        end
```

The *crossingProcess* function handles one crossing at the time. It takes a crossing ID from the set of crossing IDs and uses this as argument to the *updateCrossing* function. After *updateCrossing* has been executed *crossingProcesses* calls itself recursively after removing the mentioned crossing ID from the set of crossing IDs. The function terminates when all crossing IDs have been processed, i.e. when the set of crossing IDs is empty.

The *updateCrossing* function handles the change in the state of a crossing that has just begun to close og to open. If the crossing is open (barriers is up and signals is off) or if the crossing is closed (barriers is down and signals is off) then *updateCrossing* does not change the state of the crossing.

The first step in opening or closing the crossing is always performed by the SB controlling the crossing when it prepares the segment the train has requested a reservation for and when the train has passed the crossing.

The first step in closing the crossing is to turn on the signals. Then after an amount of time *updateCrossing* sets the barriers to be moving down. This is modelled by having an internal choice between doing nothing and setting the barriers to be moving down. It simulates that the signals are turned on a while before the barriers start to move down. Likewise, if the barriers are moving

down, then after an amount of time *updateCrossing* sets the barriers to be down and the signals to be off. The crossing is now closed.

The first step in opening the crossing is to set the barriers to be moving up. Then after an amount of time *updateCrossing* sets the barriers to be up. Now the crossing is open.

When the model is made concrete *crossing ticks* and *signal ticks* are introduced. They specify how many seconds it takes to close or open the barriers and how many seconds the signals are turned on before the barriers start to move down.

The states of a crossing are modelled as a state machine in figure 9.3 in section 9.4.

```
value
   crossingProcess : T.SBID-set × T.Tick × S.Configuration × State ~→ State
   crossingProcess(crossings,tick,con,s) ≡
      if(crossings = {})
      then
         s
      else
         let
            c : T.SBID • c ∈ crossings,
            crossings = crossings \ {c},
            s = updateCrossing(c,tick,con,s)
         in
            crossingProcess(crossings,tick,con,s)
         end
      end
   pre S.sbsAreCrossings(crossings,con),

   updateCrossing : T.SBID × T.Tick × S.Configuration × State ~→ State
   updateCrossing(cr,tick,con,s) ≡
      let
         bp = getBarrierPosition(cr,s,con),
         ss = getSignalStatus(cr,s,con)
      in
         case bp of
            T.UP →
            (
               if(ss = T.ON)
               then
                  s ⌈⌉
                  setBarrierPosition(cr,T.MOVINGDOWN,s,con)
               else
                  s
               end
            ),
            T.MOVINGDOWN →
            (
               s ⌈⌉
               (
                  let
                     bp = setBarrierPosition(cr,T.DOWN,s,con)
                  in
                     setSignalStatus(cr,T.OFF,s,con)
```

```
                end
              )
            ),
            T.DOWN → s,
            T.MOVINGUP → s ⌈⌉ setBarrierPosition(cr,T.UP,s,con)
          end
        end
    pre S.getSBType(cr,con) = T.CROSSINGSB
```

Ticking trains

The *tickTrains* finds all the train IDs and then calls the *trainProcess* function with the set of the train IDs as parameter.

```
value
    tickTrains : T.Tick × S.Configuration × State ~→ State
    tickTrains(tick,con,s) ≡
    let
        trains = { t | t : T.TrainID}
    in
        trainProcess(trains,tick,con,s)
    end
```

The *trainProcess* method handles one train at the time. It takes a train ID from the set of train IDs and uses this as argument to the unspecified *updateTrain* function. *updateTrain* calculates the new position of the train from the current position, speed and acceleration.

```
value
    trainProcess : T.TrainID-set × T.Tick × S.Configuration × State ~→ State
    trainProcess(trains,tick,con,s) ≡
        if(trains = {})
        then
            s
        else
            let
                t : T.TrainID • t ∈ trains,
                trains = trains \ {t},
                s = updateTrain(t,tick,con,s)
            in
                trainProcess(trains,tick,con,s)
            end
        end,

    updateTrain : T.TrainID × T.Tick × S.Configuration × State ~→ State
```

16.5.4 Derived observer and generators

A number of derived observers and generators are defined in terms of the basic observers and generators but they are not shown here, besides the wellformedness functions et al. below. The entire module can be found in appendix F.1.3.

16.5.5 Wellformedness

The wellformedness predicate is defined below. An axiom is stated to specify that the actual configuration is wellformed:

```
value
    is_wf : State × S.Configuration → Bool
    is_wf(s,con) ≡
        allStatesExists(con,s),

    allStatesExists : S.Configuration × State → Bool
    allStatesExists(con,s) ≡
        allTrainStatesExist(s) ∧
        train_pos_wf(con,s) ∧
        allCrossingStatesExist(con,s)  ∧
        allPointStatesExist(con,s)   ∧
        allSensorStatesExist(s)
axiom
    [is_wf]
        is_wf(conf)
```

Each of the used functions is explained shortly in the following:

allTrainStatesExist

All trains must have a state:

```
value
    allTrainStatesExist : State → Bool
    allTrainStatesExist(s) ≡
    (
        ∀ trainID : T.TrainID •
            trainStateExists(trainID,s)
        )
    /* Tells if a train has a state in the system */
    trainStateExists : T.TrainID × State → Bool
```

train_pos_wf

Front and rear position of a train must be exactly 'train length' apart:

```
value
    train_pos_wf : S.Configuration × State ~→ Bool
    train_pos_wf(con,s) ≡
    (
        ∀ t : T.TrainID •
            train_pos_ok(t,getTrainPosition(t,s),s,con)
    ),
    train_pos_ok : T.TrainID × T.TrainPosition × State × S.Configuration ~→ Bool
    train_pos_ok(t,tp,s,con) ≡
    (
        let
            T.mk_TrainPosition(posFront,posRear) = tp
        in
            (S.distance(posFront,posRear,con) = S.getTrainLength(t,con)) ∧
            train_pos_dir_ok(getTrainDirection(t,s),tp,s,con)
        end
    )
```

allCrossingStatesExist

All crossings must have a state:

```
value
    allCrossingStatesExist : S.Configuration × State → Bool
    allCrossingStatesExist(con,s) ≡
    (
        ∀ cr : T.SBID •
            S.getSBType(cr,con) = T.CROSSINGSB ⇒
                crossingStateExists(cr,s,con)
    ),
    /* Tells if a crossing has a state in the system */
    crossingStateExists : T.SBID × State × S.Configuration → Bool
```

allPointStatesExist

All points must have a state:

```
value
    allPointStatesExist : S.Configuration × State → Bool
    allPointStatesExist(con,s) ≡
    (
        ∀ p : T.SBID •
            S.getSBType(p,con) = T.POINTSB ⇒
                pointStateExists(p,s,con)
    ),  /* Tells if a point has a state in the system */
    pointStateExists : T.SBID × State × S.Configuration → Bool
```

allSensorStatesExist

All sensors must have a state:

```
value
   allSensorStatesExist : State → Bool
   allSensorStatesExist(s) ≡
   (
      ∀ sen : T.SBID •
         sensorStateExists(sen,s)
   ),
   /* Tells if a sensor has a state in the system */
   sensorStateExists : T.SBID × State → Bool
```

16.5.6 The safe predicate

The *safe* predicate is specified as:

```
   safe : State × S.Configuration ~→ Bool
   safe(s,con) ≡
      is_wf(s,con) ∧
      noCollisions(con,s) ∧
      trainPosPossible(con,s) ∧
      pointsSafe(con,s) ∧
      crossingsSafe(con,s)
```

Notice that a safe state is also wellformed. The used functions, except *is_wf* is explained shortly in the following.

noCollisions

The position of a train may not overlap with the position of other trains. This predicate is used both for the safe predicate and as precondition for the *Dynamics.setTrainPosition()*.

```
value
   noCollisions : S.Configuration × State ~→ Bool
   noCollisions(con,s) ≡
   (
      ∀ t : T.TrainID •
         ~trainPositionOccupied(t,getTrainPosition(t,s),s,con)
   )
```

trainPosPossible

Trains cannot end up on same segment driving in opposite directions away from each other.

If two trains are on same segment driving in opposite directions then the train driving up must be lower on the line than the train driving down:

```
value
  trainPosPossible : S.Configuration × State ~→ Bool
  trainPosPossible(con,ds) ≡
  (
    ∀ t1,t2 : T.TrainID, segs : T.SegmentID-set,
          tp1,tp2 : T.TrainPosition, seg : T.SegmentID •
            commonSegs(t1,t2,ds) ≠ {} ∧
            (tp1,tp2) = (getTrainPosition(t1,ds),getTrainPosition(t1,ds)) ∧
            getTrainDirection(t1,ds) ≠ getTrainDirection(t2,ds) ∧
            getTrainDirection(t1,ds) = T.UP
              ⇒
                S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
  )
```

pointsSafe

If the train is located upon a junction, the point must be connected to the branch, on which the train is located:

```
value
  pointsSafe : S.Configuration × State ~→ Bool
  pointsSafe(con,ds) ≡
  (
    ∀ sb : T.SBID, t : T.TrainID, seg : T.SegmentID •
      trainOnJunction(t,sb,con,ds) ∧
      trainOnSegment(t,seg,con,ds) ∧
      S.segIsBranch(seg,con) ⇒
        pointConnected(sb,seg,ds,con)
  )
```

crossingsSafe

When a train is located on a crossing the barriers must be down:

```
value
  crossingsSafe : S.Configuration × State ~→ Bool
  crossingsSafe(con,s) ≡
  (
    ∀ sb : T.SBID •
      S.getSBType(sb,con) = T.CROSSINGSB ∧
```

```
        trainOnSensor(sb,con,s) ⇒
            getBarrierPosition(sb,s,con) = T.DOWN
)
```

16.5.7 Initial requirement

The initial requirement specifies some requirements to how the physical state must look like initially. An axiom is stated to make sure that the requirements are satisfied:

```
value
    init_req : State × S.Configuration ~→ Bool
    init_req(s,con) ≡
        is_wf(s,con) ∧
        allTrainsInESA(s) ∧
        allTrainsFacingLine(s) ∧
        allTrainsStopped(s) ∧
        allBarriersUp(con,s) ∧
        allPointsNotShifting(con,s)
axiom
    [ wellformedness ]
        init_req(initState,S.conf)
```

Notice that the initial state must be wellformed. The used functions, except *is_wf* is explained shortly in the following.

allTrainsInESA

All trains must be in an end station area:

```
value
    allTrainsInESA : State ~→ Bool
    allTrainsInESA(s) ≡
    (
        ∀ t : T.TrainID •
            trainInESA(t,s)
    )
```

allTrainsFacingLine

All train must face the railway line:

```
value
    allTrainsFacingLine : State ~→ Bool
    allTrainsFacingLine(s) ≡
```

```
(
    ∀ t : T.TrainID, esa : T.ESAID •
        T.isESA(esa) = T.getLoc(T.frontPos(getTrainPosition(t,s))) ∧
        (esa = T.LOW ⇒ getTrainDirection(t,s) = T.UP) ∧
        (esa = T.HIGH ⇒ getTrainDirection(t,s) = T.DOWN)
)
```

allTrainsStopped

All trains must be stopped:

```
value
    allTrainsStopped : State ~→ Bool
    allTrainsStopped(s) ≡
    (
        ∀ t : T.TrainID •
            getTrainSpeed(t,s) = 0.0 ∧
            getTrainAcc(t,s) = 0.0
    )
```

allBarriersUp

All barriers must be up:

```
value
    allBarriersUp : S.Configuration × State ~→ Bool
    allBarriersUp(con,s) ≡
    (
        ∀ sb : T.SBID •
            S.getSBType(sb,con) = T.CROSSINGSB ⇒
                getBarrierPosition(sb,s,con) = T.UP
    )
```

allPointsNotShifting

All points must be in either *Up* or *Down* position:

```
value
    allPointsNotShifting : S.Configuration × State ~→ Bool
    allPointsNotShifting(con,s) ≡
    (
        ∀ sb : T.SBID •
            S.getSBType(sb,con) = T.POINTSB ⇒
                getPointPosition(sb,s,con) ∈ { T.UP, T.DOWN }
    )
```

16.5.8 Observer/generator axioms

Observer/generator axioms are added to define the relationships between the observers and generators.

One example of a observer/generator axiom is shown below. The rest can be found in appendix F.1.3.

```
axiom
  [getPointPosition_setPointPosition]
    ∀ s : State, p1,p2 : T.SBID, pp : T.PointPosition,
        con : S.Configuration •
      getPointPosition(p1,setPointPosition(p2,pp,s,con),con) ≡
        if(p1 = p2)
        then
          pp
        else
          getPointPosition(p1,s,con)
        end
    pre S.getSBType(p1,con) = T.POINTSB ∧
        S.getSBType(p2,con) = T.POINTSB ∧
        pointStateExists(p1,s,con) ∧
        pointStateExists(p2,s,con) ∧
        ∼trainOnJunction(p2,con,s)
```

If the observer and generator gets the same point (switch box ID) as argument, the observer returns the same point position as has been input to the generator. This reflects the intended behavior, that the generator only changes the position of the point it gets as argument.

16.5.9 Generator preserving wellformedness

All generators should preserve wellformedness if all preconditions are satisfied. This property is specified through a number of axioms. Only one of these are shown here. The rest of them can be found in appendix F.1.3.

```
axiom
  [gen_wf_setPointPosition]
    ∀ p : T.SBID, pp : T.PointPosition, s : State,
        con : S.Configuration •
      is_wf(s,con) ∧
      S.getSBType(p,con) = T.POINTSB ∧
      ∼trainOnJunction(p,con,s)
        ⇒
      is_wf(setPointPosition(p,pp,s,con),con)
```

If the state is wellformed and the two preconditions (S.getSBType(p,con) = T.POINTSB, trainOnJunction(p,con,s)) are satisfied the state must be wellformed after applying the generator *setPointPosition* to the state.

16.6 Control

This section describes the *Control* module.

16.6.1 Type of interest

The *ControlState* type is defined as the type of interest. In this initial specification it is a sort. A value containing the initial control state is also defined.

type
ControlState

value
initControlState : ControlState

16.6.2 Observers and generators

A number of basic observers and generators are defined and shown below grouped after which entity they concern.

SBCC

value
getSBCCLineRes : T.SBID $\times$ ControlState $\xrightarrow{\sim}$ T.HasRes,
getSBCCBranchRes : T.SBID $\times$ ControlState $\xrightarrow{\sim}$ T.HasRes,

setSBCCLineRes : T.SBID $\times$ T.HasRes $\times$ ControlState $\xrightarrow{\sim}$ ControlState,
setSBCCBranchRes : T.SBID $\times$ T.HasRes $\times$ ControlState $\xrightarrow{\sim}$ ControlState,

getLastSensorStatus : T.SBID $\times$ ControlState $\xrightarrow{\sim}$ T.SensorStatus,
setLastSensorStatus : T.SBID $\times$ T.SensorStatus $\times$ ControlState $\xrightarrow{\sim}$ ControlState,

getNextSBCCMsg : T.SBID $\times$ ControlState $\xrightarrow{\sim}$ T.HasComMsg $\times$ ControlState,
storeSBCCMsg : T.SBID $\times$ T.ComMsg $\times$ ControlState $\xrightarrow{\sim}$ ControlState,

setSBCCPrepRes : T.SBID $\times$ T.HasRes $\times$ ControlState $\xrightarrow{\sim}$ ControlState,
getSBCCPrepRes : T.SBID $\times$ ControlState $\xrightarrow{\sim}$ T.HasRes,

sbccStateExists : T.SBID $\times$ ControlState $\rightarrow$ **Bool**

getSBCCLineRes returns the line reservation of a SBCC (SB control computer)

getSBCCBranchRes returns the branch reservation of a SBCC

setSBCCLineRes changes the line reservation of a SBCC

setSBCCBranchRes changes the branch reservation of a SBCC

getLastSensorStatus returns the last known sensor status(active/inactive)

setLastSensorStatus change the last known sensor status

getNextSBCCMsg returns the next SBCC message

storeSBCCMsg stores a SBCC message (used from outside)

getSBCCPrepRes returns the reservation for the segment the SBCC is preparing

setSBCCPrepRes changes the reservation for the segment the SBCC is preparing

sbccStateExists tells if a control state exists for a SBCC

TCC

value
hasTCCRes : T.TrainID × ControlState $\stackrel{\sim}{\to}$ **Bool**,
setTCCRes : T.TrainID × **Bool** × ControlState $\stackrel{\sim}{\to}$ ControlState,

isTCCRequesting : T.TrainID × ControlState $\stackrel{\sim}{\to}$ **Bool**,
setTCCRequesting : T.TrainID × **Bool** × ControlState $\stackrel{\sim}{\to}$ ControlState,

isTrainDecelerating : T.TrainID × ControlState $\stackrel{\sim}{\to}$ **Bool**,
setTrainDecelerating : T.TrainID × **Bool** × ControlState $\stackrel{\sim}{\to}$ ControlState,

hasPassedResPoint : T.TrainID × D.State × S.Configuration $\stackrel{\sim}{\to}$ **Bool**,
hasPassedBrakePoint : T.TrainID × D.State × S.Configuration $\stackrel{\sim}{\to}$ **Bool**,

tccStateExists : T.SBID × ControlState → **Bool**

hasTCCRes tells if a TCC has a segment reservation

setTCCRes gives a TCC a segment reservation

isTCCRequesting tells if a TCC currently is requesting for a segment reservation

setTCCRequesting sets the TCC to be requesting for a segment reservation

isTrainDecelerating tells if a train currently is decelerating

setTrainDecelerating sets the train to be decelerating

hasPassedResPoint tells if a train has passed the reservation point

hasPassedBrakePoint tells if a train has passed the brake point

tccStateExists telss if a train has a control state

16.6.3 Updating the control system

The control system is ticked just like the physical system. First the physical system is ticked, then the control system is ticked. The tick function of the control system looks like:

```
value
  tick : T.Tick × ControlState × D.State × S.Configuration ~→
                                               (ControlState × D.State)
  tick(tick,cs,ds,con) ≡
  (
    let
      tSet = {t | t : T.TrainID},
      sbSet = {sb | sb : T.SBID},
      (cs,ds) = tickTCCs(tSet,tick,cs,ds,con),
      cs = tickSBCCs(sbSet,cs,ds,con)
    in
      (cs,ds)
    end
  )
```

The tick function first calls *tickTCCs* with a set of all train IDs as parameter. Then it calls *tickSBCCs* with a set of all SB IDs as parameter. These functions are described shortly in the following.

Ticking TCCs

The *tickTCCs* funtion takes one train ID out from the set of train IDs and calls the *tccProcess* function with this train ID. After executing *tccProcess*, *tickTCCs* calls itself recursively until all train IDs have been used.

```
value
  tickTCCs : T.TrainID-set × T.Tick × ControlState × D.State × S.Configuration ~→
                                               (ControlState × D.State)
  tickTCCs(tccSet,tick,cs,ds,con) ≡
    if (tccSet = {}) then
      (cs,ds)
    else
      let
        tcc : T.TrainID • tcc ∈ tccSet,
        tccSet = tccSet \ {tcc},
        (cs,ds) = tccProcess(tcc,tick,cs,ds,con)
      in
        tickTCCs(tccSet,tick,cs,ds,con)
      end
    end
```

The *tccProcess* function is called for every train (TCC). It follows the algorihtm desribed in section 11.2 by sequentially calling the functions *checkSpeed*, *clearRes* and *handleRes*.

checkSpeed checks that the speed of the train does not exceed the max allowed for the segment and the train.

clearRes removes a reservation for a segment when the segment is left.

handlesRes makes sure that the train has a reservation for the segment it is to enter. If it has not received a reservation in time the function brakes the train.

```
value
  tccProcess : T.TrainID × T.Tick × ControlState × D.State × S.Configuration ~→
                                                 ControlState × D.State
  tccProcess(t,tick,cs,ds,con) ≡
    let
      (cs,ds) = checkSpeed(t,tick,cs,ds,con),
      cs = clearRes(t,cs,ds),
      (cs,ds) = handleRes(t,cs,ds,con)
    in
      (cs,ds)
    end
```

Only the *handleRes* function is shown here. The two other functions can be found in appendix F.1.4 along with the entire *control* module.

The *handleRes* function follows the algorithm described in 11.2.3 which is a part of the TCC algorithm. If the train has passed the reservation point the TCC request the proper SB for a reservation. If it has passed the brake point without having received a reservation the TCC brakes the train.

```
value
  handleRes : T.TrainID × ControlState × D.State × S.Configuration ~→ ControlState × D.State
  handleRes(t,cs,ds,con) ≡
    if(hasPassedResPoint(t,ds,con))
    then
      if(∼hasTCCRes(t,cs))
      then
        (cs,ds)
      else
        let
          (cs,ds) = if(hasPassedBrakePoint(t,ds,con))
                    then decelerateTrain(t,cs,ds,con)
                    else (cs,ds) end
        in
          if(∼isTCCRequesting(t,cs))
          then
            tccRequestRes(t,cs,ds,con)
          else
            (cs,ds)
```

```
            end
        end
    end
else
    (cs,ds)
end
```

Ticking SBCCs

The *tickSBCCs* funtion takes one SB ID out from the set of SB IDs and calls the *sbccProcess* function with this SB ID. After executing *sbccProcess*, *tickSBCCs* calls itself recursively until all SB IDs have been used.

```
value
    tickSBCCs : T.SBID-set × ControlState × D.State × S.Configuration ~→ ControlState
    tickSBCCs(sbSet,cs,ds,con) ≡
        if (sbSet = {}) then
            cs
        else
            let
                sbcc : T.SBID • sbcc ∈ sbSet,
                sbSet = sbSet \ {sbcc},
                cs = sbccProcess(sbcc,cs,ds,con)
            in
                tickSBCCs(sbSet,cs,ds,con)
            end
        end,
```

The *sbccProcess* is called for every SB (SBCC). It follows the algorithm described in section 11.3 by using the functions *sensorProcess*, *prepareProcess* and *sbccMsgProcess*.

sensorProcess monitors the state of a sensor. If a train has passed the sensor, it dereserves reservations in the proper end SBs/point SBs.

prepareProcess prepares a segment

sbccMsgProcess handles received messages which is either a reservation request, a reservation response, or a dereservation message.

```
value
    sbccProcess : T.SBID × ControlState × D.State × S.Configuration ~→ ControlState
    sbccProcess(sb,cs,ds,con) ≡
        let
            cs = sensorProcess(sb,cs,ds,con)
        in
            if(isPreparing(sb,cs)) then
                prepareProcess(sb,cs,ds,con)
            else
```

```
        sbccMsgProcess(sb,cs,ds,con)
    end
end
```

Only the *sensorprocess* function is shown here. Some of the functions *sensorProcess* uses have not and is not shown here. They can be found in appendix F.1.4 along with the entire *control* module.

The *sensorProcess* function follows the algorithm desribed in section 11.3.1 which is a part of the SB algorithm. It retrives and saves the state of the sensor from the *Dynamics* module. If it has passed from active to inactive a train has just passed. Then it dereserves the segment the train left, if it is a end SB or point SB. The rest of the SBs does not store reservations.

```
value
  sensorProcess : T.SBID × ControlState × D.State × S.Configuration ~→ ControlState
  sensorProcess(sb,cs,ds,con) ≡
    let
      sState = D.getSensorStatus(sb,ds),
      lastState = getLastSensorStatus(sb,cs),
      cs = setLastSensorStatus(sb,sState,cs)
    in
      if((lastState = T.ACTIVE) ∧ (sState = T.INACTIVE))
      then
        let
          ds = dePrepareSeg(sb,cs,ds,con)
        in
          if(S.isLineGuard(sb,con))
          then
            makeDeRes(sb,cs,ds,con)
          else
            cs
          end
        end
      else
        cs
      end
    end
```

16.6.4 Wellformedness

The wellformedness predicate looks like:

```
value
  is_wf : ControlState × D.State × S.Configuration → Bool
  is_wf(cs,ds,con) ≡
    D.is_wf(ds,con) ∧
    tcc_has_state(cs) ∧
    sbcc_has_state(cs)
```

The control system is wellformed when its associated physical system is wellformed and a state exists for every TCC and SBCC.

tcc_has_state

Every TCC must have a state:

```
value
    tcc_has_state : ControlState → Bool
    tcc_has_state(cs) ≡
    (
        ∀ t : T.TrainID •
            tccStateExists(t,cs)
    )
```

sbcc_has_state

Every SBCC must have a state

```
value
    sbcc_has_state : ControlState → Bool
    sbcc_has_state(cs) ≡
    (
        ∀ sb : T.SBID •
            sbccStateExists(sb,cs)
    )
```

16.6.5 The *consistent* predicate

The control system and all its components must be consistent, e.g. the information stored in the control system must reflect the physical world and unintended states must not occur. Also the physical world must abide by the rules of the control system.

```
value
    consistent : ControlState × D.State × S.Configuration → Bool
    consistent(cs,ds,con) ≡
        is_wf(cs,ds,con) ∧
        train_on_branch_dir(ds,con) ∧
        tcc_hasRes_passedResPoint(cs,ds,con) ∧
        sbcc_res_wf(cs,con) ∧
        position_branch_sbcc_res_wf(cs,ds,con) ∧
        tcc_res_branch_wf(cs,ds,con) ∧
        position_sl_sbcc_res_wf(cs,ds,con) ∧
        barrierPos_signalStatus_Consistent(ds,con)
```

Notice that a consistent control state is also wellformed. The used functions, except is_wf, is explained in the following.

train_on_branch_dir

When a train is on a branch segment it must be consistent with the driving direction of the train:

```
value
  train_on_branch_dir : D.State × S.Configuration → Bool
  train_on_branch_dir(ds,con) ≡
  (
    ∀ t : T.TrainID, seg : T.SegmentID •
      D.trainOnBranch(t,con,ds) ∧
      D.trainOnSegment(t,seg,con,ds) ∧
      S.segIsBranch(seg,con) ⇒
        S.branchDir(seg,con) = D.getTrainDirection(t,ds)
  )
```

tcc_hasRes_passedResPoint

If a train has a reservation then it has passed the reservation point on the given segment:

```
value
  tcc_hasRes_passedResPoint : ControlState × D.State × S.Configuration → Bool
  tcc_hasRes_passedResPoint(cs,ds,con) ≡
  (
    ∀ t : T.TrainID •
      hasTCCRes(t,cs) ⇒ hasPassedResPoint(t,ds,con)
  )
```

sbcc_res_wf

Only POINTSB and ENDSB may have line reservations. Only POINTSB may have branch reservations:

```
value
  sbcc_res_wf : ControlState × S.Configuration → Bool
  sbcc_res_wf(cs,con) ≡
  (
    ∀ sb : T.SBID •
      (S.getSBType(sb,con) ∈ {T.PLAINSB, T.CROSSINGSB} ⇒
        {getSBCCLineRes(sb,cs)} ∪ {getSBCCBranchRes(sb,cs)} = {T.noRes})
      ∧
      (S.getSBType(sb,con) = T.ENDSB ⇒ getSBCCBranchRes(sb,cs) = T.noRes)
  )
```

position_branch_sbcc_res_wf

When a train is on a branch segment it must have a branch reservation in the SB behind:

```
value
  position_branch_sbcc_res_wf : ControlState × D.State × S.Configuration → Bool
  position_branch_sbcc_res_wf(cs,ds,con) ≡
  (
    ∀ t : T.TrainID,
          sb : T.SBID,
          tDir : T.Direction,
          seg : T.SegmentID •
            tDir = D.getTrainDirection(t,ds) ∧
            D.trainOnSegment(t,seg,con,ds) ∧
            D.trainOnBranch(t,con,ds) ∧
            S.segIsBranch(seg,con) ∧
            sb = S.getSegSB(seg,T.inverseDir(tDir),con)
              ⇒
                getSBCCBranchRes(sb,cs) = T.res(T.mk_res(t,tDir))
  )
```

tcc_res_branch_wf

If a train is (only) on a branch and has reservation then the SB in front of it and the other guard has a reservation for that train in that direction:

```
value
  tcc_res_branch_wf : ControlState × D.State × S.Configuration → Bool
  tcc_res_branch_wf(cs,ds,con) ≡
  (
    ∀ t : T.TrainID,
          seg : T.SegmentID,
          trainDir : T.Direction,
          guard1,guard2 : T.SBID,
          res : T.Reservation •
            D.trainOnSegment(t,seg,con,ds) ∧
            D.trainOnlyOnBranch(t,con,ds) ∧
            hasTCCRes(t,cs) ∧
            trainDir = D.getTrainDirection(t,ds) ∧
            guard1 = S.getSegSB(seg,T.inverseDir(trainDir),con) ∧
            guard2 = S.getSingleLineGuard(guard1,trainDir,con) ∧
            res = T.mk_res(t,trainDir)
              ⇒
                T.res(res) = getSBCCLineRes(guard1,cs) ∧
                T.res(res) = getSBCCLineRes(guard2,cs) ∧
                T.res(res) = getSBCCBranchRes(guard2,cs)
  )
```

position_sl_sbcc_res_wf

When a train is on a single line it must have a reservation in both guards with the appropriate direction + a branch reservation if driving to a point:

```
value
    position_sl_sbcc_res_wf : ControlState × D.State × S.Configuration → Bool
    position_sl_sbcc_res_wf(cs,ds,con) ≡
    (
        ∀ t : T.TrainID,
                seg : T.SegmentID,
                sb1,sb2 : T.SBID,
                dir : T.Direction,
                res : T.Reservation •
                    dir = D.getTrainDirection(t,ds) ∧
                    D.trainOnSegment(t,seg,con,ds) ∧
                    S.segIsLineSegment(seg,con) ∧
                    sb1 = S.getSingleLineGuard(seg,T.inverseDir(dir),con) ∧
                    sb2 = S.getSingleLineGuard(seg,dir,con) ∧
                    res = T.mk_res(t,dir) ⇒
                            T.res(res) = getSBCCLineRes(sb1,cs) ∧
                            T.res(res) = getSBCCLineRes(sb2,cs) ∧
                            (S.getSBType(sb2,con) = T.POINTSB ⇒
                                                T.res(res) = getSBCCBranchRes(sb2,cs))
    )
```

barrierPos_signalStatus_Consistent

Position of barriers and status of crossing signals must conform Allowed (barrier,signal) states: (UP,OFF), (UP,ON), (MOVINGDOWN,ON), (DOWN,OFF), (MOVINGUP,OFF)

```
value
    barrierPos_signalStatus_Consistent : D.State × S.Configuration → Bool
    barrierPos_signalStatus_Consistent(s,con) ≡
    (
        ∀ sb : T.SBID •
            S.getSBType(sb,con) = T.CROSSINGSB ⇒
                case D.getBarrierPosition(sb,s,con) of
                    T.UP → D.getSignalStatus(sb,s,con) ∈ { T.ON, T.OFF },
                    T.MOVINGDOWN → D.getSignalStatus(sb,s,con) = T.ON,
                    T.DOWN → D.getSignalStatus(sb,s,con) = T.OFF,
                    T.MOVINGUP → D.getSignalStatus(sb,s,con) = T.OFF
                end
    )
```

16.6.6 Initial requirement

The initial requirement specifies some requirements to how the control state must look like initially. An axiom is stated to make sure that the requirements

are satisfied:

```
value
    initReq : ControlState × D.State × S.Configuration → Bool
    initReq(cs,ds,con) ≡
        is_wf(cs,ds,con) ∧
        no_sbcc_res(cs) ∧
        sbcc_not_preparing(cs) ∧
        no_tcc_res(cs) ∧
        tcc_not_requesting(cs) ∧
        tcc_not_decelerating(cs)
axiom
    [ initial_state ]
        initReq(initControlState,D.initState,S.conf)
```

Notice that the initial requirement includes that the control state should be wellformed. The used functions is explained in the following.

no_sbcc_res

No SBCC has a reservation:

```
value
    no_sbcc_res : ControlState → Bool
    no_sbcc_res(cs) ≡
    (
        ∀ sb : T.SBID,
                branchRes,lineRes : T.HasRes •
                    branchRes = getSBCCBranchRes(sb,cs) ∧
                    lineRes = getSBCCLineRes(sb,cs)
                        ⇒
                            {branchRes} ∪ {lineRes} = {T.noRes}
    )
```

sbcc_not_preparing

No SBCC is currently preparing a segment

```
value
    sbcc_not_preparing : ControlState → Bool
    sbcc_not_preparing(cs) ≡
    (
        ∀ sb : T.SBID •
            ∼isPreparing(sb,cs)
    )
```

no_tcc_res

No TCC has a reservation:

```
value
    no_tcc_res : ControlState → Bool
    no_tcc_res(cs) ≡
    (
        ∀ t : T.TrainID •
            ∼hasTCCRes(t,cs)
    )
```

tcc_not_requesting

No TCC is requesting segment access:

```
value
    tcc_not_requesting : ControlState → Bool
    tcc_not_requesting(cs) ≡
    (
        ∀ t : T.TrainID •
            ∼isTCCRequesting(t,cs)
    )
```

tcc_not_decelerating

No TCC is requesting segment access:

```
value
    tcc_not_decelerating : ControlState → Bool
    tcc_not_decelerating(cs) ≡
    (
              ∀ t : T.TrainID •
              ∼isTrainDecelerating(t,cs)
    )
```

16.6.7 Observer/generator axioms

Observer/generator axioms are added to define the relationships between the observers and generators.

One example of a observer/generator axiom is shown below. The rest of them can be found in appendix F.1.4.

```
axiom
  [ getSBCCLineRes_setSBCCLineRes ]
    ∀ sb1,sb2 : T.SBID, cs : ControlState, sbRes : T.Reservation,
        con : S.Configuration •
      getSBCCLineRes(sb1,setSBCCLineRes(sb2,T.res(sbRes),cs)) ≡
        if(sb1 = sb2)
        then
          T.res(sbRes)
        else
          getSBCCLineRes(sb1,cs)
        end
```

If the observer and generator gets the same SB as argument, the observer returns the same line reservation as has been input to the generator. This reflects the intended behaviour, that the generator only changes the line reservation of the SB it get as argument.

16.6.8 Generator preserving wellformedness

All generators should preserve wellformedness if all preconditions are satisfied. This property is specified through a number of axioms. Only one of these are shown here. The rest of them can be found in appendix F.1.4.

```
axiom
  [ gen_wf_setSBCCLineRes ]
    ∀ sb : T.SBID, res : T.Reservation,
      ds : D.State, con : S.Configuration,
      cs : ControlState •
    is_wf(cs,ds,con) ∧
    S.getSBType(sb,con) ∈ {T.ENDSB, T.POINTSB}
      ⇒
    is_wf(setSBCCLineRes(sb,T.res(res),cs),ds,con)
```

If the state is wellformed and the precondition (S.getSBType(sb,con) isin T.ENDSB, T.POINTSB) is satisfied the state must be wellformed after applying the generator *setSBCCLineRes* to the state.

Chapter 17

Decomposed model

This chapter describes how the model is decomposed into several schemes to obtain an object oriented structure well suited for OOP[1].

The full decomposed model can be found in appendix F.2 and the method for decomposition is described in section 15.2.

17.1 Decomposed model structure

Figure 17.1 shows the structure of the schemes of the decomposed model. The arrows indicate parameterization.

17.2 Types

The Types module is not decomposed but kept exactly as in the initial model.

17.3 Statics

The Statics module is now decomposed. Four new modules are created as objects in Statics. The type of interest (*Configuration*) in Statics is now made as a product of the four objects' type of interest. The actual configuration instance (*conf)* is now made up of the four object's actual configurations:

object
  SBs : AA_SBs1(T),

[1] Object Oriented Programming

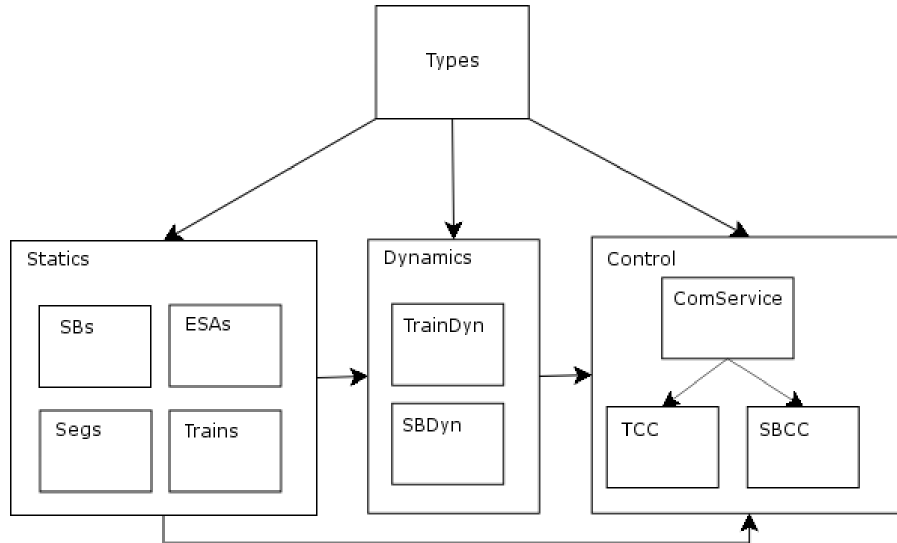

Figure 17.1: Decomposed model structure

ESAs : AA_ESAs1(T),
Segs : AA_Segs1(T),
Trains : AA_Trains1(T)

type
Configuration = SBs.SBs × Segs.Segs × ESAs.ESAs × Trains.Trains

value
conf : Configuration = (SBs.sbsConf, Segs.segsConf, ESAs.esasConf, Trains.trainsConf)

All basic and some derived observers are copied to the appropriate modules, so that each module deals with its own area. Derived observers that use observers from more than one of the new modules are kept in Statics. The other observers in Statics are changed so that they just call the equivalent observers in the appropriate modules.

The wellformedness function (is_wf) is changed so that is uses the wellformedness functions in the four new modules along with the composed wellformedness predicate, i.e. the wellformedness that deals with more than one of the new modules:

value
is_wf : Configuration → **Bool**
is_wf((sbs,segs,esas,ts)) ≡
SBs.is_wf(sbs) ∧
Segs.is_wf(segs) ∧
ESAs.is_wf(esas) ∧

```
Trains.is_wf(ts) ∧
composed_is_wf((sbs,segs,esas,ts))
```

17.3.1 SBs

The *SBs* module only deals with switch boxes. The entire module can be found in appendix F.2.2.

```
type
  SBs

value
  sbsConf : SBs,
```

17.3.2 Segs

The *Segs* module only deals with segment. The entire module can be found in appendix F.2.2.

```
type
  Segs

value
  segsConf : Segs
```

17.3.3 ESAs

The *ESAs* module only deals with end station areas. The entire module can be found in appendix F.2.2.

```
type
  ESAs

value
  esasConf : ESAs
```

17.3.4 Trains

The trains module only deals with trains. The entire module can be found in appendix F.2.2.

```
type
    Trains

value
    trainsConf : Trains
```

17.4 Dynamics

The Dynamics module is now decomposed. Two new modules are created as objects in Dynamics. The type of interest (*State*) in Dynamics becomes a product of the two objects' type of interest. The initial state (*initState)* is then made up of the two object's initial states:

```
object
    TD : AA_TrainDyn1(T,S),
    SD : AA_SBDyn1(T,S)

type
    State = TD.TrainStates × SD.SBStates

value
    initState : State = (TD.initTrainStates, SD.initSBStates)
```

All basic and some derived observers and generator are copied to the appropriate modules, so that each module deals with its own area. Derived observers and generators that use observers and generators from more than one of the new modules are kept in Dynamics. The other observers and generators in Dynamics are changed so that they just call the equivalent observers and generators in the appropriate modules.

The wellformedness predicate is changed so that is uses the wellformedness functions (is_wf) in the two new modules:

```
value
    is_wf : State × S.Configuration → Bool
    is_wf((ts,ss),con) ≡
        TD.is_wf(ts,con) ∧
        SD.is_wf(ss,con)
```

The *init_req* function is changed likewise to use the *init_req* functions in the two new modules along with the *is_wf* function in Dynamics.

```
value
    init_req : State × S.Configuration → Bool
    init_req((ts,ss),con) ≡
        is_wf((ts,ss),con) ∧
        TD.init_req(ts) ∧
        SD.init_req(ss,con)
```

17.4.1 TrainDyn

The *TrainDyn* module only deals with the state of trains. The entire module can be found in appendix F.2.3.

type
 TrainStates

value
 initTrainStates : TrainStates

17.4.2 SBDyn

The *SBDyn* module only deals with the state of switch boxes. The entire module can be found in appendix F.2.3.

type
 SBStates

value
 initSBStates : SBStates

17.5 Control

Control is also decomposed but a bit different than Statics and Dynamics as described in section 15.2.4.

object
 SBCC : AA_SBCC1(T,S,D,COM),
 TCC : AA_TCC1(T,S,D,COM)

type
 ControlState = SBCCStates × TCCStates,

 SBCCStates = T.SBID $\overrightarrow{m}$ SBCC.SBCCState,
 TCCStates = T.SBID $\overrightarrow{m}$ TCC.TCCState

value
 initControlState : ControlState

The wellformedness predicate (*is_w)*) has not been changed. It still require the dynamic state to be wellformed and a control state to exists for every TCC and SBCC.

17.5.1 TCC

The *TCC* module deals with the one TCC. The entire module can be found in appendix F.2.4.

```
type
   TCCState

value
   initTCCState : TCCState
```

17.5.2 SBCC

The *SBCC* module deals with the one SBCC. The entire module can be found in appendix F.2.4.

```
type
   SBCCState

value
   initSBCCState : SBCCState
```

17.5.3 ComService

As can be seen in the decomposed model diagram in section 17.1 a new scheme *ComService* is added. This is needed to enable the control entities *TCC* and *SBCC*, which are now moved to independent schemes, to communicate with each other.

The *ComService* scheme basically consists of a channel and functions to access the channel. A `comService` process in the *Control* scheme reads from the channel and relays the messages to the appropriate control entity (see also the message types in section 16.3.8).

The *ComService* scheme is showed below:

```
scheme AA_ComService1(T : AA_Types1) =
   class
      channel
         comChannel : T.ComMsg
      value
         sendMsg : T.ComMsg → out comChannel Unit
         sendMsg(comMsg) ≡ comChannel!comMsg,

         getMsg : Unit → in comChannel T.ComMsg
         getMsg() ≡ comChannel?
   end
```

17.6 Implementation relation

This section describes the implementation relations between the modules in the initial model and the modules in the decomposed model.

17.6.1 Types

The implementation relation between *AA_Types0* and *AA_Types1* is simply defined as:

```
theory AA0_AA1_impl_types:
axiom
    ⊢ AA_Types1 ⪯ AA_Types0
end
```

The only change from AA_Types0 to AA_Types1 is the name of the module. Therefore it is obvious that AA_Types1 directly implements AA_Types0.

17.6.2 Statics

To define the implementation relation between *AA_Statics0* and *AA_Statics1* a new module called *AA_Statics1__* is first defined. It extends AA_Statics1 with all the functions that AA_Statics1 do not have compared with AA_Statics0:

```
scheme AA_Statics1__(T : AA_Types0) =
    extend AA_Statics1(T) with
        class
            value
                trains_is_wf : Configuration → Bool,
                trainsHaveConf : Configuration → Bool,
                trainExistsInConf : T.TrainID × Configuration → Bool,
                esas_is_wf : Configuration → Bool,
                esasHaveConf : Configuration → Bool,
                esaExistsInConf : T.ESAID × Configuration → Bool,
                segs_is_wf : Configuration → Bool,
                segsHaveConf : Configuration → Bool,
                getSegSB_injective : Configuration → Bool,
                brakeResPoint_wf : Configuration → Bool,
                segExistsInConf : T.SegmentID × Configuration → Bool,
                sbs_is_wf : Configuration → Bool,
                sbsHaveConf : Configuration → Bool,
                getSBSeg_diff : Configuration → Bool,
                getSBSeg_point_wf : Configuration → Bool,
                getSBSeg_injective : Configuration → Bool,
                getSBSegType_wf : Configuration → Bool,
                sbExistsInConf : T.SBID × Configuration → Bool
        end
```

Now we can define the implementation relation by using AA_Statics1__ instead of AA_Statics1:

```
theory AA0_AA1_impl_statics:
  axiom
    in class
        object
          T : AA_Types1
        end
    ⊢ ⊢ AA_Statics1__(T) ⪯ AA_Statics0(T)
end
```

17.6.3 Dynamics

To define the implementation relation between *AA_Dynamics0* and *AA_Dynamics1* first a new module called *AA_Dynamics1__* is defined. It extends AA_Dynamics1 with all the functions that AA_Dynamics1 do not have compared with AA_Dynamics0:

```
scheme AA_Dynamics1__(T : AA_Types1, S : AA_Statics1(T)) =
  extend AA_Dynamics1(T,S) with
    class
      value
        allStatesExists : S.Configuration × State → Bool,
        allTrainStatesExist : State → Bool,
        train_pos_wf : S.Configuration × State ~→ Bool,
        train_pos_ok : T.TrainID × T.TrainPosition × State × S.Configuration ~→ Bool,
        train_pos_dir_ok : T.Direction × T.TrainPosition × State × S.Configuration → Bool,

        allCrossingStatesExist : S.Configuration × State → Bool,
        allPointStatesExist : S.Configuration × State → Bool,
        allSensorStatesExist : State → Bool,

        allTrainsInESA : State ~→ Bool,
        allTrainsFacingLine : State ~→ Bool,
        allTrainsStopped : State ~→ Bool,
        allBarriersUp : S.Configuration × State ~→ Bool,
        allPointsNotShifting : S.Configuration × State ~→ Bool,

        trainStateExists : T.TrainID × State → Bool,
        sensorStateExists : T.SBID × State → Bool,
        crossingStateExists : T.SBID × State × S.Configuration → Bool,
        pointStateExists : T.SBID × State × S.Configuration → Bool

    end
```

Now we can define the implementation relation by using AA_Dynamics1__ instead of AA_Dynamics1:

```
theory AA0_AA1_impl_dynamics:
```

```
axiom
  in
    class
      object
        T : AA_Types1,
        S : AA_Statics1__(T)

    end
    ⊢ ⊢ AA_Dynamics1__(T,S) ⪯ AA_Dynamics0(T,S)
end
```

17.6.4 Control

To define the implementation relation between *AA_Control0* and *AA_Control1* first a new module called *AA_Control__* is defined. It extends AA_Control1 with all the functions that AA_Control1 do not have compared with AA_Control0:

AA_Control1

```
scheme AA_Control1__(T : AA_Types1, S : AA_Statics1(T), D : AA_Dynamics1(T,S)) =
extend AA_Control1(T,S,D) with
class
value
  getSBCCLineRes : T.SBID × ControlState ~→ T.HasRes,
  getSBCCBranchRes : T.SBID × ControlState ~→ T.HasRes,
  setSBCCLineRes : T.SBID × T.HasRes × ControlState ~→ ControlState,
  setSBCCBranchRes : T.SBID × T.HasRes × ControlState ~→ ControlState,
  getLastSensorStatus : T.SBID × ControlState ~→ T.SensorStatus,
  setLastSensorStatus : T.SBID × T.SensorStatus × ControlState ~→
                                                        ControlState,
  getNextSBCCMsg : T.SBID × ControlState ~→ T.HasComMsg × ControlState,
  storeSBCCMsg : T.SBID × T.ComMsg × ControlState ~→ ControlState,
  getSBCCPrepRes : T.SBID × ControlState ~→ T.HasRes,
  setSBCCPrepRes : T.SBID × T.HasRes × ControlState ~→ ControlState,
  hasTCCRes : T.TrainID × ControlState ~→ Bool,
  setTCCRes : T.TrainID × Bool × ControlState ~→ ControlState,
  isTCCRequesting : T.TrainID × ControlState ~→ Bool,
  setTCCRequesting : T.TrainID × Bool × ControlState ~→ ControlState,
  isTrainDecelerating : T.TrainID × ControlState ~→ Bool,
  setTrainDecelerating : T.TrainID × Bool × ControlState ~→ ControlState,
  hasPassedResPoint : T.TrainID × D.State × S.Configuration ~→ Bool,
  hasPassedBrakePoint : T.TrainID × D.State × S.Configuration ~→ Bool,
  removeSBCCLineRes : T.SBID × ControlState ~→ ControlState,
  removeSBCCBranchRes : T.SBID × ControlState ~→ ControlState,
  removeSBCCPrepRes : T.SBID × ControlState ~→ ControlState,
  isPreparing : T.SBID × ControlState ~→ Bool,
  comService : T.ComMsg × ControlState ~→ ControlState,
  tccMsgReceiver : T.TrainID × T.ComMsg × ControlState ~→ ControlState,
  sbccMsgReceiver : T.SBID × T.ComMsg × ControlState ~→ ControlState,
```

tccProcess : T.TrainID × T.Tick × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState × D.State,
checkSpeed : T.TrainID × T.Tick × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState × D.State,
checkDeceleration : T.TrainID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState × D.State,
decelerateTrain : T.TrainID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState × D.State,
accelerateTrain : T.TrainID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState × D.State,
clearRes : T.TrainID × ControlState × D.State $\stackrel{\sim}{\rightarrow}$ ControlState,
handleRes : T.TrainID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState × D.State,
tccRequestRes : T.TrainID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState × D.State,
sendTCCReq : T.TrainID × T.SBID × T.Direction × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState,
sbccProcess : T.SBID × T.Tick × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState,
prepareProcess : T.SBID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState,
sensorProcess : T.SBID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState,
dePrepareSeg : T.SBID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState × D.State,
prepareSeg : T.SBID × T.Reservation × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState × D.State,
makeDeRes : T.SBID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState,
sendLBDeResMsg : T.SBID × T.SBID × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState,
sendLDeResMsg : T.SBID × T.SBID × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState,
sendBDeResMsg : T.SBID × T.SBID × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState,
sendLBResMsg : T.SBID × T.SBID × T.Reservation × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState,
sendSBCCMsg : T.SBID × T.ComID × T.SBCCMsg × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState,
sbccMsgProcess : T.SBID × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ ControlState,
handleSBCCMsg : T.SBID × T.Message × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState × T.ReturnSBCCMsg,
handleTCCMsg : T.SBID × T.Message × ControlState × D.State × S.Configuration $\stackrel{\sim}{\rightarrow}$ T.ReturnSBCCMsg × T.ReturnSBCCMsg × ControlState × D.State,
handleLBReq : T.SBID × T.Message × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState × T.ReturnSBCCMsg,
lineBranchFree : T.SBID × ControlState $\stackrel{\sim}{\rightarrow}$ **Bool**,
lineFree : T.SBID × ControlState $\stackrel{\sim}{\rightarrow}$ **Bool**,
handleLBResp : T.SBID × T.Message × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState × T.ReturnSBCCMsg,
handleDeResMsg : T.SBID × T.Message × ControlState $\stackrel{\sim}{\rightarrow}$ ControlState × T.ReturnSBCCMsg,
no_sbcc_res : ControlState → **Bool**,
sbcc_not_preparing : ControlState → **Bool**,
no_tcc_res : ControlState → **Bool**,
tcc_not_requesting : ControlState → **Bool**,
tcc_not_decelerating : ControlState → **Bool**,

```
    barrierPos_signalStatus_Consistent : D.State × S.Configuration → Bool
end
```

Now we can define the implementation relation by using AA_Control11__ instead of AA_Control11:

```
theory AA0_AA1_impl_control:
axiom
    in
        class
            object
                T : AA_Types1,
                S : AA_Statics1__(T),
                D : AA_Dynamics1__(T,S)

        end
        ⊢ ⊢ AA_Control1__(T,S,D) ⪯ AA_Control0(T,S,D)
end
```

Chapter 18

Concrete model

When the model is made concrete all data types that preciously were sorts are made concrete. All unspecified functions are made explicit, but no structural changes has been made.

All concrete modules in their entire can be found in appendix F.3. In the following all modules is examined and the types that has become concrete is shown:

18.1 Types

The primary change in the Types module is that *ID* is made concrete:

type
  ID = **Text**

A few types have been added as well but they are mentioned here. The entire module can be found in appendix F.3.1.

18.2 Statics

The types in Statics has not been changed since the decomposition step made Statics concrete.

18.2.1 SBs

The type of interest in SBs (*SBs*) looks like:

```
type
   SBs = T.SBID →m SBData,

   SBData == mk_sb(getUpSeg : T.SBSegment,
                   getDownSeg : T.SBSegment,
                   getType : T.SBType,
                   getPointTicks : T.HasTicks,
                   getBarrierTicks : T.HasTicks,
                   getSignalTicks : T.HasTicks)
```

The *SBData* type holds configuration data for one SB.

The *SBs* type is a mapping from *SBID* to *SBData*. In this way *SBs* can contain the configuration data for switch boxes. The wellformedness predicate says i.a. that *SBs* should contain configuration data for all SB.

18.2.2 Segs

The type of interest in Segs (*Segs*) looks like:

```
type
   Segs = SegsData × SegPoints,

   SegsData = T.SegmentID →m SegData,
   SegPoints :: getRP : T.Length
                getBP : T.Length,

   SegData == mk_seg(getUpSB : T.SBID,
                     getDownSB : T.SBID,
                     getLength : T.Length,
                     getMaxSpeed : T.Speed)
```

The *SegData* type holds configuration data for one segment.

The *SegsData* type is a mapping from *SegmentID* to *SegData*. In this way *SegsData* can contain the configuration data for segments. The wellformedness predicate says i.a. that *SegsData* should contain configuration data for all segment.

SegPoints contains the *reservation point* and *brake point*.

18.2.3 ESAs

The type of interest in ESAs (*ESAs*) looks like:

```
type
   ESAs == mk_esa(getLowSB : T.SBID,
                  getHighSB : T.SBID,
                  getLowLength : T.Length,
                  getHighLength : T.Length)
```

The *ESAs* type holds the two *end SBs* (Low and High) and the length of the two ESAs.

18.2.4 Trains

The type of interest in Trains (trains) looks like:

```
type
   Trains = T.TrainID →m TData,

   TData == mk_train(getLength : T.Length,
                     getMaxSpeed : T.Speed,
                     getMaxAcc : T.Acceleration,
                     getMaxDec : T.Acceleration)
```

The *TrainData* type holds configuration data for one train.

The trains type is a mapping from *TrainID* to *TrainData*. In this way trains can contain the configuration data for trains. The wellformedness predicate says i.a. that trains should contain configuration data for all trains.

18.3 Dynamics

The types in Dynamics has not been changed since the decomposition step made Dynamics concrete.

18.3.1 TrainDyn

The type of interest in TrainDyn (*TrainStates*) looks like:

```
type
   TrainStates = T.TrainID →m TrainState,

   TrainState == mk_tState(getTAcc : T.Acceleration ↔ setTAcc,
                           getTSpeed : T.Speed ↔ setTSpeed,
                           getTPos : T.TrainPosition ↔ setTPos,
                           getTDir : T.Direction ↔ setTDir)
```

The *TrainState* type holds the physical state data for one train.

The *TrainStates* type is a mapping from *TrainID* to *TrainStates*. In this way *TrainStates* can contain the states for trains. The wellformedness predicate says i.a. that *TrainStates* should contain states for all trains.

18.3.2 SBDyn

The type of interest in SBDyn (*SBStates*) looks like:

```
type
   SBStates = T.SBID ⇸ SBState,

   SBState == mk_sbState(getPP : T.HasPointPosition ↔ setPP,
                         getPTicks : T.HasTicks ↔ setPTicks,
                         getBP : T.HasBarrierPosition ↔ setBP,
                         getSignal : T.HasSignalStatus ↔ setSignal,
                         getBTicks : T.HasTicks ↔ setBTicks,
                         getSTicks : T.HasTicks ↔ setSTicks,
                         getSensor : T.SensorStatus ↔ setSensor)
```

The *SBState* type holds the physical state data for one SB.

The *SBStates* type is a mapping from *SBID* to *SBStates*. In this way *SBStates* can contain the states for SBs. The wellformedness predicate says i.a. that *SBStates* should contain states for all SBs.

18.4 Control

The types in Control has not been changed since the decomposition step made Control concrete.

18.4.1 TCC

The type of interest in TCC (*TCCState*) looks like:

```
type
   TCCState :: hasTCCRes : Bool ↔ setTCCRes
               isTCCRequesting : Bool ↔ setTCCRequesting
               isTrainDecelerating : Bool ↔ setTrainDecelerating
```

TCCState holds the control state for one TCC

18.4.2 SBCC

The type of interest in SBCC (*SBCCState*) looks like:

```
type
   SBCCState :: getLineRes : T.HasRes ↔ setLineRes
                getBranchRes : T.HasRes ↔ setBranchRes
                getLastSensorStatus : T.SensorStatus ↔ setLastSensorStatus
                getMsgs : T.ComMsg* ↔ setMsgs
                getPrepRes : T.HasRes ↔ setPrepRes
```

SBCCState holds the control state for one SBCC

18.5 Implementation relation

This section describes the implementation relations between the modules in the decomposed model and the modules in the concrete model.

18.5.1 Types

The implementation relation between *AA_Types1* and *CA_Types0* is simply defined as:

```
theory AA1_CA0_impl_types:
axiom
  ⊢ CA_Types0 ⪯ AA_Types1
end
```

18.5.2 Statics

The implementation relation between *AA_Statics1* and *CA_Statics0* is simply defined as:

```
theory AA1_CA0_impl_statics:
  axiom
     in class
           object
              T : CA_Types0
        end
    ⊢  ⊢ CA_Statics0(T) ⪯ AA_Statics1(T)
end
```

In the step of making the statics module concrete the signatures of the functions have not changed and no functions are deleted. Therefore the concrete statics module CA_Statics0 directly implements the decomposed statics module AA_Statics1.

18.5.3 Dynamics

The implementation relation between *AA_Dynamics1* and *CA_Dynamics0* is simply defined as:

```
theory AA1_CA0_impl_dynamics:
```

```
axiom
  in
    class
      object
        T : CA_Types0,
        S : CA_Statics0(T)
    end
    ⊢ ⊢ CA_Dynamics0(T,S) ⪯ AA_Dynamics1(T,S)
end
```

In the step of making the dynamics module concrete the signatures of the functions have not changed and no functions are deleted. Therefore the concrete dynamics module CA_Dynamics0 directly implements the decomposed dynamics module AA_Dynamics1.

18.5.4 Control

The implementation relation between *AA_Control1* and *CA_Control0* is simply defined as:

```
theory AA1_CA0_impl_control:
axiom
  in
    class
      object
        T : CA_Types0,
        S : CA_Statics0(T),
        D : CA_Dynamics0(T,S)
    end
  ⊢ ⊢

  CA_Control0(T,S,D) ⪯ AA_Control1(T,S,D)
end
```

In the step of making the control module concrete the signatures of the functions have not changed and no functions are deleted. Therefore the concrete control module CA_Control0 directly implements the decomposed control module AA_Control1.

Chapter 19

Imperative model

When the model is made imperative variables are introduced. The type of interest is removed as parameter to the functions because the functions now read and write directly at the variables instead. The signatures are changed to reflect this.

The following sections describes the variables that are introduced.

19.1 Types

The Types module does not have a type of interest. Therefore it does not need any variable(s).

19.2 Statics

Statics will not have a variable to hold the configuration since the module is decomposed. Instead the sub modules of Statics contain the variables that together form the entire configuration.

No functions write to one of the variables in the sub modules of Statics since the variables hold a configuration and not a state. Besides no generator functions exists. Therefore the configuration could be maintained as a constant rather than a variable, but that would not be suitable for implementing in JAVA since the simulator should be generic. Without a variable only one configuration could be used and it should be stated directly in the JAVA code rather than being loaded at start up.

19.2.1 SBs

The SBs module will have one variable containing the type of interest. It is initialized to the actual configuration value *sbsConf*

variable
  v_SBs : SBs := sbsConf

value
  sbsConf : SBs

19.2.2 Segs

The Segs module will have two variables. One containing the type of interest and one containing the brake- and reservation point. The variables are initialized to the actual configuration values.

variable
  v_segs : SegsData := segsDataConf,
  v_points : SegPoints := segPointsConf

value
  segsConf : Segs = (segsDataConf,segPointsConf),
  segsDataConf : SegsData,
  segPointsConf : SegPoints

19.2.3 ESAs

The ESAs module will have one variable containing the type of interest. It is initialized to the actual configuration value *esasConf*

variable
  v_ESAs : ESAs := esasConf

value
  esasConf : ESAs

19.2.4 Trains

The Trains module will have one variable containing the type of interest. It is initialized to the actual configuration value *trainsConf*

variable
v_trains : Trains := trainsConf

value
trainsConf : Trains

19.3 Dynamics

Like in the Statics module, the Dynamics module will not have variables since it is decomposed.

19.3.1 TrainDyn

The TrainDyn module will have one variable containing the type of interest. It is initialized to the initial state *initTrainStates*

variable
v_TrainStates : TrainStates := initTrainStates

value
initTrainStates : TrainStates

19.3.2 SBDyn

The SBDyn module will have one variable containing the type of interest. It is initialized to the initial state *initSBStates*

variable
v_SBStates : SBStates := initSBStates

value
initSBStates : SBStates

19.4 Control

Like in the Statics and Dynamics modules, the Control module will not have variables since it is decomposed.

A little change in the structure of the Control module has been made. Before Control contained one SBCC module and one TCC module and then two arrays containing the states of these. Now two object arrays are made containing all the TCCs and SBCCs that now contain their own control state as variables.

19.4.1 TCC

The TCC module will have three variables that together contain the three parts of the type of interest. The variables are initialized to the their parts of the initial TCC control state *initTCCState*.

```
type
   TCCState :: hasTCCRes : Bool
                         isTCCRequesting : Bool
                         isTrainDecelerating : Bool

variable
   v_tccRes : Bool := hasTCCRes(initTCCState),
   v_isReq : Bool := isTCCRequesting(initTCCState),
   v_isDec : Bool := isTrainDecelerating(initTCCState)

value
   initTCCState : TCCState
```

19.4.2 SBCC

The SBCC module will have five variables that together contain the five parts of the type of interest. The variables are initialized to the their parts of the initial SBCC control state *initSBCCState*.

```
type
   SBCCState :: getLineRes : T.HasRes
                           getBranchRes : T.HasRes
                           getSensorStatus : T.SensorStatus
                           getMsgs : T.ComMsg*
                           getPrepRes : T.HasRes

variable
   v_lineRes : T.HasRes := getLineRes(initSBCCState),
   v_branchRes : T.HasRes := getBranchRes(initSBCCState),
   v_sensorStatus : T.SensorStatus := getSensorStatus(initSBCCState),
   v_msgs : T.ComMsg* := getMsgs(initSBCCState),
   v_prepRes : T.HasRes := getPrepRes(initSBCCState)

value
   initSBCCState : SBCCState
```

19.5 Implementation relation

No implementation relations is defined between the modules in the concrete model and the modules in the imperative model, since the step of making the model imperative is not a refinement step but a transformation step.

Chapter 20

Implementing the simulator

This chapter concerns all aspects of implementing the actual simulator in JAVA. The method of translating RSL to JAVA is presented. An overview of the structure of the final JAVA code is shown an explained. A few major differences between the model and the JAVA simulator is discussed.

20.1 Translating the model to JAVA

Although the model has been refined to an imperative model the translation to JAVA is not trivial. There is no way to verify that the JAVA program is an implementation of the RSL model. The only way to make this translation is to convert the model step by step and find appropriate JAVA structures which correspond to the RSL structures.

Notation: When expressing something about JAVA code in this section it is shown in typewriter format like `int` and `Vector`. When addressing RSL structures basic types are written in bold face as **Int** and **Bool**. Derived types are emphasized like *TrainState*.

20.1.1 Schemes and objects

The semantics of schemes and objects in RSL has not been investigated in this project. It is assumed that schemes and objects are the same as class expression and can be directly translated to JAVA classes.

In the specification of the model, top level values and global objects have not been used with the goal of easing the implementation to JAVA.

20.1.2 Basic types

The basic types in the RSL model such as **Nat**, **Int**, **Bool** and **Text** have been directly translated to the JAVA built-in types `int`, `boolean` and `String`. If the types `int` and `boolean` are used in a context where an object is necessary (ex. stored in a `Vector` or `Hashtable`) the corresponding JAVA wrapper classes `Integer` and `Boolean` are used.

One disadvantage of JAVA is the typing model of the basic types. There is no way to create abbreviation types as in RSL. Therefore if two types are of the same max type in RSL then they cannot be distinguished in JAVA if directly translated:

RSL:

```
type
   ID1 = Text,
   ID2 = Text

value
   id1 : ID1,
   id2 : ID2
```

JAVA:

```
String id1;
String id2;
```

As can be seen in the examples above the RSL values *id1* and *id2* have different types whereas in JAVA they do not. Therefore the JAVA variables would be applicable to the exact same set of functions and the typing system would not catch any unintended mix-up of the two variables.

This could be solved by extending the `String` class to both a `ID1` and `ID2` class.

It has been chosen not to use the class approach but stick to the basic types. Developing the JAVA program step by step from the RSL model should avoid any function being called with wrong arguments. But as long as the basic types are used for more than one data type consistency can not be ensured.

20.1.3 Cartesian product types

Translating the cartesian product (CP) type to JAVA is fairly straight forward. The CP type is used in RSL for grouping data in a common container. This can be conveniently translated into a JAVA class with get and set methods for data access. This is illustrated in the example below:

RSL:

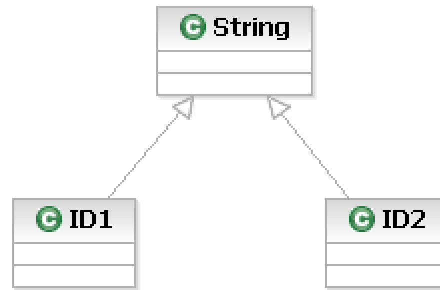

Figure 20.1: `ID1` and `ID2` as two disjoint types extending `String`

type
  Cartesian = **Int** × **Bool**

JAVA:

```
class Cartesian
{
  int var1;
  boolean var2;

  public int getVar1() {return var1;}
  public void setVar1(int var) {this.var1 = var;}
  ...
}
```

20.1.4 Map types

This section concerns converting map structures into JAVA code.

A map in RSL might look like the following:

RSL:

type
  A, B,
  ABMap = A $\overrightarrow{m}$ B

First B should be made a class with internal variables reflecting whatever information the CP contains (see section 20.1.3). The actual map structure could be translated to a `Hashtable` containing the train state objects as value having the A as keys.

One disadvantage using this interpretation is that the JAVA `Hashtable` does not have any kind of type checking (before the JAVA version 1.5). This means that though the idea of the map is only to map A to B, the key and value in the JAVA `Hashtable` can be of any type.

In this project this is solved by making a class that contains a JAVA `Hashtable` in a variable. The important methods in a JAVA `Hashtable` are then defined in this class in terms of the similar functions in the JAVA `Hashtable`. The signatures of the functions are changed so the key and value parameters are only allowed to be the types A and B respectively. This solution is illustrated in figure 20.2.

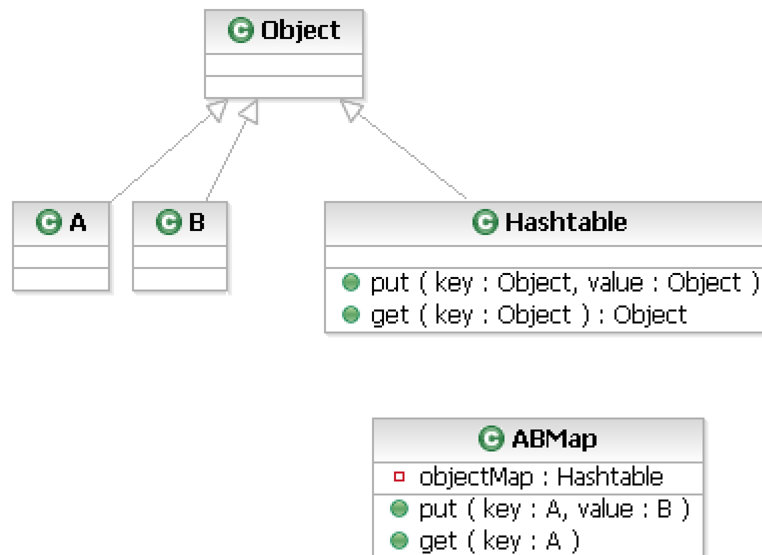

Figure 20.2: Map (`Hashtable`) with type restriction to A and B classes

20.1.5 Variant types

A variant type defines a main type which can exist in the form of several sub types (or can be constructed by different constructors). In this project we have separated the variant in three different cases:

1. All sub types are constructors with at least one argument. No constants.
2. As above but with one constant expressing the empty type or non-existense of the main type.

3. All sub types are constants with no arguments.

1. All constructors:
An example of this case is shown beneath:

RSL:

type
Variant == $con_1(dest_{11} : T_{11}, dest_{12} : T_{12})$
| $con_2(dest_{21} : T_{21}, dest_{22} : T_{22})$

Here the main type *Variant* is translated to the abstract JAVA class `Variant`. Two classes extend this class, namely `Con1`, `Con2`. The sub types or constructors are handled exactly like the cartesian product type. I.e. classes are created containing variables with same types as the variant constructor arguments. a JAVA example is shown below:

JAVA:

```
abstract class Variant
{
}

class Const1 extends Variant
{
  T11 dest11
  T12 dest12;
}

class Const2 extends Variant
{
  T11 dest21
  T12 dest22;
}
```

2. Constructors and the empty type:
An RSL example is illustrated below:

RSL:

type
Variant2 == $con_2 1(dest_2 11 : T_1)$
| empty

Here the constant *empty* is the empty value for the *Variant* type. In JAVA this can be implemented exactly as *case 1* above with the *empty* constant converted to a JAVA class with no internal values.

In this project it has been chosen to implement the empty values as `null`. Again the disadvantage of this approach is that no type checking is made on the value `null` and could therefore be any of the constants using `null` as value.

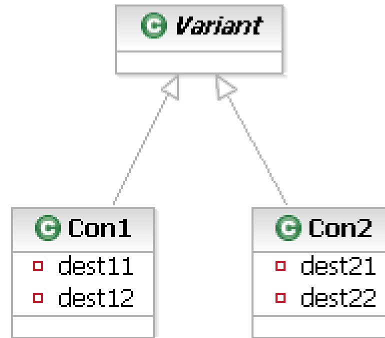

Figure 20.3: A variant with two constructors

The latter solution has been chosen, because it would be the intuitive solution in JAVA and that it is not considered that great an inconsistence. On the other hand, should a generic solution be required the prior solution with an empty class should be considered.

3. All constants:
An RSL example is illustrated below:

RSL:

type
  Variant == CONST_1 | CONST_2

Here the type *Variant* is a pure collection of constants. Again this could be implemented as classes in the generic case, to ensure type checking, in general algorithms converting variants.

But this complicates the intuitive use of these constants which would be in a **case** structure which corresponds to a `switch` in JAVA. But if these constants were implemented as classes a `switch` structure could not handle these. This should then be handled with `instanceof` like example below:

JAVA:

```
public void func(Variant3 variant)
{
  if (variant instanceof CONST1)
  {...}
  else // if (variant instanceof CONST2)
  {..}
```

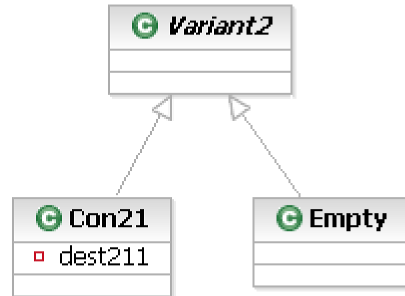

Figure 20.4: Variant class using the class `Empty` as the empty value

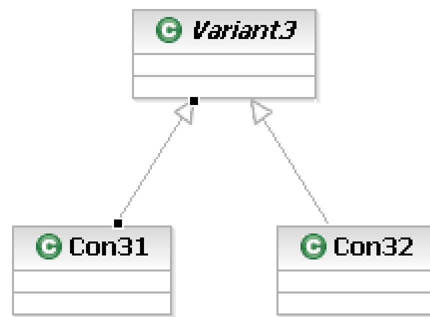

Figure 20.5: Variant class with only constants as sub classes

```
}
```

In this project it has been solved by converting all constants to integer constants. This would look like the example below:

JAVA:

```
abstract class Variant4
{
  public static final int CONST1 = 0;
  public static final int CONST2 = 1;
}
```

then the `switch` expression can be utilized:

Figure 20.6: Variant class with static integer constants

JAVA:

```
public void func(int variant)
{
  switch(variant)
  {
    case Variant4.CONST41: {...} break;
    case Variant4.CONST42: {...} break;
  }
  ...
}
```

All constants (alternative):
Another approach which is similar to the above just above. It is utilized when type checking on the variant type is wanted. Here the variant can be defined as:

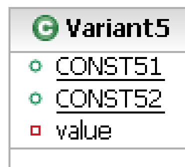

Figure 20.7: Variant class with static integer constants and internal variable

when using this variant as function parameter type checking can be used:

JAVA:

```
public void func(Variant5 variant)
{
  switch(variant.getValue())
  {
    case Variant5.CONST51: {...} break;
    case Variant5.CONST52: {...} break;
  }
  ...
}
```

20.1.6 Case expressions

Case expressions are mainly used for deciding which constructor is used to create a variant value.

As seen in section 20.1.5: "*Variants*" above, we have two different approaches to choose among which is heavily dependent on the structure of the variant.

If the RSL case expression selects among static integers then a switch in JAVA can be used, but if the variant constructors has parameters then we saw in last section that the RSL *case* expression is now an `if-then-else` in JAVA utilizing the `instanceof` operator which determines if an `object` is an instance of a specific class.

20.1.7 Preconditions

The following RSL example shows a basic precondition where *precond*() is a predicate:

RSL:

value
f : T1 → T2
f(x) ≡ ...
pre precond(x)

If a precondition exists there is obviously some unintended result of the function which should be avoided. Therefore the system should be designed so this precondition is unnecessary.

But in any case it should be handled. In JAVA it would be appropriate to throw an exception so the unintended use is noticed and some eventual precautions could be taken. The RSL code could be translated to the following:

JAVA:

```
public T2 f(T1 x)
{
```

```
  if (!precond())
   throw new Exception(``some error msg'');
  ...
}
```

Here it is checked if the precondition is not satisfied (`!precond()`). If it is not, the function call is terminated by an exception.

20.1.8 Axioms and predicates

Axioms and predicates implemented as functions are boolean expressions in RSL which - among other things - can utilize mathematical quantification. These can be quite tricky to translate to JAVA.

Therefore the normal procedure is to eliminate and change all axioms and quantification during RSL refinement. But knowing that these changes where to change quantification to some kind of *looping* over a known set of entities, this conversion could just as well be done in the translation to JAVA. If converted while refining the model, the result should probably be optimized for JAVA syntax and semantics anyway.

Ordinary axioms used in RSL to enforce some property at all times cannot be directly implemented in JAVA. Take for example a simple axioms $x > 5$. This means that "at all times" the variable x should be greater than 5. Surely some programming can guard the variable x from unintended use, but still we cannot be absolutely sure that some code which has direct access to the variable does not change the value illegally.

The rest of this section concerns translating RSL predicates, implemented as functions utilizing mathematical quantification, into JAVA.

RSL predicates can be very complex so the sections below only presents simple ideas of translating these predicates. No doubt these ideas can be applied recursively for more complex expressions but no general study has been performed.

Universal quantification

The universal quantifier (∀) expresses that the predicate following is satisfied by all possible values of some type. An example is shown below:

RSL:

axiom
  ∀ x : T • p(x)

For this expression to be translated into JAVA, it is crucial that the type T represents some finite set (`Vector`, `Hashtable` or other set types) of values. This could then be translated to:

JAVA:

```
for (; T.hasNext(); T.next())
{
  if (!p(T.current()))
    return false;
}

return true;
```

Exists quantification

The existential quantifier (∃) expresses that at least one value of some type satisfies some predicate. An example is shown below:

RSL:

axiom
  ∃ x : T • p(x)

Using an example like the JAVA example above, a structure like the one below would perform the same function in JAVA:

JAVA:

```
for (; T.hasNext(); T.next())
{
  if (p(T.current()))
    return true;
}

return false;
```

If the specific existential quantifier ($exists!$) is used then a count variable should be added to the JAVA code above. Only when a certain amount of values matches the predicate, the entire expression is true. An example is given below:

RSL:

axiom
  ∃! x1,x2 : T • p(x)

would translate to:

JAVA:

```
int counter = 2
for (; T.hasNext(); T.next())
```

```
{
  if (p(T.current()))
    counter--;
}

if (counter == 0)
  return true;
else
  return false;
```

Implication

The implication (⇒) could easily be translated to its equivalent in RSL before being translated to JAVA:

RSL:

$p(x) \Rightarrow r(x)$

is equivalent to

RSL:

$\sim p(x) \vee r(x)$

which translates nicely to

JAVA:

```
(!p(x) || r(x))
```

Complex expressions

In expression, more complex than the ones in the section above, many values can be universally quantified at the same time. Using the technique from the sections above we should create a `for-loop` for each quantified value.

But this is not always the best approach. The suggestion above would surely work but could introduce some unnecessary processing time. An example using the types of the RSL model in this project is given below:

RSL:

axiom
  ∀ t : T.TrainID, speed, maxSpeed : T.Speed •
    speed = D.getTrainSpeed(t) ∧
    maxSpeed = S.getTrainMaxSpeed(t)
      ⇒
        speed ≤ maxSpeed

When this RSL expression is implemented in JAVA then the type *T.Speed* is implemented as a `double`. Therefore it would be very expensive to loop through all `doubles` until one matches the expression *D.getTrainSpeed(t)*.

Clearly it is just the intention to extract the properties from the train and compare them. It would therefore be trivial to create a loop over all trains and compare their speed to their max speed.

initialise axiom

In the imperative specification of the model an axiom *[initial]* was added. This axiom from the *SBCC* scheme is shown below:

[initial]
  initialise post initReq()

it requires that the predicate *initReq* is true after initializing the module. This implies that all variables are fulfills this predicate. Since all variables are initialized by the initial state value in each module, this value should fulfill the *initReq* predicate.

This has been implemented in JAVA by calling the predicate in the *constructor* of a class with a such axiom. An example is shown below:

```
class StateClass
{
  State state;

  public StateClass(State initState)
  {
    initReq(initState);
    this.state = initState;
    ...
  }
}
```

The function `initReq` is designed to throw an exception if the predicate is not fulfilled. In this way the system will never run unless the predicate is true for `initState`.

20.1.9 Concurrency

If the RSL model was of concurrent nature then a lot of other considerations should be taken into account. But because this RSL model is not concurrent we have decided to put the ideas for concurrency in appendix E.

20.1.10 Example: Model translation

This section shows how the function *getNextSB* in the Statics module in the RSL model is translated to JAVA.

RSL: getNextSB

In the RSL model the *getNextSB* function looks like:

RSL:

```
value
  /* Given an SB, it returns the next SB */
  getNextSB : T.SBID × T.Direction ~→ read Segs.v_segs, SBs.v_SBs T.SBID
  getNextSB(sb,dir) ≡
    let
      nextSeg = getSBSeg(sb,dir)
    in
      case nextSeg of
        T.seg(segID) → getSegSB(segID,dir),
        T.point(upSeg,downSeg) → getSegSB(upSeg,dir)
      end
    end
  pre getSBType(sb) ≠ T.ENDSB ∨ getEndDir(sb) ≠ dir
```

JAVA: getNextSB

In JAVA the *getNextSB* function looks like:

JAVA:

```
/**
 * Given an SB, it returns the next SB
 **/
public String getNextSB(String sb,int dir)
throws Exception
{
  /* Check precondition */
  if(getSBType(sb).isEndSB() && getEndDir(sb) == dir)
      throw new Exception("No next SB after the SB (" + sb + ")");

  SBSegment nextSeg = getSBSeg(sb,dir);

  if(nextSeg instanceof Seg)
  {
    Seg s = (Seg)nextSeg;
    return getSegSB(s.getSeg(),dir);
  }
```

```
    else if(nextSeg instanceof Point)
    {
      Point p = (Point)nextSeg;
      return getSegSB(p.getUpSeg(),dir);
    }

    return null;
  }
```

Compare RSL and JAVA: getNextSB

When converting from RSL to JAVA the precondition is tested in the start of the function. If it is not satisfied the function throws an exception

In RSL *nextSeg* is created to hold the return value of *getSBSeg* by using a *let-in-end* expression:

RSL:

```
let
   nextSeg = getSBSeg(sb,dir)
in
   ...
end
```

In JAVA `nextSeg` is a variable that likewise holds the return value of `getSBSeg`.

JAVA:

```
SBSegment nextSeg = getSBSeg(sb,dir);
```

The RSL code between the *in* and *end* keywords in the *let-in-end* expression is a *case* expression:

RSL:

```
case nextSeg of
   T.seg(segID) → getSegSB(segID,dir),
   T.point(upSeg,downSeg) → getSegSB(upSeg,dir)
end
```

In JAVA this looks like:

JAVA:

```
if(nextSeg instanceof Seg
{
  Seg s = (Seg)nextSeg;
```

```
    return getSegSB(s.getSeg(),dir);
  }

  else if(nextSeg instanceof Point)
  {
    Point p = (Point)nextSeg;
    return getSegSB(p.getUpSeg(),dir);
  }

  return null;
```

In JAVA you cannot use the equivalent `switch-case` expression to the RSL expression *case*. Therefore the `if` expression and the `instanceof` operator is used to test the type of the `nextSeg` variable.

20.2 JAVA program structure

This section describes the structure of the JAVA code constituting the railway simulator.

The JAVA code has been divided into seven packages with coherent content. These packages are shown in figure 20.8.

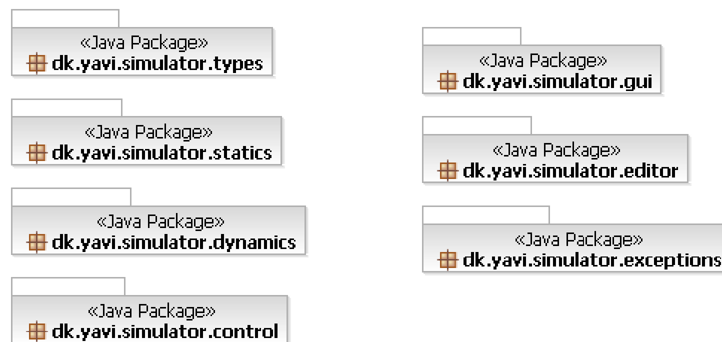

Figure 20.8: JAVA packages

types contains a number of entity classes derived from the data types in the Types module in the model.

statics contains a number of classes derived from the Statics module in the model.

dynamics contains a number of classes derived from the Dynamics module in the model.

control contains a number of classes derived from the Control module in the model.

gui contains a number of classes used to make the graphical user interface (GUI) of the simulator. The configuration editor is not a part of this package.

editor contains a number of classes used to make the graphical user interface for the configuration editor part of the simulator.

exceptions contains a number of exception classes used for error handling.

The next sections describe in further details the content of these packages.

20.2.1 The types package

The most important classes in the types package are shown in figure 20.9 and in figure 20.10.

Figure 20.9 shows a number of different classes (mostly entity classes). Some of them are composite. The classes in figure 20.9 are easy recognized in the *Types* module in the model.

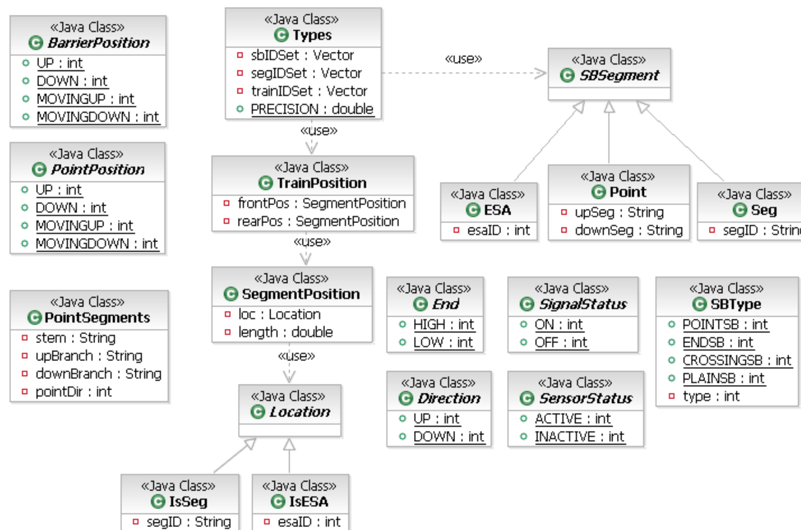

Figure 20.9: The types package

Figure 20.10 shows some classes used in the control part for communication purpose. The `ComMsg` class is used for sending messages between trains (TCCs) and SBs (SBCCs). The `ComID` class contains the ID of either a train or a SB while the `Message` class contains the actual message. The structure of the classes in figure 20.10 is easy recognized in the *Types* module in the model.

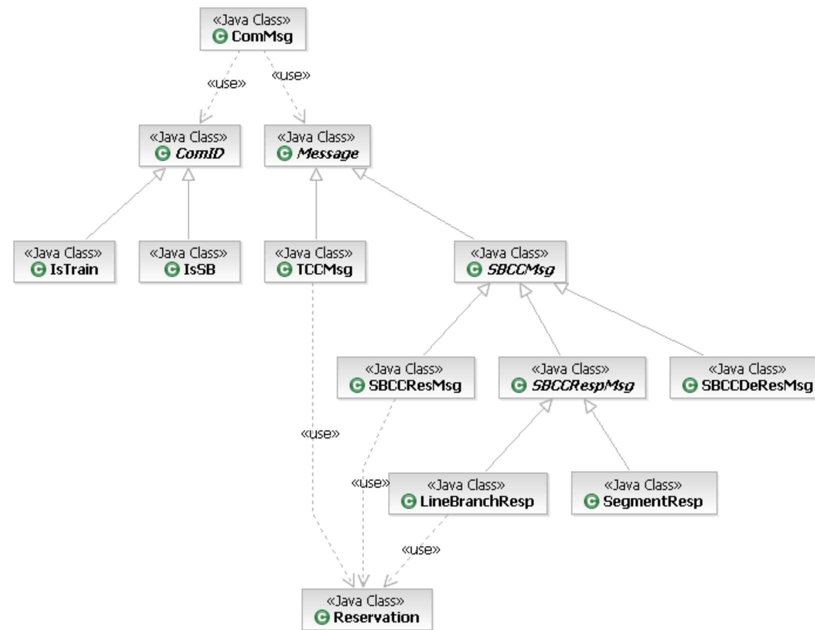

Figure 20.10: The types package

20.2.2 The statics package

The most important classes in the statics package are shown in figure 20.11. The structure of the classes is easy recognized in the *Statics* part of the model.

Statics is the main class in the `statics` package. It gathers everything dealing with the configuration of the railway line.

Trains contains the configuration part that deals with trains only and functions to read from this configuration part.

Segs contains the configuration part that deals with segments only and functions to read from this configuration part.

SBs contains the configuration part that deals with SB only and functions to read from this configuration part.

ESAs contains the configuration part that deals with the two ESAs only and functions to read from this configuration part.

TrainData is an entity class containing the configuration data for one train.

SegData is an entity class containing the configuration data for one segment.

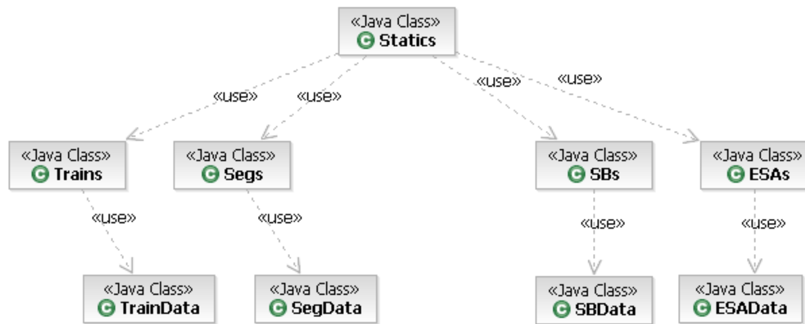

Figure 20.11: The statics package

SBData is an entity class containing the configuration data for one SB.

ESAData is an entity class containing the configuration data for both ESAs.

20.2.3 The dynamics package

The most important classes in the dynamics package are shown in figure 20.12. The structure of the classes is easy recognized in the *Dynamics* part of the model.

Figure 20.12: The dynamics package

Dynamics is the main class in the `dynamics` package. It gathers everything

dealing with the physical state of the railway line.

TrainDyn contains the part of the physical state that deals with trains only, functions updating the position and speed of the trains and functions to read from the states.

SBDyn contains the part of the physical state that deals with SBs only and functions to read from the states.

TrainStates contains the physical states of all the trains in a Hasbtable.

SBStates contains the physical states of all the SBs in a Hasbtable.

TrainState is an entity class containing the physical state of one train. The `TrainState` objects are stored in the `TrainStates` object.

SBState is an entity class containing the physical state of one SB. The `SBState` objects are stored in the `SBStates` object.

20.2.4 The control package

The most important classes in the control package are shown in figure 20.13. The structure of the classes is easy recognized in the *Control* part of the model.

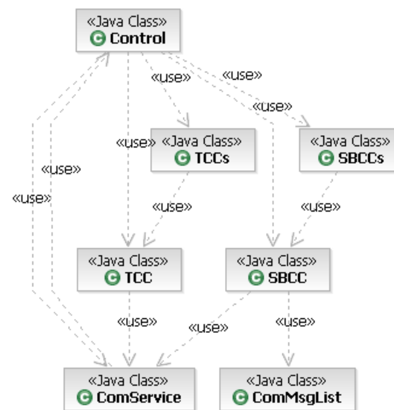

Figure 20.13: The control package

Control is the main class in the `control` package. It gathers everything dealing with the control state of the railway line. It i.a. contains two objects of the `TCCs` and `SBCCs` class.

TCCs contains the TCC objects of all the trains in a Hasbtable.

SBCCs contains the SBCC objects of all the SBs in a Hasbtable.

TCC contains the control state of one train, functions to modify the state, functions implementing the TCC algorithm and functions to communicate with other `TCCs` and `SBCCs`. The `TCC` objects are stored in the `TCCs` object.

SBCC contains the control state of one SB, functions to modify the state, functions implementing the SB algorithm and functions to communicate with other `SBCCs` and `TCCs`. The `SBCC` objects are stored in the `SBCCs` object.

ComService provides the communication service that enables the entities (`TCC` and `SBCC`) to communicate with each other.

20.2.5 The gui package

The most important classes in the gui package are shown in figure 20.14.

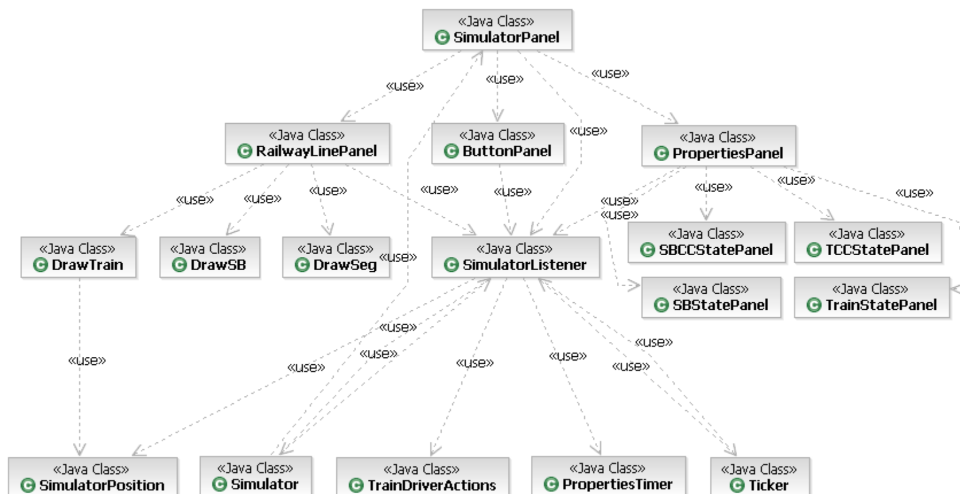

Figure 20.14: The gui package

Simulator is the main class of the simulator. It starts up the GUI etc.

SimulatorListener is an action listener class that handles the events from the GUI (pressing buttons etc.)

SimulatorPanel is a GUI panel that shows the railway simulator.

RailwayLinePanel is a GUI panel that is part of the SimulatorPanel. It shows the layout of the railway line with all the entities (ESAs, segments, SBs and trains).

ButtonPanel is a GUI panel that is a part of the SimulatorPanel. It shows a number of buttons that each corresponds to an entity in the railway line (as shown in the `RailwayLinePanel`). If a button is pressed the correspond entity's properties are shown below the buttons (PropertiesPanel).

PropertiesPanel is a GUI panel that contains the static-, dynamic- and control properties (configuration, physical state and control state) of one entity.

DrawTrain provides the ability to draw one train at the `RailwayLinePanel`.

DrawSB provides the ability to draw one SB at the `RailwayLinePanel`.

DrawSeg provides the ability to draw one SB at the `RailwayLinePanel`.

SimulatorPosition is a class that keeps track of the global position of the static entities in the `RailwayLinePanel`. I.e. it keeps track of the distance from the left most position of the railway line to the left most position of each segment, to the center of each SB and to the left most position of the high ESA (the distance to the low ESA is zero). In this way when a static entity is to be drawn `SimulatorPosition` can tell the position (in pixels) where the entity should be drawn. `SimulatorPosition` is initialized when a configuration is loaded into the simulator. A scale factor is included in the class to scale the layout of the railway line when displayed on the computer screen.

SBCCStatePanel is a GUI panel that is a used by the `PropertiesPanel` to show the control state of a SB.

SBStatePanel is a GUI panel that is a used by the `PropertiesPanel` to show the physical state of a SB.

TCCStatePanel is a GUI panel that is a used by the `PropertiesPanel` to show the control state of a train.

TrainStatePanel is a GUI panel that is a used by the `PropertiesPanel` to show the physical state of a train.

TrainDriverActions is a class that stores actions initiated by pressing a button in the GUI. These actions can be either changing the acceleration or direction of a train (train driver actions), switching a point or opening/closing a crossing. The model is not synchronized since the physical system and control system are ticked sequentially. `TrainDriverActions` are therefore used to save an action. The action is then performed after

the physical system is ticked but before the control system is ticked so that the control system always can undo the action if it is not allowed (if it will result in the speed being to large etc.)

PropertiesTimer is a timer class used for updating the `PropertiesPanel` repeatedly when an entity has been selected in the `ButtonPanel`.

Ticker is a timer class used for ticking of the physical system and the control system. At every tick it also performs the *train driver action* from the `TrainDriverAction` class and updates the `RailwayLinePanel`.

20.2.6 The editor package

The most important classes in the editor package are shown in figure 20.15.

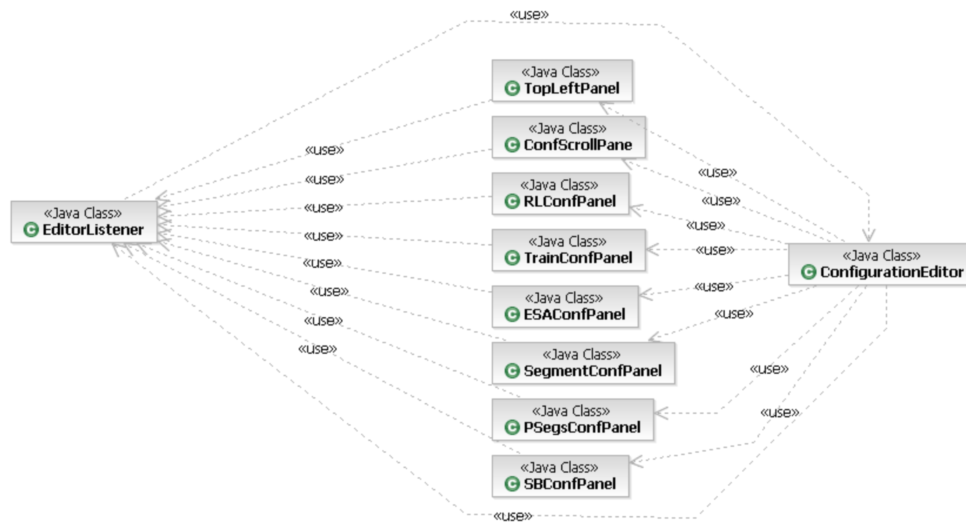

Figure 20.15: The editor package

ConfigurationEditor is a GUI panel that shows the configuration editor which is a part of the simulator program.

EditorListener is an action listener class that handles all the events from the configuration editor GUI (pressing buttons etc.).

TopLeftPanel is a GUI panel that contains the buttons in the top left corner (Import, Export, Delete, Load, Is wellformed and New).

ConfScrollPane is a GUI scroll pane that contains a list of all the configurations in the configuration editor.

RLConfPanel is a GUI panel that shows a railway line configuration in the form of buttons for each entity. When a button is pressed the configuration data of that entity is showed above and can now be updated.

TrainConfPanel is a GUI panel that shows the configuration data of one train and allows the user to change this data.

ESAConfPanel is a GUI panel that shows the configuration data of one ESA and allows the user to change this data.

SegmentConfPanel is a GUI panel that shows the configuration data of one segment and allows the user to change this data.

PSegsConfPanel is a GUI panel that shows the configuration data of two coherent branch segments and allows the user to change this data.

SBConfPanel is a GUI panel that shows the configuration data of one SB and allows the user to change this data.

20.2.7 The exceptions package

The classes in the exceptions package are shown in figure 20.16.

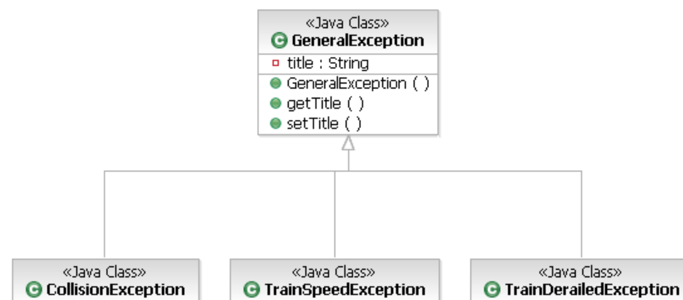

Figure 20.16: The exceptions package

GeneralException is an *Exception* class that also has a title besides the exception message.

CollisionException extends the `GeneralException` class and specializes the class for handling exceptions thrown when a collision occurs.

TrainDerailedException extends the `GeneralException` class and specializes the class for handling exceptions thrown when a train is derailed.

TrainSpeedException extends the `GeneralException` class and specializes the class for handling exceptions thrown when a train drives faster than its max allowed speed (an exception is not thrown if the train drives faster than the max allowed speed of a segment).

20.3 Translating configuration to XML

The simulator is made generic so it is possible to change the railway line configuration. Therefore it is preferable to be able to save configurations in files at the hard drive. The XML format has been chosen for this purpose. In section 21.4 it is explained how XML files can be imported/exported to/from the simulator.

To be able to save a configuration in a XML file it is necessary to make a specification of the structure of the XML. This is done using a XML DTD[1]. This section describes how the *Configuration* type in the imperative Statics module is the basis for making a DTD determining the structure of the XML files.

The `Configuration` type is defined as a product of the types of interest of the four submodules(objects) in the Statics module:

type
Configuration = SBs.SBs × Segs.Segs × ESAs.ESAs × Trains.Trains

This structure is directly reflected in the DTD:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT Configuration (SBs,Segs,ESAs,Trains)>
<!ATTLIST Configuration name CDATA #REQUIRED>
```

The first line tells that this is an XML file using the *ISO-8859-1* encoding which allows us to use the danish letters *æ*, *ø* and *å*.

The second line defines the root element of the XML file which must be a `Configuration` tag. Furhter more it specifies that the `Configuration` tag must contain the four tags `SBs`, `Segs`, `ESAs` and `Trains`, which match the names of the types of interest in the submodules of the Statics module.

The third line defines that `Configuration` must have an attribute called `name` which contains the name of the configuration. This is not used in the formal model but only in the configuration editor in the simulator to identify the different configurations.

[1]Document Type Definition

The next four sections describe the four tags encapsulated by the `Configuration` tag.

20.3.1 SBs

The formal specification of the type of interest (*SBs*) in the module SBs is defined like:

value

```
    /* Type of interest */
    SBs = T.SBID →m SBData,

    /* Data for each SB */
    SBData == mk_sb(getUpSeg : T.SBSegment,
                    getDownSeg : T.SBSegment,
                    getType : T.SBType,
                    getPointTicks : T.HasTicks,
                    getBarrierTicks : T.HasTicks,
                    getSignalTicks : T.HasTicks)
```

This structure is directly reflected in the DTD:

```
<!ELEMENT SBs (SBData,SBData+)>

<!ELEMENT SBData (SBSegment,SBSegment)>
<!ATTLIST SBData SBID CDATA #REQUIRED>
<!ATTLIST SBData sbType (POINTSB | ENDSB | CROSSINGSB | PLAINSB) #REQUIRED>
<!ATTLIST SBData pointTicks CDATA #IMPLIED>
<!ATTLIST SBData barrierTicks CDATA #IMPLIED>
<!ATTLIST SBData signalTicks CDATA #IMPLIED>

<!ELEMENT SBSegment (Seg | Point | ESA)>
<!ATTLIST SBSegment dir (UP | DOWN) #REQUIRED>

<!ELEMENT Seg EMPTY>
<!ATTLIST Seg seg CDATA #REQUIRED>

<!ELEMENT Point EMPTY>
<!ATTLIST Point upSeg CDATA #REQUIRED>
<!ATTLIST Point downSeg CDATA #REQUIRED>
```

The first line says that the `SBs` tag should contain at least two `SBData` tags (the smallest possible configuration contains two end switch boxes). In the model the *SBs* type is a mapping from *SBID* to *SBData*. In the DTD each `SBData` contains the *SBID* as a parameter and the `SBData` tags are just listed after each other in no particular order inside the `SBs`.

In the model *SBData* is a product of different static properties of a SB. In the DTD these are attributes of `SBData` except for the up- and down `SBSegments`

that are composite structures. Therefore they are encapsulated in `SBData`. As in the model (not shown here) `SBSegment` is either a `Seg`, `Point` or `ESA`.

20.3.2 Segs

The formal specification of the type of interest (*Segs*) in the module Segs is defined like:

```
value
   /* Type of interest */
   Segs = SegsData × SegPoints,

   SegsData = T.SegmentID →m SegData,
   SegPoints :: getRP : T.Length
                getBP : T.Length,

   /* Data for each Segment */
   SegData == mk_seg(getUpSB : T.SBID,
                     getDownSB : T.SBID,
                     getLength : T.Length,
                     getMaxSpeed : T.Speed)
```

The structure is a bit different in the DTD:

```
<!ELEMENT Segs (SegData+)>
<!ATTLIST Segs resPoint CDATA #REQUIRED>
<!ATTLIST Segs brakePoint CDATA #REQUIRED>

<!ELEMENT SegData EMPTY>
<!ATTLIST SegData SegmentID CDATA #REQUIRED>
<!ATTLIST SegData upSB CDATA #REQUIRED>
<!ATTLIST SegData downSB CDATA #REQUIRED>
<!ATTLIST SegData length CDATA #REQUIRED>
<!ATTLIST SegData maxSpeed CDATA #REQUIRED>
```

The reservation- and brake points stored in *SegPoints* in the model are now attributes of the `Segs` tag which also contains a number of `SegData` tags.

In the model *SegData* is a product of different static properties of a segment. In the DTD these are attributes of `SegData`.

20.3.3 ESAs

The formal specification of the type of interest (*ESAs*) in the module ESAs is defined like:

```
value
   /* Type of interest */
```

ESAs == mk_esa(getLowSB : T.SBID,
getHighSB : T.SBID,
getLowLength : T.Length,
getHighLength : T.Length)

This structure is directly reflected in the DTD:

```
<!ELEMENT ESAs EMPTY>
<!ATTLIST ESAs lowSB CDATA #REQUIRED>
<!ATTLIST ESAs highSB CDATA #REQUIRED>
<!ATTLIST ESAs lowLength CDATA #REQUIRED>
<!ATTLIST ESAs highLength CDATA #REQUIRED>
```

These four static properties of the ESAs are now attributes of the `ESAs` tag.

20.3.4 Trains

The formal specification of the type of interest (trains) in the module Trains is defined like:

value
/* Type of interest */
Trains = T.TrainID $\overrightarrow{m}$ TData,

/* Data for each Train */
TData == mk_train(getLength : T.Length,
getMaxSpeed : T.Speed,
getMaxAcc : T.Acceleration,
getMaxDec : T.Acceleration)

This structure is directly reflected in the DTD:

```
<!ELEMENT Trains (TrainData*)>

<!ELEMENT TrainData EMPTY>
<!ATTLIST TrainData TrainID CDATA #REQUIRED>
<!ATTLIST TrainData length CDATA #REQUIRED>
<!ATTLIST TrainData maxSpeed CDATA #REQUIRED>
<!ATTLIST TrainData maxAcc CDATA #REQUIRED>
<!ATTLIST TrainData maxDecel CDATA #REQUIRED>
```

In the model the trains type is a mapping from *TrainID* to *TrainData*. In the DTD each `TrainData` contains the `TrainID` as a parameter and the `TrainData` tags are just listed after each other in no particular order inside the `Trains` tag.

In the model *TrainData* is a product of different static properties of a train. In the DTD these are attributes of `TrainData`.

20.4 Differences from RSL model

This section lists the notable differences applied to the JAVA program after translation from RSL.

20.4.1 Train re-request timing

A rare problem surfaced some times which is caused by the sequential nature of the time update structure. The problem consisted in train asking for a reservation of a line from opposite ends. When these requests where timed so they almost overlapped, the control algorithm would live-lock. It happened like this:

1. The train t_1 at one end of the line made a reservation request.
2. The SB SB_1 checked its local space for reservations and if none then temporarily reserved the line locally.
3. The train t_2 makes the same request at the other end.
4. The SB SB_2 does the same as SB_1.
5. SB_1 requests a reservation at SB_2.
6. SB_2 does the same as SB_1.
7. SB_1 processes the request of SB_2 and sends negative response because the reservation is temporarily occupied while waiting for response from SB_2.
8. SB_2 does the same.
9. SB_1 processes the negative response from SB_2 and dereserves locally.
10. SB_2 does the same.

This was solved by having trains wait a random amount of ticks between each re-request, just like a re-transmission protocol in a basic Ethernet network.

This has not been implemented in RSL because we only want to deal with safety and not liveness properties which are normally much more complex to prove.

Chapter 21

Using the simulator

This chapter describes shortly how to use the simulator.

21.1 Starting the simulator

When the simulator is started the first time no configuration exists. The simulator therefore shows the Configuration Editor in which it is possible to create, edit and delete configurations that can be used in the simulator. It is also possible to export and import configurations and check whether they are wellformed.

How to create configurations is described in section 21.3 while import and export of configuration from and to XML is described in section 21.4.

A configuration that has been saved can be loaded into the simulator workspace by pressing the *Load* button. This opens the train simulator with the configuration.

The next section describes how to "play train driver", i.e. how to start and stop the trains etc.

21.2 Playing train driver

21.2.1 The railway line

When the train simulator is started with a configuration the railway line layout is showed. An simple example of this is showed in figure 21.1.

ESAs

The green rectangles in both ends of the railway line represents the ESAs. The low ESA is shown in the left side.

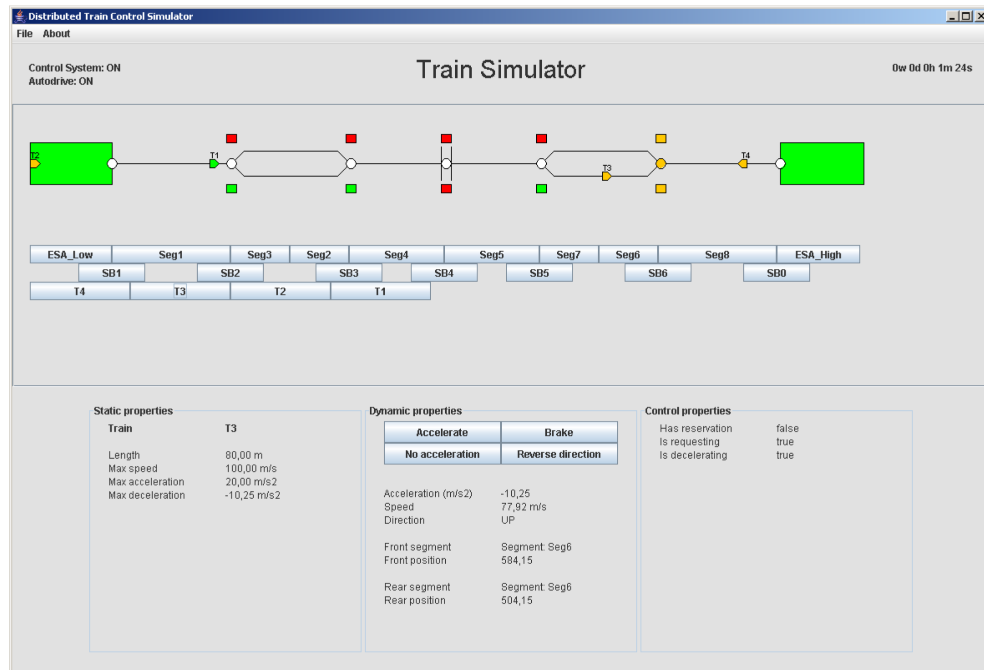

Figure 21.1: Train simulator

Switch boxes

Switch boxes are shown as white circles. When a train is located on a SB - or more precisely: on the sensor at a SB - the SB is colored red. When a SB is preparing one of its segments it is colored orange.

Crossings

Crossings are shown as two vertical lines at the left and right side of the SB. Two small boxes are placed respectively above and below the SB. These boxes represents the state of the crossing. If the boxes are red the crossing is open (barriers are up and signals are off). Green means that the crossing is closed (barriers are down, signals are off). If the crossing is opening or closing the color of both boxes are orange.

Points

When the railway line branches at a junction the branches are shown with the *down branch* above the *up branch*.

Boxes are used - as at crossing - to show the position of a point. If a box is green the

point is connected to the branch close to the box and the opposite box is then red. If both boxes are orange the point is switching from one branch to the other.

Trains

All trains are initially located in the low ESA. The ID of a train is shown just above the train. A train is red if it does not have a segment reservation, orange if it is currently asking for a segment reservation and green if it has received a segment reservation.

21.2.2 Buttons

Three rows of buttons are showed below the railway line layout. The buttons in the first row represents the two ESAs and all the segments. The buttons in the second row represents all the SBs and the buttons in the third row represents all the trains (if any). By pressing a button the static, dynamic and control properties of the entity the button represents are shown below the buttons.

Train buttons

When a train button is selected four buttons are showed in the *Dynamic properties* panel. With these it is possible to play the train driver. You can set the acceleration to either maximum (*Accelerate*), minimum (*Brake*) or to zero (*No acceleration*). Besides you can reverse the direction of the train if the train is parked (speed is zero) in an ESA facing away from the railway line. The control system changes the acceleration of the train if necessary to keep the railway line safe and consistent.

ESA- and segment buttons

When an ESA button or a segment button is pressed the static and dynamic properties of the ESA or segment are shown but without the possibility to change any of the properties.

SB buttons

When a SB button is pressed the static, dynamic and control properties of the SB are shown. If the SB is a point SB, a button is provided to be able to switch the point. If the SB is a crossing SB, a button is likewise provided to be able to open or close the crossing.

21.2.3 Control system

The control system can be switched off (but not on). This is done by pressing the *Close control system* menu in the *File* menu. Now safety in the railway line is not preserved which might result in collisions etc.

21.2.4 Autodrive

The simulator has been implemented with an *autodrive behavior*. In the *File* menu the autodrive behavior can be switched on and off.

Autodrive behavior means to let the trains drive when they can and are allowed to do so. If a TCC brakes a train because it does not have a segment reservation for the next segment, it keeps asking for this reservation. When it has received the reservation the behavior part of the TCC accelerates the train again. When the train is parked (holding with zero speed) in an ESA the TCC automatically changes the direction of the train and accelerate it.

Following this algorithm the trains continuously drives up and down the railway line.

21.3 Creating new configurations

When the Configuration Editor is started it is possible to create new configurations. The following describes how to do that.

Figure 21.2 shows the configuration editor where a configuration has been selected. The layout of the railway line is showed using buttons with icons that represents the ESAs, SBs and segments. The icons help picturing the layout of the railway line.

21.3.1 Add new configuration

To add a new configuration press the *New* button in the top left corner. The smallest possible configuration is then shown. Notice that this configuration is not saved.

21.3.2 Add a segment

To add a segment press the *Add SB/Segment* button. Then both a SB and a segment are added just to the left of the right most SB (the end SB at the high ESA).

Before adding the SB/segment pair the type of SB should be chosen by using the combo box just above the *Add SB/Segment* button. The SB can not be a end SB. If a point SB is chosen, two SBs are added along with three segments (two branches and the stem to the right).

21.3.3 Delete a segment

To delete a segment press the *Delete SB/Segment* button. Then both a SB and a segment are deleted at the same place as they are added. If the deleted SB is a point SB both of the point SBs are deleted along with both branches and the right stem.

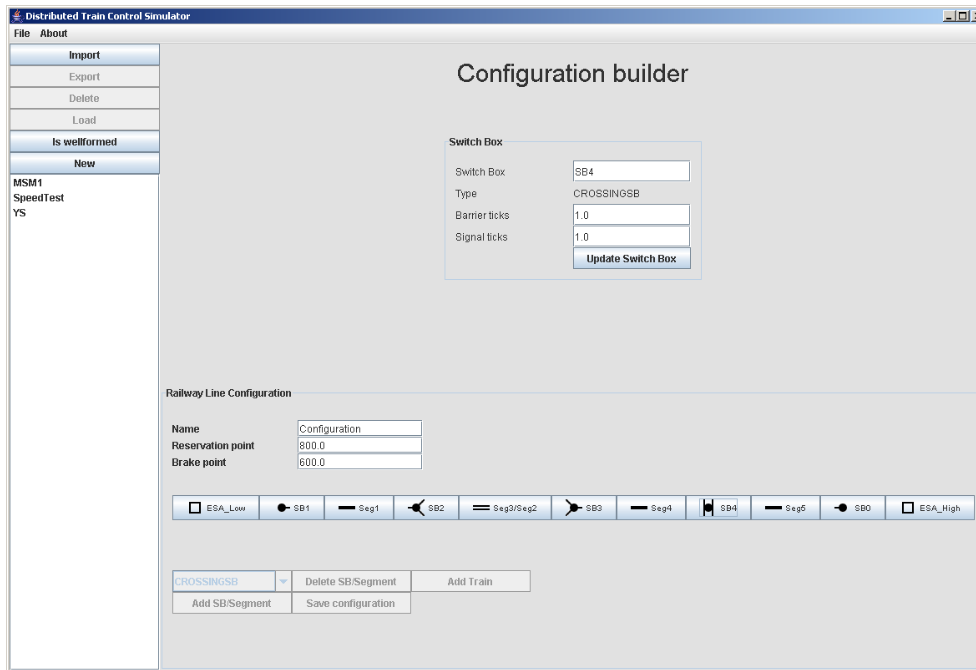

Figure 21.2: Configuration editor

21.3.4 Add a train

A train is added by pressing the *Add Train* button.

21.3.5 Delete a train

A train is deleted by pressing the *Delete* button that is shown when the train button is pressed.

21.3.6 Update properties

It is possible to update the static properties of all the entities in the configuration by pressing the button which shows the ID of the entity. Then the properties are shown above along with a button to apply the changes.

21.3.7 Save a configuration

A configuration is saved as a XML file by pressing the *Save configuration* button. If the configuration has not been saved before it will now appear in the list at the left with the name stated in the *Name* field.

By pressing one of the configurations in the list, the configuration is shown and can now be updated. A selected configuration can be deleted by pressing the *Delete* button in the upper left corner.

21.3.8 Checking wellformedness

By pressing the *Is wellformed* button is the top left corner, the currently selected configuration is checked to find out if it is wellformed. A dialog shows the result of the check.

21.3.9 Loading a configuration

A wellformed configuration that is selected in the list can be loaded into the simulator by pressing the *Load* button. The configuration editor is then closed and the simulator is opened.

21.4 XML importing/export

A configuration can be imported from a XML file by pressing the *Import* button. It will then appear in the list to the left.

When a configuration in the list is selected it can be exported to a XML file by pressing the *Export* button.

The DTD that the XML files must obey can be found in appendix C.

Chapter 22

Test

This chapter describes the strategy of how this system has been tested. The system has only been tested on system level because the complexity of the system makes a structural test too comprehensive.

22.1 Test configuration

The tests below has been performed on an IBM Thinkpad T23 laptop computer with a Pentium III 1.3 GHz Mobile processor, 1 GB RAM. The operating system is Windows XP service pack 2. The JAVA environment is Sun Standard JAVA v.1.4.2_05.

The simulator configuration contained 1 crossing, 4 points and 6 trains.

22.2 Test strategy

This section lays out two kinds of test strategies. The first is the basic tests which tests the system for basic functionality. The second is more performance and real time related.

22.2.1 Basic tests

Because of the layered structure of the system it is tested one layer at the time. Below is described what is tested in each layer:

Statics All predicates which are part of the *is_wf()* predicate is tested individually by importing a XML file (see section 21.4) with a configuration which is not wellformed in the way that it targets the predicate to be tested.

An imported configuration can be tested for wellformedness in the configuration editor by pressing the *Is wellformed* button after the configuration is imported.

An error message should be shown describing the problem found by the well-formedness check.

Dynamics All predicates which are part of the *safe()* predicate is tested individually by obtaining a state (situation) which is restricted by the targeted predicate. This is only possible of course if the control system is disabled. Else it is not possible to obtain a unsafe state.

(*This is not entirely true because the user has the power to change points and crossings even as the trains are passing them while the control system is enabled. This immediately results in an unsafe state.*)

An error message should be shown describing the unsafe situation attained.

Control Several scenarios are run while checking that the information stored in the control entities (SBCC, TCC) corresponds to the algorithm explained in section 11.1.

22.2.2 Performance tests

The following section concerns the strategy of testing the performance of the system.

The performance of the system is quite important because it is designed to be a real time simulation of a railway system. We have implemented a clock in the simulator so one can keep track of the time progress in the simulated environment.

It is a goal that this simulation time should progress in sync with the real clock. Basically this should only depend on one thing: A time update with tick time t should have a calculation time less than t.

For example let us take a tick time of 0.05s. This means that a timer calls the time update function every 0.05s. If the system should keep up with the real world it should have completed the time update calculation and be ready for the next within 0.05s. Else the simulator starts lagging behind the real world.

Average tick calculation time

The average calculation time has been calculated by running a certain amount of ticks in a loop while timing it. The results can be seen below in the following sections.

22.3 Test listings

This section lists all concrete tests and results if any.

22.3.1 Basic test listings

This section lists all the specific tests carried out for each system layer. These tests are a detailed listing of the sketched test strategy in the above section.

Statics

All sub predicates of the *is_wf()* predicate is tested individually by creating non-wellformed XML configurations and importing these using the configuration editor. Non-wellformed configurations are created by negating each invariant expression in the predicates and then creating configurations which satisfy these negated invariants.

The predicates and the associated tests are listed below:

sbsHaveConf() :

1. An SB's configuration data does not exist in the configuration even though its ID exists.
2. The reservation point is less than 0.0.
3. The break point is less than 0.0

getSBSeg_diff() : An SB has the same segments up and down.

getSBSeg_point_wf() : The two banches of a point is the same segment.

getSBSeg_injective() : Two different SB's have the same segment in the same direction.

getSBSegType_wf() :

1. An SB has the type **POINTSB** but it has not got getSBSegments(..) = point(..) in precisely one direction.
2. An SB has the type **ENDSB** but it has not got getSBSegments(..) = esa(..) in precisely one direction.
3. An SB has the type **CROSSINGSB** or **PLAINSB** but it has not got getSBSegments(..) = seg(..) in both directions.

segsHaveConf() : The data of a segment does not exist in the configuration although its ID exists.

getSegSB_injective() : Two segments have the same SB(s) in one or both directions event though they are not branches in a point.

brakeResPoint_wf() : The brake point is greater or equal to the reservation point.

esasHaveConf() : The data of an ESA does not exist in the configuration even though its ID does.

trainsHaveConf() : The data of a train does not exist in the configuration even though its ID does.

getESASBSeg_wf() : A SB with the type **ENDSB** does not have an ESA in the direction of its associated end.

getSBSeg_getSegSB_wf() : A SB has a segment in one direction but the segment does not have the SB in the opposite direction.

seg_train_length_wf() : A segment is shorter than a train.

esa_train_length_wf() : An ESA is shorter than a train.

brakePoint_wf() : The brake point is shorter than the braking distance of a train with maximum velocity.

resPoint_wf() :

1. a trains length added to the reservation distance is greater or equal to a segments length.
2. the brake point is longer than a segment.

esa_seg_length() : An ESA is shorter or equal to the brake point.

Dynamics

System states which violate the predicate *safe()* in the *Dynamics* module are obtained by running the simulator. These should be followed by an error message from the system. All images corresponding to the physical events are listed in appendix D.

noCollisions() :

1. A train crashes frontally into another train.
2. A train crashes into the rear of another train driving **UP**.
3. A train crashes into the rear of another train driving **DOWN**.

trainPosPossible() : This predicate is not tested. It only states that if two train are on same segment driving towards each other, they must be facing each other.

pointsSafe() :

1. A train drives from the stem into a point which is shifting.
2. A train drives from a branch to the stem of a point which is not shiftet to the branch of the train.

crossingsSafe() : A train drives into a crossing in which the barriers are not down.

Control

A number of scenarios should be run to ensure that the control algorithm is followed. The system should be checked, that the *consistent()* predicate holds each time the control state is changed. This happens at the following events:

Reservation point : When a train crosses this point it should request a reservation.

Brake point :

- The train has a reservation and continues.
- The train has not attained a reservation and begins braking.

Sensor :

1. A sensor is left by a train. The appropriate dereservations should be executed according to the algorithm.
2. A sensor is activated by a train. Nothing should happen.

Adding to these tests we can set the system to apply the *consistent()* predicate every some seconds. This is possible because the predicate was translated to JAVA as was the rest of the model. This increases our trust in the system to stay consistent.

22.3.2 Performance test listings

Below is shown a table of all performance test runs measuring tick calculation time:

Ticks	10	100	10^3	10^4	10^5	10^6	10^7
Real time (ms)	120	831	2944	18857	71172	401727	4291440
Simulation time (ms)	500	$5 * 10^3$	$5 * 10^4$	$5 * 10^5$	$5 * 10^6$	$5 * 10^7$	$5 * 10^8$
Average tick time (ms)	12.0	8.31	2.94	1.88	0.71	0.40	0.42
Time compression factor	4.17	6.01	16.98	26.51	70.25	124.46	116.51

Table 22.1: Performance test results

In the table above the the numbers describes the following:

Ticks The amount of ticks calculated in the test.

Real time The amount of time passed while calculating the ticks.

Simulation time The amount of time passed in the simulator which is the time interval of each tick multiplied by the amount of ticks ($tick_interval * ticks$).

Average tick time The average calculation time of a tick.

Time compression factor The factor between the time passed in the real world and the time passed in the simulator.

Startup cost

The data in the table clearly indicates that some startup cost is added to the tests. We can see that when the amount of ticks is increased, the average time for calculating a tick drops. This is probably caused by calculations taking longer time the first time being executed. But as more calculations are carried out looking very similar, a lot of calculation time is saved by caching some intermediate results in the CPU cache.

External processes

But we can also see an indication the tick calculation time rises a bit between the two last runs from 0.4 ms to 0.42 ms. This could easily be caused by JAVA running some garbage collection or another process in the windows operating system is using system resources. Therefore even if a tick takes much less time to calculate than the delay before calculating the next, we cannot guarantee that the system always calculates in real time, because the operating systems today run a lot of processes which is out of our control.

If for example JAVA garbage collection decides to run in the middle of a simulation, we cannot be sure that the simulator is given enough CPU time to complete its calculations.

Ticks threshold

At the time of writing this, the simulator is set with a tick value of 0.05s. This means that the simulator has 50 ms to complete a tick. And with an average calculation time of 0.4 ms this should be more than enough.

We still have to remember that this is only the test results of the machine which is described in the beginning of this chapter. Tick calculation time could easily vary with the amount of RAM and CPU MHz. But seen from this machines point of view, we could turn up the time update frequency without risking low performance and thereby get a more fine-grained simulation.

Configuration size

As mentioned in section 22.1 the initial test configuration contained 1 crossing, 4 points and 6 trains. Lets see how this varies when the number of points and trains is raised. We will perform these experiments with a fixed number of ticks 10^6.

+trains	+points	tick calculation time
6	0	1.02
0	6	0.47
6	6	1.15

Table 22.2: Tests with a varying number of trains and SBs

Clearly it is the number of trains which affects the system the most. This was also the expected result because a train has many processes which has to be updated at every tick where SBs does nothing until receiving a message or a sensor event.

Chapter 23

Verification

This chapter lays out the strategy of how to be able to prove / verify safety in the system developed. The actual formal verification has not been done in this report, but some informal argumentation (validation) is done later in this chapter.

23.1 The idea of provability

The idea of safety is greatly inspired from the papers [5] and [6].

23.1.1 The *safe* predicate

The term *safety* was defined in section 5.4. When the RSL model is developed a predicate $safe$ is specified. This predicate is based on the definition of safety. This predicate can be expressed as a conjunction between all the safety requirements:

$$safe(..) \equiv s_1(..) \wedge s_2(..) \wedge ... \wedge s_n(..)$$

The $safe$ predicate only expresses the fact, that for a railway line to be safe, events like train collisions and derailments has not occurred in the current state. This means that this predicate actually defines when accidents involving trains physically has occurred.

In the mentioned papers (referred to in the beginning of this section) the $safe$ predicate is defined much stronger where a safe state is when two trains are not on the same segment. This is because these models were very abstract concerning train movement so this was the obvious choice.

What we would like to hold now is the implication

$$safe(s) \Rightarrow safe(gen(s, ..))$$

where s is some state and gen is a generator (event) applying some change to the state. This means that all events applied to a safe state leads to another safe state.

The idea is that this should hold when using the developed safety / control system.

But it is not enough to require a state to be safe. For the control system to work properly the state must also be consistent to the rules of the control system. This means that the control system can only function properly if its internal state represents the actual state of the physical world. This again means that if a train is on a given segment, some reservation should exist in the control system that reflects the position of the train so that no other trains are allowed access it.

Therefore if the state of the control system is not consistent with the physical world and the control algorithm then it could possibly allow two trains to enter the same segment. Some examples of *safe* states which at some time will lead to an unsafe state is shown in figure 23.1 and 23.2.

Figure 23.1: A safe state but the trains crashes shortly

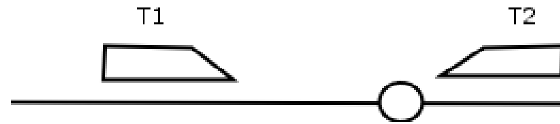

Figure 23.2: A safe state but T_2 enters the segment even though it is occupied.

23.1.2 The *consistent* predicate

A predicate *consistent* is then constructed to include the *safe* predicate

$$consistent(s) \equiv safe(s) \wedge con_1(s) \wedge ... \wedge con_n(s)$$

As mentioned above this predicate defines the consistent relationship between the physical state and the state of the control system.

The idea is now that

$$consistent(s) \Rightarrow consistent(gen(s, ..))$$

can be shown to hold and therefore

$$consistent(s) \Rightarrow safe(gen(s, ..))$$

But again this is not enough. Because all generators cannot be applied at all times. The generators in the physical module (*Dynamics*) has preconditions which are derived from the predicate (*safe*) and therefore guards the system from entering an unsafe and thereby undefined state of the system.

So this means that if only the physical system is considered we could state that:

$$safe(s) \wedge gen_guard(s) \Rightarrow safe(gen(s,..))$$

namely that IF $safe$ holds in a state AND the guard for the specific generator also holds in that state THEN the state obtained by applying the generator is also safe.

23.1.3 Preconditions (guards)

Preconditions exists to enforce intended use of functions. Preconditions are used for 3 different purposes in this model:

Wellformedness All 3 main modules (*Statics*, *Dynamics* and *Control*) have a *is_wf* predicate. Some preconditions exists to enforce that generators preserve wellformedness. Wellformedness concerns data structures to be consistent.

Safe *Dynamics* defines the $safe$ predicate. This predicate tests whether a state is safe (i.e. if train accidents has occurred in the current state). Some preconditions in this module and its sub modules ensure that $safe$ is preserved (or that is the idea. Nothing has been proved).

Consistent *Control* defines the *consistent* predicate. This predicate tests whether the control system state is consistent with the physical state of the system. Also if trains obey the procedures of the control system (use the right branch etc.).

There does not exist any preconditions that enforce the *consistent* predicate (explained below).

The *Dynamics* module

Preconditions in the *Dynamics* module can express properties concerning multiple entities from the entire *Dynamics* module. As seen in the modelling chapters the model is decomposed and the generators moved to the appropriate sub modules.

All generators that need preconditions which include other entities than the ones in its local module need to be in a higher modular level. Therefore most generators are defined in the *Dynamics* module which again calls the generator in the local module.

An example could be the *setTrainPosition* function in the *Dynamics* module which calls the local function in the *TrainDyn* (TD) module which handles all dynamics of trains:

```
setTrainPosition : T.TrainID × T.TrainPosition × State × S.Configuration ~→ State
setTrainPosition(tID,pos,(ts,ss),con) ≡
   (TD.setTrainPosition(tID,pos,ts,con),ss)
pre ~trainPositionOccupied(tID,pos,(ts,ss),con) ∧
      ~tpDerailed(pos,getTrainDirection(tID,(ts,ss)),(ts,ss),con),
```

As seen in the example above the preconditions for this function is $\sim$ *trainPositionOccupied* and $\sim$ *tpDerailed* which express that:

- Another train should not be located at the position that this generator is trying to occupy.
- The position which the generator is trying to apply to the specific train must not be at a crossing or point which is not safe for a train to pass.

The *Control* module

As mentioned the control module does not contain any preconditions to the purpose of preserving *consistent*. It is at the moment not possible to construct such preconditions because no control entity generators exists in the *Control* main module. Therefore these could only express properties about their own local state. This is not enough to ensure consistent.

Again this is out of scope of this project but these considerations have been made, if it should be relevant at some time to carry out a proof.

As of now 2 things can be done to make a proof possible:

1. Strengthen the *consistent* predicate, if possible, so the state of the other control entities can be derived from its own state and the consistent predicate.
2. Create derived generators in the top *Control* module so that stronger preconditions can be specified to the generators.

The strong properties mentioned above should be derived from the control algorithm. Some of these strong properties could be invariants of the algorithm that is not yet included in the *consistent* predicate. Some examples could be:

- When a SB receives a message from a train, it receives the direction of the train. Therefore it is known exactly which segment the train is located on (in the opposite direction of the request direction).
- When a train receives a positive response from a SB it is known that all appropriate SBs have a local copy of the reservation.
- When a train has a reservation for a segment, no other train has the same reservation.

and so on....

23.1.4 Wellformedness

In the same way that generators should preserve the *consistent* and $safe$ predicates, it should also be shown that they preserve wellformedness. If the generators does not preserve wellformedness then the system does not make sense.

The general proof obligation would look like:

$$is_wf(s) \wedge gen_wf_guard(..., s) \Rightarrow is_wf(gen(..., s))$$

These proof obligations are stated as axioms in the abstract model.

23.1.5 The *init_req* predicate

The predicate tells whether a state satisfies the requirements to an initial state. This predicate is needed to carry out an inductive proof.

If the implications in above sections have been proved, then we know that a consistent state always leads to a consistent and thereby safe state.

But for this to give any sense we need a state which is guaranteed consistent and safe. Therefore we also need to prove that:

$$init_req(s) \Rightarrow consistent(s)$$

Then a base for an inductive proof has been created.

All proof obligations mentioned in this section are summarized in section 23.2.

23.1.6 Verifying control algorithm

In the above sections several proof obligations were stated which are needed to verify safety in this system (and the simulator).

But it is not enough to ensure that if preconditions are obeyed then the system is safe. If the developed model/simulator was a discrete event simulator like in [6] the stated proof obligations would be enough. But the events in our system are not initiated by human interaction and buttons.

This is done by the physical system and the control system algorithm. Therefore it is important in this project to ensure that generator guards are implemented as a part of the algorithms in the system. Thereby we have to ensure that a function - though it has a guard - is not called unless the guard is true.

An example to clarify this is the following situation: A train travels with 100 Km/h and reaches the border of a segment. The segment is already occupied so when the generator tries to update the train position a generator guard in the control module (this is not implemented yet, refer to section 23.1.3) prevents the generator in completing its state update.

In this way the system will never reach a not-consistent state, but the system does not function properly. It is not realistic to stop a train instantly when driving 100 Km/h. Therefore it is also vital to verify that the control system algorithm is derived from the generator guards (or vice versa).

23.2 Proof obligations

This section lists the proof obligations mentioned in section 23.1. These obligations are not shown in RSL style **theory** format, but only as short mathematical implications.

The obligations are generic in the way that they are not specified for every single generator in the model but only with the generic generator *gen* which symbolizes all generators in the system.

In the same way, if a statement expresses that is_wf is preserved, it applies for all modules (*Statics*, *Dynamics* and *Control*) because they all define a such predicate. If the statement is about $safe$ it concerns only the generators in *Dynamics* and in the case of *consistent* it concerns the generators in the *Control* module.

23.2.1 *[gen_wf_pres]*

$$is_wf(t) \wedge gen_guard(..., t) \Rightarrow is_wf(gen(..., t)$$

where t is the type of interest in each module, $is_wf()$ is the associated wellformedness predicate for this type, *gen_guard* is the guard (precondition) for the generator *gen*.

This requires all generators in all modules to preserve the is_wf predicate.

23.2.2 *[gen_safe_pres]*

$$safe(s) \wedge gen_guard(..., s) \Rightarrow safe(gen(..., s))$$

where s is some state in the *Dynamics* module and *gen* is a generator in this module.

This requires all generators in the *Dynamics* module to preserve the $safe$ invariant.

23.2.3 *[gen_consistent_pres]*

$$consistent(s) \wedge gen_guard(..., s) \Rightarrow consistent(gen(..., s))$$

where s is some state in the *Control* module and *gen* is a generator in this module.

This requires all generators in the *Control* module to preserve the *consistent* invariant.

23.2.4 *[init_is_safe]*

$$init_req(s) \Rightarrow safe(s)$$

where s is some state in the *Dynamics* module.

This requires an initial state to be safe.

23.2.5 *[init_is_consistent]*

$$initReq(s) \Rightarrow consistent(s)$$

where *s* is some state in the *Control* module.

This requires an initial state to be consistent.

23.3 Proof: *[init_is_safe]*

This section sketches an informal proof that an initial state is safe. When showing mathematical formulas some RSL parameters are left out because they would complicate the expressions (no need to include *S.Configuration* in all predicates)

To begin with both predicates are listed:

```
safe : State × S.Configuration → Bool
safe(s,con) ≡
    is_wf(s,con) ∧
    noCollisions(con,s) ∧
    trainPosPossible(con,s) ∧
    pointsSafe(con,s) ∧
    crossingsSafe(con,s),
```

```
init_req : State × S.Configuration ~→ Bool
init_req(s,con) ≡
    is_wf(s,con) ∧
    allTrainsInESA(s) ∧
    allTrainsFacingLine(s) ∧
    allTrainsStopped(s) ∧
    allBarriersUp(con,s) ∧
    allPointsNotShifting(con,s),
```

Now each sub predicate of $safe$ are inspected:

- is_wf this follows directly from the definition of *init_req*.
- $noCollisions$
  1. is true if $\forall$ t_1,t_2 : T.TrainID • *commonSegs*(t_1,t_2) = {}.
  2. *commonSegs* returns an empty set if trains are in ESA.
  3. by knowing that $allTrainsInESA$ restricts trains to be in ESA, then $noCollisions$ must be true.
- *trainPosPossible* is also true when $\forall$ t_1,t_2 : T.TrainID • *commonSegs*(t_1,t_2) = {}, so following above result this also holds.
- $pointsSafe$
  1. is true when $\forall$ t : T.TrainID • ~ *trainOnJunction*(t).
  2. *trainOnJunction* is false when the specific train is not at a junction (point).
  3. again following $allTrainsInESA$ we know that no train is on a junction.

- *crossingsSafe*
  1. is true when ∀ t : T.TrainID • ∼ *trainOnSensor*(*t*).
  2. *trainOnSensor* is false when a train is not at two different segments at once.
  3. following the predicate *allTrainsInESA* we know that all trains are in the ESAs and therefore cannot be at two segments.

Hereby we have proved informally that the requirements to an initial state also satisfies the *safe* predicate.

Chapter 24

Ideas & future work

This chapter describes ideas to how the model and simulator could be extended or changed with different functionality.

24.1 Improved exceptions

A good idea would be to improve the exceptions used in the simulator. Exceptions are thrown whenever a precondition in the dynamic physical system (*Dynamics*) is violated. The messages in these exception could be more descriptive and more accurate. The structure and handling of the exceptions could also be improved.

24.2 Pipelining of trains

The current model is developed to avoid deadlock. Therefore only one train is allowed to be at a single line and its coherent branch segment (if any) at the time.

If we do not wish to avoid deadlock the algorithm could be changed to allow pipelining of trains, i.e. more than one train could be located at a single line if they drive in the same direction. This would enable more traffic in the railway line but still, the trains have to be able to pass each other which they only can at the branches of a junction. This means that it might not a great advantage to allow pipelining as long the layout of the railway line is as suggested in this project.

24.3 Complex networks

The railway line of this project is very simple. Basically it is a connection between two ends that some times branches into two tracks but immediately after gathers again.

This simple railway line could be extended so that it did not have to gather again when

it branches into two tracks. In this way complex railway networks could be created. An example of this is shown in figure 24.1

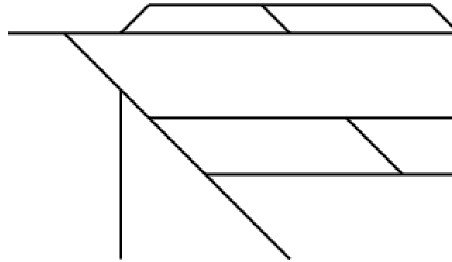

Figure 24.1: A complex railway network

In such a complex network it would a bit more difficult to guarantee that deadlock does not occur. Route plans might play a greater role in that kind of railway networks.

24.4 Automatic reservation- and brake points

In the model the reservation- and brake points are static values common for all segments and ESAs. Instead of being a part of the configuration these values could be calculated by the TCC on the fly dependent on the current speed of the train etc.

This would make the calculation of whether the train has passed the reservation- and brake point at bit longer and thereby a bit more time consuming but this might not be a problem. The advantage would be that the reservation- and brake point would be more accurate.

24.5 Speed reduction before entering segment

Each segment has a max allowed speed that no train is not allowed to exceed. This value can be different for two neighboring segments. The TCC of a train repeatedly checks the speed of the train and makes sure that the speed does not exceed the max allowed speed (both for the train and for the segment).

If the two segments *seg1* and *seg2* have the max allowed train speeds of *40 m/s* and *30 m/s* respectively and a train drives from seg1 into seg2 with 39 m/s, then the train drives to fast when it reaches seg2. This is because the TCC do not brake the train before the front of the train has entered seg2 where the TCC finds out that the train drives to fast.

An solution to this could be that when a train has passed the brake point its TCC should also check if it drives faster than the max allowed for the next segment. If so it should brake the train until it does not drive too fast.

24.6 Time tables and stations

A extension to the model could be to include time tables and stations. The time tables should specify the expected time of departure of trains from the stations including the end station areas. The position of stations should then be specified. Some kind of notification in the simulator of when the time of departure has been exceeded could then be added.

24.7 Automatic train behavior

Another extension to the model could be to give the trains some other kind of automatic behavior. Currently the *Autodrive behavior* behavior is used in the simulator but with the possibility to turn it off.

The Autodrive behavior is explained in section 21.2.4. The next section describes the time table based behavior.

24.7.1 Time table based behavior

If time tables and stations are introduced the TCCs could be changed so they obey these time tables by accelerating the trains at the time of departure when it is possible. This would be a bit more complex to implement but it would be a nice feature.

24.8 Ideas for concurrency

Following in the wake of a distributed control system one would say that concurrency would naturally come into consideration. In this implementation we have not found it necessary.

But even though we did not implement a concurrent RSL model we investigated how this could be done, and how this would fit an implementation in JAVA.

Because this is not directly relevant for our implementation, the theory of concurrency in RSL and JAVA has been moved to appendix E.

Chapter 25

Related work

25.1 Automatic translation from RSL to JAVA

At fall 2004 Ulrik Hjarnaa did a thesis [8] on automatic translation from a subset of RSL to JAVA. Only a subset of RSL can be translated to JAVA since RSL allows datatypes that are sorts (unspecified) and implicit specifications and JAVA does not allow either of these types.

The requirements to the translation were that the translated JAVA code must be semantically equivalent to the RSL specification and it should be possible to recognize the RSL specification in the translated JAVA code.

The main idea of having an automatic translation from RSL to JAVA is that this translation is done systematically and fast. If the translator is sound it is guaranteed that the translated JAVA code is correct. If you should do the translation manually human factors might influence the result and the manual translation of two persons would possibly differ. The automatic translation might save you some time.

The disadvantage of automatic translation is that, since only a subset of RSL can be translated, you need to make sure that the RSL specification does not contain RSL parts that are not part of this subset. If it does, you need to change the syntax of these parts to something that is translatable but that has the same semantics.

Included in the `rsltc` program is the functionality to translate RSL to `C++` and `SML`. The translator from RSL to JAVA is a nice extension of this functionality.

25.2 Formal Development and Verification of a Distributed Railway Control System

Another project dealing with modelling a distributed railway and its control system was carried out by Anne E. Haxthausen and Jan Peleska i year 2000 [5].

The focus in that project was on both modelling/development and verification of a distributed control system. Therefore the events were made discrete without dealing with time aspects at all, because this would complicate matters greatly. Therefore only the discrete events which represent changes in the control system state were considered. When dealing with spacing of trains this is really the only events which is needed to be considered.

In this project we have had another focus. The control system is still distributed but time plays a large role in both the physical system and the control system. The RSL model uses time to continuously update the position and speed of the trains. In this way there are many more events and the state space is much bigger than in [5]. Therefore it is much more difficult to verify the safety formally.

The algorithms used in these two projects has some similarities but they differ in particular in two ways. First of all, the algorithm in this project prevents deadlock from occurring, which has huge impact on the algorithm. This feature is not included in [5].

Secondly, in this project most of the complexity are placed in the SBs in stead of in the TCCs. The TCCs ask for a reservation and brakes the train if it has not obtained a reservation in time (plus speed checking). But all decisions whether the train/TCC may get a reservation is placed in the SBs. Since the control system prevents deadlock, the SBs have to communicate with each other. This also increases the complexity of the SBs.

In [5], the TCCs plays a greater role. Anne and Jan chose a design where the entire state of the SB is sent to the TCC. It is then up to the TCC to decide whether is is safe to go forward.

Our argumentation for choosing the other strategy is that SBs are statically positioned in a network an will never move. Therefore it would seem beneficial to let these entities handle all location specific decisions. Especially when trains of different designs were to utilize this control system. Then the amount of equipment would be small and minimal in complexity which would seem beneficial in such cases.

The discussed paper served as the starting point and inspiration for this project.

25.3 Domain Specific Languages

This section gives a short description of the thesis [7]. A brief discussion of similarities in this project is given.

The thesis concerns construction a domain specific language for tramway control systems. A domain specific language is a language used to construct certain structures in a specific domain.

What is special about such a language is that it is constructed only for a specific domain and therefore uses domain specific terms in the language constructs. This enables domain experts as well as computer modelling specialists to read and understand the structures expressed in this language.

This language is implemented in XML and used as input for a generic control system

to generate a control system specific to the structure expressed by the XML construct. In this way one can easily construct new control systems without having to validate for safety every time. It is only relevant to validate the generic control system.

25.3.1 Domain Specific language

In this project we also construct a kind of domain specific language. This language is also implemented in XML (in the way that a DTD or syntax definition is created) which can express railway configuration within the boundaries of the specified system wellformedness conditions.

In some way the control system constructed in this thesis can also be considered a generic control system in relation to all possible configurations expressible in the developed configuration language.

25.3.2 Verifying safety

This thesis uses the same concept of provability of safety as in [7]. We also in this thesis present the concept of the predicates $safe()$ and $consistent()$ and how the method of proving these should work.

25.4 Modelling interlocking systems

In this thesis a basic interlocking system is modelled in RSL and then translated to a Simulator written in the JAVA language.

The model and simulator models train movement as discrete steps in and out of entire segments of track.

Much of the system safety is formally verified. The rest is verified informally in a structured way.

25.4.1 Train dynamics

In our thesis the train dynamics are modelled much more realistic which also would complicate the matters of proving safety. In the discussed thesis the train movements are abstract steps in and out of the safety segments. Therefore no speed considerations or braking requirements were mentioned.

25.4.2 Verifying safety

The verification method in [6] is basically the same as in this thesis. One difference though is considerable. In the discussed thesis it was only necessary to verify that $consistent(s) \wedge gen_guard() \Rightarrow consistent(gen(..., s))$ because all state changes (events) were initiated by pressing buttons. Guards were implemented in the button handling code so if a guard condition was not satisfied, then no action was performed.

Thereby it was enough to prove that a satisfied guard implied a consistent state. It was known that a function would never be executed without the guard being satisfied.

But further validation is needed to ensure safety in this system (see section 23.1.6).

Chapter 26

Discussion

26.1 Predicates and preconditions

In the model - and therefore also in the simulator - some predicates and preconditions were defined. This section describes how these predicates and preconditions have eased the process of writing software which is not formally verified.

26.1.1 Predicates

The three main modules in the model: Statics, Dynamics and Control all contain a predicate that checks if the module is wellformed.

For Statics to be wellformed the configuration must be wellformed. Checking this before the simulation of the railway line is started is very useful, since a lot of errors may occur at run time, if the configuration is not wellformed. A lot of time might be spent on searching for errors in the code even though the error is in the configuration that has been loaded. Rejecting configurations that are not wellformed saves us from this work.

when using the RAISE development method such predicates are normally specified on a very early state of development. This forces the developers to agree on what defines a wellformed system, and how it should work, at the very beginning of the development phase. If this discussion were to take place deep into the development phase, discovery of error could possibly prove fatal for the project in form of re-designing and development costs.

26.1.2 Preconditions

Many of the generators in the model have a precondition that is used to make sure that the system state stays wellformed, safe and consistent. In the JAVA code these preconditions are checked in the beginning of each function call. If they are not satisfied

an exception is thrown.

None of the preconditions should be violated when the control system is turned on. They are, however, stated anyway. If the control system is turned off, a precondition might be violated, e.g. if a train passes a crossing that is not closed. In that kind of situations we get response from the simulator telling what have happened. This is very useful, since we know when the state has become unsafe.

Some functions have preconditions specifying what proper input is. If these preconditions are violated, the program code might contain an error. I.e. the implementation of the simulator is not correct. In these situations the preconditions help us finding these errors. It proved very helpful when debugging the system. It greatly increases the trust in the software, if the system is run without any preconditions being violated.

26.2 A safe algorithm

This section explains why we believe the algorithm of the control system keeps the railway safe. This does not include any formal verification or proof but is presented informally. The algorithm is explained in chapter 11.

In section 5.4 *safety* is defined and explained. It also describes the situations to avoid to keep the railway line safe. The next sections go through these situations and explain why they do not occur when the control system is turned on. If the control system is turned off, safety is not guaranteed.

26.2.1 Two trains collide

Before a train is allowed to leave an ESA or a branch segment it must have a line-branch reservation to the next ESA or branch segment (always the right branch). When one train has reserved a single line and the coherent branch segment (if any) no other train will get a such reservation. Trains that try to leave a segment or an ESA without a reservation is braked by the train's onboard computer (TCC) when the train passes the brake point. The brake point must be large enough so that any train can brake from max speed to zero before it enters the next segment. Therefore two trains will never be at the same segment at the same time.

Actually two trains will never be at the same single line at the same time. If they drive in the same direction two trains will not get a reservation for a single line if another train is located at the coherent branch segment (following the single line). This assures that deadlock does not occur.

More than one train is allowed to be in an ESA in which it is assumed that collisions do not take place. Since collision at a segment will not occur, collisions between two trains cannot take place.

26.2.2 Collisions at a crossing

When a train is to enter any new segment, it must have a segment reservation (besides the line-branch reservation) which is given when the segment is prepared. If the SB to

pass to enter the new segment is a crossing SB, preparing the segment means to close the crossing. The reservation is not given before the crossing is closed. If the SB fails to prepare the segment - e.g. if the barriers do not work properly - a reservation is not given. A state chart describing the different states of a crossing can be found in section 9.4.

If the train does not have a reservation to enter the next segment the TCC brakes the train in a safe distance. Therefore the barriers will be down when the train passes through the crossing. A road vehicle and a train will therefore only be able to collide at a crossing if the road vehicle violates the traffic rules by driving through the barriers (or around them if they do not cover the whole road). The control system cannot control this kind of situations and it is assumed that they will not happen.

In the simulator it is possible to open and close a crossing manually - also after the segment is prepared. Using this possibility might violate the safety of the railway line. Therefore this possibility must not be used when the control system is on, if the railway line must be safe.

Assuming that manually opening of a crossing does not occur, collisions at a crossing cannot take place.

26.2.3 Derailing at a junction

As described in section 26.2.2 a segment must be prepared before a segment reservation is given. If the SB to pass to enter the new segment is a point SB, preparing the segment means to switch the point to be in the correct position (always use the right branch). If the train comes from the stem, the point is switched to the right. If the train comes from a branch, the point is switched to this branch.

In the simulator it is possible to switch a point manually - also after the segment is prepared. Using this possibility might violate the safety of the railway line. Therefore this possibility must not be used when the control system is on, if the railway line must be safe.

Assuming that manually switching of a point does not occur, no train that enters a junction will be derailed.

26.2.4 External events

At least two external events can occur that violates the safety of the control system:

- A person or an animal walks on the railway line and is hit by a train.
- Any physical defects in the railway system like cracks in the railway line and broken electrical wires that causes the system to fail .

These events cannot be avoided by the control system. Therefore it is assumed that they do not occur, so that the safety of the railway line is not violated.

Chapter 27

Conclusion

In this chapter we summarize the results of this project and evaluate each part of the project and associated development phases.

27.1 Summary of results

This section lists the results of this project.

27.1.1 RSL model

A RSL model of a railway line with a distributed control system was developed. The model was developed using the RAISE method [3] to refine the model from an abstract applicative to a concrete applicative model and at last to an imperative model (method summarized in appendix B).

The model was constructed in 3 main modules: *Statics*, *Dynamics* and *Control*. In the order mentioned each module were based on the previous module and added new functionality.

Each main module was decomposed into several sub module which each concerned some specific area of the main module.

Each module was constructed with provability of wellformedness, safety and consistency in mind. Therefore predicates expressing these properties were constructed and axioms stating how these were to be applied, were also added.

27.1.2 Control system / algorithm

A control algorithm was developed with the purpose of preventing train accidents and deadlocks. The control algorithm was kept as simple as possible by making the messages sent between the control entities minimal.

The control system was designed so that only the most necessary processing was done in the train control computer (TCC). Therefore only a YES or NO was sent from a switch box (SB) in response to a TCC reservation request. The reason for this was, that if different kind of trains where to utilize this control system, then the complexity of the equipment to be installed on each train should be minimal. Almost all complexity and control decisions are handled in the SBs.

27.1.3 XML configuration language DTD

The data type of a *Configuration* was transformed into a XML DTD (syntax definition file). This DTD can express all configurations allowed in the considered domain.

Though some wellformedness requirements are directly implemented in the DTD, many of the wellformedness invariants are enforced by the JAVA configuration editor. The editor is able to do this through the predicates from the RSL model which are implemented in JAVA functions.

This means that a XML file can express a configuration which is not wellformed even if obeying the rules of the DTD.

27.1.4 JAVA train simulator

The RSL model was translated into JAVA by using a method also developed in this thesis (section 20.1). All predicates where translated to JAVA functions and the essence of the *initial* axioms were also implemented.

A graphical user interface (GUI) was constructed to visualize the state of the model. The functionality to manipulate trains, points and crossings where also implemented.

The simulator was constructed so it generically could take a configuration XML file as input and simulate a control system on the given configuration.

The top level framework to *tick* the model (see section 5.3), which was not specified in the RSL model, was also developed.

The design of the GUI was specified and prototyped as diagrams before the actual implementation took place. The GUI requirements was based on the functionality of the model.

27.1.5 JAVA configuration editor

A JAVA configuration editor was constructed as a tool to help constructing wellformed XML configuration files which can be used as input to the simulator.

All wellformedness predicates from the *Statics* module was implemented so that the user gets a warning if the edited configuration violates the invariants.

27.2 Evaluation of results

In this section various aspects of the project are evaluated and commented.

27.2.1 Design method

In this thesis it was decided to have a preliminary design phase before the modelling phase. Some might wonder about this decision because the modelling phase is often where design decisions are made.

The argument for having such a design phase before modelling is that the decisions made in the design phase were on a very abstract and algorithmic level.

Therefore all decisions made in the design phases were (mostly) WHAT should be modelled and not HOW it should be modelled. The concrete data structures and function structure were left to the modelling phase.

One of the main products of the design phase was the control algorithm (chapter 11). We found it necessary to have the basic concepts of the control system and algorithm in place before modelling, because the choice of control system functionality would probably have great impact on initial model structure.

27.2.2 Train dynamics analysis

Many calculations in basic mechanical physics were performed to create invariants which should ensure safety in the system. It was surprising to see how many requirements that were necessary to ensure safety.

All of these calculation were performed without any form of safety margin. In a real system where the calculations are much more complex, it would be wise to consider some kind of margins for error. So in the invariant concerning brake point (bp) distance the inequality would look like:

$$bp > s_{brk} + safety$$

where s_{brk} is the braking distance for a train in max speed and $safety$ is a safety distance that the train should never cross when beginning to brake at the brake point.

27.2.3 Model

The development of the model progressed without too many problems. A challenge was to create a structure which was finely balanced between redundancy and calculation complexity. By this is meant that there are two approaches when developing such a model:

1. The model could be **optimized mathematically** for **provability** and **simplicity in the data structure**. Normally the data structure can be kept minimal where all other properties can be derived by some calculations. The down side of this is that the processing of operations in the model get very heavy

if it is implemented in a programming language. Every time some property is needed, it is first calculated before it can be used.

2. The model could also be **optimized for performance** for implementation in a programming language. This means that **redundancy** and pre-calculated data often are great advantages. The down side of this is that the mathematical specification of the model gets much harder because much redundancy increases the amount of wellformedness predicates needed to keep the data structure consistent.

27.2.4 Verification

Safety was not verified formally in this thesis but the model has been constructed with a formal proof in mind. The theory and method of a such proof is described in chapter 23.

The mentioned chapter does not take into account that timing is an issue when verifying the system, but of course this is of great concern. The possibility of the proof getting too complex when including timing considerations is to be considered. When carrying out such a proof one should abstract from the time considerations.

Instead it is encouraged that an abstract model is developed where only the changes in control state are considered (i.e. trains entering and leaving segments, and reservations added and removed.). It should be shown that a negative reservation sent to a train implies that the train stops before it exceeds the segment border. Then this can be left out of the proof.

Though the model structure is prepared for such a proof, many preconditions and changes to predicates are probably necessary as pointed out in section 23.1.3. Since a proof has not been performed, we have no way of knowing whether the current set of preconditions and predicates are correct and sufficient to carry out such a proof.

Many of the proof obligation in section 23.2 are not stated in the model and therefore a great deal of axioms and preconditions are still to be defined in the model.

Before an actual proof of safety can be performed, it should also be shown / proved that generators preserve consistency (wellformedness). Else the system would not make sense.

27.2.5 Modelling method

After trying different approaches to the model it was decided that the model should resemble the capabilities of a JAVA program as closely as possible, even in the initial abstract model.

Therefore it was decided not to use the approach utilizing wellformed sub types as suggested in some examples in [3].

Instead we added a value to our model modules which resembled the initial state of the system and added an axiom that the initial state should be wellformed, safe and consistent. By proving that the generators preserved these properties, the system should stay that way.

27.2.6 JAVA translation method

Before translating the model to JAVA a method for translation was agreed on. The translation method in section 20.1 was inspired by the structure in [3], chapter 5, where each RSL expression is systematically considered one by one.

Many alternative translation techniques are suggested in some cases, and arguments for the chosen technique are presented.

Chapter 28

Tools used in this project

This chapter describes the tools that have been used in this project. This includes tools for writing the report, developing the model, developing the simulator etc.

TeXnicCenter / Latex This report has been written by using Latex in TeXnicCenter, which is a dedicated Latex editor with functionality to handle projects, shortcuts to Latex commandoes, etc.

CVS[1] has been used to be able to work concurrently with version control.

UltraEdit is a plain editor that has been used to develop the model and to view *.txt* files.

DIA is a diagram creation program (UML diagrams etc.). It has been used to draw different diagrams, the railway line layout, the simulator layout etc.

XDE is a professional tool for drawing UML diagrams that is based on the *Eclipse* tool. It has been used to draw state charts when making the algorithms of the control system and to generate class diagrams from the JAVA code.

Eclipse is a programming IDE that has been used for implementing the simulator in JAVA.

JAVA is the language the simulator is implemented in. The simulator is implemented and tested in both version 1.4.2 and 1.5.0

Adope Photoshop has been used to draw and manipulate images.

Chapter 29

Bibliography

Bibliography

[1] Morten S. Madsen, Martin M. Bæk, "*Modelling a Distributed Control System for a Railway Network*", special project, Institute of mathematical modelling, Technical University of Denmark, Lyngby, fall 2004.

[2] The RAISE Language Group, "*The RAISE Specification Language*", The BCS Practitioners Series, Prentice Hall, 1992.

[3] The RAISE Method Group, "The RAISE Development Method", The BCS Practitioners Series, Prentice Hall, 1995.

[4] Joern Pachl, "*Railway Operation and Control*", Technical University Braunschweig, VTD Rail Publishing, USA, 2002.

[5] Anne E. Haxthausen, Jan Peleska, "*Formal Development and Verification of aDistributed Railway Control System*", IEEE Transactions On Software Engineering Vol. 26 No. 8, August 2000.

[6] Torben Gjaldbæk, "*Modelling Interlocking Systems for Railway Lines*", Master Thesis, Institute of mathematical modelling, Technical University of Denmark, Lyngby, 2002.

[7] Rasmus Dyhrberg, Nikolaj Christensen, "*A Domain Specific Language for Tramway Control Systems*", Master Thesis, Institute of mathematical modelling, Technical University of Denmark, Lyngby, 2004.

[8] Ulrik Hjarnaa. "*Translation of a Subset of RSL into Java*". Master of Science Thesis, DTU, 2004.

[9] Simon Bennett, Steve McRobb, Ray Farmer, "*Object-Oriented System Analysis And Design, Using UML, Second Edition*", McGraw-Hill Education, 2002.

[10] Gregory R. Andrews, "*Foundations of Multithreaded, Parallel, and Distributed Programming*", Addison Wesley Longmann, Inc., USA, 2002.

[11] Banestyrelsen, "*Oplæg om Jernbanesikkerhed*", September 2000.

[12] ETCS Web-site, http://etcs.uic.asso.fr/index.html.

Appendix A

Design of GUI

A.1 TrainSimulator

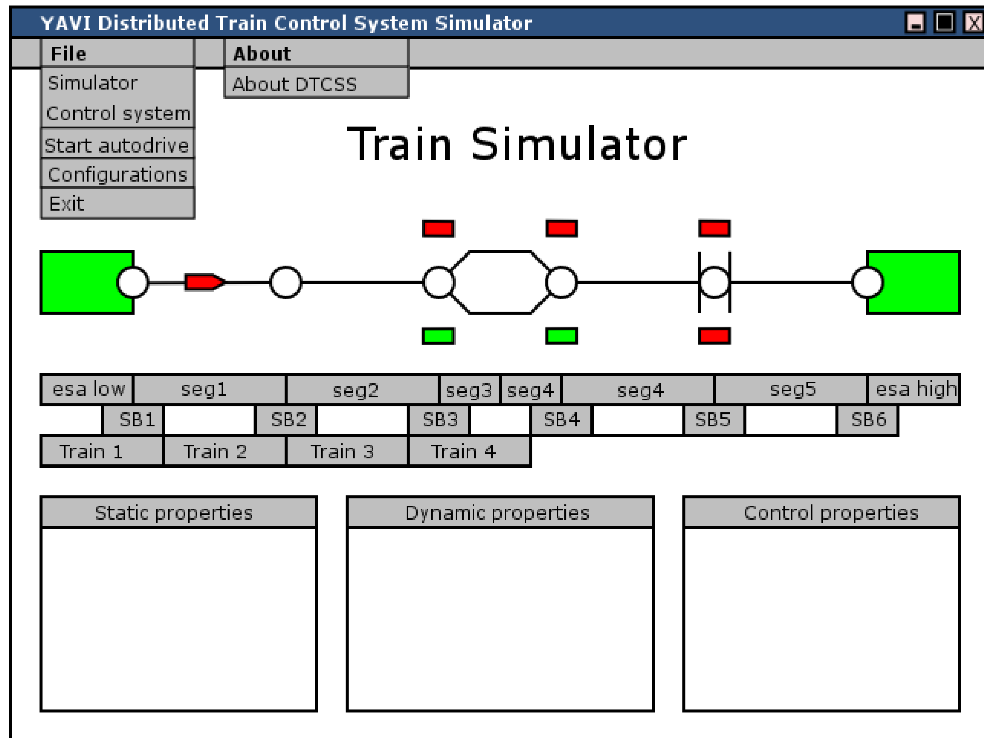

Figure A.1:

A.2 Configuration builder

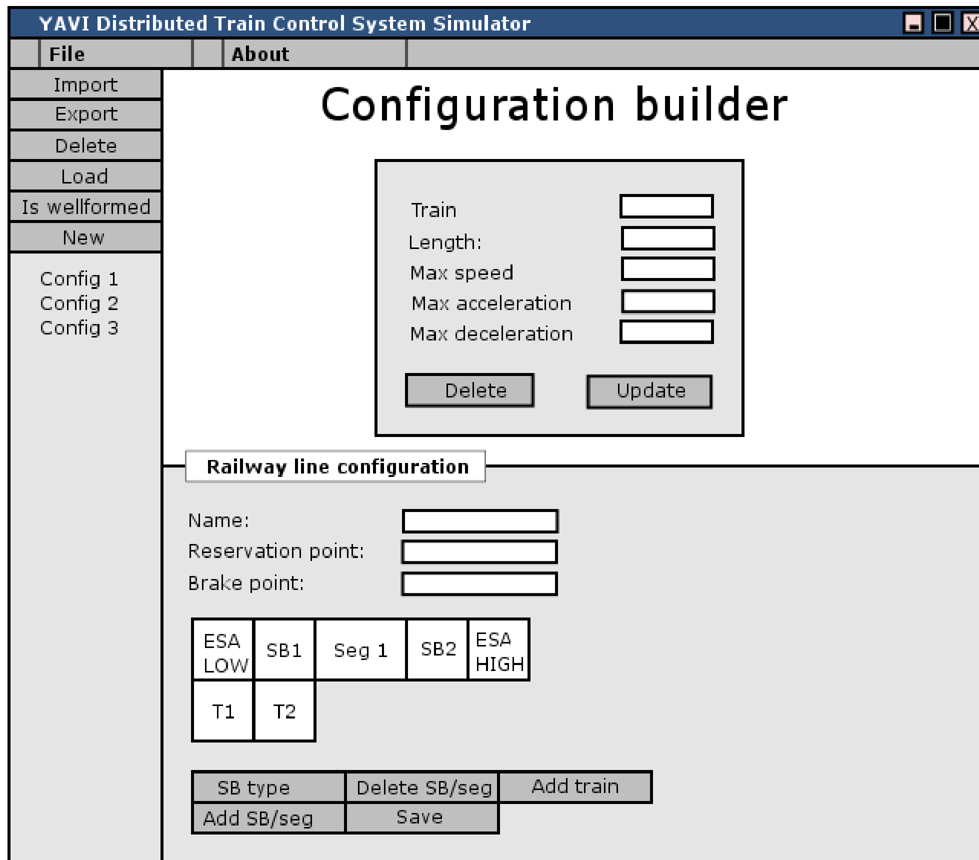

Figure A.2:

Appendix B

RSL method description

This section describes the method used and steps taken developing the RSL model. The description of the method is not specific in references to the actual model but describe the development method in a more general manner. To read this section the reader should know of formal development and RSL to some degree.

While developing this model we have tried to follow the *RAISE Development Method* from [3] as closely as possible.

The following development steps were carried out:

B.1 Abstract applicative

The initial modelling steps are carried out using the abstract applicative technique. It gives a better overview of the entire model when abstracting from all the detailed data handling of a normal program.

Introduce types:

- The main type of interest i added as a sort[1]. This type is usually some global state of the considered system or module scope.
- Several other types can be introduced which often represents some entities or entity related information in the system. These smaller types can be declared as sorts but can also be respresented by a concrete data type such as **Text** or **Int**.

type
  T,
  T_1,
  T_2,
  T_3 = **Text**

[1] An abstract type declaration in RSL without any concrete data structure or type.

Introduce functions:

- Functions which represent the functionality (the possible changes to the system) are added. As the bodies of these functions are created the need for lower level functions which directly handles the main type of interest is discovered.
- Basic functions which extracts (observes) information from the main data type are called *observers*.
- Basic functions which generates values of the main type and thereby changes the system state are called *generators*.
- At this abstract level it is not yet possible to define bodies for the basic observer and generator functions. Therefore these are left as signatures.
- Classify the functions as observers, generators and derived and sort them appropriately.

```
value
  obs₁ : T × .. → T₁ × ..,
  obs₂ : T × .. → T₂ × ..,

  gen₁ : T × T₁ × .. → T,
  gen₂ : T × T₂ × .. → T
```

Signature formulation:

- Decide whether functions are total or partial.
- If partial decide on the necessary preconditions for the generators.
- Partial functions can be made total by changing the return type to indicate success or failure when executing the function. In this way the function can return *failure* for all un-wished input.

Formulate observational axioms:

- Formulate observational axioms defining the relationship between all observer-generator pairs in each module. If the generator has any precondition this should be specified here. These axioms have the following form:

```
axiom
  [obs_gen]
  ∀ t : T, a₁ : T₁, .. , aₙ : Tₙ •
      obsᵢ(genⱼ(a₁,..,aₙ,t)) ≡ val_exp
  pre genⱼ_guard(a₁,..,aₙ,t)
```

where T is the type of interest, a_i are some arguments for the generator and *val_exp* is some value expression. *gen_j_guard* is the generator guard predicate which must be true to be applied.

Add invariants: invariants are formulated as functions and a *is_wf*() function which is a conjunction of all invariants is added:

```
value
  inv₁ : T → Bool,
  ..
  invₙ : T → Bool,
```

```
is_wf_T : T → Bool
is_wf_T(t) ≡
    inv1(t) ∧ .. ∧ invn(t)
```

- An axiom for each generator is added stating that the generator preserves the invariant:

```
axiom
  [genj_wf_preserve]
  ∀ t : T •
      genj_guard(t,...) ∧ is_wf(t) ⇒ is_wf(genj(t,...))
```

Add constant or init value:

- If appropriate, add a constant or initial value of the type of interest. This should be used as constant or initial value for the type of interest variable in the imperative version.
- Add axiom that states that the value fulfills the invariants and maybe some other requirements specific to the initial value:

```
value
  init_T : T,

  init_reqs : T → Bool

axiom
  [init_wf]
      is_wf(init_T) ∧ init_reqs(init_T)
```

Hide internal properties: Hide all properties (functions etc.) in the modules which are not for external use.

B.2 Type decomposition (optional)

- If the type of interest has many observers one can consider the possibility to decompose this type into several types.
- If a collection of observers seemingly has a common area of interest, a new type can be formed, and the observers / generators are changed to use this type instead.
- The type of interest can then be formulated as a product of all the new minor types. This makes the major type partially concrete by the fact that it is now a cartesian product but only of types that are abstract themselves.
- The minor types can be moved to seperate modules if appropriate.
- This procedure can be repeated if the minor types again contain too much information (have too many basic observers and generators) - The method book defines this as a type having 2 or more observers.

B.3 Concrete applicative

During this step the model is made concrete applicative. All data types are made concrete and basic functions made explicit. Following steps are performed:

Concrete data types:

- Concrete data structures are chosen for all currently abstract data types.
- It should be preferred to design the data types such that some of the invariants (wellformedness requirements) of the system are enforced directly by the data type and therefore can be removed. This contributes to the simplicity of the system.

Explicit functions: Basic observers / generators are made explicit to utilize the concrete data types.

Remove observational axioms:

- All observational axioms are removed, and it is verified that these are fulfilled by the explicit observers / generators.
- Any preconditions on these axioms are added as preconditions for the definition of the appropriate generator.

Remove generator invariant axioms: The axioms concerning generators preserving the invariant are removed because the function bodies are now concrete. It is verified that the axioms are fulfilled by the new concrete definition.

B.4 Concrete imperative

The model is made imperative so that functions utilizing the type of interest now operates with the internal variables instead of parameters and writes to these instead of returning main type as result:

Introduce variables:

- Variables for the types of interest are introduced.
- If the type of interest is a product of several types then variables for these minor types are introduced (in their seperate modules).
- If the minor types exist in seperate modules then objects for each module is introduced in the utilizing module.

Transform signatures:

- All functions with type of interest as parameter replaces this with **Unit** and **read any** in its signature.
- If a type of interest occur in the result type, it is replaced by a **Unit** and a **write any** (or the specific variable name(s) instead of **any**).

Transform function bodies: the return value expression in the bodies of the generators are replaced by an assignment to the appropriate variable. All references to the type of interest formal parameter is replaced with the variable name.

Introduce loops: Recursive function definitions are replaced by loops which is the imperative counterpart.

Initial value: The initial (constant) value is used to initialize the variable of the same type:

type
  T

value
  init_T : T

variable
  T_var : T := init_T

Appendix C

XML DTD

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT Configuration (SBs,Segs,ESAs,Trains)>
<!ATTLIST Configuration name CDATA #REQUIRED>

<!ELEMENT SBs (SBData,SBData+)>

<!ELEMENT SBData (SBSegment,SBSegment)>
<!ATTLIST SBData SBID CDATA #REQUIRED>
<!ATTLIST SBData sbType (POINTSB | ENDSB | CROSSINGSB | PLAINSB) #REQUIRED>
<!ATTLIST SBData pointTicks CDATA #IMPLIED>
<!ATTLIST SBData barrierTicks CDATA #IMPLIED>
<!ATTLIST SBData signalTicks CDATA #IMPLIED>

<!ELEMENT SBSegment (Seg | Point | ESA)>
<!ATTLIST SBSegment dir (UP | DOWN) #REQUIRED>

<!ELEMENT Seg EMPTY>
<!ATTLIST Seg seg CDATA #REQUIRED>

<!ELEMENT Point EMPTY>
<!ATTLIST Point upSeg CDATA #REQUIRED>
<!ATTLIST Point downSeg CDATA #REQUIRED>

<!ELEMENT ESA EMPTY>
<!ATTLIST ESA esa (HIGH | LOW) #REQUIRED>

<!ELEMENT Segs (SegData+)>
<!ATTLIST Segs resPoint CDATA #REQUIRED>
<!ATTLIST Segs brakePoint CDATA #REQUIRED>

<!ELEMENT SegData EMPTY>
```

```
<!ATTLIST SegData SegmentID CDATA #REQUIRED>
<!ATTLIST SegData upSB CDATA #REQUIRED>
<!ATTLIST SegData downSB CDATA #REQUIRED>
<!ATTLIST SegData length CDATA #REQUIRED>
<!ATTLIST SegData maxSpeed CDATA #REQUIRED>


<!ELEMENT ESAs EMPTY>
<!ATTLIST ESAs lowSB CDATA #REQUIRED>
<!ATTLIST ESAs highSB CDATA #REQUIRED>
<!ATTLIST ESAs lowLength CDATA #REQUIRED>
<!ATTLIST ESAs highLength CDATA #REQUIRED>


<!ELEMENT Trains (TrainData*)>

<!ELEMENT TrainData EMPTY>
<!ATTLIST TrainData TrainID CDATA #REQUIRED>
<!ATTLIST TrainData length CDATA #REQUIRED>
<!ATTLIST TrainData maxSpeed CDATA #REQUIRED>
<!ATTLIST TrainData maxAcc CDATA #REQUIRED>
<!ATTLIST TrainData maxDecel CDATA #REQUIRED>
```

Appendix D

Test images

This section lists the images illustrating the test scenarios of the *Dynamics* module.

D.1 Collisions

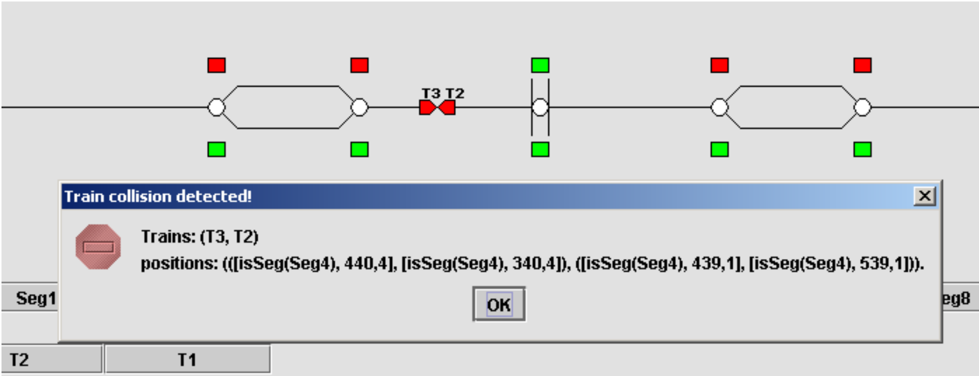

Figure D.1: A frontal collision

D.2 Derailings

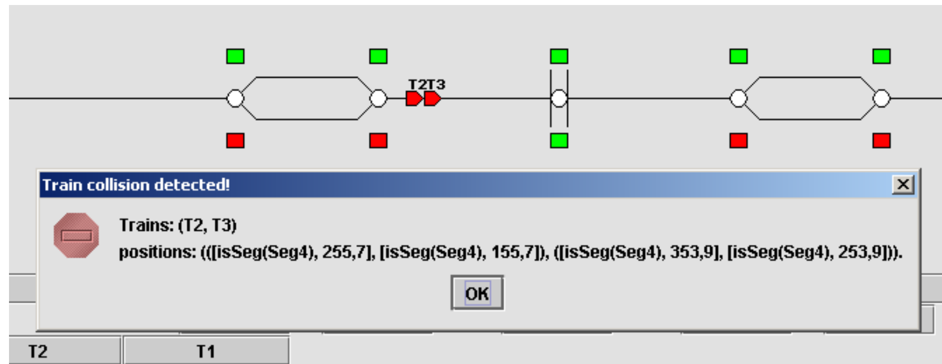

Figure D.2: A rear collision driving UP

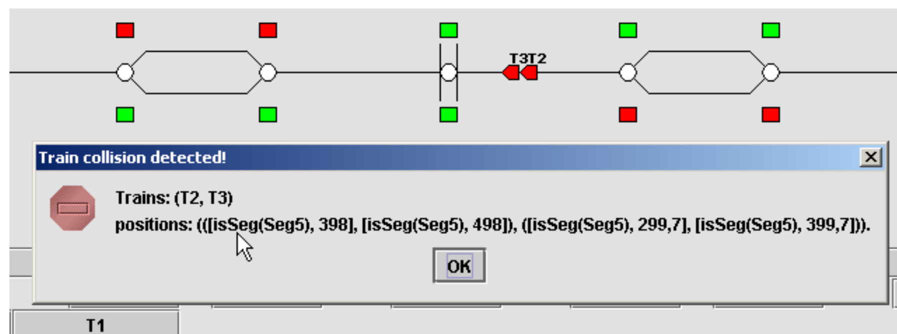

Figure D.3: A rear collision driving DOWN

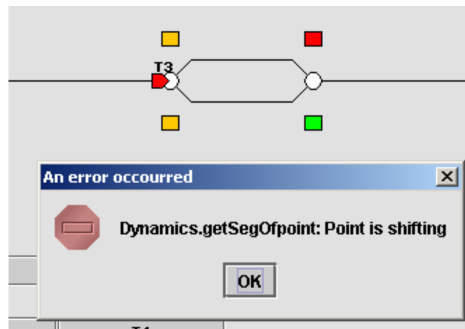

Figure D.4: A derailing caused by shifting point

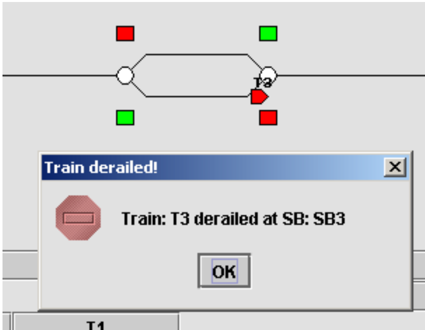

Figure D.5: A derailing caused by wrong point position

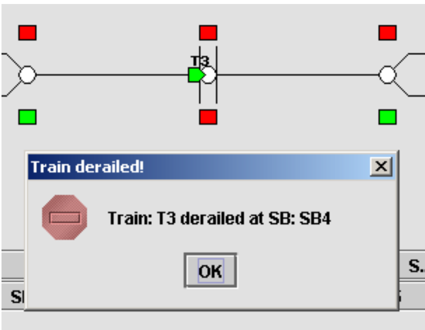

Figure D.6: A train crashing at a open crossing

Appendix E

Concurrency

This section covers the ideas for using concurrency in the RSL model and to translate this into JAVA.

E.1 Concurrency in RSL

Concurrency in RSL are obtained by using channels. Sending values over a channel is another way besides function calls to pass a value to another function (process). Processes in RSL are functions which sends and receives values over channels.

Concurrency is used in RSL to obtain two main purposes:

1. Synchronization between processes and (mainly) controlling access to shared variables.
2. Detaching function processing from the caller of the function. This enables objects to call each other to execute some process, and then later fetch the result either but listening to a channel or by function call.

In this section both aspects are covered.

E.2 Concurrency in JAVA

Concurrency in JAVA has the same two main goals as mentioned above. In JAVA though, the concept of *synchronized* functions (explained below) ensures exclusive access to functions and variables.

JAVA does not have channels as RSL does, and therefore processes takes another form. In JAVA it is possible to spawn a thread which perform the calculation of a function call in an object to detach the caller from the actual calculation.

E.3 Shared variables in RSL

When the imperative concept with variables is introduced the issue of shared variables arises. It is needed to ensure that all write access to shared variables is atomic (exclusive). This can be solved by using a semaphore module. The semaphore keeps a token which is passed around using a channel in the class. Below the class is sketched:

```
scheme Semaphore =
class
    type
        SemReq == TOKEN

    channel
        semChan : SemReq

    value
        semProcess : Unit → in semChan out semChan Unit
        semProcess() ≡
            while (true) do
                semChan!TOKEN;

                let
                    token = semChan?
                in
                    ()
                end
            end,
end
```

The scheme *Semaphore* is based on the semaphore concept which is well known in the world of parallel programming - e.g. programming involving several processes or threads with common data access. The basic idea is that only one process or thread has write access to a variable at a time. This is ensured by having a *token* for such a shared variable. A process that requires access to this variable needs to have its token first.

The functions below has been created for easy access to token:

```
getToken : Unit → in semChan Unit
getToken() is
    let
        token = semChan?
    in
        ()
    end,

putToken : Unit → out semChan Unit
putToken() is
    semChan!TOKEN
```

As can be seen in the scheme above the token is requested / returned through the get and put method. Token access is provided through a channel so access to the token is

synchronized. The data accessing functions uses the token as shown below:

object
  tStateSem : Semaphore

variable
  v_trainStates : TrainStates

axiom
  setTrainState(tid,ts) ≡
    tStateSem.getToken();
    v_trainStates := v_trainStates !! [tid ↦ ts];
    tStateSem.putToken(),

All other functions which also requires variable access will then have to use the synchronized functions which accesses the appropriate token.

When several threads request the token at a time the token is sometimes passed directly from the utilizing thread to the next. The semaphore process itself is necessary to initially store the token and to store the token when not in use.

E.4 Shared variables in JAVA

Synchronization can be solved by using semaphore objects (utilizing channels) in RSL (explained in above section).

The JAVA solution translated from RSL could be made to look very similar to the RSL solution:

```
Object semaphore1 = new Object();

public void atomicFunc(int input)
{
  ...
  synchronized(semaphore1)
  {
    < some critical section >
  }
  ...
}
```

Thus keeping the semaphore concept from the RSL model and only synchronizing the exact same code as between the *get()* and *put()* statements in the RSL synchronized sections.

An easier approach has been used in this project as the whole function is synchronized if just one section of the function is critical. It will look like this:

```
public synchronized void atomicFunc(int input)
```

```
{
  ...
  < some critical section >
  ...
}
```

E.5 Channel communication in JAVA

A process in RSL is per definition a function which reads / writes on a channel. Channels are necessary in RSL if some processing is required to be synchronized or is to be executed in parallel.

Take an example where a train wants to communicate with a switch box (SB) and ask for permission to enter a segment. If the SB has a point then it switches the point to the appropriate branch and first then send an acknowledgment.

If the train was to communicate with the SB via. a function call, the train process would seemingly lock until the function call to the SB returned. This can be avoided by using channel communication instead of a direct function call.

The problem is that there (seemingly) is nothing in JAVA that corresponds to the RSL channel. The only way two processes or threads can communicate is by:

1. Shared variables
2. Socket (TCP / IP) communication
3. Direct function calls

E.5.1 Socket communication

Unless the RSL model is to be implemented as a distributed system there is no argument for using socket communication. Not only does this add to the complexity of the program but it does also increase the system requirements of the system running the software. Though it in fact is a distributed system that we are implementing it is still just a simulator and has no need to implement socket communication.

E.5.2 Shared varibles

Another approach is to implement the RSL channel as a JAVA class. The two communicating processes could then each have a reference to this channel. A such solution is sketched below:

```
public abstract class RSLChannel
{
   private Object storedObject = null;

   private synchronized Object readChannel()
   {
```

```
    while (storedObject == null)
            wait();

      Object temp = storedObject;
      storedObject = null;
      notifyAll();
      return temp;
   }

   private synchronized boolean writeChannel(Object obj)
   {
      storedObject = obj;

      // wait for object to be read
      while(storedObject != null)
         wait();

      return true; // object written and read successfully
   }
}
```

For type dependent channels the above solution needs to be extended with synchronized functions `read()` / `write()` which typecasts the return / input values for the functions above to the appropriate type.

E.5.3 Direct function call

One last approach is to let the communicating objects call each other directly. Instead of having a process listening to a channel then a function can be implemented with a normal parameter as input.

Again the disadvantage is that the calling object or process locks until the called function returns. This can be avoided by spawning the function call to another thread while execution continues in the calling process. To give an example of this the class `MethodThreadRunner` has been created. It only needs following arguments:

1. The object on which to invoke a function call.
2. The name of the function to invoke.
3. The arguments for the invoked function.

In fact the `MethodThreadRunner` class can also be used in the sense of parallel execution. Take for example the RSL expression func1() ‖ func2(). This could easily be translated to JAVA. It would look like the below:

```
MethodThreadRunner func1Runner = new MethodThreadRunner("func1",this);
MethodThreadRunner func2Runner = new MethodThreadRunner("func2",this);

func1Runner.start(); func2Runner.start();
func1Runner.join();  func2Runner.join();
```

This piece of code terminates when both `func1()` and `func2()` has terminated.

Appendix F

RSL modules

F.1 Initial model

F.1.1 Types

```
scheme AA_Types0 =
  class
    type
      /* Entity ID types */
      ID,
      ESAID = End,
      SBID = {| sb : ID • sbIDLimit(sb) |},
      SegmentID = {| seg : ID • segIDLimit(seg) |},
      TrainID = {| t : ID • trainIDLimit(t) |},

      /* Location of a train, on an esa or an segment */
      Location == isESA(getESA : ESAID) | isSeg(getSeg : SegmentID),

      /* The ends (esa ends) */
      End == HIGH | LOW,
      /* Driving direction on the line */
      Direction == UP | DOWN,

      /* Physical parameters */
      Length = Real,
      Speed = Real,
      Acceleration = Real,

      /* Neighbour of a SB in a direction */
      SBSegment == seg(getSeg : SegmentID) |
                   point(getUpSeg : SegmentID,
                         getDownSeg : SegmentID) |
                   esa(getESA : ESAID),
      /* The type of a SB */
      SBType == POINTSB | ENDSB | CROSSINGSB | PLAINSB,
      /* The segments etc around a point */
```

```
    PointSegments == pointSegments(getStem : SegmentID,
                                   getUpBranch : SegmentID,
                                   getDownBranch : SegmentID,
                                   getPointDir : Direction),

    /* The state of different entities */
    PointPosition == UP | DOWN | MOVINGUP | MOVINGDOWN,
    BarrierPosition == UP | DOWN | MOVINGUP | MOVINGDOWN,
    SignalStatus == ON | OFF,
    SensorStatus == ACTIVE | INACTIVE,

    /* Location/position of a train */
    TrainPosition :: frontPos : SegmentPosition ↔ setFrontPos
                     rearPos : SegmentPosition ↔ setRearPos,

    /* Location / position of a train end */
    SegmentPosition :: getLoc : Location
                       getLength : Length,

    Tick = Real,

    /* From control */
    HasRes == res(Reservation) | noRes,
    HasSeg == isSeg(SegmentID) | noSeg,

    Message = TCCMsg | SBCCMsg,
    TCCMsg == segReq(Reservation),
    SBCCMsg = SBCCResMsg | SBCCDeResMsg | SBCCRespMsg,
    SBCCResMsg == lineBranchReq(Reservation),
    SBCCDeResMsg == lineBranchDeRes | lineDeRes
                    | branchDeRes,
    SBCCRespMsg = LineBranchResp | SegmentResp,
    LineBranchResp == lineBranchResp(getRes : Reservation,

    SegmentResp == segResp(isPos : Bool),

    Reservation == mk_res(getTrain : TrainID, getDir : Direction),

    ReturnSBCCMsg == hasMsg(SBCCMsg) | noSBCCMsg,

    ComID == isSB(SBID) | isTrain(TrainID),
    ComMsg == mk_comMsg(getSender : ComID,
                        getReceiver : ComID,
                        getMsg : Message),
    HasComMsg == comMsg(ComMsg) | noComMsg

value
    /* The tick interval in seconds */
    tick_interval : Tick,

    /* Limits the ID of SBs Segments and Trains */
    sbIDLimit : ID → Bool,
    segIDLimit : ID → Bool,
    trainIDLimit : ID → Bool,

    /* Inverse the direction */
    inverseDir : Direction → Direction
```

```
inverseDir(dir) ≡
    case dir of
        UP → DOWN,
        DOWN → UP
    end,

/* Determines if a certain location is
     included in a SBSegment */
segPosInSBSeg : SegmentPosition × SBSegment → Bool
segPosInSBSeg(segPos,sbSeg) ≡
    case sbSeg of
        seg(seg) → isSeg(seg) = getLoc(segPos),
        point(upSeg,downSeg) → isSeg(upSeg) = getLoc(segPos) ∨
                                    isSeg(downSeg) = getLoc(segPos),
        esa(esa) → isESA(esa) = getLoc(segPos)
    end,

/* Returns all the segments in a 'SBSegment' */
sbSegToSet : SBSegment → SegmentID-set
sbSegToSet(sbSeg) ≡
    case sbSeg of
        seg(seg1) → { seg1 },
        point(seg1,seg2) → { seg1,seg2 },
        _ → {}
    end,

/* Returns the end to reach when
        following a certain direction */
dir2End : Direction → End
dir2End(dir) ≡
    case dir of
        DOWN → LOW,
        UP → HIGH
    end,

/* Returns the direction to go
     if is to reach a certain End */
end2Dir : End → Direction
end2Dir(end1) ≡
    case end1 of
        LOW → DOWN,
        HIGH → UP
    end,

/* Determines if a location is an ESA */
segPosIsESA : SegmentPosition → Bool
segPosIsESA(segPos) ≡
    case getLoc(segPos) of
        isESA(esa) → true,
        _ → false
    end,

/* Determines if an end position is a segment */
segPosIsSeg : SegmentPosition → Bool
segPosIsSeg(segPos) ≡
    case getLoc(segPos) of
        isSeg(_) → true
```

```
            end,

          /* Determines if a TrainPosition is
               located on one segment */
          trainOnlyOnESA : TrainPosition → Bool
          trainOnlyOnESA(pos) ≡
            case getLoc(frontPos(pos)) of
              isESA(_) →
              (
                case getLoc(rearPos(pos)) of
                  isESA(_) → true,
                  _ → false
                end
              ),

              _ → false
            end,

          /* Returns a set containing a segment if the
               position is on a segment else an empty set */
          segPosSeg : SegmentPosition → SegmentID-set
          segPosSeg(segPos) ≡
            case getLoc(segPos) of
              isSeg(seg) → {seg},
              _ → {}
            end,

          /* Returns a set containing all the
               segments in a TrainPosition */
          trainPosSegs : TrainPosition → SegmentID-set
          trainPosSegs(tp) ≡
            segPosSeg(frontPos(tp)) ∪ segPosSeg(rearPos(tp)),

          frontLoc : TrainPosition → Location
          frontLoc(tp) ≡
            getLoc(frontPos(tp)),

          rearLoc : TrainPosition → Location
          rearLoc(tp) ≡
            getLoc(rearPos(tp)),

          oneLoc : TrainPosition → Bool
          oneLoc(tp) ≡
            frontLoc(tp) = rearLoc(tp)

  end
```

F.1.2 Statics

```
context: AA_Types0
scheme AA_Statics0(T : AA_Types0) =
  class
    type
      /* Main railway line configuration type */
      Configuration
```

```
value
   conf : Configuration,

   /∗ Basic Observers ∗/

   /∗ ESA observers ∗/
   getESASB : T.ESAID × Configuration ~→ T.SBID,
   getESALength : T.ESAID × Configuration ~→ T.Length,
   esaExistsInConf : T.ESAID × Configuration → Bool,

   /∗ SB observers ∗/
   getSBSeg : T.SBID × T.Direction × Configuration ~→
                                                  T.SBSegment,
   getSBType : T.SBID × Configuration ~→ T.SBType,
   sbExistsInConf : T.SBID × Configuration → Bool,

   /∗ Segment observers ∗/
   getSegSB : T.SegmentID × T.Direction ×
                                   Configuration ~→ T.SBID,
   getSegLength : T.SegmentID × Configuration ~→ T.Length,
   getSegMaxSpeed : T.SegmentID × Configuration ~→ T.Speed,
   segExistsInConf : T.SegmentID × Configuration → Bool,

   /∗ Train observers ∗/
   getTrainLength : T.TrainID × Configuration ~→ T.Length,
   getTrainMaxSpeed : T.TrainID × Configuration ~→ T.Speed,
   getTrainMaxAcc : T.TrainID × Configuration ~→ T.Acceleration,
   getTrainMaxDec : T.TrainID × Configuration ~→ T.Acceleration,
   trainExistsInConf : T.TrainID × Configuration → Bool,

   /∗ Reservation− and brake−point observers ∗/
   getResPoint : Configuration ~→ T.Length,
   getBrakePoint : Configuration ~→ T.Length,

   /∗ Auxiliary functions ∗/

   /∗ Determines if a SB is a Single Line Guard ∗/
   isLineGuard : T.SBID × Configuration ~→ Bool
   isLineGuard(sb,con) ≡
      getSBType(sb,con) ∈ {T.POINTSB, T.ENDSB},

   /∗ Determines if a SB is a PointSB ∗/
   isPointSB : T.SBID × Configuration ~→ Bool
   isPointSB(sb,con) ≡
      getSBType(sb,con) = T.POINTSB,

   /∗ Determines if a segment is a branch segment ∗/
   segIsBranch : T.SegmentID × Configuration ~→ Bool
   segIsBranch(seg,con) ≡
      getSBType(getSegSB(seg,T.UP,con),con) = T.POINTSB ∧
      getSBType(getSegSB(seg,T.DOWN,con),con) = T.POINTSB,

   /∗ Determines if a segment is a line segment,
```

```
    i.e. a segment in a single line */
segIsLineSegment : T.SegmentID × Configuration ~→ Bool
segIsLineSegment(seg,con) ≡
    ∼segIsBranch(seg,con),

neighbours : T.Location × T.Location × Configuration ~→ Bool
neighbours(loc1,loc2,con) ≡
    (getLocSBs(loc1,con) ∪ getLocSBs(loc2,con)) ≠ {},

/* Finds the distance (T.Length)
    between two T.SegmentPosition */
distance : T.SegmentPosition × T.SegmentPosition ×
                                    Configuration ~→ T.Length
distance(segPos1,segPos2,con) ≡
    if (T.getLoc(segPos1) = T.getLoc(segPos2))
    then
        if (T.getLength(segPos1) < T.getLength(segPos2))
        then
            T.getLength(segPos2) − T.getLength(segPos1)
        else
            T.getLength(segPos1) − T.getLength(segPos2)
        end
    else
        if (segPosLower(segPos1,segPos2,con))
        then
            getLocLength(T.getLoc(segPos1),con) −
                    T.getLength(segPos1) + T.getLength(segPos2)
        else
            getLocLength(T.getLoc(segPos2),con) −
                    T.getLength(segPos2) + T.getLength(segPos1)
        end
    end
pre neighbours(T.getLoc(segPos1),T.getLoc(segPos2),con) ∨
        T.getLoc(segPos1) = T.getLoc(segPos2),

segPosLower : T.SegmentPosition × T.SegmentPosition ×
                                        Configuration ~→ Bool
segPosLower(segPos1,segPos2,con) ≡
    if (T.getLoc(segPos1) = T.getLoc(segPos2)) then
        T.getLength(segPos1) < T.getLength(segPos2)
    else
        locLower(T.getLoc(segPos1),T.getLoc(segPos2),con)
    end,


/* If a location is immediatedly lower than
    another location. If one location is an ESA,
    the result will depend on the orientation
    of the ESA. */
locLower : T.Location × T.Location × Configuration ~→ Bool
locLower(loc1,loc2,con) ≡
    case loc1 of
        T.isESA(esa1) → (esa1 = T.LOW),
        T.isSeg(seg1) →
        (
            case loc2 of
```

```
                T.isESA(esa2) → (esa2 = T.HIGH),
                T.isSeg(seg2) →
                (
                    seg2 ∈ getNextSegSet(seg1,T.UP,con)
                )
            end
        )
    end,

getLocLength : T.Location × Configuration ~→ T.Length
getLocLength(loc,con) ≡
    case loc of
        T.isESA(esa) → getESALength(esa,con),
        T.isSeg(seg) → getSegLength(seg,con)
    end,

getLocSBs : T.Location × Configuration ~→ T.SBID-set
getLocSBs(loc,con) ≡
    case loc of
        T.isESA(esa) → {getESASB(esa,con)},
        T.isSeg(seg) → {getSegSB(seg,T.UP,con),
                              getSegSB(seg,T.DOWN,con)}
    end,

/* Returns the segments around a point */
getSBPointSegs : T.SBID × Configuration ~→ T.PointSegments
getSBPointSegs(sb,con) ≡
    let
        dir = getPointDir(sb,con),
        pointSegs = getSBSeg(sb,dir,con),
        T.seg(stemSeg) = getSBSeg(sb,T.inverseDir(dir),con)
    in
        T.pointSegments(stemSeg,
                          T.getUpSeg(pointSegs),
                          T.getDownSeg(pointSegs),
                          dir)
    end
pre getSBType(sb,con) = T.POINTSB,

/* Returns the driving direction of a branch */
branchDir : T.SegmentID × Configuration ~→ T.Direction
branchDir(seg,con) ≡
    let
        T.point(up,down) =
                getSBSeg(getSegSB(seg,T.UP,con),T.DOWN,con)
    in
        if (seg = up)
        then
            T.UP
        else
            T.DOWN
        end
    end
pre segIsBranch(seg,con),

/* Given a single line guard, it returns the single
   line guard at the opposite end of the single line */
```

```
getOppositeGuard : T.SBID × Configuration ~→ T.SBID
getOppositeGuard(sb,con) ≡
    let
        sbType = getSBType(sb,con),
        dir = if(sbType = T.POINTSB) then getPointDir(sb,con)
                    else getEndDir(sb,con) end,
        lineDir = T.inverseDir(dir)
    in
        getSingleLineGuard(getNextSB(sb,lineDir,con),lineDir,con)
    end
pre isLineGuard(sb,con),

/∗ Given a point SB, it returns the point SB
     at the opposite end of the branches ∗/
getOppositePointSB : T.SBID × Configuration ~→ T.SBID
getOppositePointSB(sb,con) ≡
    let
        dir = getPointDir(sb,con)
    in
         getNextSB(sb,dir,con)
    end
pre getSBType(sb,con) = T.POINTSB,

/∗ Given an SB, it returns the next SB ∗/
getNextSB : T.SBID × T.Direction × Configuration ~→ T.SBID
getNextSB(sb,dir,con) ≡
    let
        nextSeg = getSBSeg(sb,dir,con)
    in
        case nextSeg of
            T.seg(segID) → getSegSB(segID,dir,con),
            T.point(upSeg,downSeg) → getSegSB(upSeg,dir,con)
        end
    end
pre getSBType(sb,con) ≠ T.ENDSB ∨ getEndDir(sb,con) ≠ dir,

getNextSegSet : T.SegmentID × T.Direction ×
                                        Configuration ~→ T.SegmentID-set
getNextSegSet(seg,dir,con) ≡
    T.sbSegToSet(getSBSeg(getSegSB(seg,dir,con),dir,con)),

/∗ Returns the first single line guard in a direction ∗/
getSingleLineGuard : T.SBID × T.Direction ×
                                              Configuration ~→ T.SBID
getSingleLineGuard(sb,dir,con) ≡
    if(isLineGuard(sb,con))
    then
        sb
    else
        getSingleLineGuard(getNextSB(sb,dir,con),dir,con)
    end,

/∗ Returns the two single line
     guards of a line−segment ∗/
getSingleLineGuards : T.SegmentID ×
                                        Configuration ~→ T.SBID-set
```

```
getSingleLineGuards(seg,con) ≡
    let
        sb = getSegSB(seg,T.UP,con)
    in
        { getSingleLineGuard(sb,T.UP,con),
            getSingleLineGuard(sb,T.DOWN,con) }
    end
pre ∼segIsBranch(seg,con),

/∗ Returns the direction of a point SB
   from the stem towards the branches ∗/
getPointDir : T.SBID × Configuration ~→ T.Direction
getPointDir(sb,con) ≡
    let sbSeg = getSBSeg(sb,T.UP,con)
    in
        case sbSeg of
            T.point(_,_) → T.UP,
            _ → T.DOWN
        end
    end
pre getSBType(sb,con) = T.POINTSB,

/∗ Returns the direction against an ESA from an END SB ∗/
getEndDir : T.SBID × Configuration ~→ T.Direction
getEndDir(sb,con) ≡
    case getSBSeg(sb,T.UP,con) of
        T.esa(_) → T.UP,
        _ → T.DOWN
    end
pre getSBType(sb,con) = T.ENDSB,

sbsAreCrossings : T.SBID-set × Configuration ~→ Bool
sbsAreCrossings(sbs,con) ≡
(
    ∀ sb : T.SBID •
        sb ∈ sbs ⇒
            getSBType(sb,con) = T.CROSSINGSB
),

sbsArePoints : T.SBID-set × Configuration ~→ Bool
sbsArePoints(sbs,con) ≡
(
    ∀ sb : T.SBID •
        sb ∈ sbs ⇒
            getSBType(sb,con) = T.POINTSB
),

/∗ Invariants ∗/
is_wf : Configuration → Bool
is_wf(con) ≡
    sbs_is_wf(con) ∧
    segs_is_wf(con) ∧
    esas_is_wf(con) ∧
    trains_is_wf(con) ∧
    composed_is_wf(con),

sbs_is_wf : Configuration → Bool
```

```
sbs_is_wf(con) ≡
    sbsHaveConf(con) ∧
    getSBSeg_diff(con) ∧
    getSBSeg_point_wf(con) ∧
    getSBSeg_injective(con) ∧
    getSBSegType_wf(con),

/∗ A configuration for each SB must exists ∗/
sbsHaveConf : Configuration → Bool
sbsHaveConf(con) ≡
(
    (∀ seg : T.SegmentID •
        sbExistsInConf(seg,con)) ∧
     getResPoint(con) > 0.0 ∧
     getBrakePoint(con) > 0.0
),

/∗ The segments next to a SB are different
     in the T.UP and the T.DOWN direction.
     I.e. the line is not circular ∗/
getSBSeg_diff : Configuration → Bool
getSBSeg_diff(con) ≡
(
    ∀ sb : T.SBID •
        getSBSeg(sb,T.UP,con) ≠ getSBSeg(sb,T.DOWN,con)
),

/∗ The two branches of a junction are different ∗/
getSBSeg_point_wf : Configuration → Bool
getSBSeg_point_wf(con) ≡
(
    ∀ sb : T.SBID,
            seg1,seg2 : T.SegmentID,
            dir : T.Direction •
                T.point(seg1,seg2) = getSBSeg(sb,dir,con) ⇒
                    seg1 ≠ seg2
),

/∗ Two different SBs have different SBSegments
     in the same direction ∗/
getSBSeg_injective : Configuration → Bool
getSBSeg_injective(con) ≡
(
    ∀ sb1, sb2 : T.SBID,
            dir : T.Direction •
                sb1 ≠ sb2 ⇒
                    getSBSeg(sb1,dir,con) ≠ getSBSeg(sb2,dir,con)
),

/∗ The type of a SB must conform
     with the result of getSBSeg ∗/
getSBSegType_wf : Configuration → Bool
getSBSegType_wf(con) ≡
(
    ∀ sb : T.SBID •
        case getSBType(sb,con) of
            T.ENDSB →
```

```
            (∃! dir : T.Direction, esa : T.ESAID •
                esa = T.dir2End(dir) ∧
                getSBSeg(sb,dir,con) = T.esa(esa)),
        T.POINTSB →
            (∃! dir : T.Direction, seg1,seg2 : T.SegmentID •
                getSBSeg(sb,dir,con) = T.point(seg1,seg2)),
        T.CROSSINGSB →
            (∀ dir : T.Direction •
                ∃ seg : T.SegmentID •
                    getSBSeg(sb,dir,con) = T.seg(seg)),
        T.PLAINSB →
            (∃! dir : T.Direction •
                ∃ seg : T.SegmentID •
                    getSBSeg(sb,dir,con) = T.seg(seg))
    end
),

segs_is_wf : Configuration → Bool
segs_is_wf(con) ≡
    segsHaveConf(con) ∧
    getSegSB_injective(con) ∧
    brakeResPoint_wf(con),

/* A configuration for each Segment must exists */
segsHaveConf : Configuration → Bool
segsHaveConf(con) ≡
(
    ∀ seg : T.SegmentID •
        segExistsInConf(seg,con)
),

/**
* The SB in the end of a segment is different
* for two different segments or they are the
* same in both direction (being branches)
**/
getSegSB_injective : Configuration → Bool
getSegSB_injective(con) ≡
(
    ∀ seg1, seg2 : T.SegmentID,
        dir : T.Direction •
            seg1 ≠ seg2 ⇒
            (
                getSegSB(seg1,dir,con) ≠ getSegSB(seg2,dir,con)
            )
                ∨
            (
            getSegSB(seg1,T.UP,con) = getSegSB(seg2,T.UP,con) ∧
            getSegSB(seg1,T.DOWN,con) = getSegSB(seg2,T.DOWN,con)
            )
),

/* The reservation−point should be placed before
    the brake−point, i.e. there is a greater
    distance from the end of a segment to the
    reservation−point than to the brake−point */
brakeResPoint_wf : Configuration → Bool
```

```
brakeResPoint_wf(con) ≡
    getResPoint(con) > getBrakePoint(con),

esas_is_wf : Configuration → Bool
esas_is_wf(con) ≡
    esasHaveConf(con),

/* A configuration for each ESA must exists */
esasHaveConf : Configuration → Bool
esasHaveConf(con) ≡
(
    ∀ esa : T.ESAID •
        esaExistsInConf(esa,con)
),

trains_is_wf : Configuration → Bool
trains_is_wf(con) ≡
    trainsHaveConf(con),

/* A configuration for each Train must exists */
trainsHaveConf : Configuration → Bool
trainsHaveConf(con) ≡
(
    ∀ t : T.TrainID •
        trainExistsInConf(t,con)
),

composed_is_wf : Configuration → Bool
composed_is_wf(con) ≡
    pointSegs_wf(con) ∧
    getESASBSeg_wf(con) ∧
    getSBSeg_getSegSB_wf(con)∧
    seg_train_length_wf(con) ∧
    esa_train_length_wf(con) ∧
    brakePoint_wf(con) ∧
    resPoint_wf(con) ∧
    collisions_detectable(con),

/* All associated point (points next to each other)
     must have same up and down branches */
pointSegs_wf : Configuration → Bool
pointSegs_wf(con) ≡
(
    ∀ sb : T.SBID •
        getSBType(sb,con) = T.POINTSB
            ⇒
            let
                pSegs1 = getSBPointSegs(sb,con),
                dir1 = T.getPointDir(pSegs1),

                sb2 = getNextSB(sb,dir1,con),
                pSegs2 = getSBPointSegs(sb2,con)
            in
                T.getUpBranch(pSegs1) = T.getUpBranch(pSegs2) ∧
                T.getDownBranch(pSegs1) = T.getDownBranch(pSegs2)
            end
 ),
```

```
/∗ Given an ESA. From the coherent END SB
     the next SBSegment directed against the
     ESA must be the ESA ∗/
getESASBSeg_wf : Configuration → Bool
getESASBSeg_wf(con) ≡
(
    ∀ esa : T.ESAID •
        getSBSeg(getESASB(esa,con),T.end2Dir(esa),con) = T.esa(esa)
),

/∗ Calculating the SB in a direction from each segment
     in the SBSegment calculated from a SB in the opposite
     direction must give the original SB ∗/
getSBSeg_getSegSB_wf : Configuration → Bool
getSBSeg_getSegSB_wf(con) ≡
(
    ∀ sb : T.SBID, dir : T.Direction, seg : T.SegmentID •
        seg ∈ T.sbSegToSet(getSBSeg(sb,dir,con)) ⇒
            getSegSB(seg,T.inverseDir(dir),con) = sb
),

/∗ All segments must be longer than any train ∗/
seg_train_length_wf : Configuration → Bool
seg_train_length_wf(con) ≡
(
    ∀ seg : T.SegmentID, t : T.TrainID •
        getSegLength(seg,con) > getTrainLength(t,con)
),

/∗ An ESA must be longer than a brake point.
     This ensures that all the axioms above
     (concerning braking) also applies to the ESAs
∗/
esa_train_length_wf : Configuration → Bool
esa_train_length_wf(con) ≡
(
    ∀ esa : T.ESAID, t : T.TrainID •
        getESALength(esa,con) >
            getBrakePoint(con) + getTrainLength(t,con)
),

/∗ If a train starts to brake at the brakepoint
     it must be able to stop entirely before
     entering the next segment
∗/
brakePoint_wf : Configuration → Bool
brakePoint_wf(con) ≡
(
    ∀ t : T.TrainID, tAcc : T.Acceleration,
            brakeP, brakeL, s_err : T.Length,
            tSpeed : T.Speed •
                tAcc = getTrainMaxDec(t,con) ∧
                brakeP = getBrakePoint(con) ∧
                tSpeed = getTrainMaxSpeed(t,con) ∧
                s_err = tSpeed ∗ T.tick_interval ∧
                brakeL = −0.5 ∗ tSpeed ∗ tSpeed / tAcc
```

```
                    ⇒
                        brakeP > brakeL + s_err
        ),

        /* When a train reach the brake point it must be entirely
             on a single segment and the brake point must be smaller
             than the length of any segment
        */
        resPoint_wf : Configuration → Bool
        resPoint_wf(con) ≡
        (
            ∀ t : T.TrainID, seg : T.SegmentID,
                    tlen, slen, resPoint, brakePoint : T.Length •
                        tlen = getTrainLength(t,con) ∧
                        slen = getSegLength(seg,con) ∧
                        resPoint = getResPoint(con) ∧
                        brakePoint = getBrakePoint(con)
                            ⇒
                                slen  > (resPoint + tlen) ∧
                                brakePoint < slen
        ),

        /* Ensures that collisions can be detected before
             trains passes through each other */
        collisions_detectable : Configuration → Bool
        collisions_detectable(con) ≡
        (
            ∀ t1, t2 : T.TrainID, sp1, sp2 : T.Speed,
                    s_err1, s_err2, s_col : T.Length •
                        sp1 = getTrainMaxSpeed(t1,con) ∧
                        sp2 = getTrainMaxSpeed(t2,con) ∧
                        s_err1 = sp1 * T.tick_interval ∧
                        s_err2 = sp2 * T.tick_interval ∧
                        s_col = s_err1 + s_err2
                            ⇒
                                s_col < getTrainLength(t1,con)
        )

    axiom
        [is_wf]
            is_wf(conf)

end
```

F.1.3 Dynamics

```
context: AA_Statics0
scheme AA_Dynamics0(T : AA_Types0,S : AA_Statics0(T)) =
    class
        type
            /* Type of interest */
            State

        value
            initState : State,
```

```
/* Point observer */
getPointPosition : T.SBID × State × S.Configuration ~→ T.PointPosition,
getPointTicks : T.SBID × State × S.Configuration ~→ T.Tick,

/* Point generator */
setPointPosition : T.SBID × T.PointPosition × State ×
                                        S.Configuration ~→ State,
setPointTicks : T.SBID × T.Tick × State × S.Configuration ~→ State,

/* Crossing observer */
getBarrierPosition : T.SBID × State × S.Configuration ~→ T.BarrierPosition,
getSignalStatus : T.SBID × State × S.Configuration ~→ T.SignalStatus,

/* Crossing generator */
setBarrierPosition : T.SBID × T.BarrierPosition × State ×
                                        S.Configuration ~→ State,
setSignalStatus : T.SBID × T.SignalStatus × State × S.Configuration ~→ State,

/* Sensor observer */
getSensorStatus : T.SBID × State → T.SensorStatus,

/* Sensor generator */
setSensorStatus : T.SBID × T.SensorStatus × State × S.Configuration ~→ State,

/* Train observer */
getTrainAcc : T.TrainID × State → T.Acceleration,
getTrainSpeed : T.TrainID × State → T.Speed,
getTrainPosition : T.TrainID × State → T.TrainPosition,
getTrainDirection : T.TrainID × State → T.Direction,

/* Train generator */
setTrainAcc : T.TrainID × T.Acceleration × State × S.Configuration ~→ State,
setTrainSpeed : T.TrainID × T.Speed × State × S.Configuration ~→ State,
setTrainPosition : T.TrainID × T.TrainPosition × State ×
                                        S.Configuration ~→ State,
setTrainDirection : T.TrainID × T.Direction × State ~→ State,

changeTrainDirection : T.TrainID × State × S.Configuration → State
changeTrainDirection(t,s,con) ≡
  let
    dir = T.inverseDir(getTrainDirection(t,s)),
    tp = getTrainPosition(t,s),

    front = T.frontPos(tp),
    rear = T.rearPos(tp),

    tp = T.mk_TrainPosition(rear,front),

    s = setTrainDirection(t,dir,s)
  in
    setTrainPosition(t,tp,s,con)
  end,

/* Processes */
```

```
tick : T.Tick × S.Configuration × State ~→ State
tick(tick,con,s) ≡
    let
        s = tickPoints(tick,con,s),
        s = tickCrossings(tick,con,s),
        s = tickTrains(tick,con,s)
    in
        s
    end,

tickPoints : T.Tick × S.Configuration × State ~→ State
tickPoints(tick,con,s) ≡
    let
        points = { p | p : T.SBID • S.getSBType(p,con) = T.POINTSB }
    in
        pointProcess(points,tick,con,s)
    end,

pointProcess : T.SBID-set × T.Tick × S.Configuration × State ~→ State
pointProcess(points,tick,con,s) ≡
    if(points = {})
    then
        s
    else
        let
            p : T.SBID • p ∈ points,
            points = points \ {p},
            s = updatePoint(p,tick,con,s)
        in
            pointProcess(points,tick,con,s)
        end
    end
pre S.sbsArePoints(points,con),

updatePoint : T.SBID × T.Tick × S.Configuration × State ~→ State
updatePoint(p,tick,con,s) ≡
    let
        pp = getPointPosition(p,s,con)
    in
        case pp of
            T.MOVINGDOWN → s ⌈⌉ setPointPosition(p,T.DOWN,s,con),
            T.MOVINGUP → s ⌈⌉ setPointPosition(p,T.UP,s,con),
            _ → s
        end
    end
pre S.getSBType(p,con) = T.POINTSB,

tickCrossings : T.Tick × S.Configuration × State ~→ State
tickCrossings(tick,con,s) ≡
    let
        crossings = { c | c : T.SBID • S.getSBType(c,con) = T.CROSSINGSB }
    in
        crossingProcess(crossings,tick,con,s)
    end,

crossingProcess : T.SBID-set × T.Tick × S.Configuration × State ~→ State
```

```
crossingProcess(crossings,tick,con,s) ≡
    if(crossings = {})
    then
        s
    else
        let
            c : T.SBID • c ∈ crossings,
            crossings = crossings \ {c},
            s = updateCrossing(c,tick,con,s)
        in
            crossingProcess(crossings,tick,con,s)
        end
    end
pre S.sbsAreCrossings(crossings,con),

updateCrossing : T.SBID × T.Tick × S.Configuration × State ~→ State
updateCrossing(cr,tick,con,s) ≡
    let
        bp = getBarrierPosition(cr,s,con),
        ss = getSignalStatus(cr,s,con)
    in
        case bp of
            T.UP →
            (
                if(ss = T.ON)
                then
                    s ⊓
                    setBarrierPosition(cr,T.MOVINGDOWN,s,con)
                else
                    s
                end
            ),
            T.MOVINGDOWN →
            (
                s ⊓
                (
                    let
                        bp = setBarrierPosition(cr,T.DOWN,s,con)
                    in
                        setSignalStatus(cr,T.OFF,s,con)
                    end
                )
            ),
            T.DOWN → s,
            T.MOVINGUP → s ⊓ setBarrierPosition(cr,T.UP,s,con)
        end
    end
pre S.getSBType(cr,con) = T.CROSSINGSB,

tickTrains : T.Tick × S.Configuration × State ~→ State
tickTrains(tick,con,s) ≡
let
    trains = { t | t : T.TrainID}
in
    trainProcess(trains,tick,con,s)
end,
```

```
trainProcess : T.TrainID-set × T.Tick × S.Configuration × State ~→ State
trainProcess(trains,tick,con,s) ≡
    if(trains = {})
    then
        s
    else
        let
            t : T.TrainID • t ∈ trains,
            trains = trains \ {t},
            s = updateTrain(t,tick,con,s)
        in
            trainProcess(trains,tick,con,s)
        end
    end,

updateTrain : T.TrainID × T.Tick × S.Configuration × State ~→ State,

/* Auxiliary functions */

/* Returns the front segment of a train. If front is on ESA then
     the rear segment is returned. This is used for speed checking */
getTrainLoc : T.TrainID × State → T.Location
getTrainLoc(t,ds) ≡
    let
        tp = getTrainPosition(t,ds),
        frontLoc = T.getLoc(T.frontPos(tp)),
        rearLoc = T.getLoc(T.rearPos(tp))
    in
        case frontLoc of
            T.isESA(esa) → rearLoc,
            _ → frontLoc
        end
    end
pre ∼trainInESA(t,ds),

tpDerailed : T.TrainPosition × T.Direction × State × S.Configuration → Bool
tpDerailed(tp,dir,s,con) ≡
    if (∼T.oneLoc(tp) ∧ ∼T.segPosIsESA(T.frontPos(tp))) then
        let
            seg = T.getSeg(T.frontLoc(tp)),
            sb = S.getSegSB(seg,T.inverseDir(dir),con)
        in
            case S.getSBType(sb,con) of
                    T.POINTSB →
                    (
                        if (dir = S.getPointDir(sb,con)) then
                            pointConnected(sb,T.getSeg(T.frontLoc(tp)),s,con)
                        else
                            pointConnected(sb,T.getSeg(T.rearLoc(tp)),s,con)
                        end
                    ),

                    T.CROSSINGSB →
                    (
                        getBarrierPosition(sb,s,con) = T.DOWN
                    ),
```

```
                _ → false
            end
        end
    else
        false
    end,

getESATrains : T.ESAID × State ~→ T.TrainID-set
getESATrains(esa,s) ≡
    { t | t : T.TrainID • T.trainOnlyOnESA(getTrainPosition(t,s)) },

/∗ Front and rear position of a train must be exactly
     'train length' apart ∗/
train_pos_ok : T.TrainID × T.TrainPosition × State × S.Configuration ~→ Bool
train_pos_ok(t,tp,s,con) ≡
(
    let
        T.mk_TrainPosition(posFront,posRear) = tp
    in
        (S.distance(posFront,posRear,con) = S.getTrainLength(t,con)) ∧
        train_pos_dir_ok(getTrainDirection(t,s),tp,s,con)
    end
),

/∗ If train drives UP then rear pos must be lower than front pos
     and vice versa ∗/
train_pos_dir_ok : T.Direction × T.TrainPosition × State ×
                                                S.Configuration → Bool
train_pos_dir_ok(dir,tp,s,con) ≡
(
    case dir of
        T.UP →
        (
            S.segPosLower(T.rearPos(tp),T.frontPos(tp),con)
        ),

        T.DOWN →
        (
            S.segPosLower(T.frontPos(tp),T.rearPos(tp),con)
        )
    end
),

getTrainSegments : T.TrainID × State ~→ T.SegmentID-set
getTrainSegments(t,s) ≡
    T.trainPosSegs(getTrainPosition(t,s)),

getTrainBranch : T.TrainID × State × S.Configuration ~→ T.SegmentID
getTrainBranch(t,s,con) ≡
(
    let
        seg : T.SegmentID • seg ∈ getTrainSegments(t,s) ∧
                                    S.segIsBranch(seg,con)
    in
        seg
    end
)
```

```
pre (∃ sb : T.SBID • trainOnJunction(t,sb,con,s)),

trainOnSegment : T.TrainID × T.SegmentID × S.Configuration × State →̃ Bool
trainOnSegment(tID,seg,con,ds) ≡
    seg ∈ getTrainSegments(tID,ds),

trainOnJunction : T.TrainID × T.SBID × S.Configuration × State →̃ Bool
trainOnJunction(t,sb,con,ds) ≡
(
    S.getSBType(sb,con) = T.POINTSB ∧
    trainOnSensor(t,sb,con,ds)
),

trainOnJunction : T.SBID × S.Configuration × State →̃ Bool
trainOnJunction(sb,con,s) ≡
    S.getSBType(sb,con) = T.POINTSB ∧
    trainOnSensor(sb,con,s),

trainOnSensor : T.TrainID × T.SBID × S.Configuration × State →̃ Bool
trainOnSensor(t,sb,con,ds) ≡
(
    ∃ dir : T.Direction, tPos : T.TrainPosition,
                sp1,sp2 : T.SegmentPosition •
                    tPos = getTrainPosition(t,ds) ∧
        T.segPosInSBSeg(sp1, S.getSBSeg(sb,dir,con)) ∧
        T.segPosInSBSeg(sp2, S.getSBSeg(sb,T.inverseDir(dir),con))
),

trainOnSensor : T.SBID × S.Configuration × State →̃ Bool
trainOnSensor(sb,con,s) ≡
(
    ∃ t : T.TrainID, dir : T.Direction, tPos : T.TrainPosition,
                sp1,sp2 : T.SegmentPosition •
                    tPos = getTrainPosition(t,s) ∧
        T.segPosInSBSeg(sp1, S.getSBSeg(sb,dir,con)) ∧
        T.segPosInSBSeg(sp2, S.getSBSeg(sb,T.inverseDir(dir),con))
),

trainInESA : T.TrainID × State →̃ Bool
trainInESA(t,s) ≡
    T.trainOnlyOnESA(getTrainPosition(t,s)),

trainInESADrivingOut : T.TrainID × State → Bool
trainInESADrivingOut(t,s) ≡
    if(∼T.trainOnlyOnESA(getTrainPosition(t,s)))
    then
        false
    else
        let
            segPos = T.frontPos(getTrainPosition(t,s)),
            esa = T.getESA(T.getLoc(segPos)),
            dir = getTrainDirection(t,s)
        in
            T.end2Dir(esa) ≠ dir
        end
    end,
```

```
trainFrontInESA : T.TrainID × State ~→ Bool
trainFrontInESA(t,s) ≡
    let
        tPos = getTrainPosition(t,s)
    in
        T.segPosIsESA(T.frontPos(tPos))
    end,

/∗ Telling if a train is (partly) on a single line ∗/
trainOnSingleLine : T.TrainID × S.Configuration × State ~→ Bool
trainOnSingleLine(t,con,s) ≡
    let
        tPos = getTrainPosition(t,s),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∃ s : T.SegmentID •
                    s ∈ segSet ⇒
                        ∼S.segIsBranch(s,con)
            )
        else
            false
        end
    end,

/∗ Telling if a train is (partly) on a branch ∗/
trainOnBranch : T.TrainID × S.Configuration × State ~→ Bool
trainOnBranch(t,con,s) ≡
    let
        tPos = getTrainPosition(t,s),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∃ s : T.SegmentID •
                    s ∈ segSet ⇒
                        S.segIsBranch(s,con)
            )
        else
            false
        end
    end,

/∗ Telling if a train is only on a branch ∗/
trainOnlyOnBranch : T.TrainID × S.Configuration × State ~→ Bool
trainOnlyOnBranch(t,con,s) ≡
    let
        tPos = getTrainPosition(t,s),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∀ s : T.SegmentID •
                    s ∈ segSet ⇒
                        S.segIsBranch(s,con)
```

```
        )
      else
        false
      end
    end,

  pointConnected : T.SBID × T.SegmentID × State × S.Configuration ~→ Bool
  pointConnected(sbID,seg,ds,con) ≡
    let
      pointSegs = S.getSBPointSegs(sbID,con)
    in
      case getPointPosition(sbID,ds,con) of
        T.UP → (seg = T.getUpBranch(pointSegs)),
        T.DOWN → (seg = T.getDownBranch(pointSegs)),
        _ → false
      end
    end
  pre S.getSBType(sbID,con) = T.POINTSB,

  trainFrontLoc : T.TrainID × State ~→ T.Location
  trainFrontLoc(t,ds) ≡
    case T.frontLoc(getTrainPosition(t,ds)) of
      T.isESA(_) → T.rearLoc(getTrainPosition(t,ds)),
      T.isSeg(seg) → T.isSeg(seg)
    end,

  sensor_guard : T.SBID × T.SensorStatus × S.Configuration × State ~→ Bool
  sensor_guard(sen,ss,con,s) ≡
    (ss = T.ACTIVE ∧ trainOnSensor(sen,con,s)) ∨
    (ss = T.INACTIVE ∧ ∼trainOnSensor(sen,con,s)),

  decelerateTrain : T.TrainID × S.Configuration × State ~→ State
  decelerateTrain(t,con,s) ≡
    if(getTrainSpeed(t,s) ≠ 0.0)
    then
      let
        maxDec = S.getTrainMaxDec(t,con),
        curDec = getTrainAcc(t,s)
      in
        if(maxDec ≠ curDec)
        then
          setTrainAcc(t,maxDec,s,con)
        else
          s
        end
      end
    else
      setTrainAcc(t,0.0,s,con)
    end,

  accelerateTrain : T.TrainID × S.Configuration × State ~→ State
  accelerateTrain(tID,con,s) ≡
    setTrainAcc(tID,S.getTrainMaxAcc(tID,con),s,con),

  commonSegs : T.TrainPosition × T.TrainID × State ~→ T.SegmentID-set
  commonSegs(tp1,t2,ds) ≡
```

```
    T.trainPosSegs(tp1) ∩ getTrainSegments(t2,ds),

commonSegs : T.TrainID × T.TrainID × State ⥲ T.SegmentID-set
commonSegs(t1,t2,ds) ≡
    getTrainSegments(t1,ds) ∩ getTrainSegments(t2,ds),

trainPositionOccupied : T.TrainID × T.TrainPosition × State ×
                                                S.Configuration ⥲ Bool
trainPositionOccupied(t1,tp1,ds,con) ≡
(
    ∀ segs : T.SegmentID-set, dir1,dir2 : T.Direction,
            tp1,tp2 : T.TrainPosition •
                ∃ t2 : T.TrainID •
                    t2 ≠ t1 ∧
                    segs = commonSegs(tp1,t2,ds) ∧
                    segs ≠ {} ∧
                    (dir1,dir2) = (getTrainDirection(t1,ds),getTrainDirection(t2,ds)) ∧
                    tp2 = getTrainPosition(t2,ds) ∧

                    case dir1 of
                        T.UP →
                        (
                            if (dir1 = dir2)
                            then
                                S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con) ⇒
                                    ∼S.segPosLower(T.frontPos(tp1),T.rearPos(tp2),con)
                            else
                                ∼S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
                            end
                        ),

                        T.DOWN →
                        (
                            if (dir1 = dir2) then
                                ∼S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con) ⇒
                                    S.segPosLower(T.frontPos(tp1),T.rearPos(tp2),con)
                            else
                                S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
                            end
                        )
                    end
),

/* Tells if a train has a state in the system */
trainStateExists : T.TrainID × State → Bool,

/* Tells if a sensor has a state in the system */
sensorStateExists : T.SBID × State → Bool,

/* Tells if a crossing has a state in the system */
crossingStateExists : T.SBID × State × S.Configuration → Bool,

/* Tells if a point has a state in the system */
pointStateExists : T.SBID × State × S.Configuration → Bool,

/* Invariants etc. */
```

```
/∗ Telling if the railway line is safe ∗/
safe : State × S.Configuration ~→ Bool
safe(s,con) ≡
    is_wf(s,con) ∧
    noCollisions(con,s) ∧
    trainPosPossible(con,s) ∧
    pointsSafe(con,s) ∧
    crossingsSafe(con,s),

/∗∗
∗   The position of a train may not overlap
∗   with the position of other trains
∗∗/
noCollisions : S.Configuration × State ~→ Bool
noCollisions(con,s) ≡
(
    ∀ t : T.TrainID •
        ∼trainPositionOccupied(t,getTrainPosition(t,s),s,con)
),

/∗∗
∗   Trains cannot end up on same segment
∗   driving in opposite directions away from each other.
∗
∗   If two train are on same segment driving in opposite
∗   directions then the train driving up must be lower
∗   on the line than the train driving down.
∗∗/
trainPosPossible : S.Configuration × State ~→ Bool
trainPosPossible(con,ds) ≡
(
    ∀ t1,t2 : T.TrainID, segs : T.SegmentID-set,
            tp1,tp2 : T.TrainPosition, seg : T.SegmentID •
                commonSegs(t1,t2,ds) ≠ {} ∧
                (tp1,tp2) = (getTrainPosition(t1,ds),getTrainPosition(t1,ds)) ∧
                getTrainDirection(t1,ds) ≠ getTrainDirection(t2,ds) ∧
                getTrainDirection(t1,ds) = T.UP
                    ⇒
                        S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
),

/∗∗
∗   If the train is located upon a junction,
∗   the point must be connected to the
∗   branch, on which the train is located
∗∗/
pointsSafe : S.Configuration × State ~→ Bool
pointsSafe(con,ds) ≡
(
    ∀ sb : T.SBID, t : T.TrainID, seg : T.SegmentID •
        trainOnJunction(t,sb,con,ds) ∧
        trainOnSegment(t,seg,con,ds) ∧
        S.segIsBranch(seg,con) ⇒
            pointConnected(sb,seg,ds,con)
),

/∗ When a train is located on a crossing
```

```
     the barriers must be down */
crossingsSafe : S.Configuration × State ~→ Bool
crossingsSafe(con,s) ≡
(
    ∀ sb : T.SBID •
        S.getSBType(sb,con) = T.CROSSINGSB ∧
        trainOnSensor(sb,con,s) ⇒
            getBarrierPosition(sb,s,con) = T.DOWN
),

/* Wellformedness */
is_wf : State × S.Configuration → Bool
is_wf(s,con) ≡
    allStatesExists(con,s),

/* All states must exist to be wf */
allStatesExists : S.Configuration × State → Bool
allStatesExists(con,s) ≡
    allTrainStatesExist(s) ∧
    train_pos_wf(con,s) ∧
    allCrossingStatesExist(con,s)  ∧
    allPointStatesExist(con,s)   ∧
    allSensorStatesExist(s),

/* All trains must have a state */
allTrainStatesExist : State → Bool
allTrainStatesExist(s) ≡
(
    ∀ trainID : T.TrainID •
        trainStateExists(trainID,s)
),

/* Front and rear position of a train must be exactly
     'train length' apart */
train_pos_wf : S.Configuration × State ~→ Bool
train_pos_wf(con,s) ≡
(
    ∀ t : T.TrainID •
        train_pos_ok(t,getTrainPosition(t,s),s,con)
),

/* All crossings must have a state */
allCrossingStatesExist : S.Configuration × State → Bool
allCrossingStatesExist(con,s) ≡
(
    ∀ cr : T.SBID •
        S.getSBType(cr,con) = T.CROSSINGSB ⇒
            crossingStateExists(cr,s,con)
),

/* All points must have a state */
allPointStatesExist : S.Configuration × State → Bool
allPointStatesExist(con,s) ≡
(
    ∀ p : T.SBID •
        S.getSBType(p,con) = T.POINTSB ⇒
            pointStateExists(p,s,con)
```

```
),

/* All sensors must have a state */
allSensorStatesExist : State → Bool
allSensorStatesExist(s) ≡
(
    ∀ sen : T.SBID •
        sensorStateExists(sen,s)
),

init_req : State × S.Configuration ~→ Bool
init_req(s,con) ≡
    is_wf(s,con) ∧
    allTrainsInESA(s) ∧
    allTrainsFacingLine(s) ∧
    allTrainsStopped(s) ∧
    allBarriersUp(con,s) ∧
    allPointsNotShifting(con,s),

allTrainsInESA : State ~→ Bool
allTrainsInESA(s) ≡
(
    ∀ t : T.TrainID •
        trainInESA(t,s)
),

/* All trains must initially face the railway line */
allTrainsFacingLine : State ~→ Bool
allTrainsFacingLine(s) ≡
(
    ∀ t : T.TrainID, esa : T.ESAID •
        T.isESA(esa) = T.getLoc(T.frontPos(getTrainPosition(t,s))) ∧
        (esa = T.LOW ⇒ getTrainDirection(t,s) = T.UP) ∧
        (esa = T.HIGH ⇒ getTrainDirection(t,s) = T.DOWN)
),

allTrainsStopped : State ~→ Bool
allTrainsStopped(s) ≡
(
    ∀ t : T.TrainID •
        getTrainSpeed(t,s) = 0.0 ∧
        getTrainAcc(t,s) = 0.0
),

allBarriersUp : S.Configuration × State ~→ Bool
allBarriersUp(con,s) ≡
(
    ∀ sb : T.SBID •
        S.getSBType(sb,con) = T.CROSSINGSB ⇒
            getBarrierPosition(sb,s,con) = T.UP
),

allPointsNotShifting : S.Configuration × State ~→ Bool
allPointsNotShifting(con,s) ≡
(
    ∀ sb : T.SBID •
```

```
        S.getSBType(sb,con) = T.POINTSB ⇒
            getPointPosition(sb,s,con) ∈ { T.UP, T.DOWN }
  )

axiom
  [ wellformedness ]
      init_req(initState,S.conf),

  /**
  * Observer_generator axioms
  **/

  /* getPointPosition_gen */

  [ getPointPosition_setPointPosition ]
  ∀ sb1,sb2 : T.SBID, pp : T.PointPosition,
        s : State,
        con : S.Configuration •
      getPointPosition(sb1,setPointPosition(sb2,pp,s,con),con) ≡
        if(sb1 = sb2)
        then
            pp
        else
            getPointPosition(sb1,s,con)
        end
  pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con) ∧
        ∼trainOnJunction(sb2,con,s),

  [ getPointPosition_setPointTicks ]
  ∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : State,
        con : S.Configuration •
      getPointPosition(sb1,setPointTicks(sb2,ticks,s,con),con) ≡
        getPointPosition(sb1,s,con)
  pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

  [ getPointPosition_setBarrierPosition ]
  ∀ sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        s : State,
        con : S.Configuration •
      getPointPosition(sb1,setBarrierPosition(sb2,bp,s,con),con) ≡
        getPointPosition(sb1,s,con)
  pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        pointStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

  [ getPointPosition_setSignalStatus ]
  ∀ sb1,sb2 : T.SBID, ss : T.SignalStatus,
        s : State,
        con : S.Configuration •
```

```
    getPointPosition(sb1,setSignalStatus(sb2,ss,s,con),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        pointStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[ getPointPosition_setSensorStatus ]
∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
        s : State,
        con : S.Configuration •
    getPointPosition(sb1,setSensorStatus(sb2,ss,s,con),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        sensor_guard(sb2,ss,con,s) ∧
        pointStateExists(sb1,s,con) ∧
        sensorStateExists(sb2,s),

[ getPointPosition_setTrainAcc ]
∀ sb1 : T.SBID, t : T.TrainID, acc : T.Acceleration,
        s : State,
        con : S.Configuration •
    getPointPosition(sb1,setTrainAcc(t,acc,s,con),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        acc ≤ S.getTrainMaxAcc(t,con) ∧
        S.getTrainMaxDec(t,con) ≤ acc,

[ getPointPosition_setTrainSpeed ]
∀ sb1 : T.SBID, t : T.TrainID, sp : T.Speed,
        s : State,
        con : S.Configuration •
    getPointPosition(sb1,setTrainSpeed(t,sp,s,con),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        sp ≤ S.getTrainMaxSpeed(t,con),

[ getPointPosition_setTrainPosition ]
∀ sb1 : T.SBID, t : T.TrainID, pos : T.TrainPosition,
        s : State,
        con : S.Configuration •
    getPointPosition(sb1,setTrainPosition(t,pos,s,con),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        ∼trainPositionOccupied(t,pos,s,con) ∧
        ∼tpDerailed(pos,getTrainDirection(t,s),s,con) ∧
        train_pos_ok(t,pos,s,con),

[ getPointPosition_setTrainDirection ]
∀ sb1 : T.SBID, t : T.TrainID, dir : T.Direction,
        s : State,
```

```
        con : S.Configuration •
    getPointPosition(sb1,setTrainDirection(t,dir,s),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        (
            getTrainSpeed(t,s) = 0.0 ∨
            getTrainDirection(t,s) = dir
        ),

/∗ getPointTicks_gen ∗/

[getPointTicks_setPointPosition]
∀ sb1,sb2 : T.SBID, pp : T.PointPosition,
        s : State,
        con : S.Configuration •
    getPointTicks(sb1,setPointPosition(sb2,pp,s,con),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con) ∧
        ∼trainOnJunction(sb2,con,s),

[getPointTicks_setPointTicks]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : State,
        con : S.Configuration •
    getPointTicks(sb1,setPointTicks(sb2,ticks,s,con),con) ≡
        if(sb1 = sb2)
        then
            ticks
        else
            getPointTicks(sb1,s,con)
        end
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[getPointTicks_setBarrierPosition]
∀ sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        s : State,
        con : S.Configuration •
    getPointTicks(sb1,setBarrierPosition(sb2,bp,s,con),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        pointStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[getPointTicks_setSignalStatus]
∀ sb1,sb2 : T.SBID, ss : T.SignalStatus,
        s : State,
        con : S.Configuration •
    getPointTicks(sb1,setSignalStatus(sb2,ss,s,con),con) ≡
```

```
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        pointStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[ getPointTicks_setSensorStatus ]
∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
        s : State,
        con : S.Configuration •
    getPointTicks(sb1,setSensorStatus(sb2,ss,s,con),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        sensor_guard(sb2,ss,con,s) ∧
        pointStateExists(sb1,s,con) ∧
        sensorStateExists(sb2,s),

[ getPointTicks_setTrainAcc ]
∀ sb1 : T.SBID, t : T.TrainID, acc : T.Acceleration,
        s : State,
        con : S.Configuration •
    getPointTicks(sb1,setTrainAcc(t,acc,s,con),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        acc ≤ S.getTrainMaxAcc(t,con) ∧
        S.getTrainMaxDec(t,con) ≤ acc,

[ getPointTicks_setTrainSpeed ]
∀ sb1 : T.SBID, t : T.TrainID, sp : T.Speed,
        s : State,
        con : S.Configuration •
    getPointTicks(sb1,setTrainSpeed(t,sp,s,con),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        sp ≤ S.getTrainMaxSpeed(t,con),

[ getPointTicks_setTrainPosition ]
∀ sb1 : T.SBID, t : T.TrainID, pos : T.TrainPosition,
        s : State,
        con : S.Configuration •
    getPointTicks(sb1,setTrainPosition(t,pos,s,con),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        ∼trainPositionOccupied(t,pos,s,con) ∧
        ∼tpDerailed(pos,getTrainDirection(t,s),s,con) ∧
        train_pos_ok(t,pos,s,con),

[ getPointTicks_setTrainDirection ]
∀ sb1 : T.SBID, t : T.TrainID, dir : T.Direction,
        s : State,
        con : S.Configuration •
```

```
    getPointTicks(sb1,setTrainDirection(t,dir,s),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        (
            getTrainSpeed(t,s) = 0.0 ∨
            getTrainDirection(t,s) = dir
        ),

/* getBarrierPosition_gen */

[ getBarrierPosition_setPointPosition ]
∀ sb1,sb2 : T.SBID, pp : T.PointPosition,
        s : State,
        con : S.Configuration •
    getBarrierPosition(sb1,setPointPosition(sb2,pp,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        crossingStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con) ∧
        ∼trainOnJunction(sb2,con,s),

[ getBarrierPosition_setPointTicks ]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : State,
        con : S.Configuration •
    getBarrierPosition(sb1,setPointTicks(sb2,ticks,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        crossingStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[ getBarrierPosition_setBarrierPosition ]
∀ s : State, sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        con : S.Configuration •
    getBarrierPosition(sb1,setBarrierPosition(sb2,bp,s,con),con) ≡
        if(sb1 = sb2)
        then
            bp
        else
            getBarrierPosition(sb1,s,con)
        end
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[ getBarrierPosition_setSignalStatus ]
∀ sb1,sb2 : T.SBID, ss : T.SignalStatus,
        s : State,
        con : S.Configuration •
    getBarrierPosition(sb1,setSignalStatus(sb2,ss,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
```

```
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[ getBarrierPosition_setSensorStatus ]
∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
        s : State,
        con : S.Configuration •
    getBarrierPosition(sb1,setSensorStatus(sb2,ss,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        sensor_guard(sb2,ss,con,s) ∧
        crossingStateExists(sb1,s,con) ∧
        sensorStateExists(sb2,s),

[ getBarrierPosition_setTrainAcc ]
∀ sb1 : T.SBID, t : T.TrainID, acc : T.Acceleration,
        s : State,
        con : S.Configuration •
    getBarrierPosition(sb1,setTrainAcc(t,acc,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        acc ≤ S.getTrainMaxAcc(t,con) ∧
        S.getTrainMaxDec(t,con) ≤ acc,

[ getBarrierPosition_setTrainSpeed ]
∀ sb1 : T.SBID, t : T.TrainID, sp : T.Speed,
        s : State,
        con : S.Configuration •
    getBarrierPosition(sb1,setTrainSpeed(t,sp,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        sp ≤ S.getTrainMaxSpeed(t,con),

[ getBarrierPosition_setTrainPosition ]
∀ sb1 : T.SBID, t : T.TrainID, pos : T.TrainPosition,
        s : State,
        con : S.Configuration •
    getBarrierPosition(sb1,setTrainPosition(t,pos,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        ∼trainPositionOccupied(t,pos,s,con) ∧
        ∼tpDerailed(pos,getTrainDirection(t,s),s,con) ∧
        train_pos_ok(t,pos,s,con),

[ getBarrierPosition_setTrainDirection ]
∀ sb1 : T.SBID, t : T.TrainID, dir : T.Direction,
        s : State,
        con : S.Configuration •
    getBarrierPosition(sb1,setTrainDirection(t,dir,s),con) ≡
        getBarrierPosition(sb1,s,con)
```

```
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        (
            getTrainSpeed(t,s) = 0.0 ∨
            getTrainDirection(t,s) = dir
        ),


/* getSignalStatus_gen */

[ getSignalStatus_setPointPosition ]
∀ sb1,sb2 : T.SBID, pp : T.PointPosition,
        s : State,
        con : S.Configuration •
    getSignalStatus(sb1,setPointPosition(sb2,pp,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        crossingStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con) ∧
        ∼trainOnJunction(sb2,con,s),

[ getSignalStatus_setPointTicks ]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : State,
        con : S.Configuration •
    getSignalStatus(sb1,setPointTicks(sb2,ticks,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        crossingStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[ getSignalStatus_setBarrierPosition ]
∀ s : State, sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        con : S.Configuration •
    getSignalStatus(sb1,setBarrierPosition(sb2,bp,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[ getSignalStatus_setSignalStatus ]
∀ s : State, sb1,sb2 : T.SBID, ss : T.SignalStatus,
    con : S.Configuration •
    getSignalStatus(sb1,setSignalStatus(sb2,ss,s,con),con) ≡
        if(sb1 = sb2)
        then
            ss
        else
            getSignalStatus(sb1,s,con)
        end
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
```

```
        crossingStateExists(sb2,s,con),

[ getSignalStatus_setSensorStatus ]
∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
        s : State,
        con : S.Configuration •
    getSignalStatus(sb1,setSensorStatus(sb2,ss,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        sensor_guard(sb2,ss,con,s) ∧
        crossingStateExists(sb1,s,con) ∧
        sensorStateExists(sb2,s),

[ getSignalStatus_setTrainAcc ]
∀ sb1 : T.SBID, t : T.TrainID, acc : T.Acceleration,
        s : State,
        con : S.Configuration •
    getSignalStatus(sb1,setTrainAcc(t,acc,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        acc ≤ S.getTrainMaxAcc(t,con) ∧
        S.getTrainMaxDec(t,con) ≤ acc,

[ getSignalStatus_setTrainSpeed ]
∀ sb1 : T.SBID, t : T.TrainID, sp : T.Speed,
        s : State,
        con : S.Configuration •
    getSignalStatus(sb1,setTrainSpeed(t,sp,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        sp ≤ S.getTrainMaxSpeed(t,con),

[ getSignalStatus_setTrainPosition ]
∀ sb1 : T.SBID, t : T.TrainID, pos : T.TrainPosition,
        s : State,
        con : S.Configuration •
    getSignalStatus(sb1,setTrainPosition(t,pos,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        trainStateExists(t,s) ∧
        ∼trainPositionOccupied(t,pos,s,con) ∧
        ∼tpDerailed(pos,getTrainDirection(t,s),s,con) ∧
        train_pos_ok(t,pos,s,con),

[ getSignalStatus_setTrainDirection ]
∀ sb1 : T.SBID, t : T.TrainID, dir : T.Direction,
        s : State,
        con : S.Configuration •
    getSignalStatus(sb1,setTrainDirection(t,dir,s),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
```

```
        trainStateExists(t,s) ∧
        (
            getTrainSpeed(t,s) = 0.0 ∨
            getTrainDirection(t,s) = dir
        ),

/* getSensorStatus_gen */

[ getSensorStatus_setPointPosition ]
∀ sb1,sb2 : T.SBID, pp : T.PointPosition,
        s : State,
        con : S.Configuration •
    getSensorStatus(sb1,setPointPosition(sb2,pp,s,con)) ≡
        getSensorStatus(sb1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        sensorStateExists(sb1,s) ∧
        pointStateExists(sb2,s,con) ∧
        ~trainOnJunction(sb2,con,s),

[ getSensorStatus_setPointTicks ]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : State,
        con : S.Configuration •
    getSensorStatus(sb1,setPointTicks(sb2,ticks,s,con)) ≡
        getSensorStatus(sb1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        sensorStateExists(sb1,s) ∧
        pointStateExists(sb2,s,con),

[ getSensorStatus_setBarrierPosition ]
∀ s : State, sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        con : S.Configuration •
    getSensorStatus(sb1,setBarrierPosition(sb2,bp,s,con)) ≡
        getSensorStatus(sb1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
        sensorStateExists(sb1,s) ∧
        crossingStateExists(sb2,s,con),

[ getSensorStatus_setSignalStatus ]
∀ s : State, sb1,sb2 : T.SBID, ss : T.SignalStatus,
    con : S.Configuration •
    getSensorStatus(sb1,setSignalStatus(sb2,ss,s,con)) ≡
        getSensorStatus(sb1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
        sensorStateExists(sb1,s) ∧
        crossingStateExists(sb2,s,con),

[ getSensorStatus_setSensorStatus ]
∀ s : State, sb1,sb2 : T.SBID, ss : T.SensorStatus,
        con : S.Configuration •
    getSensorStatus(sb1,setSensorStatus(sb2,ss,s,con)) ≡
        if(sb1 = sb2)
        then
            ss
        else
            getSensorStatus(sb1,s)
        end
```

```
pre sensor_guard(sb2,ss,con,s) ∧
        sensorStateExists(sb1,s) ∧
        sensorStateExists(sb2,s),

[ getSensorStatus_setTrainAcc ]
∀ sb1 : T.SBID, t : T.TrainID, acc : T.Acceleration,
        s : State,
        con : S.Configuration •
    getSensorStatus(sb1,setTrainAcc(t,acc,s,con)) ≡
        getSensorStatus(sb1,s)
pre sensorStateExists(sb1,s) ∧
        trainStateExists(t,s) ∧
        acc ≤ S.getTrainMaxAcc(t,con) ∧
        S.getTrainMaxDec(t,con) ≤ acc,

[ getSensorStatus_setTrainSpeed ]
∀ sb1 : T.SBID, t : T.TrainID, sp : T.Speed,
        s : State,
        con : S.Configuration •
    getSensorStatus(sb1,setTrainSpeed(t,sp,s,con)) ≡
        getSensorStatus(sb1,s)
pre sensorStateExists(sb1,s) ∧
        trainStateExists(t,s) ∧
        sp ≤ S.getTrainMaxSpeed(t,con),

[ getSensorStatus_setTrainPosition ]
∀ sb1 : T.SBID, t : T.TrainID, pos : T.TrainPosition,
        s : State,
        con : S.Configuration •
    getSensorStatus(sb1,setTrainPosition(t,pos,s,con)) ≡
        getSensorStatus(sb1,s)
pre sensorStateExists(sb1,s) ∧
        trainStateExists(t,s) ∧
        ∼trainPositionOccupied(t,pos,s,con) ∧
        ∼tpDerailed(pos,getTrainDirection(t,s),s,con) ∧
        train_pos_ok(t,pos,s,con),

[ getSensorStatus_setTrainDirection ]
∀ sb1 : T.SBID, t : T.TrainID, dir : T.Direction,
        s : State,
        con : S.Configuration •
    getSensorStatus(sb1,setTrainDirection(t,dir,s)) ≡
        getSensorStatus(sb1,s)
pre sensorStateExists(sb1,s) ∧
        trainStateExists(t,s) ∧
        (
            getTrainSpeed(t,s) = 0.0 ∨
            getTrainDirection(t,s) = dir
        ),

/∗ getTrainAcc_gen ∗/

[ getTrainAcc_setPointPosition ]
∀ t1 : T.TrainID, sb2 : T.SBID, pp : T.PointPosition,
        s : State,
        con : S.Configuration •
    getTrainAcc(t1,setPointPosition(sb2,pp,s,con)) ≡
```

```
        getTrainAcc(t1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        trainStateExists(t1,s) ∧
        pointStateExists(sb2,s,con) ∧
        ∼trainOnJunction(sb2,con,s),

[ getTrainAcc_setPointTicks ]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : State,
        con : S.Configuration •
    getTrainAcc(sb1,setPointTicks(sb2,ticks,s,con)) ≡
        getTrainAcc(sb1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        trainStateExists(sb1,s) ∧
        pointStateExists(sb2,s,con),

[ getTrainAcc_setBarrierPosition ]
∀ t1 : T.TrainID, sb2 : T.SBID, bp : T.BarrierPosition,
        s : State,
        con : S.Configuration •
    getTrainAcc(t1,setBarrierPosition(sb2,bp,s,con)) ≡
        getTrainAcc(t1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
        trainStateExists(t1,s) ∧
        crossingStateExists(sb2,s,con),

[ getTrainAcc_setSignalStatus ]
∀ t1 : T.TrainID, sb2 : T.SBID, ss : T.SignalStatus,
        s : State,
        con : S.Configuration •
    getTrainAcc(t1,setSignalStatus(sb2,ss,s,con)) ≡
        getTrainAcc(t1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
        trainStateExists(t1,s) ∧
        crossingStateExists(sb2,s,con),

[ getTrainAcc_setSensorStatus ]
∀ t1 : T.TrainID, sb2 : T.SBID, ss : T.SensorStatus,
        s : State,
        con : S.Configuration •
    getTrainAcc(t1,setSensorStatus(sb2,ss,s,con)) ≡
        getTrainAcc(t1,s)
pre sensor_guard(sb2,ss,con,s) ∧
        trainStateExists(t1,s) ∧
        sensorStateExists(sb2,s),

[ getTrainAcc_setTrainAcc ]
∀ t1,t2 : T.TrainID, acc : T.Acceleration,
        s : State,
        con : S.Configuration •
    getTrainAcc(t1,setTrainAcc(t2,acc,s,con)) ≡
        if(t1 = t2)
        then
            acc
        else
            getTrainAcc(t1,s)
        end
```

```
pre acc ≤ S.getTrainMaxAcc(t2,con) ∧
        S.getTrainMaxDec(t2,con) ≤ acc ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainAcc_setTrainSpeed ]
∀ t1,t2 : T.TrainID, sp : T.Speed,
        s : State,
        con : S.Configuration •
    getTrainAcc(t1,setTrainSpeed(t2,sp,s,con)) ≡
        getTrainAcc(t1,s)
pre trainStateExists(t1,s) ∧
        trainStateExists(t2,s) ∧
        sp ≤ S.getTrainMaxSpeed(t2,con),

[ getTrainAcc_setTrainPosition ]
∀ t1,t2 : T.TrainID, pos : T.TrainPosition,
        s : State,
        con : S.Configuration •
    getTrainAcc(t1,setTrainPosition(t2,pos,s,con)) ≡
        getTrainAcc(t1,s)
pre trainStateExists(t1,s) ∧
        trainStateExists(t2,s) ∧
        ∼trainPositionOccupied(t2,pos,s,con) ∧
        ∼tpDerailed(pos,getTrainDirection(t2,s),s,con) ∧
        train_pos_ok(t2,pos,s,con),

[ getTrainAcc_setTrainDirection ]
∀ t1,t2 : T.TrainID, dir : T.Direction,
        s : State,
        con : S.Configuration •
    getTrainAcc(t1,setTrainDirection(t2,dir,s)) ≡
        getTrainAcc(t1,s)
pre trainStateExists(t1,s) ∧
        trainStateExists(t2,s) ∧
        (
            getTrainSpeed(t2,s) = 0.0 ∨
            getTrainDirection(t2,s) = dir
        ),

/∗ getTrainSpeed_gen ∗/

[ getTrainSpeed_setPointPosition ]
∀ t1 : T.TrainID, sb2 : T.SBID, pp : T.PointPosition,
        s : State,
        con : S.Configuration •
    getTrainSpeed(t1,setPointPosition(sb2,pp,s,con)) ≡
        getTrainSpeed(t1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        trainStateExists(t1,s) ∧
        pointStateExists(sb2,s,con) ∧
        ∼trainOnJunction(sb2,con,s),

[ getTrainSpeed_setPointTicks ]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : State,
        con : S.Configuration •
```

```
    getTrainSpeed(sb1,setPointTicks(sb2,ticks,s,con)) ≡
        getTrainSpeed(sb1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        trainStateExists(sb1,s) ∧
        pointStateExists(sb2,s,con),

[ getTrainSpeed_setBarrierPosition ]
∀ t1 : T.TrainID, sb2 : T.SBID, bp : T.BarrierPosition,
        s : State,
        con : S.Configuration •
    getTrainSpeed(t1,setBarrierPosition(sb2,bp,s,con)) ≡
        getTrainSpeed(t1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
        trainStateExists(t1,s) ∧
        crossingStateExists(sb2,s,con),

[ getTrainSpeed_setSignalStatus ]
∀ t1 : T.TrainID, sb2 : T.SBID, ss : T.SignalStatus,
        s : State,
        con : S.Configuration •
    getTrainSpeed(t1,setSignalStatus(sb2,ss,s,con)) ≡
        getTrainSpeed(t1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
        trainStateExists(t1,s) ∧
        crossingStateExists(sb2,s,con),

[ getTrainSpeed_setSensorStatus ]
∀ t1 : T.TrainID, sb2 : T.SBID, ss : T.SensorStatus,
        s : State,
        con : S.Configuration •
    getTrainSpeed(t1,setSensorStatus(sb2,ss,s,con)) ≡
        getTrainSpeed(t1,s)
pre sensor_guard(sb2,ss,con,s) ∧
        trainStateExists(t1,s) ∧
        sensorStateExists(sb2,s),

[ getTrainSpeed_setTrainAcc ]
∀ t1,t2 : T.TrainID, acc : T.Acceleration,
        s : State,
        con : S.Configuration •
    getTrainSpeed(t1,setTrainAcc(t2,acc,s,con)) ≡
        getTrainSpeed(t1,s)
pre acc ≤ S.getTrainMaxAcc(t2,con) ∧
        S.getTrainMaxDec(t2,con) ≤ acc ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainSpeed_setTrainSpeed ]
∀ s : State, t1,t2 : T.TrainID, sp : T.Speed,
    con : S.Configuration •
    getTrainSpeed(t1,setTrainSpeed(t2,sp,s,con)) ≡
        if(t1 = t2)
        then
            sp
        else
            getTrainSpeed(t1,s)
        end
```

```
pre sp ≤ S.getTrainMaxSpeed(t2,con) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainSpeed_setTrainPosition ]
∀ t1,t2 : T.TrainID, pos : T.TrainPosition,
        s : State,
        con : S.Configuration •
    getTrainSpeed(t1,setTrainPosition(t2,pos,s,con)) ≡
        getTrainSpeed(t1,s)
pre trainStateExists(t1,s) ∧
        trainStateExists(t2,s) ∧
        ∼trainPositionOccupied(t2,pos,s,con) ∧
        ∼tpDerailed(pos,getTrainDirection(t2,s),s,con) ∧
        train_pos_ok(t2,pos,s,con),

[ getTrainSpeed_setTrainDirection ]
∀ t1,t2 : T.TrainID, dir : T.Direction,
        s : State,
        con : S.Configuration •
    getTrainSpeed(t1,setTrainDirection(t2,dir,s)) ≡
        getTrainSpeed(t1,s)
pre trainStateExists(t1,s) ∧
        trainStateExists(t2,s) ∧
        (
            getTrainSpeed(t2,s) = 0.0 ∨
            getTrainDirection(t2,s) = dir
        ),

/∗ getTrainPosition_gen ∗/

[ getTrainPosition_setPointPosition ]
∀ t1 : T.TrainID, sb2 : T.SBID, pp : T.PointPosition,
        s : State,
        con : S.Configuration •
    getTrainPosition(t1,setPointPosition(sb2,pp,s,con)) ≡
        getTrainPosition(t1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        trainStateExists(t1,s) ∧
        pointStateExists(sb2,s,con) ∧
        ∼trainOnJunction(sb2,con,s),

[ getTrainPosition_setPointTicks ]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : State,
        con : S.Configuration •
    getTrainPosition(sb1,setPointTicks(sb2,ticks,s,con)) ≡
        getTrainPosition(sb1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        trainStateExists(sb1,s) ∧
        pointStateExists(sb2,s,con),

[ getTrainPosition_setBarrierPosition ]
∀ t1 : T.TrainID, sb2 : T.SBID, bp : T.BarrierPosition,
        s : State,
        con : S.Configuration •
    getTrainPosition(t1,setBarrierPosition(sb2,bp,s,con)) ≡
```

```
        getTrainPosition(t1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
        trainStateExists(t1,s) ∧
        crossingStateExists(sb2,s,con),

[ getTrainPosition_setSignalStatus ]
∀ t1 : T.TrainID, sb2 : T.SBID, ss : T.SignalStatus,
        s : State,
        con : S.Configuration •
    getTrainPosition(t1,setSignalStatus(sb2,ss,s,con)) ≡
        getTrainPosition(t1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
        trainStateExists(t1,s) ∧
        crossingStateExists(sb2,s,con),

[ getTrainPosition_setSensorStatus ]
∀ t1 : T.TrainID, sb2 : T.SBID, ss : T.SensorStatus,
        s : State,
        con : S.Configuration •
    getTrainPosition(t1,setSensorStatus(sb2,ss,s,con)) ≡
        getTrainPosition(t1,s)
pre sensor_guard(sb2,ss,con,s) ∧
        trainStateExists(t1,s) ∧
        sensorStateExists(sb2,s),

[ getTrainPosition_setTrainAcc ]
∀ t1,t2 : T.TrainID, acc : T.Acceleration,
        s : State,
        con : S.Configuration •
    getTrainPosition(t1,setTrainAcc(t2,acc,s,con)) ≡
        getTrainPosition(t1,s)
pre acc ≤ S.getTrainMaxAcc(t2,con) ∧
        S.getTrainMaxDec(t2,con) ≤ acc ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainPosition_setTrainSpeed ]
∀ s : State, t1,t2 : T.TrainID, sp : T.Speed,
    con : S.Configuration •
    getTrainPosition(t1,setTrainSpeed(t2,sp,s,con)) ≡
        getTrainPosition(t1,s)
pre sp ≤ S.getTrainMaxSpeed(t2,con) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainPosition_setTrainPosition ]
∀ s : State, t1,t2 : T.TrainID, tp : T.TrainPosition,
        con : S.Configuration •
    getTrainPosition(t1,setTrainPosition(t2,tp,s,con)) ≡
        if(t1 = t2)
        then
            tp
        else
            getTrainPosition(t1,s)
        end
pre ∼trainPositionOccupied(t2,tp,s,con) ∧
        ∼tpDerailed(tp,getTrainDirection(t2,s),s,con) ∧
```

```
        train_pos_ok(t2,tp,s,con) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainPosition_setTrainDirection ]
∀ t1,t2 : T.TrainID, dir : T.Direction,
        s : State,
        con : S.Configuration •
    getTrainPosition(t1,setTrainDirection(t2,dir,s)) ≡
        getTrainPosition(t1,s)
pre trainStateExists(t1,s) ∧
        trainStateExists(t2,s) ∧
        (
            getTrainSpeed(t2,s) = 0.0 ∨
            getTrainDirection(t2,s) = dir
        ),

/∗ getTrainDirection_gen ∗/

[ getTrainDirection_setPointPosition ]
∀ t1 : T.TrainID, sb2 : T.SBID, pp : T.PointPosition,
        s : State,
        con : S.Configuration •
    getTrainDirection(t1,setPointPosition(sb2,pp,s,con)) ≡
        getTrainDirection(t1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        trainStateExists(t1,s) ∧
        pointStateExists(sb2,s,con) ∧
        ∼trainOnJunction(sb2,con,s),

[ getTrainDirection_setPointTicks ]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : State,
        con : S.Configuration •
    getTrainDirection(sb1,setPointTicks(sb2,ticks,s,con)) ≡
        getTrainDirection(sb1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        trainStateExists(sb1,s) ∧
        pointStateExists(sb2,s,con),

[ getTrainDirection_setBarrierPosition ]
∀ t1 : T.TrainID, sb2 : T.SBID, bp : T.BarrierPosition,
        s : State,
        con : S.Configuration •
    getTrainDirection(t1,setBarrierPosition(sb2,bp,s,con)) ≡
        getTrainDirection(t1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
        trainStateExists(t1,s) ∧
        crossingStateExists(sb2,s,con),

[ getTrainDirection_setSignalStatus ]
∀ t1 : T.TrainID, sb2 : T.SBID, ss : T.SignalStatus,
        s : State,
        con : S.Configuration •
    getTrainDirection(t1,setSignalStatus(sb2,ss,s,con)) ≡
        getTrainDirection(t1,s)
pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
```

```
        trainStateExists(t1,s) ∧
        crossingStateExists(sb2,s,con),

[ getTrainDirection_setSensorStatus ]
∀ t1 : T.TrainID, sb2 : T.SBID, ss : T.SensorStatus,
        s : State,
        con : S.Configuration •
    getTrainDirection(t1,setSensorStatus(sb2,ss,s,con)) ≡
        getTrainDirection(t1,s)
pre sensor_guard(sb2,ss,con,s) ∧
        trainStateExists(t1,s) ∧
        sensorStateExists(sb2,s),

[ getTrainDirection_setTrainAcc ]
∀ t1,t2 : T.TrainID, acc : T.Acceleration,
        s : State,
        con : S.Configuration •
    getTrainDirection(t1,setTrainAcc(t2,acc,s,con)) ≡
        getTrainDirection(t1,s)
pre acc ≤ S.getTrainMaxAcc(t2,con) ∧
        S.getTrainMaxDec(t2,con) ≤ acc ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainDirection_setTrainSpeed ]
∀ s : State, t1,t2 : T.TrainID, sp : T.Speed,
    con : S.Configuration •
    getTrainDirection(t1,setTrainSpeed(t2,sp,s,con)) ≡
        getTrainDirection(t1,s)
pre sp ≤ S.getTrainMaxSpeed(t2,con) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainDirection_setTrainPosition ]
∀ s : State, t1,t2 : T.TrainID, pos : T.TrainPosition,
        con : S.Configuration •
    getTrainDirection(t1,setTrainPosition(t2,pos,s,con)) ≡
        getTrainDirection(t1,s)
pre ∼trainPositionOccupied(t2,pos,s,con) ∧
        ∼tpDerailed(pos,getTrainDirection(t2,s),s,con) ∧
        train_pos_ok(t2,pos,s,con) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainDirection_setTrainDirection ]
∀ s : State, t1,t2 : T.TrainID, dir : T.Direction•
    getTrainDirection(t1,setTrainDirection(t2,dir,s)) ≡
        if(t1 = t2)
        then
            dir
        else
            getTrainDirection(t1,s)
        end
pre (getTrainSpeed(t2,s) = 0.0 ∨
        getTrainDirection(t2,s) = dir) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),
```

```
/* end of obs_gen axioms */

    /** Maintaining the wellformedness of the state when
    *   applied to a generator with its precondition
    *   General form:
    *     is_wf(state) ∧ pre_cond(..,state) ⇒
    *       is_wf(gen_(state))
    **/

    [ gen_wf_setPointPosition ]
    ∀ p : T.SBID, pp : T.PointPosition, s : State,
          con : S.Configuration •
       is_wf(s,con) ∧
       S.getSBType(p,con) = T.POINTSB ∧
       ∼trainOnJunction(p,con,s)
          ⇒
       is_wf(setPointPosition(p,pp,s,con),con),

    [ gen_wf_setPointTicks ]
    ∀ p : T.SBID, ticks : T.Tick, s : State,
          con : S.Configuration •
       is_wf(s,con) ∧
       S.getSBType(p,con) = T.POINTSB
          ⇒
       is_wf(setPointTicks(p,ticks,s,con),con),

    [ gen_wf_setBarrierPosition ]
    ∀ cr : T.SBID, bp : T.BarrierPosition, s : State,
          con : S.Configuration •
       is_wf(s,con) ∧
       S.getSBType(cr,con) = T.CROSSINGSB
          ⇒
       is_wf(setBarrierPosition(cr,bp,s,con),con),

    [ gen_wf_setSignalStatus ]
    ∀ cr : T.SBID, ss : T.SignalStatus, s : State,
          con : S.Configuration •
       is_wf(s,con) ∧
       S.getSBType(cr,con) = T.CROSSINGSB
          ⇒
       is_wf(setSignalStatus(cr,ss,s,con),con),

    [ gen_wf_setSensorStatus ]
    ∀ sen : T.SBID, ss : T.SensorStatus, s : State,
          con : S.Configuration •
       is_wf(s,con) ∧
       sensor_guard(sen,ss,con,s)
          ⇒
       is_wf(setSensorStatus(sen,ss,s,con),con),

    [ gen_wf_setTrainAcc ]
    ∀ t : T.TrainID, acc : T.Acceleration,
       con : S.Configuration, s : State •
       is_wf(s,con) ∧
```

```
        acc ≤ S.getTrainMaxAcc(t,con) ∧
        S.getTrainMaxDec(t,con) ≤ acc
            ⇒
        is_wf(setTrainAcc(t,acc,s,con),con),

    [gen_wf_setTrainSpeed]
    ∀ t : T.TrainID, sp : T.Speed,
        con : S.Configuration, s : State •
        is_wf(s,con) ∧
        sp ≤ S.getTrainMaxSpeed(t,con)
            ⇒
        is_wf(setTrainSpeed(t,sp,s,con),con),

    [gen_wf_setTrainPosition]
    ∀ t : T.TrainID, pos : T.TrainPosition, s : State,
            con : S.Configuration •
        is_wf(s,con) ∧
        ∼trainPositionOccupied(t,pos,s,con) ∧
        train_pos_ok(t,pos,s,con)
            ⇒
        is_wf(setTrainPosition(t,pos,s,con),con),

    [gen_wf_setTrainDirection]
    ∀ t : T.TrainID, dir : T.Direction, s : State,
            con : S.Configuration •
        is_wf(s,con) ∧
        (
            getTrainSpeed(t,s) = 0.0 ∨
            getTrainDirection(t,s) = dir
        )
            ⇒
        is_wf(setTrainDirection(t,dir,s),con)

end
```

F.1.4 Control

```
context: AA_Dynamics0
scheme AA_Control0(T : AA_Types0, S : AA_Statics0(T),
                                        D : AA_Dynamics0(T,S)) =
  class
    type
      ControlState

    value
      initControlState : ControlState,

      getSBCCLineRes : T.SBID × ControlState ~→ T.HasRes,
      getSBCCBranchRes : T.SBID × ControlState ~→ T.HasRes,

      setSBCCLineRes : T.SBID × T.HasRes ×
                                ControlState ~→ ControlState,
      setSBCCBranchRes : T.SBID × T.HasRes ×
                                ControlState ~→ ControlState,
```

```
getLastSensorStatus : T.SBID × ControlState ~→
                                        T.SensorStatus,
setLastSensorStatus : T.SBID × T.SensorStatus ×
                                   ControlState ~→ ControlState,

getNextSBCCMsg : T.SBID × ControlState ~→
                                   T.HasComMsg × ControlState,
storeSBCCMsg : T.SBID × T.ComMsg ×
                                   ControlState ~→ ControlState,

getSBCCPrepRes : T.SBID × ControlState ~→ T.HasRes,
setSBCCPrepRes : T.SBID × T.HasRes ×
                                   ControlState ~→ ControlState,

sbccStateExists : T.SBID × ControlState → Bool,

hasTCCRes : T.TrainID × ControlState ~→ Bool,
setTCCRes : T.TrainID × Bool × ControlState ~→
                                        ControlState,

isTCCRequesting : T.TrainID × ControlState ~→ Bool,
setTCCRequesting : T.TrainID × Bool ×
                                   ControlState ~→ ControlState,

isTrainDecelerating : T.TrainID × ControlState ~→ Bool,
setTrainDecelerating : T.TrainID × Bool ×
                                   ControlState ~→ ControlState,

hasPassedResPoint : T.TrainID × D.State ×
                                     S.Configuration ~→ Bool,
hasPassedBrakePoint : T.TrainID × D.State ×
                                     S.Configuration ~→ Bool,

tccStateExists : T.SBID × ControlState → Bool,

/* Processes */
tick : T.Tick × ControlState × D.State ×
                    S.Configuration ~→ ControlState × D.State
tick(tick,cs,ds,con) ≡
(
   let
      tSet = {t | t : T.TrainID},
      sbSet = {sb | sb : T.SBID},
      (cs,ds) = tickTCCs(tSet,tick,cs,ds,con),
      cs = tickSBCCs(sbSet,tick,cs,ds,con)
   in
      (cs,ds)
   end
),

tickSBCCs : T.SBID-set × T.Tick × ControlState ×
                       D.State × S.Configuration ~→ ControlState
tickSBCCs(sbSet,tick,cs,ds,con) ≡
```

```
    if (sbSet = {}) then
        cs
    else
        let
            sbcc : T.SBID • sbcc ∈ sbSet,
            sbSet = sbSet \ {sbcc},
            cs = sbccProcess(sbcc,tick,cs,ds,con)
        in
            tickSBCCs(sbSet,tick,cs,ds,con)
        end
    end,

tickTCCs : T.TrainID-set × T.Tick × ControlState ×
                                    D.State × S.Configuration ~→
                                    ControlState × D.State
tickTCCs(tccSet,tick,cs,ds,con) ≡
    if (tccSet = {}) then
        (cs,ds)
    else
        let
            tcc : T.TrainID • tcc ∈ tccSet,
            tccSet = tccSet \ {tcc},
            (cs,ds) = tccProcess(tcc,tick,cs,ds,con)
        in
            tickTCCs(tccSet,tick,cs,ds,con)
        end
    end,

/∗ Auxiliary ∗/
removeSBCCLineRes : T.SBID × ControlState ~→ ControlState
removeSBCCLineRes(sb,cs) ≡
    setSBCCLineRes(sb,T.noRes,cs),

removeSBCCBranchRes : T.SBID × ControlState ~→ ControlState
removeSBCCBranchRes(sb,cs) ≡
    setSBCCBranchRes(sb,T.noRes,cs),

removeSBCCPrepRes : T.SBID × ControlState ~→ ControlState
removeSBCCPrepRes(sb,cs) ≡
    setSBCCPrepRes(sb,T.noRes,cs),

isPreparing : T.SBID × ControlState ~→ Bool
isPreparing(sb,cs) ≡
    case getSBCCPrepRes(sb,cs) of
        T.noRes → false,
        _ → true
    end,

comService : T.ComMsg × ControlState ~→ ControlState
comService(comMsg,cs) ≡
    case T.getReceiver(comMsg) of
        T.isSB(sb) → sbccMsgReceiver(sb,comMsg,cs),
        T.isTrain(t) → tccMsgReceiver(t,comMsg,cs)
    end,

tccMsgReceiver : T.TrainID × T.ComMsg ×
```

```
                                          ControlState ~→ ControlState
tccMsgReceiver(t,comMsg,cs) ≡
    let
        resp = T.getMsg(comMsg)
    in
        case resp of
            T.segResp(resGranted) →
            (
                let
                    cs = setTCCRequesting(t,false,cs)
                in
                    if(resGranted)
                    then
                        setTCCRes(t,true,cs)
                    else
                        cs
                    end
                end
            ),
            _ → cs
        end
    end,

sbccMsgReceiver : T.SBID × T.ComMsg ×
                                          ControlState ~→ ControlState
sbccMsgReceiver(sb,comMsg,cs) ≡
    storeSBCCMsg(sb,comMsg,cs),

/* Processes */
tccProcess : T.TrainID × T.Tick × ControlState ×
                                          D.State × S.Configuration ~→
                                          ControlState × D.State
tccProcess(t,tick,cs,ds,con) ≡
    let
        (cs,ds) = checkSpeed(t,tick,cs,ds,con),
        cs = clearRes(t,cs,ds),
        (cs,ds) = handleRes(t,cs,ds,con)
    in
        (cs,ds)
    end,

checkSpeed : T.TrainID × T.Tick × ControlState ×
                                          D.State × S.Configuration ~→
                                          ControlState × D.State
checkSpeed(t,tick,cs,ds,con) ≡
    let
        dts = S.getTrainMaxAcc(t,con) * tick,
        ts = D.getTrainSpeed(t,ds) + dts,
        trainMaxSpeed = S.getTrainMaxSpeed(t,con)
    in
        /* If train is entirely in an ESA */
        if(D.trainInESA(t,ds))
        then
            if(ts > trainMaxSpeed)
            then
                decelerateTrain(t,cs,ds,con)
            else
```

```
                checkDeceleration(t,cs,ds,con)
            end
        else /* Train on segment and perhaps an ESA */
            let
                tp = D.getTrainPosition(t,ds),

                /* Will always be an IsSeg */
                isSeg = D.getTrainLoc(t,ds),

                frontSeg = T.getSeg(isSeg),
                segMaxSpeed = S.getSegMaxSpeed(frontSeg,con)
            in
                if (ts > segMaxSpeed ∨ ts > trainMaxSpeed)
                then
                    decelerateTrain(t,cs,ds,con)
                else
                    checkDeceleration(t,cs,ds,con)
                end
            end
        end
    end, /* let */

checkDeceleration : T.TrainID × ControlState ×
                                D.State × S.Configuration ∼→
                                ControlState × D.State
checkDeceleration(t,cs,ds,con) ≡
    if (isTrainDecelerating(t,cs))
    then
        let
            ds = D.setTrainAcc(t,0.0,ds,con),
            cs = setTrainDecelerating(t,false,cs)
        in
            (cs,ds)
        end
    else
        (cs,ds)
    end,

decelerateTrain : T.TrainID × ControlState ×
                                D.State × S.Configuration ∼→
                                ControlState × D.State
decelerateTrain(t,cs,ds,con) ≡
    let
        cs = setTrainDecelerating(t,true,cs),
        ds = D.decelerateTrain(t,con,ds)
    in
        (cs,ds)
    end,

accelerateTrain : T.TrainID × ControlState ×
                                D.State × S.Configuration ∼→
                                ControlState × D.State
accelerateTrain(t,cs,ds,con) ≡
    let
        cs = setTrainDecelerating(t,false,cs),
        ds = D.accelerateTrain(t,con,ds)
    in
```

```
        (cs,ds)
    end,

clearRes : T.TrainID × ControlState × D.State ~→
                                                ControlState
clearRes(t,cs,ds) ≡
    if(∼T.oneLoc(D.getTrainPosition(t,ds)))
    then
        setTCCRes(t,false,cs)
    else
        cs
    end,

handleRes : T.TrainID × ControlState ×
                                D.State × S.Configuration ~→
                                ControlState × D.State
handleRes(t,cs,ds,con) ≡
    if(hasPassedResPoint(t,ds,con))
    then
        if(∼hasTCCRes(t,cs))
        then
            (cs,ds)
        else
            let
                (cs,ds) = if(hasPassedBrakePoint(t,ds,con))
                          then decelerateTrain(t,cs,ds,con)
                          else (cs,ds) end
            in
                if(∼isTCCRequesting(t,cs))
                then
                    tccRequestRes(t,cs,ds,con)
                else
                    (cs,ds)
                end
            end
        end
    else
        (cs,ds)
    end,

tccRequestRes : T.TrainID × ControlState ×
                                D.State × S.Configuration ~→
                                ControlState × D.State
tccRequestRes(t,cs,ds,con) ≡
    let
        loc = T.frontLoc(D.getTrainPosition(t,ds)),
        dir = D.getTrainDirection(t,ds)
    in
        case loc of
            T.isESA(esa) →
            (
                /∗ If not facing line, then do nothing.
                   Will brake at brakepoint ∗/
                if (dir = T.end2Dir(esa))
                then
                    (cs,ds)
                else
```

```
                    (sendTCCReq(t,S.getESASB(esa,con),dir,cs),ds)
                end
            ),

            T.isSeg(seg) →
            (
                (sendTCCReq(t,S.getSegSB(seg,dir,con),dir,cs),ds)
            )
        end /* case */
    end, /* let */

sendTCCReq : T.TrainID × T.SBID × T.Direction ×
                                        ControlState ~→ ControlState
sendTCCReq(t,sb,dir,cs) ≡
    let
        sender = T.isTrain(t),
        receiver = T.isSB(sb),
        res = T.mk_res(t,dir),
        msg = T.segReq(res),
        comMsg = T.mk_comMsg(sender,receiver,msg),
        cs = setTCCRequesting(t,true,cs)
    in
        comService(comMsg,cs)
    end,

sbccProcess : T.SBID × T.Tick × ControlState × D.State ×
                                        S.Configuration ~→ ControlState
sbccProcess(sb,tick,cs,ds,con) ≡
    let
        cs = sensorProcess(sb,cs,ds,con)
    in
        if (isPreparing(sb,cs)) then
            prepareProcess(sb,cs,ds,con)
        else
            sbccMsgProcess(sb,cs,ds,con)
        end
    end,

/* Waits for the preparation of a segment
   before a train is allowed to enter */
prepareProcess : T.SBID × ControlState × D.State ×
                                        S.Configuration ~→ ControlState
prepareProcess(sb,cs,ds,con) ≡
    case S.getSBType(sb,con) of
        /* case POINTSB → wait for !moving */
        T.POINTSB →
        (
            if (D.getPointPosition(sb,ds,con) ∈ {T.UP, T.DOWN})
            then
                let
                    T.res(res) = getSBCCPrepRes(sb,cs),
                    train = T.getTrain(res),
                    cs = removeSBCCPrepRes(sb,cs)
                in
                    sendSBCCMsg(sb,T.isTrain(train),T.segResp(true),cs)
                end
            else
```

```
            cs
        end
    ),

    /* case crossingsb → wait for DOWN */
    T.CROSSINGSB →
    (
        if (D.getBarrierPosition(sb,ds,con) = T.DOWN) then
            let
                T.res(res) = getSBCCPrepRes(sb,cs),
                train = T.getTrain(res),
                cs = removeSBCCPrepRes(sb,cs)
            in
                sendSBCCMsg(sb,T.isTrain(train),T.segResp(true),cs)
            end
        else
            cs
        end
    ),

    _ → cs
end,

sensorProcess : T.SBID × ControlState × D.State ×
                                    S.Configuration ~→ ControlState
sensorProcess(sb,cs,ds,con) ≡
    let
        sState = D.getSensorStatus(sb,ds),
        lastState = getLastSensorStatus(sb,cs),
        cs = setLastSensorStatus(sb,sState,cs)
    in
        if((lastState = T.ACTIVE) ∧ (sState = T.INACTIVE))
        then
            let
                ds = dePrepareSeg(sb,cs,ds,con)
            in
                if(S.isLineGuard(sb,con))
                then
                    makeDeRes(sb,cs,ds,con)
                else
                    cs
                end
            end
        else
            cs
        end
    end,


dePrepareSeg : T.SBID × ControlState × D.State ×
                                        S.Configuration ~→
                                        ControlState × D.State
dePrepareSeg(sb,cs,ds,con) ≡
    let
        cs = removeSBCCPrepRes(sb,cs)
    in
        case S.getSBType(sb,con) of
```

```
            T.CROSSINGSB →
            (
                (cs,D.setBarrierPosition(sb,T.MOVINGUP,ds,con))
            ),

            _ → (cs,ds)
        end
    end,

prepareSeg : T.SBID × T.Reservation × ControlState ×
                                        D.State × S.Configuration ~→
                                        ControlState × D.State
prepareSeg(sb,res,cs,ds,con) ≡
    let
        cs = setSBCCPrepRes(sb,T.res(res),cs)
    in
        case S.getSBType(sb,con) of
            T.CROSSINGSB →
            (
                let
                    ds = D.setSignalStatus(sb,T.ON,ds,con)
                in
                    (cs,ds)
                end
            ),

            T.POINTSB →
            (
                case T.getDir(res) of
                    T.UP →
                    (
                        let
                            ds = D.setPointPosition(sb,T.MOVINGUP,ds,con)
                        in
                            (cs,ds)
                        end
                    ),

                    T.DOWN →
                    (
                        let
                            ds = D.setPointPosition(sb,T.MOVINGDOWN,ds,con)
                        in
                            (cs,ds)
                        end
                    )
                end
            ),

            _ → (cs,ds)
        end
    end,


makeDeRes : T.SBID × ControlState × D.State ×
                                        S.Configuration ~→ ControlState
makeDeRes(sb,cs,ds,con) ≡
```

```
        case S.getSBType(sb,con) of
            T.ENDSB →
            (
                let
                    T.res(lineRes) = getSBCCLineRes(sb,cs),
                    endDir = S.getEndDir(sb,con)
                in
                    if (T.getDir(lineRes) = endDir)
                    then
                        let
                            cs = removeSBCCLineRes(sb,cs)
                        in
                            sendLDeResMsg(sb,S.getOppositeGuard(sb,con),cs)
                        end
                    else
                        cs
                    end /∗ if ∗/
                end /∗ let ∗/
            ),

            T.POINTSB →
            (
                let
                    T.res(lineRes) = getSBCCLineRes(sb,cs),
                    pointDir = S.getPointDir(sb,con)
                in
                    if (T.getDir(lineRes) = pointDir)
                    then
                        let
                            cs = removeSBCCLineRes(sb,cs)
                        in
                            sendLDeResMsg(sb,S.getOppositeGuard(sb,con),cs)
                        end
                    else
                        sendBDeResMsg(sb,S.getOppositeGuard(sb,con),cs)
                    end /∗ if ∗/
                end /∗ let ∗/
            )
        end /∗ case ∗/
    pre S.isLineGuard(sb,con),

    sendLBDeResMsg : T.SBID × T.SBID × ControlState ~→
                                                    ControlState
    sendLBDeResMsg(thisSB,remoteSB,cs) ≡
        sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineBranchDeRes,cs),

    sendLDeResMsg : T.SBID × T.SBID × ControlState ~→
                                                    ControlState
    sendLDeResMsg(thisSB,remoteSB,cs) ≡
        sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineDeRes,cs),

    sendBDeResMsg : T.SBID × T.SBID × ControlState ~→
                                                    ControlState
    sendBDeResMsg(thisSB,remoteSB,cs) ≡
        sendSBCCMsg(thisSB,T.isSB(remoteSB),T.branchDeRes,cs),

    sendLBResMsg : T.SBID × T.SBID × T.Reservation ×
```

```
                                        ControlState ~→ ControlState
sendLBResMsg(thisSB,remoteSB,aRes,cs) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),
                        T.lineBranchReq(aRes),cs),

sendSBCCMsg : T.SBID × T.ComID × T.SBCCMsg ×
                                        ControlState ~→ ControlState
sendSBCCMsg(sb,receiver,sbccMsg,cs) ≡
    comService(T.mk_comMsg(T.isSB(sb),receiver,sbccMsg),cs),

sbccMsgProcess : T.SBID × ControlState × D.State ×
                                        S.Configuration ~→ ControlState
sbccMsgProcess(sb,cs,ds,con) ≡
    let
        (hasComMsg,cs) = getNextSBCCMsg(sb,cs)
    in
        case hasComMsg of
            T.comMsg(T.mk_comMsg(sender,receiver,msg)) →
            (
                case sender of
                    T.isTrain(_) →
                    (
                        let
                            (retMsg,cs,ds) = handleTCCMsg(sb,msg,cs,ds,con)
                        in
                            case retMsg of
                                T.hasMsg(aMsg) →
                                        sendSBCCMsg(sb,sender,aMsg,cs),
                                _ → cs
                            end
                        end
                    ),

                    T.isSB(_) →
                    (
                        let
                            (cs,retMsg) = handleSBCCMsg(sb,msg,cs)
                        in
                            case retMsg of
                                T.hasMsg(aMsg) →
                                        sendSBCCMsg(sb,sender,aMsg,cs),
                                _ → cs
                            end
                        end
                    )
                end /* case */
            ),
            _ → cs /* no message to process */
        end
    end, /* let */

handleSBCCMsg : T.SBID × T.Message × ControlState ~→
                                        ControlState × T.ReturnSBCCMsg
handleSBCCMsg(sb,msg,cs) ≡
    case msg of
        /* Request */
        T.lineBranchReq(_) → handleLBReq(sb,msg,cs),
```

```
        /* Response */
        T.lineBranchResp(_) → handleLBResp(sb,msg,cs),

        /* De reservation */
        T.lineBranchDeRes → handleDeResMsg(sb,msg,cs),
        T.lineDeRes → handleDeResMsg(sb,msg,cs),
        T.branchDeRes → handleDeResMsg(sb,msg,cs)
    end,

handleTCCMsg : T.SBID × T.Message × ControlState ×
                    D.State × S.Configuration ~→
                    T.ReturnSBCCMsg × ControlState × D.State
handleTCCMsg(sb,msg,cs,ds,con) ≡
    let
        T.segReq(res) = msg
    in
        case S.getSBType(sb,con) of
            T.ENDSB →
            (
                /* if direction away from end → send msg
                     else OK
                */
                if (T.getDir(res) = S.getEndDir(sb,con))
                then
                    let
                        (cs,ds) = prepareSeg(sb,res,cs,ds,con)
                    in
                        (T.noSBCCMsg,cs,ds)
                    end
                else
                    if (lineFree(sb,cs))
                    then
                        let
                            cs = setSBCCLineRes(sb,T.res(res),cs),
                            cs = sendLBResMsg(sb,
                                        S.getOppositeGuard(sb,con),res,cs)
                        in
                            (T.noSBCCMsg,cs,ds)
                        end
                    else
                        (T.hasMsg(T.segResp(false)),cs,ds)
                    end
                end
            ),

            /* If direction away from point → send msg
                 else OK
            */
            T.POINTSB →
            (
                if(T.getDir(res) = S.getPointDir(sb,con))
                then
                    let
                        (cs,ds) = prepareSeg(sb,res,cs,ds,con)
                    in
                        (T.noSBCCMsg,cs,ds)
```

```
            end
        else
            if (lineFree(sb,cs))
            then
                let
                    cs = setSBCCLineRes(sb,T.res(res),cs),
                    cs = sendLBResMsg(sb,
                                S.getOppositeGuard(sb,con),res,cs)
                in
                    (T.noSBCCMsg,cs,ds)
                end
            else
                (T.hasMsg(T.segResp(false)),cs,ds)
            end
        end
    ),

    _ → /∗ PLAINSB, CROSSINGSB ∗/
    (
        let
            (cs,ds) = prepareSeg(sb,res,cs,ds,con)
        in
            (T.noSBCCMsg,cs,ds)
        end
    )
  end /∗ case ∗/
end, /∗ let ∗/

/∗ if (!line free) then NO
   reserve line;
   if (end type) then YES
   if (!branch free) then deres line; NO
   res branch; YES;
∗/
handleLBReq : T.SBID × T.Message × ControlState ~→
                                ControlState × T.ReturnSBCCMsg
handleLBReq(sb,msg,cs) ≡
  let
    T.lineBranchReq(res) = msg
  in
    if (lineBranchFree(sb,cs))
    then
      let
        cs = setSBCCLineRes(sb,T.res(res),cs),
        cs = setSBCCBranchRes(sb,T.res(res),cs)
      in
        (cs,T.hasMsg(T.lineBranchResp(res,true)))
      end
    else
      (cs,T.hasMsg(T.lineBranchResp(res,false)))
    end
  end,

lineBranchFree : T.SBID × ControlState ~→ Bool
lineBranchFree(sb,cs) ≡
  (getSBCCLineRes(sb,cs) = T.noRes) ∧
  (getSBCCBranchRes(sb,cs) = T.noRes),
```

```
lineFree : T.SBID × ControlState ~→ Bool
lineFree(sb,cs) ≡
    (getSBCCLineRes(sb,cs) = T.noRes),

/* if (response = NO) deres; NO;
     if (!prepare_segment()) deres; NO;
     OK;
*/
handleLBResp : T.SBID × T.Message × ControlState ~→
                                        ControlState × T.ReturnSBCCMsg
handleLBResp(sb,msg,cs) ≡
    let
        T.lineBranchResp(res,granted) = msg
    in
        if (granted)
        then
            let
                cs = sendSBCCMsg(sb,T.isTrain(T.getTrain(res)),
                                            T.segResp(true),cs)
            in
                (cs,T.noSBCCMsg)
            end
        else
            let
                cs = removeSBCCLineRes(sb,cs),
                cs = sendSBCCMsg(sb,T.isTrain(T.getTrain(res)),
                                            T.segResp(false),cs)
            in
                (cs,T.noSBCCMsg)
            end
        end
    end,

/* case(msg)
            lb → deres line; deres branch
            l → deres line;
            b → deres branch;
*/
handleDeResMsg : T.SBID × T.Message × ControlState ~→
                                        ControlState × T.ReturnSBCCMsg
handleDeResMsg(sb,msg,cs) ≡
    case msg of
        T.lineBranchDeRes →
        (
            let
                cs = removeSBCCLineRes(sb,cs),
                cs = removeSBCCBranchRes(sb,cs)
            in
                (cs,T.noSBCCMsg)
            end
        ),

        T.lineDeRes →
        (
            let
                cs = removeSBCCLineRes(sb,cs)
```

```
            in
                (cs,T.noSBCCMsg)
            end
        ),

        T.branchDeRes →
        (
            let
                cs = removeSBCCBranchRes(sb,cs)
            in
                (cs,T.noSBCCMsg)
            end
        )
    end,

/* Wellformednes */
is_wf : ControlState × D.State × S.Configuration → Bool
is_wf(cs,ds,con) ≡
    D.is_wf(ds,con) ∧
    tcc_has_state(cs) ∧
    sbcc_has_state(cs),

/* Every TCC must have a state */
tcc_has_state : ControlState → Bool
tcc_has_state(cs) ≡
(
    ∀ t : T.TrainID •
        tccStateExists(t,cs)
),

/* Every SBCC must have a state */
sbcc_has_state : ControlState → Bool
sbcc_has_state(cs) ≡
(
    ∀ sb : T.SBID •
        sbccStateExists(sb,cs)
),

/**
* Defines that the control system and all its
* components must be consistent e.g. the information
* stored in the control system must reflect the physical
* world and unintended states must not occur.
*
* Also the physical world must abide by
* the rules of the control system.
**/
consistent : ControlState × D.State × S.Configuration → Bool
consistent(cs,ds,con) ≡
    is_wf(cs,ds,con) ∧
    train_on_branch_dir(ds,con) ∧
    tcc_hasRes_passedResPoint(cs,ds,con) ∧
    sbcc_res_wf(cs,con) ∧
    position_branch_sbcc_res_wf(cs,ds,con) ∧
    tcc_res_branch_wf(cs,ds,con) ∧
    position_sl_sbcc_res_wf(cs,ds,con) ∧
    barrierPos_signalStatus_Consistent(ds,con),
```

```
/∗ When a train is on a branch segment it is consitent
     with the driving direction of the train ∗/
train_on_branch_dir : D.State × S.Configuration → Bool
train_on_branch_dir(ds,con) ≡
(
    ∀ t : T.TrainID, seg : T.SegmentID •
        D.trainOnBranch(t,con,ds) ∧
        D.trainOnSegment(t,seg,con,ds) ∧
        S.segIsBranch(seg,con) ⇒
            S.branchDir(seg,con) = D.getTrainDirection(t,ds)
),

/∗ If a train has a reservation then it
     has passed the reservation point
     on the given segment ∗/
tcc_hasRes_passedResPoint : ControlState × D.State ×
                                                    S.Configuration → Bool
tcc_hasRes_passedResPoint(cs,ds,con) ≡
(
    ∀ t : T.TrainID •
        hasTCCRes(t,cs) ⇒ hasPassedResPoint(t,ds,con)
),

/∗ Only POINTSB and ENDSB may have line reservations
     Only POINTSB may have branch reservations ∗/
sbcc_res_wf : ControlState × S.Configuration → Bool
sbcc_res_wf(cs,con) ≡
(
    ∀ sb : T.SBID •
        (S.getSBType(sb,con) ∈ {T.PLAINSB, T.CROSSINGSB} ⇒
                {getSBCCLineRes(sb,cs)} ∪
                {getSBCCBranchRes(sb,cs)} = {T.noRes})
        ∧
        (S.getSBType(sb,con) = T.ENDSB ⇒
            getSBCCBranchRes(sb,cs) = T.noRes)
),

/∗ When a train is on a branch segment it must have
     a branch reservation in the SB behind ∗/
position_branch_sbcc_res_wf : ControlState × D.State ×
                                                    S.Configuration → Bool
position_branch_sbcc_res_wf(cs,ds,con) ≡
(
    ∀ t : T.TrainID,
            sb : T.SBID,
            tDir : T.Direction,
            seg : T.SegmentID •
                tDir = D.getTrainDirection(t,ds) ∧
                D.trainOnSegment(t,seg,con,ds) ∧
                D.trainOnBranch(t,con,ds) ∧
                S.segIsBranch(seg,con) ∧
                sb = S.getSegSB(seg,T.inverseDir(tDir),con)
                    ⇒
                        getSBCCBranchRes(sb,cs) = T.res(T.mk_res(t,tDir))
),
```

```
/* If a train is (only) on a branch and has reservation
   then the SB in front of it and the other guard has a
   reservation for that train in that direction */
  tcc_res_branch_wf : ControlState × D.State ×
                                      S.Configuration → Bool
  tcc_res_branch_wf(cs,ds,con) ≡
  (
    ∀ t : T.TrainID,
        seg : T.SegmentID,
        trainDir : T.Direction,
        guard1,guard2 : T.SBID,
        res : T.Reservation •
          D.trainOnSegment(t,seg,con,ds) ∧
          D.trainOnlyOnBranch(t,con,ds) ∧
          hasTCCRes(t,cs) ∧
          trainDir = D.getTrainDirection(t,ds) ∧
          guard1 = S.getSegSB(seg,T.inverseDir(trainDir),con) ∧
          guard2 = S.getSingleLineGuard(guard1,trainDir,con) ∧
          res = T.mk_res(t,trainDir)
            ⇒
              T.res(res) = getSBCCLineRes(guard1,cs) ∧
              T.res(res) = getSBCCLineRes(guard2,cs) ∧
              T.res(res) = getSBCCBranchRes(guard2,cs)
  ),

  /* When a train is on a single line it must
     have a reservation in both guards with the
     appropriate direction + a branch
     reservation if driving to a point */
  position_sl_sbcc_res_wf : ControlState × D.State ×
                                      S.Configuration → Bool
  position_sl_sbcc_res_wf(cs,ds,con) ≡
  (
    ∀ t : T.TrainID,
        seg : T.SegmentID,
        sb1,sb2 : T.SBID,
        dir : T.Direction,
        res : T.Reservation •
          dir = D.getTrainDirection(t,ds) ∧
          D.trainOnSegment(t,seg,con,ds) ∧
          S.segIsLineSegment(seg,con) ∧
          sb1 = S.getSingleLineGuard(seg,
                              T.inverseDir(dir),con) ∧
          sb2 = S.getSingleLineGuard(seg,dir,con) ∧
          res = T.mk_res(t,dir) ⇒
              T.res(res) = getSBCCLineRes(sb1,cs) ∧
              T.res(res) = getSBCCLineRes(sb2,cs) ∧
              (S.getSBType(sb2,con) = T.POINTSB ⇒
                T.res(res) = getSBCCBranchRes(sb2,cs))
  ),

  /* Position of barriers and status of
     crossing signals must conform
     Allowed (barrier,signal) states:
     (UP,OFF), (UP,ON), (MOVINGDOWN,ON), (DOWN,ON),
     (DOWN,OFF), (MOVINGUP,OFF)
  */
```

```
barrierPos_signalStatus_Consistent : D.State ×
                                            S.Configuration → Bool
barrierPos_signalStatus_Consistent(s,con) ≡
(
    ∀ sb : T.SBID •
        S.getSBType(sb,con) = T.CROSSINGSB ⇒
            case D.getBarrierPosition(sb,s,con) of
                T.UP → D.getSignalStatus(sb,s,con) ∈ {T.ON,T.OFF},
                T.MOVINGDOWN → D.getSignalStatus(sb,s,con) = T.ON,
                T.DOWN → D.getSignalStatus(sb,s,con) = T.OFF,
                T.MOVINGUP → D.getSignalStatus(sb,s,con) = T.OFF
            end
),

initReq : ControlState × D.State × S.Configuration → Bool
initReq(cs,ds,con) ≡
    is_wf(cs,ds,con) ∧
    no_sbcc_res(cs) ∧
    sbcc_not_preparing(cs) ∧
    no_tcc_res(cs) ∧
    tcc_not_requesting(cs) ∧
    tcc_not_decelerating(cs),

no_sbcc_res : ControlState → Bool
no_sbcc_res(cs) ≡
(
    ∀ sb : T.SBID,
            branchRes,lineRes : T.HasRes •
                branchRes = getSBCCBranchRes(sb,cs) ∧
                lineRes = getSBCCLineRes(sb,cs)
                    ⇒
                        {branchRes} ∪ {lineRes} = {T.noRes}
),

/∗ No SBCC is currently preparing a segment ∗/
sbcc_not_preparing : ControlState → Bool
sbcc_not_preparing(cs) ≡
(
    ∀ sb : T.SBID •
        ∼isPreparing(sb,cs)
),

no_tcc_res : ControlState → Bool
no_tcc_res(cs) ≡
(
    ∀ t : T.TrainID •
        ∼hasTCCRes(t,cs)
),

/∗ No TCC is requesting segment access ∗/
tcc_not_requesting : ControlState → Bool
tcc_not_requesting(cs) ≡
(
    ∀ t : T.TrainID •
        ∼isTCCRequesting(t,cs)
),
```

```
    /* No TCC is requesting segment access */
    tcc_not_decelerating : ControlState → Bool
    tcc_not_decelerating(cs) ≡
    (
        ∀ t : T.TrainID •
            ~isTrainDecelerating(t,cs)
    )

axiom
    /* The initial state has to be wellformed and
       fullfill the initial state requirements */
    [ initial_state ]
        initReq(initControlState,D.initState,S.conf),

    /**
    * Observer_generator axioms
    **/

    /* getSBCCLineRes_gen */

    [ getSBCCLineRes_setSBCCLineRes ]
        ∀ sb1,sb2 : T.SBID, sbRes : T.Reservation,
                cs : ControlState,
                con : S.Configuration •
            getSBCCLineRes(sb1,
                        setSBCCLineRes(sb2,T.res(sbRes),cs)) ≡
                if(sb1 = sb2)
                then
                    T.res(sbRes)
                else
                    getSBCCLineRes(sb1,cs)
                end,

    [ getSBCCLineRes_removeSBCCLineRes ]
        ∀ sb1,sb2 : T.SBID, cs : ControlState •
            getSBCCLineRes(sb1,removeSBCCLineRes(sb2,cs)) ≡
                if(sb1 = sb2)
                then
                    T.noRes
                else
                    getSBCCLineRes(sb1,cs)
                end,

    [ getSBCCLineRes_setSBCCBranchRes ]
        ∀ sb1,sb2 : T.SBID, sbRes : T.Reservation,
                cs : ControlState,
                con : S.Configuration •
            getSBCCLineRes(sb1,
                    setSBCCBranchRes(sb2,T.res(sbRes),cs)) ≡
                getSBCCLineRes(sb1,cs),

    [ getSBCCLineRes_setLastSensorStatus ]
        ∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
                cs : ControlState,
                con : S.Configuration •
            getSBCCLineRes(sb1,setLastSensorStatus(sb2,ss,cs)) ≡
                getSBCCLineRes(sb1,cs),
```

```
[ getSBCCLineRes_storeSBCCMsg ]
    ∀ sb1,sb2 : T.SBID, cm : T.ComMsg,
            cs : ControlState,
            con : S.Configuration •
        getSBCCLineRes(sb1,storeSBCCMsg(sb2,cm,cs)) ≡
            getSBCCLineRes(sb1,cs),

[ getSBCCLineRes_setSBCCPrepRes ]
    ∀ sb1,sb2 : T.SBID, hr : T.HasRes,
            cs : ControlState,
            con : S.Configuration •
        getSBCCLineRes(sb1,setSBCCPrepRes(sb2,hr,cs)) ≡
            getSBCCLineRes(sb1,cs),

[ getSBCCLineRes_setTCCRes ]
    ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
        getSBCCLineRes(sb1,setTCCRes(sb2,b,cs)) ≡
            getSBCCLineRes(sb1,cs),

[ getSBCCLineRes_setTCCRequesting ]
    ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
        getSBCCLineRes(sb1,setTCCRequesting(sb2,b,cs)) ≡
            getSBCCLineRes(sb1,cs),

[ getSBCCLineRes_setTrainDecelerating ]
    ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
        getSBCCLineRes(sb1,setTrainDecelerating(sb2,b,cs)) ≡
            getSBCCLineRes(sb1,cs),

/∗ getSBCCBranchRes_gen ∗/

[ getSBCCBranchRes_setSBCCLineRes ]
    ∀ sb1,sb2 : T.SBID, sbRes : T.Reservation,
            cs : ControlState,
            con : S.Configuration •
        getSBCCBranchRes(sb1,
                setSBCCLineRes(sb2,T.res(sbRes),cs)) ≡
            getSBCCBranchRes(sb1,cs),

[ getSBCCBranchRes_setSBCCBranchRes ]
    ∀ sb1,sb2 : T.SBID, cs : ControlState,
            sbRes : T.Reservation,
            con : S.Configuration •
        getSBCCBranchRes(sb1,
                setSBCCBranchRes(sb2,T.res(sbRes),cs)) ≡
            if(sb1 = sb2)
            then
                T.res(sbRes)
            else
                getSBCCBranchRes(sb1,cs)
```

```
        end,

[ getSBCCBranchRes_removeSBCCBranchRes ]
  ∀ sb1,sb2 : T.SBID, cs : ControlState •
    getSBCCBranchRes(sb1,removeSBCCBranchRes(sb2,cs)) ≡
      if(sb1 = sb2)
      then
        T.noRes
      else
        getSBCCBranchRes(sb1,cs)
      end,

[ getSBCCBranchRes_setLastSensorStatus ]
  ∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
      cs : ControlState,
      con : S.Configuration •
    getSBCCBranchRes(sb1,setLastSensorStatus(sb2,ss,cs)) ≡
      getSBCCBranchRes(sb1,cs),

[ getSBCCBranchRes_storeSBCCMsg ]
  ∀ sb1,sb2 : T.SBID, cm : T.ComMsg,
      cs : ControlState,
      con : S.Configuration •
    getSBCCBranchRes(sb1,storeSBCCMsg(sb2,cm,cs)) ≡
      getSBCCBranchRes(sb1,cs),

[ getSBCCBranchRes_setSBCCPrepRes ]
  ∀ sb1,sb2 : T.SBID, hr : T.HasRes,
      cs : ControlState,
      con : S.Configuration •
    getSBCCBranchRes(sb1,setSBCCPrepRes(sb2,hr,cs)) ≡
      getSBCCBranchRes(sb1,cs),

[ getSBCCBranchRes_setTCCRes ]
  ∀ sb1,sb2 : T.SBID, b : Bool,
      cs : ControlState,
      con : S.Configuration •
    getSBCCBranchRes(sb1,setTCCRes(sb2,b,cs)) ≡
      getSBCCBranchRes(sb1,cs),

[ getSBCCBranchRes_setTCCRequesting ]
  ∀ sb1,sb2 : T.SBID, b : Bool,
      cs : ControlState,
      con : S.Configuration •
    getSBCCBranchRes(sb1,setTCCRequesting(sb2,b,cs)) ≡
      getSBCCBranchRes(sb1,cs),

[ getSBCCBranchRes_setTrainDecelerating ]
  ∀ sb1,sb2 : T.SBID, b : Bool,
      cs : ControlState,
      con : S.Configuration •
    getSBCCBranchRes(sb1,setTrainDecelerating(sb2,b,cs)) ≡
      getSBCCBranchRes(sb1,cs),

/* getLastSensorStatus_gen */

[ getLastSensorStatus_setSBCCLineRes ]
```

```
    ∀ sb1,sb2 : T.SBID, sbRes : T.Reservation,
            cs : ControlState,
            con : S.Configuration •
        getLastSensorStatus(sb1,
                setSBCCLineRes(sb2,T.res(sbRes),cs)) ≡
            getLastSensorStatus(sb1,cs),

[ getLastSensorStatus_setSBCCBranchRes ]
    ∀ sb1,sb2 : T.SBID, cs : ControlState,
            sbRes : T.Reservation,
            con : S.Configuration •
        getLastSensorStatus(sb1,
                setSBCCBranchRes(sb2,T.res(sbRes),cs)) ≡
            getLastSensorStatus(sb1,cs),

[ getLastSensorStatus_setLastSensorStatus ]
    ∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
            cs : ControlState •
        getLastSensorStatus(sb1,
                setLastSensorStatus(sb2,ss,cs)) ≡
            if(sb1 = sb2)
            then
                ss
            else
                getLastSensorStatus(sb1,cs)
            end,

[ getLastSensorStatus_storeSBCCMsg ]
    ∀ sb1,sb2 : T.SBID, cm : T.ComMsg,
            cs : ControlState,
            con : S.Configuration •
        getLastSensorStatus(sb1,storeSBCCMsg(sb2,cm,cs)) ≡
            getLastSensorStatus(sb1,cs),

[ getLastSensorStatus_setSBCCPrepRes ]
    ∀ sb1,sb2 : T.SBID, hr : T.HasRes,
            cs : ControlState,
            con : S.Configuration •
        getLastSensorStatus(sb1,setSBCCPrepRes(sb2,hr,cs)) ≡
            getLastSensorStatus(sb1,cs),

[ getLastSensorStatus_setTCCRes ]
    ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
        getLastSensorStatus(sb1,setTCCRes(sb2,b,cs)) ≡
            getLastSensorStatus(sb1,cs),

[ getLastSensorStatus_setTCCRequesting ]
    ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
        getLastSensorStatus(sb1,setTCCRequesting(sb2,b,cs)) ≡
            getLastSensorStatus(sb1,cs),

[ getLastSensorStatus_setTrainDecelerating ]
    ∀ sb1,sb2 : T.SBID, b : Bool,
```

```
            cs : ControlState,
            con : S.Configuration •
         getLastSensorStatus(sb1,
               setTrainDecelerating(sb2,b,cs)) ≡
            getLastSensorStatus(sb1,cs),

/* getNextSBCCMsg_gen */

[getNextSBCCMsg_setSBCCLineRes]
      ∀ sb1,sb2 : T.SBID, sbRes : T.Reservation,
            cs : ControlState,
            con : S.Configuration •
         getNextSBCCMsg(sb1,
               setSBCCLineRes(sb2,T.res(sbRes),cs)) ≡
            getNextSBCCMsg(sb1,cs),

   [getNextSBCCMsg_setSBCCBranchRes]
      ∀ sb1,sb2 : T.SBID, cs : ControlState,
            sbRes : T.Reservation,
            con : S.Configuration •
         getNextSBCCMsg(sb1,
               setSBCCBranchRes(sb2,T.res(sbRes),cs)) ≡
            getNextSBCCMsg(sb1,cs),

   [getNextSBCCMsg_setLastSensorStatus]
      ∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
            cs : ControlState •
         getNextSBCCMsg(sb1,setLastSensorStatus(sb2,ss,cs)) ≡
            getNextSBCCMsg(sb1,cs),

   [getNextSBCCMsg_setSBCCPrepRes]
      ∀ sb1,sb2 : T.SBID, hr : T.HasRes,
            cs : ControlState,
            con : S.Configuration •
         getNextSBCCMsg(sb1,setSBCCPrepRes(sb2,hr,cs)) ≡
            getNextSBCCMsg(sb1,cs),

   [getNextSBCCMsg_setTCCRes]
      ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
         getNextSBCCMsg(sb1,setTCCRes(sb2,b,cs)) ≡
            getNextSBCCMsg(sb1,cs),

   [getNextSBCCMsg_setTCCRequesting]
      ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
         getNextSBCCMsg(sb1,setTCCRequesting(sb2,b,cs)) ≡
            getNextSBCCMsg(sb1,cs),

   [getNextSBCCMsg_setTrainDecelerating]
      ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
         getNextSBCCMsg(sb1,setTrainDecelerating(sb2,b,cs)) ≡
            getNextSBCCMsg(sb1,cs),
```

```
/* getSBCCPrepRes_gen */

[ getSBCCPrepRes_setSBCCLineRes ]
    ∀ sb1,sb2 : T.SBID, sbRes : T.Reservation,
            cs : ControlState,
            con : S.Configuration •
        getSBCCPrepRes(sb1,setSBCCLineRes(sb2,T.res(sbRes),cs)) ≡
            getSBCCPrepRes(sb1,cs),

[ getSBCCPrepRes_setSBCCBranchRes ]
    ∀ sb1,sb2 : T.SBID, cs : ControlState,
            sbRes : T.Reservation,
            con : S.Configuration •
        getSBCCPrepRes(sb1,
                setSBCCBranchRes(sb2,T.res(sbRes),cs)) ≡
            getSBCCPrepRes(sb1,cs),

[ getSBCCPrepRes_setLastSensorStatus ]
    ∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
            cs : ControlState •
        getSBCCPrepRes(sb1,setLastSensorStatus(sb2,ss,cs)) ≡
            getSBCCPrepRes(sb1,cs),

[ getSBCCPrepRes_storeSBCCMsg ]
    ∀ sb1,sb2 : T.SBID, cm : T.ComMsg,
            cs : ControlState,
            con : S.Configuration •
        getSBCCPrepRes(sb1,storeSBCCMsg(sb2,cm,cs)) ≡
            getSBCCPrepRes(sb1,cs),

[ getSBCCPrepRes_setSBCCPrepRes ]
    ∀ sb1,sb2 : T.SBID, hr : T.HasRes,
            cs : ControlState,
            con : S.Configuration •
        getSBCCPrepRes(sb1,setSBCCPrepRes(sb2,hr,cs)) ≡
            if(sb1 = sb2)
            then
                hr
            else
                getSBCCPrepRes(sb1,cs)
            end,

[ getSBCCPrepRes_setTCCRes ]
    ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
        getSBCCPrepRes(sb1,setTCCRes(sb2,b,cs)) ≡
            getSBCCPrepRes(sb1,cs),

[ getSBCCPrepRes_setTCCRequesting ]
    ∀ sb1,sb2 : T.SBID, b : Bool,
            cs : ControlState,
            con : S.Configuration •
        getSBCCPrepRes(sb1,setTCCRequesting(sb2,b,cs)) ≡
            getSBCCPrepRes(sb1,cs),
```

```
[getSBCCPrepRes_setTrainDecelerating]
  ∀ sb1,sb2 : T.SBID, b : Bool,
      cs : ControlState,
      con : S.Configuration •
    getSBCCPrepRes(sb1,setTrainDecelerating(sb2,b,cs)) ≡
      getSBCCPrepRes(sb1,cs),

/* hasTCCRes_gen */

[hasTCCRes_setSBCCLineRes]
  ∀ sb1,sb2 : T.SBID, sbRes : T.Reservation,
      cs : ControlState,
      con : S.Configuration •
    hasTCCRes(sb1,setSBCCLineRes(sb2,T.res(sbRes),cs)) ≡
      hasTCCRes(sb1,cs),

[hasTCCRes_setSBCCBranchRes]
  ∀ sb1,sb2 : T.SBID, cs : ControlState,
      sbRes : T.Reservation,
      con : S.Configuration •
    hasTCCRes(sb1,setSBCCBranchRes(sb2,T.res(sbRes),cs)) ≡
      hasTCCRes(sb1,cs),

[hasTCCRes_setLastSensorStatus]
  ∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
      cs : ControlState •
    hasTCCRes(sb1,setLastSensorStatus(sb2,ss,cs)) ≡
      hasTCCRes(sb1,cs),

[hasTCCRes_storeSBCCMsg]
  ∀ sb1,sb2 : T.SBID, cm : T.ComMsg,
      cs : ControlState,
      con : S.Configuration •
    hasTCCRes(sb1,storeSBCCMsg(sb2,cm,cs)) ≡
      hasTCCRes(sb1,cs),

[hasTCCRes_setSBCCPrepRes]
  ∀ sb1,sb2 : T.SBID, hr : T.HasRes,
      cs : ControlState,
      con : S.Configuration •
    hasTCCRes(sb1,setSBCCPrepRes(sb2,hr,cs)) ≡
      hasTCCRes(sb1,cs),

[hasTCCRes_setTCCRes]
  ∀ t1, t2 : T.TrainID, b : Bool, cs : ControlState •
    hasTCCRes(t1,setTCCRes(t2,b,cs)) ≡
      if(t1 = t2)
      then
        b
      else
        hasTCCRes(t1,cs)
      end,

[hasTCCRes_setTCCRequesting]
  ∀ sb1,sb2 : T.SBID, b : Bool,
      cs : ControlState,
      con : S.Configuration •
```

```
      hasTCCRes(sb1,setTCCRequesting(sb2,b,cs)) ≡
        hasTCCRes(sb1,cs),

[ hasTCCRes_setTrainDecelerating ]
  ∀ sb1,sb2 : T.SBID, b : Bool,
        cs : ControlState,
        con : S.Configuration •
      hasTCCRes(sb1,setTrainDecelerating(sb2,b,cs)) ≡
        hasTCCRes(sb1,cs),

/∗ isTCCRequesting_gen ∗/

[ isTCCRequesting_setSBCCLineRes ]
  ∀ sb1,sb2 : T.SBID, sbRes : T.Reservation,
        cs : ControlState,
        con : S.Configuration •
      isTCCRequesting(sb1,
          setSBCCLineRes(sb2,T.res(sbRes),cs)) ≡
        isTCCRequesting(sb1,cs),

[ isTCCRequesting_setSBCCBranchRes ]
  ∀ sb1,sb2 : T.SBID, cs : ControlState,
        sbRes : T.Reservation,
        con : S.Configuration •
      isTCCRequesting(sb1,
          setSBCCBranchRes(sb2,T.res(sbRes),cs)) ≡
        isTCCRequesting(sb1,cs),

[ isTCCRequesting_setLastSensorStatus ]
  ∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
        cs : ControlState •
      isTCCRequesting(sb1,setLastSensorStatus(sb2,ss,cs)) ≡
        isTCCRequesting(sb1,cs),

[ isTCCRequesting_storeSBCCMsg ]
  ∀ sb1,sb2 : T.SBID, cm : T.ComMsg,
        cs : ControlState,
        con : S.Configuration •
      isTCCRequesting(sb1,storeSBCCMsg(sb2,cm,cs)) ≡
        isTCCRequesting(sb1,cs),

[ isTCCRequesting_setSBCCPrepRes ]
  ∀ sb1,sb2 : T.SBID, hr : T.HasRes,
        cs : ControlState,
        con : S.Configuration •
      isTCCRequesting(sb1,setSBCCPrepRes(sb2,hr,cs)) ≡
        isTCCRequesting(sb1,cs),

[ isTCCRequesting_setTCCRes ]
  ∀ t1, t2 : T.TrainID, b : Bool, cs : ControlState •
      isTCCRequesting(t1,setTCCRes(t2,b,cs)) ≡
        isTCCRequesting(t1,cs),

[ isTCCRequesting_setTCCRequesting ]
  ∀ t1, t2 : T.TrainID, b : Bool, cs : ControlState •
      isTCCRequesting(t1,setTCCRequesting(t2,b,cs)) ≡
        if(t1 = t2)
```

```
            then
              b
            else
              isTCCRequesting(t1,cs)
            end,

[ isTCCRequesting_setTrainDecelerating ]
  ∀ sb1,sb2 : T.SBID, b : Bool,
          cs : ControlState,
          con : S.Configuration •
      isTCCRequesting(sb1,setTrainDecelerating(sb2,b,cs)) ≡
          isTCCRequesting(sb1,cs),

/∗ isTrainDecelerating_gen ∗/

[ isTrainDecelerating_setSBCCLineRes ]
  ∀ sb1,sb2 : T.SBID, sbRes : T.Reservation,
          cs : ControlState,
          con : S.Configuration •
      isTrainDecelerating(sb1,
            setSBCCLineRes(sb2,T.res(sbRes),cs)) ≡
          isTrainDecelerating(sb1,cs),

[ isTrainDecelerating_setSBCCBranchRes ]
  ∀ sb1,sb2 : T.SBID, cs : ControlState,
          sbRes : T.Reservation,
          con : S.Configuration •
      isTrainDecelerating(sb1,
            setSBCCBranchRes(sb2,T.res(sbRes),cs)) ≡
          isTrainDecelerating(sb1,cs),

[ isTrainDecelerating_setLastSensorStatus ]
  ∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
          cs : ControlState •
      isTrainDecelerating(sb1,
            setLastSensorStatus(sb2,ss,cs)) ≡
          isTrainDecelerating(sb1,cs),

[ isTrainDecelerating_storeSBCCMsg ]
  ∀ sb1,sb2 : T.SBID, cm : T.ComMsg,
          cs : ControlState,
          con : S.Configuration •
      isTrainDecelerating(sb1,storeSBCCMsg(sb2,cm,cs)) ≡
          isTrainDecelerating(sb1,cs),

[ isTrainDecelerating_setSBCCPrepRes ]
  ∀ sb1,sb2 : T.SBID, hr : T.HasRes,
          cs : ControlState,
          con : S.Configuration •
      isTrainDecelerating(sb1,setSBCCPrepRes(sb2,hr,cs)) ≡
          isTrainDecelerating(sb1,cs),

[ isTrainDecelerating_setTCCRes ]
  ∀ t1, t2 : T.TrainID, b : Bool, cs : ControlState •
      isTrainDecelerating(t1,setTCCRes(t2,b,cs)) ≡
          isTrainDecelerating(t1,cs),
```

```
[ isTrainDecelerating_setTCCRequesting ]
    ∀ t1, t2 : T.TrainID, b : Bool, cs : ControlState •
        isTrainDecelerating(t1,setTCCRequesting(t2,b,cs)) ≡
            isTrainDecelerating(t1,cs),

[ isTrainDecelerating_setTrainDecelerating ]
    ∀ t1, t2 : T.TrainID, b : Bool, cs : ControlState •
        isTrainDecelerating(t1,setTrainDecelerating(t2,b,cs)) ≡
            if(t1 = t2)
            then
                b
            else
                isTrainDecelerating(t1,cs)
            end,

/* end of obs_gen axioms */


/** Maintaining the wellformedness of the state when
*   applied to a generator with its precondition
*   General form:
*     is_wf(state) ∧ pre_cond(..,state) ⇒
*       is_wf(gen_(state))
**/

[ gen_wf_setSBCCLineRes ]
    ∀ sb : T.SBID, res : T.Reservation,
        ds : D.State, con : S.Configuration,
        cs : ControlState •
    is_wf(cs,ds,con) ∧
    S.getSBType(sb,con) ∈ {T.ENDSB, T.POINTSB}
        ⇒
    is_wf(setSBCCLineRes(sb,T.res(res),cs),ds,con),

[ gen_wf_setSBCCBranchRes ]
    ∀ sb : T.SBID, res : T.Reservation,
    ds : D.State, con : S.Configuration,
    cs : ControlState •
    is_wf(cs,ds,con) ∧
    S.getSBType(sb,con) = T.POINTSB
        ⇒
    is_wf(setSBCCBranchRes(sb,T.res(res),cs),ds,con),

[ gen_wf_removeSBCCLineRes ]
    ∀ sb : T.SBID,
        ds : D.State, con : S.Configuration,
        cs : ControlState •
    is_wf(cs,ds,con) ∧
    S.getSBType(sb,con) ∈ {T.ENDSB, T.POINTSB}
        ⇒
    is_wf(removeSBCCLineRes(sb,cs),ds,con),

[ gen_wf_removeSBCCBranchRes ]
    ∀ sb : T.SBID,
        ds : D.State, con : S.Configuration,
        cs : ControlState •
    is_wf(cs,ds,con) ∧
```

```
      S.getSBType(sb,con) = T.POINTSB
          ⇒
      is_wf(removeSBCCBranchRes(sb,cs),ds,con),

  [ gen_wf_setLastSensorStatus ]
      ∀ sb : T.SBID, ss : T.SensorStatus,
          ds : D.State, con : S.Configuration,
          cs : ControlState •
      is_wf(cs,ds,con)
          ⇒
      is_wf(setLastSensorStatus(sb,ss,cs),ds,con),

  [ gen_wf_storeSBCCMsg ]
      ∀ sb : T.SBID,
          ds : D.State, con : S.Configuration,
          cs : ControlState,
          cm : T.ComMsg •
      is_wf(cs,ds,con)
          ⇒
      is_wf(storeSBCCMsg(sb,cm,cs),ds,con),

  [ gen_wf_setSBCCPrepRes ]
      ∀ sb : T.SBID,
          ds : D.State, con : S.Configuration,
          cs : ControlState,
          hr : T.HasRes •
      is_wf(cs,ds,con)
          ⇒
      is_wf(setSBCCPrepRes(sb,hr,cs),ds,con),

  [ gen_wf_setTCCRes ]
      ∀ sb : T.SBID, b : Bool,
          ds : D.State, con : S.Configuration,
          cs : ControlState
          •
      is_wf(cs,ds,con)
          ⇒
      is_wf(setTCCRes(sb,b,cs),ds,con),

  [ gen_wf_setTCCRequesting ]
      ∀ sb : T.SBID, b : Bool,
          ds : D.State, con : S.Configuration,
          cs : ControlState •
      is_wf(cs,ds,con)
          ⇒
      is_wf(setTCCRequesting(sb,b,cs),ds,con),

  [ gen_wf_setTrainDecelerating ]
      ∀ sb : T.SBID, b : Bool,
          ds : D.State, con : S.Configuration,
          cs : ControlState •
      is_wf(cs,ds,con)
          ⇒
      is_wf(setTrainDecelerating(sb,b,cs),ds,con)

end
```

F.2 Decomposed model

F.2.1 Types

The decomposed Types module is the same as the Types module in the initial model, except that the name is changed from *AA_Types0* to *AA_Types1*. The module can be found in appendix F.1.1.

F.2.2 Statics

```
context: AA_Types1, AA_SBs1, AA_Segs1, AA_Trains1, AA_ESAs1
scheme AA_Statics1(T : AA_Types1) =
  class
    object
      SBs : AA_SBs1(T),
      ESAs : AA_ESAs1(T),
      Segs : AA_Segs1(T),
      Trains : AA_Trains1(T)

    type
      /* Main railway line configuration type */
      Configuration = SBs.SBs × Segs.Segs ×
                                ESAs.ESAs × Trains.Trains

    value
      conf : Configuration = (SBs.sbsConf, Segs.segsConf,
                                ESAs.esasConf, Trains.trainsConf),

      /* Observers */

      /* ESA observers */
      getESASB : T.ESAID × Configuration ~→ T.SBID
      getESASB(esa,(sbs,segs,esas,ts)) ≡
        ESAs.getESASB(esa,esas),

      getESALength : T.ESAID × Configuration ~→ T.Length
      getESALength(esa,(sbs,segs,esas,ts)) ≡
        ESAs.getESALength(esa,esas),

      /* SB observers */
      getSBSeg : T.SBID × T.Direction × Configuration ~→
                                                  T.SBSegment
      getSBSeg(sb,dir,(sbs,segs,esas,ts)) ≡
        SBs.getSBSeg(sb,dir,sbs),

      getSBType : T.SBID × Configuration ~→ T.SBType
      getSBType(sb,(sbs,segs,esas,ts)) ≡
        SBs.getSBType(sb,sbs),

      /* Segment observers */
      getSegSB : T.SegmentID × T.Direction ×
                                   Configuration ~→ T.SBID
      getSegSB(seg,dir,(sbs,segs,esas,ts)) ≡
        Segs.getSegSB(seg,dir,segs),
```

```
getSegLength : T.SegmentID × Configuration ~→ T.Length
getSegLength(segID,(sbs,segs,esas,trains)) ≡
    Segs.getSegLength(segID,segs),

getSegMaxSpeed : T.SegmentID × Configuration ~→ T.Speed
getSegMaxSpeed(segID,(sbs,segs,esas,trains)) ≡
    Segs.getSegMaxSpeed(segID,segs),

/* Train observers */
getTrainLength : T.TrainID × Configuration ~→ T.Length
getTrainLength(tID,(sbs,segs,esas,trains)) ≡
    Trains.getTrainLength(tID,trains),

getTrainMaxSpeed : T.TrainID × Configuration ~→
                                                        T.Acceleration
getTrainMaxSpeed(t,(sbs,segs,esas,ts)) ≡
    Trains.getTrainMaxSpeed(t,ts),

getTrainMaxAcc : T.TrainID × Configuration ~→ T.Acceleration
getTrainMaxAcc(t,(sbs,segs,esas,ts)) ≡
    Trains.getTrainMaxAcc(t,ts),

getTrainMaxDec : T.TrainID × Configuration ~→ T.Acceleration
getTrainMaxDec(t,(sbs,segs,esas,ts)) ≡
    Trains.getTrainMaxDec(t,ts),

/* Reservation− and brake−point observers */
getResPoint : Configuration ~→ T.Length
getResPoint((sbs,segs,esas,ts)) ≡
    Segs.getResPoint(segs),

getBrakePoint : Configuration ~→ T.Length
getBrakePoint((sbs,segs,esas,ts)) ≡
    Segs.getBrakePoint(segs),

/* Auxiliary functions */

/* Determines if a SB is a Single Line Guard */
isLineGuard : T.SBID × Configuration ~→ Bool
isLineGuard(sbID,con) ≡
    getSBType(sbID,con) ∈ {T.POINTSB, T.ENDSB},

/* Determines if a SB is a PointSB */
isPointSB : T.SBID × Configuration ~→ Bool
isPointSB(sbID,con) ≡
    getSBType(sbID,con) = T.POINTSB,

/* Determines if a segment is a branch segment */
segIsBranch : T.SegmentID × Configuration ~→ Bool
segIsBranch(seg,con) ≡
    getSBType(getSegSB(seg,T.UP,con),con) = T.POINTSB ∧
    getSBType(getSegSB(seg,T.DOWN,con),con) = T.POINTSB,

/* Determines if a segment is a line segment,
```

```
    i.e. a segment in a single line */
segIsLineSegment : T.SegmentID × Configuration ~→ Bool
segIsLineSegment(seg,con) ≡
  ∼segIsBranch(seg,con),

neighbours : T.Location × T.Location × Configuration ~→ Bool
neighbours(loc1,loc2,con) ≡
  (getLocSBs(loc1,con) ∪ getLocSBs(loc2,con)) ≠ {},

/* Finds the distance (T.Length)
    between two T.SegmentPosition */
distance : T.SegmentPosition × T.SegmentPosition ×
                              Configuration ~→ T.Length
distance(segPos1,segPos2,con) ≡
  if (T.getLoc(segPos1) = T.getLoc(segPos2))
  then
    if (T.getLength(segPos1) < T.getLength(segPos2))
    then
      T.getLength(segPos2) − T.getLength(segPos1)
    else
      T.getLength(segPos1) − T.getLength(segPos2)
    end
  else
    if (segPosLower(segPos1,segPos2,con))
    then
      getLocLength(T.getLoc(segPos1),con) −
        T.getLength(segPos1) + T.getLength(segPos2)
    else
      getLocLength(T.getLoc(segPos2),con) −
        T.getLength(segPos2) + T.getLength(segPos1)
    end
  end
pre neighbours(T.getLoc(segPos1),T.getLoc(segPos2),con) ∨
    T.getLoc(segPos1) = T.getLoc(segPos2),

segPosLower : T.SegmentPosition × T.SegmentPosition ×
                              Configuration ~→ Bool
segPosLower(segPos1,segPos2,con) ≡
  if (T.getLoc(segPos1) = T.getLoc(segPos2)) then
    T.getLength(segPos1) < T.getLength(segPos2)
  else
    locLower(T.getLoc(segPos1),T.getLoc(segPos2),con)
  end,


/* If a location is immediatedly lower than
    another location. If one location is an ESA,
    the result will depend on the orientation
    of the ESA. */
locLower : T.Location × T.Location × Configuration ~→ Bool
locLower(loc1,loc2,con) ≡
  case loc1 of
    T.isESA(esa1) → (esa1 = T.LOW),
    T.isSeg(seg1) →
    (
      case loc2 of
```

```
                T.isESA(esa2) → (esa2 = T.HIGH),
                T.isSeg(seg2) →
                (
                    seg2 ∈ getNextSegSet(seg1,T.UP,con)
                )
            end
        )
    end,

getLocLength : T.Location × Configuration ~→ T.Length
getLocLength(loc,con) ≡
    case loc of
        T.isESA(esa) → getESALength(esa,con),
        T.isSeg(seg) → getSegLength(seg,con)
    end,

getLocSBs : T.Location × Configuration ~→ T.SBID-set
getLocSBs(loc,con) ≡
    case loc of
        T.isESA(esa) → {getESASB(esa,con)},
        T.isSeg(seg) → {getSegSB(seg,T.UP,con),
                                    getSegSB(seg,T.DOWN,con)}
    end,

/∗ Returns the segments around a point ∗/
getSBPointSegs : T.SBID × Configuration ~→ T.PointSegments
getSBPointSegs(sb,(sbs,segs,esas,ts)) ≡
    SBs.getSBPointSegs(sb,sbs)
pre getSBType(sb,(sbs,segs,esas,ts)) = T.POINTSB,

/∗ Returns the driving direction of a branch ∗/
branchDir : T.SegmentID × Configuration ~→ T.Direction
branchDir(seg,con) ≡
    let
        T.point(up,down) =
                    getSBSeg(getSegSB(seg,T.UP,con),T.DOWN,con)
    in
        if (seg = up)
        then
            T.UP
        else
            T.DOWN
        end
    end
pre segIsBranch(seg,con),

/∗ Given a single line guard, it returns the single
     line guard at the opposite end of the single line ∗/
getOppositeGuard : T.SBID × Configuration ~→ T.SBID
getOppositeGuard(sb,con) ≡
    let
        sbType = getSBType(sb,con),
        dir = if(sbType = T.POINTSB) then getPointDir(sb,con)
                    else getEndDir(sb,con) end,
        lineDir = T.inverseDir(dir)
    in
```

```
        getSingleLineGuard(getNextSB(sb,lineDir,con),lineDir,con)
    end
pre isLineGuard(sb,con),

/∗ Given a point SB, it returns the point SB
     at the opposite end of the branches ∗/
getOppositePointSB : T.SBID × Configuration ~→ T.SBID
getOppositePointSB(sb,con) ≡
    let
        dir = getPointDir(sb,con)
    in
         getNextSB(sb,dir,con)
    end
pre getSBType(sb,con) = T.POINTSB,

/∗ Given an SB, it returns the next SB ∗/
getNextSB : T.SBID × T.Direction × Configuration ~→ T.SBID
getNextSB(sb,dir,con) ≡
    let
        nextSeg = getSBSeg(sb,dir,con)
    in
        case nextSeg of
            T.seg(segID) → getSegSB(segID,dir,con),
            T.point(upSeg,downSeg) → getSegSB(upSeg,dir,con)
        end
    end
pre getSBType(sb,con) ≠ T.ENDSB ∨ getEndDir(sb,con) ≠ dir,

getNextSegSet : T.SegmentID × T.Direction ×
                                        Configuration ~→ T.SegmentID-set
getNextSegSet(seg,dir,con) ≡
    T.sbSegToSet(getSBSeg(getSegSB(seg,dir,con),dir,con)),

/∗ Returns the first single line guard in a direction ∗/
getSingleLineGuard : T.SBID × T.Direction ×
                                            Configuration ~→ T.SBID
getSingleLineGuard(sb,dir,con) ≡
    if(isLineGuard(sb,con))
    then
        sb
    else
        getSingleLineGuard(getNextSB(sb,dir,con),dir,con)
    end,

/∗ Returns the two single line
     guards of a line−segment ∗/
getSingleLineGuards : T.SegmentID ×
                                          Configuration ~→ T.SBID-set
getSingleLineGuards(seg,con) ≡
    let
        sb = getSegSB(seg,T.UP,con)
    in
        { getSingleLineGuard(sb,T.UP,con),
            getSingleLineGuard(sb,T.DOWN,con) }
    end
pre ∼segIsBranch(seg,con),
```

```
/∗ Returns the direction of a point SB
    from the stem towards the branches ∗/
getPointDir : T.SBID × Configuration ~→ T.Direction
getPointDir(sbID,(sbs,segs,esas,trains)) ≡
   SBs.getPointDir(sbID,sbs)
pre getSBType(sbID,(sbs,segs,esas,trains)) = T.POINTSB,

/∗ Returns the direction against an ESA from an END SB ∗/
getEndDir : T.SBID × Configuration ~→ T.Direction
getEndDir(sbID,(sbs,segs,esas,trains)) ≡
   SBs.getEndDir(sbID,sbs)
pre getSBType(sbID,(sbs,segs,esas,trains)) = T.ENDSB,

sbsAreCrossings : T.SBID-set × Configuration ~→ Bool
sbsAreCrossings(sbs,con) ≡
(
   ∀ sb : T.SBID •
      sb ∈ sbs ⇒
         getSBType(sb,con) = T.CROSSINGSB
),

sbsArePoints : T.SBID-set × Configuration ~→ Bool
sbsArePoints(sbs,con) ≡
(
   ∀ sb : T.SBID •
      sb ∈ sbs ⇒
         getSBType(sb,con) = T.POINTSB
),

/∗ Invariants ∗/
is_wf : Configuration → Bool
is_wf((sbs,segs,esas,ts)) ≡
   SBs.is_wf(sbs) ∧
   Segs.is_wf(segs) ∧
   ESAs.is_wf(esas) ∧
   Trains.is_wf(ts) ∧
   composed_is_wf((sbs,segs,esas,ts)),

composed_is_wf : Configuration → Bool
composed_is_wf(con) ≡
   pointSegs_wf(con) ∧
   getESASBSeg_wf(con) ∧
   getSBSeg_getSegSB_wf(con)∧
   seg_train_length_wf(con) ∧
   esa_train_length_wf(con) ∧
   brakePoint_wf(con) ∧
   resPoint_wf(con) ∧
   collisions_detectable(con),

/∗ All associated point (points next to each other)
    must have same up and down branches ∗/
pointSegs_wf : Configuration → Bool
pointSegs_wf(con) ≡
(
   ∀ sb : T.SBID •
      getSBType(sb,con) = T.POINTSB
```

```
            ⇒
              let
                pSegs1 = getSBPointSegs(sb,con),
                dir1 = T.getPointDir(pSegs1),

                sb2 = getNextSB(sb,dir1,con),
                pSegs2 = getSBPointSegs(sb2,con)
              in
                T.getUpBranch(pSegs1) = T.getUpBranch(pSegs2) ∧
                T.getDownBranch(pSegs1) = T.getDownBranch(pSegs2)
              end
 ),

/∗ Given an ESA. From the coherent END SB
    the next SBSegment directed against the
    ESA must be the ESA ∗/
getESASBSeg_wf : Configuration → Bool
getESASBSeg_wf(con) ≡
(
    ∀ esa : T.ESAID •
        getSBSeg(getESASB(esa,con),T.end2Dir(esa),con) = T.esa(esa)
),

/∗ Calculating the SB in a direction from each segment
    in the SBSegment calculated from a SB in the opposite
    direction must give the original SB ∗/
getSBSeg_getSegSB_wf : Configuration → Bool
getSBSeg_getSegSB_wf(con) ≡
(
    ∀ sb : T.SBID, dir : T.Direction, seg : T.SegmentID •
        seg ∈ T.sbSegToSet(getSBSeg(sb,dir,con)) ⇒
            getSegSB(seg,T.inverseDir(dir),con) = sb
),

/∗ All segments must be longer than any train ∗/
seg_train_length_wf : Configuration → Bool
seg_train_length_wf(con) ≡
(
    ∀ seg : T.SegmentID, t : T.TrainID •
        getSegLength(seg,con) > getTrainLength(t,con)
),

/∗ An ESA must be longer than a brake point.
    This ensures that all the axioms above
    (concerning braking) also applies to the ESAs
∗/
esa_train_length_wf : Configuration → Bool
esa_train_length_wf(con) ≡
(
    ∀ esa : T.ESAID, t : T.TrainID •
        getESALength(esa,con) >
            getBrakePoint(con) + getTrainLength(t,con)
),

/∗ If a train starts to brake at the brakepoint
    it must be able to stop entirely before
    entering the next segment
```

```
    */
    brakePoint_wf : Configuration → Bool
    brakePoint_wf(con) ≡
    (
      ∀ t : T.TrainID, tAcc : T.Acceleration,
           brakeP, brakeL, s_err : T.Length,
           tSpeed : T.Speed •
             tAcc = getTrainMaxDec(t,con) ∧
             brakeP = getBrakePoint(con) ∧
             tSpeed = getTrainMaxSpeed(t,con) ∧
             s_err = tSpeed ∗ T.tick_interval ∧
             brakeL = −0.5 ∗ tSpeed ∗ tSpeed / tAcc
               ⇒
                 brakeP > brakeL + s_err
    ),

    /∗ ensures that resPoint > brakePoint and that a train
       is entirely on a single segment when resPoint is
       exceeded also that brakePoint < segment length∗/
    resPoint_wf : Configuration → Bool
    resPoint_wf(con) ≡
    (
      ∀ t : T.TrainID, seg : T.SegmentID,
           tlen, slen, resPoint, brakePoint : T.Length •
             tlen = getTrainLength(t,con) ∧
             slen = getSegLength(seg,con) ∧
             resPoint = getResPoint(con) ∧
             brakePoint = getBrakePoint(con)
               ⇒
                 slen  > (resPoint + tlen) ∧
                 brakePoint < slen
    ),

    /∗ Ensures that collisions can be detected
       before  trains passes through each other ∗/
    collisions_detectable : Configuration → Bool
    collisions_detectable(con) ≡
    (
      ∀ t1, t2 : T.TrainID, sp1, sp2 : T.Speed,
           s_err1, s_err2, s_col : T.Length •
             sp1 = getTrainMaxSpeed(t1,con) ∧
             sp2 = getTrainMaxSpeed(t2,con) ∧
             s_err1 = sp1 ∗ T.tick_interval ∧
             s_err2 = sp2 ∗ T.tick_interval ∧
             s_col = s_err1 + s_err2
               ⇒
                 s_col < getTrainLength(t1,con)
    )

  axiom
    [is_wf]
      is_wf(conf)

end
```

SBs

```
context: AA_Types1
scheme AA_SBs1(T : AA_Types1) =
  class
    type
      /* Type of interest */
      SBs

    value
      sbsConf : SBs,

      /* SB observers */
      getSBSeg : T.SBID × T.Direction × SBs ~→ T.SBSegment,
      getSBType : T.SBID × SBs ~→ T.SBType,
      sbExistsInConf : T.SBID × SBs → Bool,

      /* Auxiliary functions */

      /* Returns the direction of a point SB
         from the stem towards the branches */
      getPointDir : T.SBID × SBs ~→ T.Direction
      getPointDir(sb,sbs) ≡
        let sbSeg = getSBSeg(sb,T.UP,sbs)
        in
          case sbSeg of
            T.point(_,_) → T.UP,
            _ → T.DOWN
          end
        end
      pre getSBType(sb,sbs) = T.POINTSB,

      /* Returns the segments around a point */
      getSBPointSegs : T.SBID × SBs ~→ T.PointSegments
      getSBPointSegs(sb,sbs) ≡
        let
          dir = getPointDir(sb,sbs),
          pointSegs = getSBSeg(sb,dir,sbs),
          T.seg(stemSeg) = getSBSeg(sb,T.inverseDir(dir),sbs)
        in
          T.pointSegments(stemSeg,
                          T.getUpSeg(pointSegs),
                          T.getDownSeg(pointSegs),
                          dir)
        end
      pre getSBType(sb,sbs) = T.POINTSB,

      /* Returns the direction against an ESA from an END SB */
      getEndDir : T.SBID × SBs ~→ T.Direction
      getEndDir(sb,sbs) ≡
        case getSBSeg(sb,T.UP,sbs) of
          T.esa(_) → T.UP,
          _ → T.DOWN
        end
      pre getSBType(sb,sbs) = T.ENDSB,
```

```
/* Invariants */
is_wf : SBs → Bool
is_wf(sbs) ≡
    sbsHaveConf(sbs) ∧
    getSBSeg_diff(sbs) ∧
    getSBSeg_point_wf(sbs) ∧
    getSBSeg_injective(sbs) ∧
    getSBSegType_wf(sbs),

/* A configuration for each SB must exists */
sbsHaveConf : SBs → Bool
sbsHaveConf(sbs) ≡
(
    ∀ sb : T.SBID •
        sbExistsInConf(sb,sbs)
),

/* The segments next to a SB are different
     in the UP and the DOWN direction.
     I.e. the line is not circular */
getSBSeg_diff : SBs → Bool
getSBSeg_diff(sbs) ≡
(
    ∀ sb : T.SBID •
        getSBSeg(sb,T.UP,sbs) ≠ getSBSeg(sb,T.DOWN,sbs)
),

/* The two branches of a junction are different */
getSBSeg_point_wf : SBs → Bool
getSBSeg_point_wf(sbs) ≡
(
    ∀ sb : T.SBID,
            seg1,seg2 : T.SegmentID,
            dir : T.Direction •
                T.point(seg1,seg2) = getSBSeg(sb,dir,sbs) ⇒
                    seg1 ≠ seg2
),

/* Two different SBs have different SBSegments
     in the same direction */
getSBSeg_injective : SBs → Bool
getSBSeg_injective(sbs) ≡
(
    ∀ sb1, sb2 : T.SBID,
            dir : T.Direction •
                sb1 ≠ sb2 ⇒
                    getSBSeg(sb1,dir,sbs) ≠ getSBSeg(sb2,dir,sbs)
),

/* The type of a SB must conform
     with the result of getSBSeg */
getSBSegType_wf : SBs → Bool
getSBSegType_wf(sbs) ≡
(
    ∀ sb : T.SBID •
        case getSBType(sb,sbs) of
            T.ENDSB →
```

```
                    (∃! dir : T.Direction, esaID : T.ESAID •
                        esaID = T.dir2End(dir) ∧
                        getSBSeg(sb,dir,sbs) = T.esa(esaID)),
                T.POINTSB →
                    (∃! dir : T.Direction, seg1,seg2 : T.SegmentID •
                        getSBSeg(sb,dir,sbs) = T.point(seg1,seg2)),
                T.CROSSINGSB →
                    (∀ dir : T.Direction •
                        ∃ seg : T.SegmentID •
                            getSBSeg(sb,dir,sbs) = T.seg(seg)),
                T.PLAINSB →
                    (∀ dir : T.Direction •
                        ∃ seg : T.SegmentID •
                            getSBSeg(sb,dir,sbs) = T.seg(seg))
            end
        )

    end
```

Segs

```
context: AA_Types1
scheme AA_Segs1(T : AA_Types1) =
    class
        type
            /* Type of interest */
            Segs

        value
            segsConf : Segs,

            /* Segment observers */
            getSegSB : T.SegmentID × T.Direction × Segs ~→ T.SBID,
            getSegLength : T.SegmentID × Segs ~→ T.Length,
            getSegMaxSpeed : T.SegmentID × Segs ~→ T.Speed,
            segExistsInConf : T.SegmentID × Segs → Bool,

            /* Reservation− and brake−point observers */
            getResPoint : Segs → T.Length,
            getBrakePoint : Segs → T.Length,

            /* Invariant */
            is_wf : Segs → Bool
            is_wf(segs) ≡
                segsHaveConf(segs) ∧
                getSegSB_injective(segs) ∧
                brakeResPoint_wf(segs),

            /* A configuration for each Segment must exists */
            segsHaveConf : Segs → Bool
            segsHaveConf(segs) ≡
            (
                (∀ seg : T.SegmentID •
                    segExistsInConf(seg,segs)) ∧
                 getResPoint(segs) > 0.0 ∧
```

```
            getBrakePoint(segs) > 0.0
        ),

        /**
        * The SB in the end of a segment is different
        * for two different segments or they are the
        * same in both direction (being branches)
        **/
        getSegSB_injective : Segs → Bool
        getSegSB_injective(segs) ≡
        (
            ∀ seg1, seg2 : T.SegmentID,
                dir : T.Direction •
                    seg1 ≠ seg2 ⇒
                    (
                        getSegSB(seg1,dir,segs) ≠ getSegSB(seg2,dir,segs)
                    )
                        ∨
                    (
                    getSegSB(seg1,T.UP,segs) = getSegSB(seg2,T.UP,segs) ∧
                    getSegSB(seg1,T.DOWN,segs) = getSegSB(seg2,T.DOWN,segs)
                    )
        ),

        /* The reservation−point should be placed before
            the brake−point, i.e. there is a greater
            distance from the end of a segment to the
            reservation−point than to the brake−point */
        brakeResPoint_wf : Segs → Bool
        brakeResPoint_wf(segs) ≡
            getResPoint(segs) > getBrakePoint(segs)

    end
```

ESAs

```
context: AA_Types1
scheme AA_ESAs1(T : AA_Types1) =
    class
        type
            /* Type of interest */
            ESAs

        value
            esasConf : ESAs,

            /* ESA observers */
            getESASB : T.ESAID × ESAs ~→ T.SBID,
            getESALength : T.ESAID × ESAs ~→ T.Length,
            esaExistsInConf : T.ESAID × ESAs → Bool,

            /* Invariants */
            is_wf : ESAs → Bool
            is_wf(esas) ≡
                esasHaveConf(esas),
```

```
        /* A configuration for each ESA must exists */
        esasHaveConf : ESAs → Bool
        esasHaveConf(esas) ≡
        (
            ∀ esa : T.ESAID •
                esaExistsInConf(esa,esas)
        )

  end
```

Trains

```
context: AA_Types1
scheme AA_Trains1(T : AA_Types1) =
  class
    type
        /* Type of interest */
        Trains

    value
        trainsConf : Trains,

        /* Train observers */
        getTrainLength : T.TrainID × Trains ~→ T.Length,
        getTrainMaxSpeed : T.TrainID × Trains ~→ T.Speed,
        getTrainMaxAcc : T.TrainID × Trains ~→ T.Acceleration,
        getTrainMaxDec : T.TrainID × Trains ~→ T.Acceleration,
        trainExistsInConf : T.TrainID × Trains → Bool,

        /* Invariants */
        is_wf : Trains → Bool
        is_wf(trains) ≡
            trainsHaveConf(trains),

        /* A configuration for each Train must exists */
        trainsHaveConf : Trains → Bool
        trainsHaveConf(trains) ≡
        (
            ∀ t : T.TrainID •
                trainExistsInConf(t,trains)
        )

  end
```

F.2.3 Dynamics

```
context: AA_Statics1, AA_TrainDyn1, AA_SBDyn1, AA_Types1
scheme AA_Dynamics1(T : AA_Types1, S : AA_Statics1(T)) =
  class
    object
        TD : AA_TrainDyn1(T,S),
```

```
    SD : AA_SBDyn1(T,S)

type
    /∗ Type of interest ∗/
    State = TD.TrainStates × SD.SBStates

value
    initState : State = (TD.initTrainStates, SD.initSBStates),

    /∗ Point observer ∗/
    getPointPosition : T.SBID × State × S.Configuration ~→
                                                    T.PointPosition
    getPointPosition(sb,(ts,sbs),con) ≡
        SD.getPointPosition(sb,sbs,con)
    pre S.getSBType(sb,con) = T.POINTSB,

    getPointTicks : T.SBID × State × S.Configuration ~→ T.Tick
    getPointTicks(sb,(ts,sbs),con) ≡
        SD.getPointTicks(sb,sbs,con)
    pre S.getSBType(sb,con) = T.POINTSB,

    /∗ Point generator ∗/
    setPointPosition : T.SBID × T.PointPosition × State ×
                                          S.Configuration ~→ State
    setPointPosition(sb,ppos,(ts,ss),con) ≡
        (ts,SD.setPointPosition(sb,ppos,ss,con))
    pre S.getSBType(sb,con) = T.POINTSB ∧
            ∼trainOnJunction(sb,con,(ts,ss)),

    setPointTicks : T.SBID × T.Tick × State × S.Configuration ~→ State
    setPointTicks(sb,ticks,(ts,ss),con) ≡
        (ts,SD.setPointTicks(sb,ticks,ss,con))
    pre S.getSBType(sb,con) = T.POINTSB,

    /∗ Crossing observer ∗/
    getBarrierPosition : T.SBID × State × S.Configuration ~→ T.BarrierPosition
    getBarrierPosition(sb,(ts,sbs),con) ≡
        SD.getBarrierPosition(sb,sbs,con)
    pre S.getSBType(sb,con) = T.CROSSINGSB,

    getSignalStatus : T.SBID × State × S.Configuration ~→ T.SignalStatus
    getSignalStatus(sb,(ts,sbs),con) ≡
        SD.getSignalStatus(sb,sbs,con)
    pre S.getSBType(sb,con) = T.CROSSINGSB,

    /∗ Crossing generator ∗/
    setBarrierPosition : T.SBID × T.BarrierPosition × State ×
                                          S.Configuration ~→ State
    setBarrierPosition(sb,bPos,(ts,sbs),con) ≡
        (ts,SD.setBarrierPosition(sb,bPos,sbs,con))
    pre S.getSBType(sb,con) = T.CROSSINGSB,

    setSignalStatus : T.SBID × T.SignalStatus × State ×
                                          S.Configuration ~→ State
    setSignalStatus(sb,sigStat,(ts,sbs),con) ≡
        (ts,SD.setSignalStatus(sb,sigStat,sbs,con))
```

```
pre S.getSBType(sb,con) = T.CROSSINGSB,

/* Sensor observer */
getSensorStatus : T.SBID × State → T.SensorStatus
getSensorStatus(sb,(ts,sbs)) ≡
    SD.getSensorStatus(sb,sbs),

/* Sensor generator */
setSensorStatus : T.SBID × T.SensorStatus × State ×
                                         S.Configuration ~→ State
setSensorStatus(sb,senStat,(ts,ss),con) ≡
    (ts,SD.setSensorStatus(sb,senStat,ss))
pre sensor_guard(sb,senStat,con,(ts,ss)),

/* Train observer */
getTrainAcc : T.TrainID × State → T.Acceleration
getTrainAcc(tID,(ts,sbs)) ≡
    TD.getTrainAcc(tID,ts),

getTrainSpeed : T.TrainID × State → T.Speed
getTrainSpeed(tID,(ts,sbs)) ≡
    TD.getTrainSpeed(tID,ts),

getTrainPosition : T.TrainID × State → T.TrainPosition
getTrainPosition(tID,(ts,sbs)) ≡
    TD.getTrainPosition(tID,ts),

getTrainDirection : T.TrainID × State → T.Direction
getTrainDirection(tID,(ts,sbs)) ≡
    TD.getTrainDirection(tID,ts),

/* Train generator */
setTrainAcc : T.TrainID × T.Acceleration × State ×
                                         S.Configuration ~→ State
setTrainAcc(tID,acc,(ts,ss),con) ≡
    (TD.setTrainAcc(tID,acc,ts,con),ss)
pre acc ≤ S.getTrainMaxAcc(tID,con) ∧
        S.getTrainMaxDec(tID,con) ≤ acc,

setTrainSpeed : T.TrainID × T.Speed × State ×
                                         S.Configuration ~→ State
setTrainSpeed(tID,speed,(ts,ss),con) ≡
    (TD.setTrainSpeed(tID,speed,ts,con),ss)
pre speed ≤ S.getTrainMaxSpeed(tID,con),

setTrainPosition : T.TrainID × T.TrainPosition × State ×
                                         S.Configuration ~→ State
setTrainPosition(tID,pos,(ts,ss),con) ≡
    (TD.setTrainPosition(tID,pos,ts,con),ss)
pre ∼trainPositionOccupied(tID,pos,(ts,ss),con) ∧
        ∼tpDerailed(pos,getTrainDirection(tID,(ts,ss)),(ts,ss),con),

setTrainDirection : T.TrainID × T.Direction × State ~→ State
setTrainDirection(tID,dir,(ts,ss)) ≡
    (TD.setTrainDirection(tID,dir,ts),ss)
pre getTrainSpeed(tID,(ts,ss)) = 0.0 ∨
```

```
        getTrainDirection(tID,(ts,ss)) = dir,

changeTrainDirection : T.TrainID × State × S.Configuration → State
changeTrainDirection(t,s,con) ≡
    let
        dir = T.inverseDir(getTrainDirection(t,s)),
        tp = getTrainPosition(t,s),

        front = T.frontPos(tp),
        rear = T.rearPos(tp),

        tp = T.mk_TrainPosition(rear,front),

        s = setTrainDirection(t,dir,s)
    in
        setTrainPosition(t,tp,s,con)
    end,

/* Processes */
tick : T.Tick × S.Configuration × State ~→ State
tick(tick,con,s) ≡
    let
        s = tickPoints(tick,con,s),
        s = tickCrossings(tick,con,s),
        s = tickTrains(tick,con,s)
    in
        s
    end,

tickPoints : T.Tick × S.Configuration × State ~→ State
tickPoints(tick,con,s) ≡
    let
        points = { p | p : T.SBID • S.getSBType(p,con) = T.POINTSB }
    in
        pointProcess(points,tick,con,s)
    end,

pointProcess : T.SBID-set × T.Tick × S.Configuration ×
                                                    State ~→ State
pointProcess(points,tick,con,s) ≡
    if(points = {})
    then
        s
    else
        let
            p : T.SBID • p ∈ points,
            points = points \ {p},
            s = updatePoint(p,tick,con,s)
        in
            pointProcess(points,tick,con,s)
        end
    end
pre S.sbsArePoints(points,con),

updatePoint : T.SBID × T.Tick × S.Configuration × State ~→ State
updatePoint(p,tick,con,s) ≡
    let
```

```
            pp = getPointPosition(p,s,con)
        in
            case pp of
                T.MOVINGDOWN → s † setPointPosition(p,T.DOWN,s,con),
                T.MOVINGUP → s † setPointPosition(p,T.UP,s,con),
                _ → s
            end
        end
    pre S.getSBType(p,con) = T.POINTSB,

    tickCrossings : T.Tick × S.Configuration × State ~→ State
    tickCrossings(tick,con,s) ≡
        let
            crossings = { c | c : T.SBID • S.getSBType(c,con) = T.CROSSINGSB }
        in
            crossingProcess(crossings,tick,con,s)
        end,

    crossingProcess : T.SBID-set × T.Tick × S.Configuration ×
                                                        State ~→ State
    crossingProcess(crossings,tick,con,s) ≡
        if(crossings = {})
        then
            s
        else
            let
                c : T.SBID • c ∈ crossings,
                crossings = crossings \ {c},
                s = updateCrossing(c,tick,con,s)
            in
                crossingProcess(crossings,tick,con,s)
            end
        end
    pre S.sbsAreCrossings(crossings,con),

    updateCrossing : T.SBID × T.Tick × S.Configuration × State ~→ State
    updateCrossing(cr,tick,con,s) ≡
        let
            bp = getBarrierPosition(cr,s,con),
            ss = getSignalStatus(cr,s,con)
        in
            case bp of
                T.UP →
                (
                    if(ss = T.ON)
                    then
                        s †
                        setBarrierPosition(cr,T.MOVINGDOWN,s,con)
                    else
                        s
                    end
                ),
                T.MOVINGDOWN →
                (
                    s †
                    (
                        let
```

```
                  bp = setBarrierPosition(cr,T.DOWN,s,con)
                in
                  setSignalStatus(cr,T.OFF,s,con)
                end
              )
          ),
          T.DOWN → s,
          T.MOVINGUP → s ⌈⌉ setBarrierPosition(cr,T.UP,s,con)
        end
      end
    pre S.getSBType(cr,con) = T.CROSSINGSB,

    tickTrains : T.Tick × S.Configuration × State ~→ State
    tickTrains(tick,con,s) ≡
    let
      trains = { t | t : T.TrainID}
    in
      trainProcess(trains,tick,con,s)
    end,

    trainProcess : T.TrainID-set × T.Tick × S.Configuration × State ~→ State
    trainProcess(trains,tick,con,s) ≡
      if(trains = {})
      then
        s
      else
        let
          t : T.TrainID • t ∈ trains,
          trains = trains \ {t},
          s = updateTrain(t,tick,con,s)
        in
          trainProcess(trains,tick,con,s)
        end
      end,

    updateTrain : T.TrainID × T.Tick × S.Configuration × State ~→ State,

    /* Auxiliary functions */

    /* Returns the front segment of a train. If front is on ESA then
       the rear segment is returned. This is used for speed checking */
    getTrainLoc : T.TrainID × State → T.Location
    getTrainLoc(t,ds) ≡
      let
        tp = getTrainPosition(t,ds),
        frontLoc = T.getLoc(T.frontPos(tp)),
        rearLoc = T.getLoc(T.rearPos(tp))
      in
        case frontLoc of
          T.isESA(esa) → rearLoc,
          _ → frontLoc
        end
      end
    pre ∼trainInESA(t,ds),

  tpDerailed : T.TrainPosition × T.Direction × State × S.Configuration → Bool
  tpDerailed(tp,dir,s,con) ≡
```

```
if (~T.oneLoc(tp) ∧ ~T.segPosIsESA(T.frontPos(tp))) then
    let
        seg = T.getSeg(T.frontLoc(tp)),
        sb = S.getSegSB(seg,T.inverseDir(dir),con)
    in
        case S.getSBType(sb,con) of
                T.POINTSB →
                (
                    if (dir = S.getPointDir(sb,con)) then
                        pointConnected(sb,T.getSeg(T.frontLoc(tp)),s,con)
                    else
                        pointConnected(sb,T.getSeg(T.rearLoc(tp)),s,con)
                    end
                ),

                T.CROSSINGSB →
                (
                    getBarrierPosition(sb,s,con) = T.DOWN
                ),

                _ → false
        end
    end
else
    false
end,

getESATrains : T.ESAID × State → T.TrainID-set
getESATrains(esa,s) ≡
    { t | t : T.TrainID • T.trainOnlyOnESA(getTrainPosition(t,s)) },

getTrainSegments : T.TrainID × State → T.SegmentID-set
getTrainSegments(t,(ts,ss)) ≡
    TD.getTrainSegments(t,ts),

getTrainBranch : T.TrainID × State × S.Configuration ~→ T.SegmentID
getTrainBranch(t,s,con) ≡
(
    let
        seg : T.SegmentID • seg ∈ getTrainSegments(t,s) ∧
                                        S.segIsBranch(seg,con)
    in
        seg
    end
)
pre (∃ sb : T.SBID • trainOnJunction(t,sb,con,s)),

trainOnSegment : T.TrainID × T.SegmentID × S.Configuration × State → Bool
trainOnSegment(tID,seg,con,ds) ≡
    seg ∈ getTrainSegments(tID,ds),

trainOnJunction : T.TrainID × T.SBID × S.Configuration × State → Bool
trainOnJunction(t,sb,con,ds) ≡
(
    S.getSBType(sb,con) = T.POINTSB ∧
    trainOnSensor(t,sb,con,ds)
),
```

```
trainOnJunction : T.SBID × S.Configuration × State → Bool
trainOnJunction(sb,con,s) ≡
    S.getSBType(sb,con) = T.POINTSB ∧
    trainOnSensor(sb,con,s),

trainOnSensor : T.TrainID × T.SBID × S.Configuration × State → Bool
trainOnSensor(t,sb,con,ds) ≡
(
    ∃ dir : T.Direction, tPos : T.TrainPosition,
                sp1,sp2 : T.SegmentPosition •
                    tPos = getTrainPosition(t,ds) ∧
        T.segPosInSBSeg(sp1, S.getSBSeg(sb,dir,con)) ∧
        T.segPosInSBSeg(sp2, S.getSBSeg(sb,T.inverseDir(dir),con))
),

trainOnSensor : T.SBID × S.Configuration × State → Bool
trainOnSensor(sb,con,s) ≡
(
    ∃ t : T.TrainID, dir : T.Direction, tPos : T.TrainPosition,
                sp1,sp2 : T.SegmentPosition •
                    tPos = getTrainPosition(t,s) ∧
        T.segPosInSBSeg(sp1, S.getSBSeg(sb,dir,con)) ∧
        T.segPosInSBSeg(sp2, S.getSBSeg(sb,T.inverseDir(dir),con))
),

trainInESA : T.TrainID × State → Bool
trainInESA(t,(ts,ss)) ≡
    TD.trainInESA(t,ts),

trainInESADrivingOut : T.TrainID × State ~→ Bool
trainInESADrivingOut(t,(ts,ss)) ≡
    TD.trainInESADrivingOut(t,ts),

trainFrontInESA : T.TrainID × State → Bool
trainFrontInESA(t,(ts,ss)) ≡
    TD.trainFrontInESA(t,ts),

/* Telling if a train is (partly) on a single line */
trainOnSingleLine : T.TrainID × S.Configuration × State → Bool
trainOnSingleLine(t,con,s) ≡
    let
        tPos = getTrainPosition(t,s),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∃ s : T.SegmentID •
                    s ∈ segSet ⇒
                        ∼S.segIsBranch(s,con)
            )
        else
            false
        end
    end,

/* Telling if a train is (partly) on a branch */
```

```
trainOnBranch : T.TrainID × S.Configuration × State → Bool
trainOnBranch(t,con,s) ≡
    let
        tPos = getTrainPosition(t,s),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∃ s : T.SegmentID •
                    s ∈ segSet ⇒
                        S.segIsBranch(s,con)
            )
        else
            false
        end
    end,

/* Telling if a train is only on a branch */
trainOnlyOnBranch : T.TrainID × S.Configuration × State → Bool
trainOnlyOnBranch(t,con,s) ≡
    let
        tPos = getTrainPosition(t,s),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∀ s : T.SegmentID •
                    s ∈ segSet ⇒
                        S.segIsBranch(s,con)
            )
        else
            false
        end
    end,

pointConnected : T.SBID × T.SegmentID × State × S.Configuration → Bool
pointConnected(sb,seg,ds,con) ≡
    let
        pointSegs = S.getSBPointSegs(sb,con)
    in
        case getPointPosition(sb,ds,con) of
            T.UP → (seg = T.getUpBranch(pointSegs)),
            T.DOWN → (seg = T.getDownBranch(pointSegs)),
            _ → false
        end
    end
pre S.getSBType(sb,con) = T.POINTSB,

trainFrontLoc : T.TrainID × State → T.Location
trainFrontLoc(t,(ts,ss)) ≡
    TD.trainFrontLoc(t,ts),


sensor_guard : T.SBID × T.SensorStatus × S.Configuration × State → Bool
sensor_guard(sen,ss,con,s) ≡
    (ss = T.ACTIVE ∧ trainOnSensor(sen,con,s)) ∨
    (ss = T.INACTIVE ∧ ∼trainOnSensor(sen,con,s)),
```

```
decelerateTrain : T.TrainID × S.Configuration × State → State
decelerateTrain(t,con,s) ≡
    if(getTrainSpeed(t,s) ≠ 0.0)
    then
        let
            maxDec = S.getTrainMaxDec(t,con),
            curDec = getTrainAcc(t,s)
        in
            if(maxDec ≠ curDec)
            then
                setTrainAcc(t,maxDec,s,con)
            else
                s
            end
        end
    else
        setTrainAcc(t,0.0,s,con)
    end,

accelerateTrain : T.TrainID × S.Configuration × State ~→ State
accelerateTrain(tID,con,s) ≡
    setTrainAcc(tID,S.getTrainMaxAcc(tID,con),s,con),

commonSegs : T.TrainPosition × T.TrainID × State → T.SegmentID-set
commonSegs(tp1,t2,ds) ≡
    T.trainPosSegs(tp1) ∩ getTrainSegments(t2,ds),

commonSegs : T.TrainID × T.TrainID × State → T.SegmentID-set
commonSegs(t1,t2,ds) ≡
    getTrainSegments(t1,ds) ∩ getTrainSegments(t2,ds),

trainPositionOccupied : T.TrainID × T.TrainPosition × State ×
                                        S.Configuration → Bool
trainPositionOccupied(t1,tp1,ds,con) ≡
(
    ∀ segs : T.SegmentID-set, dir1,dir2 : T.Direction,
            tp1,tp2 : T.TrainPosition •
                ∃ t2 : T.TrainID •
                    t2 ≠ t1 ∧
                    segs = commonSegs(tp1,t2,ds) ∧
                    segs ≠ {} ∧
                    (dir1,dir2) = (getTrainDirection(t1,ds),getTrainDirection(t2,ds)) ∧
                    tp2 = getTrainPosition(t2,ds) ∧

                    case dir1 of
                        T.UP →
                        (
                            if (dir1 = dir2)
                            then
                                S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con) ⇒
                                    ∼S.segPosLower(T.frontPos(tp1),T.rearPos(tp2),con)
                            else
                                ∼S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
                            end
                        ),
```

```
                        T.DOWN →
                        (
                            if (dir1 = dir2) then
                                ~S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con) ⇒
                                    S.segPosLower(T.frontPos(tp1),T.rearPos(tp2),con)
                            else
                                S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
                            end
                        )
                    end
),

/* Invariants etc. */

/* Telling if the railway line is safe */
safe : State × S.Configuration → Bool
safe(s,con) ≡
    is_wf(s,con) ∧
    noCollisions(con,s) ∧
    trainPosPossible(con,s) ∧
    pointsSafe(con,s) ∧
    crossingsSafe(con,s),

/**
*   The position of a train may not overlap
*   with the position of other trains
**/
noCollisions : S.Configuration × State → Bool
noCollisions(con,s) ≡
(
    ∀ t : T.TrainID •
        ~trainPositionOccupied(t,getTrainPosition(t,s),s,con)
),

/**
*   Trains cannot end up on same segment
*   driving in opposite directions away from each other.
*
*   If two train are on same segment driving in opposite
*   directions then the train driving up must be lower
*   on the line than the train driving down.
**/
trainPosPossible : S.Configuration × State → Bool
trainPosPossible(con,ds) ≡
(
    ∀ t1,t2 : T.TrainID, segs : T.SegmentID-set,
            tp1,tp2 : T.TrainPosition, seg : T.SegmentID •
                commonSegs(t1,t2,ds) ≠ {} ∧
                (tp1,tp2) = (getTrainPosition(t1,ds),getTrainPosition(t1,ds)) ∧
                getTrainDirection(t1,ds) ≠ getTrainDirection(t2,ds) ∧
                getTrainDirection(t1,ds) = T.UP
                    ⇒
                        S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
),

/**
*   If the train is located upon a junction,
```

```
*   the point must be connected to the
*   branch, on which the train is located
**/
pointsSafe : S.Configuration × State → Bool
pointsSafe(con,ds) ≡
(
    ∀ sb : T.SBID, t : T.TrainID, seg : T.SegmentID •
        trainOnJunction(t,sb,con,ds) ∧
        trainOnSegment(t,seg,con,ds) ∧
        S.segIsBranch(seg,con) ⇒
            pointConnected(sb,seg,ds,con)
),

/* When a train is located on a crossing
     the barriers must be down */
crossingsSafe : S.Configuration × State → Bool
crossingsSafe(con,s) ≡
(
    ∀ sb : T.SBID •
        S.getSBType(sb,con) = T.CROSSINGSB ∧
        trainOnSensor(sb,con,s) ⇒
            getBarrierPosition(sb,s,con) = T.DOWN
),

/* Wellformedness */
is_wf : State × S.Configuration → Bool
is_wf((ts,ss),con) ≡
    TD.is_wf(ts,con) ∧
    SD.is_wf(ss,con),

init_req : State × S.Configuration → Bool
init_req((ts,ss),con) ≡
    is_wf((ts,ss),con) ∧
    TD.init_req(ts) ∧
    SD.init_req(ss,con)


axiom
    [wellformedness]
        init_req(initState,S.conf),

    /** Maintaining the wellformedness of the state when
    *   applied to a generator with its precondition
    *   General form:
    *     is_wf(state) ∧ pre_cond(..,state) ⇒
    *       is_wf(gen_(state))
    **/

    [gen_wf_setPointPosition]
    ∀ p : T.SBID, pp : T.PointPosition, s : State,
            con : S.Configuration •
        is_wf(s,con) ∧
        S.getSBType(p,con) = T.POINTSB ∧
        ∼trainOnJunction(p,con,s)
            ⇒
        is_wf(setPointPosition(p,pp,s,con),con),
```

```
[ gen_wf_setPointTicks ]
∀ p : T.SBID, ticks : T.Tick, s : State,
      con : S.Configuration •
   is_wf(s,con) ∧
   S.getSBType(p,con) = T.POINTSB
      ⇒
   is_wf(setPointTicks(p,ticks,s,con),con),

[ gen_wf_setBarrierPosition ]
∀ cr : T.SBID, bp : T.BarrierPosition, s : State,
      con : S.Configuration •
   is_wf(s,con) ∧
   S.getSBType(cr,con) = T.CROSSINGSB
      ⇒
   is_wf(setBarrierPosition(cr,bp,s,con),con),

[ gen_wf_setSignalStatus ]
∀ cr : T.SBID, ss : T.SignalStatus, s : State,
      con : S.Configuration •
   is_wf(s,con) ∧
   S.getSBType(cr,con) = T.CROSSINGSB
      ⇒
   is_wf(setSignalStatus(cr,ss,s,con),con),

[ gen_wf_setSensorStatus ]
∀ sen : T.SBID, ss : T.SensorStatus, s : State,
      con : S.Configuration •
   is_wf(s,con) ∧
   sensor_guard(sen,ss,con,s)
      ⇒
   is_wf(setSensorStatus(sen,ss,s,con),con),

[ gen_wf_setTrainAcc ]
∀ t : T.TrainID, acc : T.Acceleration,
   con : S.Configuration, s : State •
   is_wf(s,con) ∧
   acc ≤ S.getTrainMaxAcc(t,con) ∧
   S.getTrainMaxDec(t,con) ≤ acc
      ⇒
   is_wf(setTrainAcc(t,acc,s,con),con),

[ gen_wf_setTrainSpeed ]
∀ t : T.TrainID, sp : T.Speed,
   con : S.Configuration, s : State •
   is_wf(s,con) ∧
   sp ≤ S.getTrainMaxSpeed(t,con)
      ⇒
   is_wf(setTrainSpeed(t,sp,s,con),con),

[ gen_wf_setTrainPosition ]
∀ t : T.TrainID, pos : T.TrainPosition, s : State,
      con : S.Configuration •
   is_wf(s,con) ∧
   ∼trainPositionOccupied(t,pos,s,con)
      ⇒
   is_wf(setTrainPosition(t,pos,s,con),con),
```

```
      [ gen_wf_setTrainDirection ]
      ∀ t : T.TrainID, dir : T.Direction, s : State,
            con : S.Configuration •
         is_wf(s,con) ∧
         (
            getTrainSpeed(t,s) = 0.0 ∨
            getTrainDirection(t,s) = dir
         )
            ⇒
         is_wf(setTrainDirection(t,dir,s),con)

   end
```

TrainDyn

```
context: AA_Types1, AA_Statics1
scheme AA_TrainDyn1(T : AA_Types1, S : AA_Statics1(T)) =
   class
      type
         TrainStates

      value
         initTrainStates : TrainStates,

         /* Train observer */
         getTrainAcc : T.TrainID × TrainStates ~→ T.Acceleration,
         getTrainSpeed : T.TrainID × TrainStates ~→ T.Speed,
         getTrainPosition : T.TrainID × TrainStates~→T.TrainPosition,
         getTrainDirection : T.TrainID × TrainStates ~→ T.Direction,

         /* Train generator */
         setTrainAcc : T.TrainID × T.Acceleration × TrainStates ×
                                        S.Configuration ~→ TrainStates,
         setTrainSpeed : T.TrainID × T.Speed × TrainStates ×
                                        S.Configuration ~→ TrainStates,
         setTrainPosition : T.TrainID × T.TrainPosition ×
                                     TrainStates × S.Configuration ~→
                                     TrainStates,
         setTrainDirection : T.TrainID × T.Direction ×
                                        TrainStates ~→ TrainStates,

         /* Tells if a train has a state in the system */
         trainStateExists : T.TrainID × TrainStates → Bool,

         /* Front and rear position of a train must be exactly
             'train length' apart */
         train_pos_ok : T.TrainID × T.TrainPosition ×
                                  TrainStates × S.Configuration ~→ Bool
         train_pos_ok(t,tp,s,con) ≡
         (
            let
               T.mk_TrainPosition(posFront,posRear) = tp
            in
               (S.distance(posFront,posRear,con) =
```

```
            S.getTrainLength(t,con)) ∧
        train_pos_dir_ok(getTrainDirection(t,s),tp,s,con)
    end
),

/∗ If train drives UP then rear pos must be
     lower than front pos and vice versa ∗/
train_pos_dir_ok : T.Direction × T.TrainPosition ×
                                  TrainStates × S.Configuration → Bool
train_pos_dir_ok(dir,tp,s,con) ≡
(
    case dir of
        T.UP →
        (
            S.segPosLower(T.rearPos(tp),T.frontPos(tp),con)
        ),

        T.DOWN →
        (
            S.segPosLower(T.frontPos(tp),T.rearPos(tp),con)
        )
    end
),

getTrainSegments : T.TrainID × TrainStates ~→
                                                T.SegmentID-set
getTrainSegments(t,s) ≡
    T.trainPosSegs(getTrainPosition(t,s)),

trainInESA : T.TrainID × TrainStates ~→ Bool
trainInESA(t,s) ≡
    T.trainOnlyOnESA(getTrainPosition(t,s)),

trainInESADrivingOut : T.TrainID × TrainStates → Bool
trainInESADrivingOut(t,s) ≡
    if(∼T.trainOnlyOnESA(getTrainPosition(t,s)))
    then
        false
    else
        let
            segPos = T.frontPos(getTrainPosition(t,s)),
            esa = T.getESA(T.getLoc(segPos)),
            dir = getTrainDirection(t,s)
        in
            T.end2Dir(esa) ≠ dir
        end
    end,

trainFrontInESA : T.TrainID × TrainStates ~→ Bool
trainFrontInESA(t,s) ≡
    let
        tPos = getTrainPosition(t,s)
    in
        T.segPosIsESA(T.frontPos(tPos))
    end,

trainFrontLoc : T.TrainID × TrainStates ~→ T.Location
```

```
trainFrontLoc(t,ds) ≡
    case T.frontLoc(getTrainPosition(t,ds)) of
        T.isESA(_) → T.rearLoc(getTrainPosition(t,ds)),
        T.isSeg(seg) → T.isSeg(seg)
    end,

is_wf : TrainStates × S.Configuration ~→ Bool
is_wf(s,con) ≡
    allTrainStatesExist(s) ∧
    train_pos_wf(con,s),

/* All trains must have a state */
allTrainStatesExist : TrainStates → Bool
allTrainStatesExist(s) ≡
(
    ∀ trainID : T.TrainID •
        trainStateExists(trainID,s)
),

/* Front and rear position of a train must be exactly
    'train length' apart */
train_pos_wf : S.Configuration × TrainStates ~→ Bool
train_pos_wf(con,s) ≡
(
    ∀ t : T.TrainID •
        train_pos_ok(t,getTrainPosition(t,s),s,con)
),

init_req : TrainStates ~→ Bool
init_req(s) ≡
    allTrainsInESA(s) ∧
    allTrainsStopped(s) ∧
    allTrainsFacingLine(s),

allTrainsInESA : TrainStates ~→ Bool
allTrainsInESA(s) ≡
(
    ∀ t : T.TrainID •
        trainInESA(t,s)
),

allTrainsStopped : TrainStates ~→ Bool
allTrainsStopped(s) ≡
(
    ∀ t : T.TrainID •
        getTrainSpeed(t,s) = 0.0 ∧
        getTrainAcc(t,s) = 0.0
),

allTrainsFacingLine : TrainStates ~→ Bool
allTrainsFacingLine(s) ≡
(
    ∀ t : T.TrainID, esa : T.ESAID •
        T.isESA(esa) =
                T.getLoc(T.frontPos(getTrainPosition(t,s))) ∧
        (esa = T.LOW ⇒ getTrainDirection(t,s) = T.UP) ∧
```

```
            (esa = T.HIGH ⇒ getTrainDirection(t,s) = T.DOWN)
      )

axiom
      /**
      * Observer_generator axioms
      **/

      /* getTrainAcc_gen */

      [ getTrainAcc_setTrainAcc ]
      ∀ s : TrainStates, t1,t2 : T.TrainID,
            acc : T.Acceleration,
            con : S.Configuration •
         getTrainAcc(t1,setTrainAcc(t2,acc,s,con)) ≡
            if(t1 = t2)
            then
               acc
            else
               getTrainAcc(t1,s)
            end
      pre acc ≤ S.getTrainMaxAcc(t2,con) ∧
            S.getTrainMaxDec(t2,con) ≤ acc ∧
            trainStateExists(t1,s) ∧
            trainStateExists(t2,s),

      [ getTrainAcc_setTrainSpeed ]
      ∀ s : TrainStates, t1,t2 : T.TrainID, sp : T.Speed,
         con : S.Configuration •
         getTrainAcc(t1,setTrainSpeed(t2,sp,s,con)) ≡
            getTrainAcc(t1,s)
      pre sp ≤ S.getTrainMaxSpeed(t2,con) ∧
            trainStateExists(t1,s) ∧
            trainStateExists(t2,s),

      [ getTrainAcc_setTrainPosition ]
      ∀ s : TrainStates, t1,t2 : T.TrainID, tp : T.TrainPosition,
            con : S.Configuration •
         getTrainAcc(t1,setTrainPosition(t2,tp,s,con)) ≡
            getTrainAcc(t1,s)
      pre train_pos_ok(t2,tp,s,con) ∧
            trainStateExists(t1,s) ∧
            trainStateExists(t2,s),

      [ getTrainAcc_setTrainDirection ]
      ∀ s : TrainStates, t1,t2 : T.TrainID, dir : T.Direction•
         getTrainAcc(t1,setTrainDirection(t2,dir,s)) ≡
            getTrainAcc(t1,s)
      pre (getTrainSpeed(t2,s) = 0.0 ∨
            getTrainDirection(t2,s) = dir) ∧
            trainStateExists(t1,s) ∧
            trainStateExists(t2,s),

      /* getTrainSpeed_gen */

      [ getTrainSpeed_setTrainAcc ]
      ∀ s : TrainStates, t1,t2 : T.TrainID, acc : T.Acceleration,
```

```
    con : S.Configuration •
    getTrainSpeed(t1,setTrainAcc(t2,acc,s,con)) ≡
        getTrainSpeed(t1,s)
pre acc ≤ S.getTrainMaxAcc(t2,con) ∧
        S.getTrainMaxDec(t2,con) ≤ acc ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainSpeed_setTrainSpeed ]
∀ s : TrainStates, t1,t2 : T.TrainID, sp : T.Speed,
    con : S.Configuration •
    getTrainSpeed(t1,setTrainSpeed(t2,sp,s,con)) ≡
        if(t1 = t2)
        then
            sp
        else
            getTrainSpeed(t1,s)
        end
pre sp ≤ S.getTrainMaxSpeed(t2,con) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainSpeed_setTrainPosition ]
∀ s : TrainStates, t1,t2 : T.TrainID, tp : T.TrainPosition,
        con : S.Configuration •
    getTrainSpeed(t1,setTrainPosition(t2,tp,s,con)) ≡
        getTrainSpeed(t1,s)
pre train_pos_ok(t2,tp,s,con) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainSpeed_setTrainDirection ]
∀ s : TrainStates, t1,t2 : T.TrainID, dir : T.Direction•
    getTrainSpeed(t1,setTrainDirection(t2,dir,s)) ≡
        getTrainSpeed(t1,s)
pre (getTrainSpeed(t2,s) = 0.0 ∨
        getTrainDirection(t2,s) = dir) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

/* getTrainPosition_gen */

[ getTrainPosition_setTrainAcc ]
∀ s : TrainStates, t1,t2 : T.TrainID, acc : T.Acceleration,
    con : S.Configuration •
    getTrainPosition(t1,setTrainAcc(t2,acc,s,con)) ≡
        getTrainPosition(t1,s)
pre acc ≤ S.getTrainMaxAcc(t2,con) ∧
        S.getTrainMaxDec(t2,con) ≤ acc ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s),

[ getTrainPosition_setTrainSpeed ]
∀ s : TrainStates, t1,t2 : T.TrainID, sp : T.Speed,
    con : S.Configuration •
    getTrainPosition(t1,setTrainSpeed(t2,sp,s,con)) ≡
        getTrainPosition(t1,s)
```

```
pre sp ≤ S.getTrainMaxSpeed(t2,con) ∧
      trainStateExists(t1,s) ∧
      trainStateExists(t2,s),

[ getTrainPosition_setTrainPosition ]
∀ s : TrainStates, t1,t2 : T.TrainID, tp : T.TrainPosition,
      con : S.Configuration •
   getTrainPosition(t1,setTrainPosition(t2,tp,s,con)) ≡
      if(t1 = t2)
      then
         tp
      else
         getTrainPosition(t1,s)
      end
pre train_pos_ok(t2,tp,s,con) ∧
      trainStateExists(t1,s) ∧
      trainStateExists(t2,s),

[ getTrainPosition_setTrainDirection ]
∀ s : TrainStates, t1,t2 : T.TrainID, dir : T.Direction•
   getTrainPosition(t1,setTrainDirection(t2,dir,s)) ≡
      getTrainPosition(t1,s)
pre (getTrainSpeed(t2,s) = 0.0 ∨
      getTrainDirection(t2,s) = dir) ∧
      trainStateExists(t1,s) ∧
      trainStateExists(t2,s),

/∗ getTrainDirection_gen ∗/

[ getTrainDirection_setTrainAcc ]
∀ s : TrainStates, t1,t2 : T.TrainID, acc : T.Acceleration,
   con : S.Configuration •
   getTrainDirection(t1,setTrainAcc(t2,acc,s,con)) ≡
      getTrainDirection(t1,s)
pre acc ≤ S.getTrainMaxAcc(t2,con) ∧
      S.getTrainMaxDec(t2,con) ≤ acc ∧
      trainStateExists(t1,s) ∧
      trainStateExists(t2,s),

[ getTrainDirection_setTrainSpeed ]
∀ s : TrainStates, t1,t2 : T.TrainID, sp : T.Speed,
   con : S.Configuration •
   getTrainDirection(t1,setTrainSpeed(t2,sp,s,con)) ≡
      getTrainDirection(t1,s)
pre sp ≤ S.getTrainMaxSpeed(t2,con) ∧
      trainStateExists(t1,s) ∧
      trainStateExists(t2,s),

[ getTrainDirection_setTrainPosition ]
∀ s : TrainStates, t1,t2 : T.TrainID, tp : T.TrainPosition,
      con : S.Configuration •
   getTrainDirection(t1,setTrainPosition(t2,tp,s,con)) ≡
      getTrainDirection(t1,s)
pre train_pos_ok(t2,tp,s,con) ∧
      trainStateExists(t1,s) ∧
      trainStateExists(t2,s),
```

```
[ getTrainDirection_setTrainDirection ]
∀ s : TrainStates, t1,t2 : T.TrainID, dir : T.Direction•
    getTrainDirection(t1,setTrainDirection(t2,dir,s)) ≡
        if(t1 = t2)
        then
            dir
        else
            getTrainDirection(t1,s)
        end
pre (getTrainSpeed(t2,s) = 0.0 ∨
        getTrainDirection(t2,s) = dir) ∧
        trainStateExists(t1,s) ∧
        trainStateExists(t2,s)

end
```

SBDyn

```
context: AA_Types1, AA_Statics1
scheme AA_SBDyn1(T : AA_Types1, S : AA_Statics1(T)) =
    class
        type
            SBStates

        value
            initSBStates : SBStates,

            /* Point observer */
            getPointPosition : T.SBID × SBStates ×
                                        S.Configuration ~→ T.PointPosition,
            getPointTicks : T.SBID × SBStates ×
                                        S.Configuration ~→ T.Tick,

            /* Point generator */
            setPointPosition : T.SBID × T.PointPosition × SBStates ×
                                        S.Configuration ~→ SBStates,
            setPointTicks : T.SBID × T.Tick × SBStates ×
                                        S.Configuration ~→ SBStates,

            /* Crossing observer */
            getBarrierPosition : T.SBID × SBStates ×
                                        S.Configuration ~→ T.BarrierPosition,
            getSignalStatus : T.SBID × SBStates ×
                                        S.Configuration ~→ T.SignalStatus,

            /* Crossing generator */
            setBarrierPosition : T.SBID × T.BarrierPosition × SBStates ×
                                        S.Configuration ~→ SBStates,
            setSignalStatus : T.SBID × T.SignalStatus × SBStates ×
                                        S.Configuration ~→ SBStates,

            /* Sensor observer */
            getSensorStatus : T.SBID × SBStates → T.SensorStatus,
```

```
/∗ Sensor generator ∗/
setSensorStatus : T.SBID × T.SensorStatus ×
                                        SBStates → SBStates,

/∗ Tells if a sensor has a state in the system ∗/
sensorStateExists : T.SBID × SBStates → Bool,

/∗ Tells if a crossing has a state in the system ∗/
crossingStateExists : T.SBID × SBStates ×
                                    S.Configuration ~→ Bool,

/∗ Tells if a point has a state in the system ∗/
pointStateExists : T.SBID × SBStates ×
                                    S.Configuration ~→ Bool,

is_wf : SBStates × S.Configuration → Bool
is_wf(s,con) ≡
    allCrossingStatesExist(con,s) ∧
    allPointStatesExist(con,s) ∧
    allSensorStatesExist(s),

/∗ All crossings must have a state ∗/
allCrossingStatesExist : S.Configuration × SBStates → Bool
allCrossingStatesExist(con,s) ≡
(
    ∀ cr : T.SBID •
        S.getSBType(cr,con) = T.CROSSINGSB ⇒
            crossingStateExists(cr,s,con)
),

/∗ All points must have a state ∗/
allPointStatesExist : S.Configuration × SBStates → Bool
allPointStatesExist(con,s) ≡
(
    ∀ p : T.SBID •
        S.getSBType(p,con) = T.POINTSB ⇒
            pointStateExists(p,s,con)
),

/∗ All sensors must have a state ∗/
allSensorStatesExist : SBStates → Bool
allSensorStatesExist(s) ≡
(
    ∀ sen : T.SBID •
        sensorStateExists(sen,s)
),

init_req : SBStates × S.Configuration → Bool
init_req(s,con) ≡
    allBarriersUp(con,s) ∧
    allPointsNotShifting(con,s),

allBarriersUp : S.Configuration × SBStates → Bool
allBarriersUp(con,s) ≡
(
    ∀ sb : T.SBID •
        S.getSBType(sb,con) = T.CROSSINGSB ⇒
```

```
            getBarrierPosition(sb,s,con) = T.UP
),

allPointsNotShifting : S.Configuration × SBStates → Bool
allPointsNotShifting(con,s) ≡
(
    ∀ sb : T.SBID •
        S.getSBType(sb,con) = T.POINTSB ⇒
            getPointPosition(sb,s,con) ∈ { T.UP, T.DOWN }
)
```

axiom

```
/**
* Observer_generator axioms
* Only the relevant axioms are shown
* All other observer generator pairs are by definition
* unaffected by each other
**/

/* getPointPosition */
[getPointPosition_setPointPosition]
∀ sb1,sb2 : T.SBID, pp : T.PointPosition,
        s : SBStates,
        con : S.Configuration •
    getPointPosition(sb1,setPointPosition(sb2,pp,s,con),con) ≡
        if(sb1 = sb2)
        then
            pp
        else
            getPointPosition(sb1,s,con)
        end
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[getPointPosition_setPointTicks]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : SBStates,
        con : S.Configuration •
    getPointPosition(sb1,setPointTicks(sb2,ticks,s,con),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[getPointPosition_setBarrierPosition]
∀ sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        s : SBStates,
        con : S.Configuration •
    getPointPosition(sb1,setBarrierPosition(sb2,bp,s,con),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        pointStateExists(sb1,s,con) ∧
```

```
        crossingStateExists(sb2,s,con),

[getPointPosition_setSignalStatus]
∀ sb1,sb2 : T.SBID, ss : T.SignalStatus,
        s : SBStates,
        con : S.Configuration •
    getPointPosition(sb1,setSignalStatus(sb2,ss,s,con),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        pointStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[getPointPosition_setSensorStatus]
∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
        s : SBStates,
        con : S.Configuration •
    getPointPosition(sb1,setSensorStatus(sb2,ss,s),con) ≡
        getPointPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        sensorStateExists(sb2,s),

/* getPointTicks_gen*/

[getPointTicks_setPointPosition]
∀ sb1,sb2 : T.SBID, pp : T.PointPosition,
        s : SBStates,
        con : S.Configuration •
    getPointTicks(sb1,setPointPosition(sb2,pp,s,con),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[getPointTicks_setPointTicks]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : SBStates,
        con : S.Configuration •
    getPointTicks(sb1,setPointTicks(sb2,ticks,s,con),con) ≡
        if(sb1 = sb2)
        then
            ticks
        else
            getPointTicks(sb1,s,con)
        end
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[getPointTicks_setBarrierPosition]
∀ sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        s : SBStates,
        con : S.Configuration •
    getPointTicks(sb1,setBarrierPosition(sb2,bp,s,con),con) ≡
```

```
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        pointStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[getPointTicks_setSignalStatus]
∀ sb1,sb2 : T.SBID, ss : T.SignalStatus,
        s : SBStates,
        con : S.Configuration •
    getPointTicks(sb1,setSignalStatus(sb2,ss,s,con),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        pointStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[getPointTicks_setSensorStatus]
∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
        s : SBStates,
        con : S.Configuration •
    getPointTicks(sb1,setSensorStatus(sb2,ss,s),con) ≡
        getPointTicks(sb1,s,con)
pre S.getSBType(sb1,con) = T.POINTSB ∧
        pointStateExists(sb1,s,con) ∧
        sensorStateExists(sb2,s),

/* getBarrierPosition_gen */

[getBarrierPosition_setPointPosition]
∀ sb1, sb2 : T.SBID, pp : T.PointPosition,
        s : SBStates,
        con : S.Configuration •
    getBarrierPosition(sb1,setPointPosition(sb2,pp,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        crossingStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[getBarrierPosition_setPointTicks]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : SBStates,
        con : S.Configuration •
    getBarrierPosition(sb1,setPointTicks(sb2,ticks,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        crossingStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[getBarrierPosition_setBarrierPosition]
∀ s : SBStates, sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        con : S.Configuration •
    getBarrierPosition(sb1,setBarrierPosition(sb2,bp,s,con),con) ≡
        if(sb1 = sb2)
        then
```

```
          bp
        else
          getBarrierPosition(sb1,s,con)
        end
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[ getBarrierPosition_setSignalStatus ]
∀ sb1,sb2 : T.SBID, ss : T.SignalStatus,
        s : SBStates,
        con : S.Configuration •
    getBarrierPosition(sb1,setSignalStatus(sb2,ss,s,con),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[ getBarrierPosition_setSensorStatus ]
∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
        s : SBStates,
        con : S.Configuration •
    getBarrierPosition(sb1,setSensorStatus(sb2,ss,s),con) ≡
        getBarrierPosition(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        sensorStateExists(sb2,s),

/* getSignalStatus_gen */

[ getSignalStatus_setPointPosition ]
∀ sb1, sb2 : T.SBID, pp : T.PointPosition,
        s : SBStates,
        con : S.Configuration •
    getSignalStatus(sb1,setPointPosition(sb2,pp,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        crossingStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[ getSignalStatus_setPointTicks ]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : SBStates,
        con : S.Configuration •
    getSignalStatus(sb1,setPointTicks(sb2,ticks,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.POINTSB ∧
        crossingStateExists(sb1,s,con) ∧
        pointStateExists(sb2,s,con),

[ getSignalStatus_setBarrierPosition ]
∀ s : SBStates, sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        con : S.Configuration •
```

```
    getSignalStatus(sb1,setBarrierPosition(sb2,bp,s,con),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[ getSignalStatus_setSignalStatus ]
∀ s : SBStates, sb1,sb2 : T.SBID, ss : T.SignalStatus,
    con : S.Configuration •
    getSignalStatus(sb1,setSignalStatus(sb2,ss,s,con),con) ≡
        if(sb1 = sb2)
        then
            ss
        else
            getSignalStatus(sb1,s,con)
        end
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        S.getSBType(sb2,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        crossingStateExists(sb2,s,con),

[ getSignalStatus_setSensorStatus ]
∀ sb1,sb2 : T.SBID, ss : T.SensorStatus,
        s : SBStates,
        con : S.Configuration •
    getSignalStatus(sb1,setSensorStatus(sb2,ss,s),con) ≡
        getSignalStatus(sb1,s,con)
pre S.getSBType(sb1,con) = T.CROSSINGSB ∧
        crossingStateExists(sb1,s,con) ∧
        sensorStateExists(sb2,s),

/* getSensorStatus_gen */

[ getSensorStatus_setPointPosition ]
∀ sb1, sb2 : T.SBID, pp : T.PointPosition,
        s : SBStates,
        con : S.Configuration •
    getSensorStatus(sb1,setPointPosition(sb2,pp,s,con)) ≡
        getSensorStatus(sb1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        sensorStateExists(sb1,s) ∧
        pointStateExists(sb2,s,con),

[ getSensorStatus_setPointTicks ]
∀ sb1,sb2 : T.SBID, ticks : T.Tick,
        s : SBStates,
        con : S.Configuration •
    getSensorStatus(sb1,setPointTicks(sb2,ticks,s,con)) ≡
        getSensorStatus(sb1,s)
pre S.getSBType(sb2,con) = T.POINTSB ∧
        sensorStateExists(sb1,s) ∧
        pointStateExists(sb2,s,con),

[ getSensorStatus_setBarrierPosition ]
∀ s : SBStates, sb1,sb2 : T.SBID, bp : T.BarrierPosition,
        con : S.Configuration •
```

```
        getSensorStatus(sb1,setBarrierPosition(sb2,bp,s,con)) ≡
            getSensorStatus(sb1,s)
    pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
            sensorStateExists(sb1,s) ∧
            crossingStateExists(sb2,s,con),

    [ getSensorStatus_setSignalStatus ]
    ∀ s : SBStates, sb1,sb2 : T.SBID, ss : T.SignalStatus,
        con : S.Configuration •
        getSensorStatus(sb1,setSignalStatus(sb2,ss,s,con)) ≡
            getSensorStatus(sb1,s)
    pre S.getSBType(sb2,con) = T.CROSSINGSB ∧
            sensorStateExists(sb1,s) ∧
            crossingStateExists(sb2,s,con),

    [ getSensorStatus_setSensorStatus ]
    ∀ s : SBStates, sb1,sb2 : T.SBID, ss : T.SensorStatus,
            con : S.Configuration •
        getSensorStatus(sb1,setSensorStatus(sb2,ss,s)) ≡
            if(sb1 = sb2)
            then
                ss
            else
                getSensorStatus(sb1,s)
            end
    pre sensorStateExists(sb1,s) ∧
            sensorStateExists(sb2,s)

end
```

F.2.4 Control

```
context: AA_Dynamics1, AA_ComService1, AA_SBCC1, AA_TCC1
scheme AA_Control1(T : AA_Types1, S : AA_Statics1(T),
                                D : AA_Dynamics1(T,S)) =
  class
    object
      COM : AA_ComService1(T),
      SBCC : AA_SBCC1(T,S,D,COM),
      TCC : AA_TCC1(T,S,D,COM)

    type
      ControlState = SBCCStates × TCCStates,

      SBCCStates = T.SBID →m SBCC.SBCCState,
      TCCStates = T.SBID →m TCC.TCCState

    value
      initControlState : ControlState,

      updateSBCCState : T.SBID × SBCC.SBCCState ×
                                    ControlState ~→ ControlState
      updateSBCCState(sb,sbcc,(sbccs,tccs)) ≡
        (sbccs † [ sb ↦ sbcc ],tccs)
      pre sbccStateExists(sb,(sbccs,tccs)),
```

```
getSBCCState : T.SBID × ControlState ∼→ SBCC.SBCCState
getSBCCState(sb,(sbccs,tccs)) ≡
    sbccs(sb)
pre sbccStateExists(sb,(sbccs,tccs)),

updateTCCState : T.TrainID × TCC.TCCState ×
                                    ControlState ∼→ ControlState
updateTCCState(t,tcc,(sbccs,tccs)) ≡
    (sbccs,tccs † [ t ↦ tcc ])
pre tccStateExists(t,(sbccs,tccs)),

getTCCState : T.TrainID × ControlState ∼→ TCC.TCCState
getTCCState(t,(sbccs,tccs)) ≡
    tccs(t)
pre tccStateExists(t,(sbccs,tccs)),

sbccStateExists : T.SBID × ControlState → Bool
sbccStateExists(sb,(sbccs,tccs)) ≡
    sb ∈ dom sbccs,

tccStateExists : T.TrainID × ControlState → Bool
tccStateExists(t,(sbccs,tccs)) ≡
    t ∈ dom tccs,

/∗ Processes ∗/
tick : T.Tick × ControlState × D.State ×
                              S.Configuration ∼→ out COM.comChannel
                                        (ControlState × D.State)
tick(tick,cs,ds,con) ≡
(
    let
        tSet = {t | t : T.TrainID},
        sbSet = {sb | sb : T.SBID},
        (cs,ds) = tickTCCs(tSet,tick,cs,ds,con),
        cs = tickSBCCs(sbSet,tick,cs,ds,con)
    in
        (cs,ds)
    end
),

tickSBCCs : T.SBID-set × T.Tick × ControlState ×
                                D.State × S.Configuration ∼→
                                out COM.comChannel ControlState
tickSBCCs(sbSet,tick,cs,ds,con) ≡
    if (sbSet = {}) then
        cs
    else
        let
            sbcc : T.SBID • sbcc ∈ sbSet,
            sbSet = sbSet \ {sbcc},
            sbccState = getSBCCState(sbcc,cs),
            sbccState = SBCC.sbccProcess(sbcc,tick,sbccState,ds,con),
            cs = updateSBCCState(sbcc,sbccState,cs)
        in
            tickSBCCs(sbSet,tick,cs,ds,con)
```

```
        end
    end,

tickTCCs : T.TrainID-set × T.Tick × ControlState ×
                    D.State × S.Configuration ~→
                    out COM.comChannel (ControlState × D.State)
tickTCCs(tccSet,tick,cs,ds,con) ≡
    if (tccSet = {}) then
        (cs,ds)
    else
        let
            tcc : T.TrainID • tcc ∈ tccSet,
            tccSet = tccSet \ {tcc},
            tccState = getTCCState(tcc,cs),
            tccState = TCC.tccProcess(tcc,tick,tccState,ds,con),
            cs = updateTCCState(tcc,tccState,cs)
        in
            tickTCCs(tccSet,tick,cs,ds,con)
        end
    end,

/* Communication */
comService : ControlState ~→ in COM.comChannel ControlState
comService(cs) ≡
    let
        comMsg = COM.getMsg()
    in
        case T.getReceiver(comMsg) of
            T.isSB(sb) →
            (
                let
                    sbcc = getSBCCState(sb,cs),
                    sbcc = SBCC.msgReceiver(comMsg, sbcc)
                in
                    updateSBCCState(sb,sbcc,cs)
                end
            ),
            T.isTrain(t) →
            (
                let
                    tcc = getTCCState(t,cs),
                    tcc = TCC.tccMsgReceiver(comMsg,tcc)
                in
                    updateTCCState(t,tcc,cs)
                end
            )
        end
    end,

/* Invariants */
is_wf : ControlState × D.State × S.Configuration → Bool
is_wf(cs,ds,con) ≡
    D.is_wf(ds,con) ∧
    tcc_has_state(cs) ∧
    sbcc_has_state(cs),

tcc_has_state : ControlState → Bool
```

```
tcc_has_state(cs) ≡
(
    ∀ t : T.TrainID •
        tccStateExists(t,cs)
),

sbcc_has_state : ControlState → Bool
sbcc_has_state(cs) ≡
(
    ∀ sb : T.SBID •
        sbccStateExists(sb,cs)
),

/**
* Defines that the control system and all its
* components must be consistent e.g. the information
* stored in the control system must reflect the
* physical world and unintended states must not occur.
* Also the physical world must abide by the rules
* of the control system.
**/
consistent : ControlState × D.State × S.Configuration → Bool
consistent(cs,ds,con) ≡
    is_wf(cs,ds,con) ∧
    train_on_branch_dir(ds,con) ∧
    tcc_hasRes_passedResPoint(cs,ds,con) ∧
    sbcc_res_wf(cs,con) ∧
    position_branch_sbcc_res_wf(cs,ds,con) ∧
    tcc_res_branch_wf(cs,ds,con) ∧
    position_sl_sbcc_res_wf(cs,ds,con),


/* When a train is on a branch segment it is consitent
     with the driving direction of the train */
train_on_branch_dir : D.State × S.Configuration → Bool
train_on_branch_dir(ds,con) ≡
(
    ∀ t : T.TrainID, seg : T.SegmentID •
        D.trainOnBranch(t,con,ds) ∧
        D.trainOnSegment(t,seg,con,ds) ∧
        S.segIsBranch(seg,con) ⇒
            S.branchDir(seg,con) = D.getTrainDirection(t,ds)
),

/* If a train has a reservation then it
     has passed the reservation point
     on the given segment */
tcc_hasRes_passedResPoint : ControlState × D.State ×
                                        S.Configuration → Bool
tcc_hasRes_passedResPoint(cs,ds,con) ≡
(
    ∀ t : T.TrainID,
            tccState : TCC.TCCState •
                tccState = getTCCState(t,cs) ∧
                TCC.hasTCCRes(tccState) ⇒
                    TCC.hasPassedResPoint(t,ds,con)
),
```

```
/* Only POINTSB and ENDSB may have line reservations
     Only POINTSB may have branch reservations */
sbcc_res_wf : ControlState × S.Configuration → Bool
sbcc_res_wf(cs,con) ≡
(
    ∀ sb : T.SBID,
            sbcc : SBCC.SBCCState,
            lineRes, branchRes : T.HasRes •
        (
            S.getSBType(sb,con) ∈ {T.PLAINSB, T.CROSSINGSB} ∧
            sbcc = getSBCCState(sb,cs) ∧
            lineRes = SBCC.getLineRes(sbcc) ∧
            branchRes = SBCC.getBranchRes(sbcc)
                ⇒
                    {lineRes} ∪ {branchRes} = {T.noRes}
        )
            ∧
        (
            S.getSBType(sb,con) = T.ENDSB ∧
            sbcc = getSBCCState(sb,cs)
                ⇒
                    SBCC.getBranchRes(sbcc) = T.noRes
        )
),

/* When a train is on a branch segment it must
     have a branch reservation in the SB behind */
position_branch_sbcc_res_wf : ControlState × D.State ×
                                        S.Configuration → Bool
position_branch_sbcc_res_wf(cs,ds,con) ≡
(
    ∀ t : T.TrainID,
            sb : T.SBID,
            tDir : T.Direction,
            seg : T.SegmentID,
            sbcc : SBCC.SBCCState •
                tDir = D.getTrainDirection(t,ds) ∧
                D.trainOnSegment(t,seg,con,ds) ∧
                D.trainOnBranch(t,con,ds) ∧
                S.segIsBranch(seg,con) ∧
                sb = S.getSegSB(seg,T.inverseDir(tDir),con) ∧
                sbcc = getSBCCState(sb,cs)
                    ⇒
                        SBCC.getBranchRes(sbcc) = T.res(T.mk_res(t,tDir))
),

/* If a train is (only) on a branch and has reservation
     then the SB in front of it and the other guard has
     a reservation for that train in that direction */
tcc_res_branch_wf : ControlState × D.State ×
                                        S.Configuration  → Bool
tcc_res_branch_wf(cs,ds,con) ≡
(
    ∀ t : T.TrainID,
            seg : T.SegmentID,
            trainDir : T.Direction,
```

```
            guard1,guard2 : T.SBID,
            sbcc1,sbcc2 : SBCC.SBCCState,
            res : T.Reservation •
               D.trainOnSegment(t,seg,con,ds) ∧
               D.trainOnlyOnBranch(t,con,ds) ∧
               TCC.hasTCCRes(getTCCState(t,cs)) ∧
               trainDir = D.getTrainDirection(t,ds) ∧
               guard1 = S.getSegSB(seg,T.inverseDir(trainDir),con) ∧
               guard2 = S.getSingleLineGuard(guard1,trainDir,con) ∧
               sbcc1 = getSBCCState(guard1,cs) ∧
               sbcc2 = getSBCCState(guard2,cs) ∧
               res = T.mk_res(t,trainDir)
                  ⇒
                     T.res(res) = SBCC.getLineRes(sbcc1) ∧
                     T.res(res) = SBCC.getLineRes(sbcc2) ∧
                     T.res(res) = SBCC.getBranchRes(sbcc2)
),

/∗ When a train is on a single line it must
   have a reservation in both guards with the
   appropriate direction + a branch
   reservation if driving to a point ∗/
position_sl_sbcc_res_wf : ControlState × D.State ×
                                        S.Configuration → Bool
position_sl_sbcc_res_wf(cs,ds,con) ≡
(
   ∀ t : T.TrainID,
            seg : T.SegmentID,
            sb1,sb2 : T.SBID,
            sbcc1,sbcc2 : SBCC.SBCCState,
            dir : T.Direction,
            res : T.Reservation •
               dir = D.getTrainDirection(t,ds) ∧
               D.trainOnSegment(t,seg,con,ds) ∧
               S.segIsLineSegment(seg,con) ∧
               sb1 = S.getSingleLineGuard(seg,
                                    T.inverseDir(dir),con) ∧
               sb2 = S.getSingleLineGuard(seg,dir,con) ∧
               sbcc1 = getSBCCState(sb1,cs) ∧
               sbcc2 = getSBCCState(sb2,cs) ∧
               res = T.mk_res(t,dir) ⇒
                     T.res(res) = SBCC.getLineRes(sbcc1) ∧
                     T.res(res) = SBCC.getLineRes(sbcc2) ∧
                     (S.getSBType(sb2,con) = T.POINTSB ⇒
                           T.res(res) = SBCC.getBranchRes(sbcc2))
),

initReq : ControlState × D.State × S.Configuration → Bool
initReq(cs,ds,con) ≡
   is_wf(cs,ds,con) ∧
   is_tcc_init(cs) ∧
   is_sbcc_init(cs) ∧
   all_tcc_initReq(cs) ∧
   all_sbcc_initReq(cs),

/∗ Requires that the init constants in TCC
   scheme is used for initialization ∗/
```

```
is_tcc_init : ControlState → Bool
is_tcc_init((sbccs,tccs)) ≡
(
    ∀ tcc : TCC.TCCState •
        tcc ∈ rng(tccs) ⇒ tcc = TCC.initTCCState
),

/∗ Requires that the init constants in SBCC
     scheme is used for initialization ∗/
is_sbcc_init : ControlState → Bool
is_sbcc_init((sbccs,tccs)) ≡
(
    ∀ sbcc : SBCC.SBCCState •
        sbcc ∈ rng(sbccs) ⇒ sbcc = SBCC.initSBCCState
),

all_tcc_initReq : ControlState → Bool
all_tcc_initReq(cs) ≡
(
    ∀ t : T.TrainID, tcc : TCC.TCCState •
        tcc = getTCCState(t,cs) ⇒
            TCC.initReq(tcc)
),

all_sbcc_initReq : ControlState → Bool
all_sbcc_initReq(cs) ≡
(
    ∀ sb : T.SBID, sbcc : SBCC.SBCCState •
        sbcc = getSBCCState(sb,cs) ⇒
            SBCC.initReq(sbcc)
)

axiom
/∗ The initial state has to be wellformed and fullfill the
     initial state requirements ∗/
[initial_state]
    initReq(initControlState,D.initState,S.conf),

/∗ SBCC gen wf ∗/
[gen_wf_setSBCCLineRes]
    ∀ con : S.Configuration,
        ds : D.State,
        res : T.HasRes,
        sb : T.SBID,
        sbcc : SBCC.SBCCState,
        cs : ControlState •
            is_wf(cs,ds,con) ∧
            S.getSBType(sb,con) ∈ {T.ENDSB, T.POINTSB} ∧
            sbcc = getSBCCState(sb,cs)
                ⇒
            is_wf(updateSBCCState(sb,
                                SBCC.setLineRes(res,sbcc),cs),ds,con),

[gen_wf_setSBCCBranchRes]
    ∀ con : S.Configuration,
        ds : D.State,
        sb : T.SBID,
```

```
      sbcc : SBCC.SBCCState,
      res : T.HasRes, cs : ControlState •
         is_wf(cs,ds,con) ∧
         S.getSBType(sb,con) = T.POINTSB ∧
         sbcc = getSBCCState(sb,cs)
            ⇒
         is_wf(updateSBCCState(sb,
                  SBCC.setBranchRes(res,sbcc),cs),ds,con),

[ gen_wf_removeSBCCLineRes ]
   ∀ con : S.Configuration,
         sb : T.SBID,
         sbcc : SBCC.SBCCState,
         ds : D.State, cs : ControlState •
            is_wf(cs,ds,con) ∧
            S.getSBType(sb,con) ∈ {T.ENDSB, T.POINTSB} ∧
            sbcc = getSBCCState(sb,cs)
               ⇒
            is_wf(updateSBCCState(sb,
                        SBCC.removeLineRes(sbcc),cs),ds,con),

[ gen_wf_removeSBCCBranchRes ]
   ∀ con : S.Configuration,
         sb : T.SBID,
         sbcc : SBCC.SBCCState,
         ds : D.State, cs : ControlState •
            is_wf(cs,ds,con) ∧
            S.getSBType(sb,con) = T.POINTSB ∧
            sbcc = getSBCCState(sb,cs)
               ⇒
            is_wf(updateSBCCState(sb,
                     SBCC.removeBranchRes(sbcc),cs),ds,con),

[ gen_wf_setLastSensorStatus ]
   ∀ con : S.Configuration,
         sb : T.SBID,
         sbcc : SBCC.SBCCState,
         ds : D.State, cs : ControlState,
         ss : T.SensorStatus •
            is_wf(cs,ds,con) ∧
            sbcc = getSBCCState(sb,cs)
               ⇒
            is_wf(updateSBCCState(sb,
                  SBCC.setLastSensorStatus(ss,sbcc),cs),ds,con),

[ gen_wf_storeSBCCMsg ]
   ∀ con : S.Configuration,
         sb : T.SBID,
         sbcc : SBCC.SBCCState,
         ds : D.State, cs : ControlState,
         cm : T.ComMsg •
            is_wf(cs,ds,con) ∧
            sbcc = getSBCCState(sb,cs)
               ⇒
            is_wf(updateSBCCState(sb,
                     SBCC.storeMsg(cm,sbcc),cs),ds,con),
```

```
[ gen_wf_setSBCCPrepRes ]
    ∀ con : S.Configuration,
            sb : T.SBID,
            sbcc : SBCC.SBCCState,
            ds : D.State, cs : ControlState,
            hr : T.HasRes •
                is_wf(cs,ds,con) ∧
                sbcc = getSBCCState(sb,cs)
                    ⇒
                is_wf(updateSBCCState(sb,
                                SBCC.setPrepRes(hr,sbcc),cs),ds,con),

/∗ TCC gen wf ∗/
[ gen_wf_setTCCRes ]
    ∀ con : S.Configuration,
            cs : ControlState,
            t : T.TrainID,
            ds : D.State,
            tcc : TCC.TCCState,
            b : Bool •
                is_wf(cs,ds,con) ∧
                tcc = getTCCState(t,cs)
                    ⇒
                is_wf(updateTCCState(t,
                                TCC.setTCCRes(b,tcc),cs),ds,con),

[ gen_wf_setTCCRequesting ]
    ∀ con : S.Configuration,
            ds : D.State,
            tcc : TCC.TCCState,
            cs : ControlState,
            t : T.TrainID,
            b : Bool •
                is_wf(cs,ds,con) ∧
                tcc = getTCCState(t,cs)
                    ⇒
                is_wf(updateTCCState(t,
                                TCC.setTCCRequesting(b,tcc),cs),ds,con),

[ gen_wf_setTrainDecelerating ]
    ∀ con : S.Configuration,
            ds : D.State,
            cs : ControlState,
            tcc : TCC.TCCState,
            t : T.TrainID,
            b : Bool •
                is_wf(cs,ds,con) ∧
                tcc = getTCCState(t,cs)
                    ⇒
                is_wf(updateTCCState(t,
                        TCC.setTrainDecelerating(b,tcc),cs),ds,con)

end
```

TCC

```
context: AA_Types1, AA_Statics1, AA_Dynamics1, AA_ComService1
scheme AA_TCC1(T : AA_Types1, S : AA_Statics1(T),
                    D : AA_Dynamics1(T,S), COM : AA_ComService1(T)) =
  class
    type
      TCCState

    value
      initTCCState : TCCState,

      hasTCCRes : TCCState → Bool,
      setTCCRes : Bool × TCCState → TCCState,

      isTCCRequesting : TCCState → Bool,
      setTCCRequesting : Bool × TCCState → TCCState,

      isTrainDecelerating : TCCState → Bool,
      setTrainDecelerating : Bool × TCCState → TCCState,

      hasPassedResPoint : T.TrainID × D.State ×
                                        S.Configuration → Bool,
      hasPassedBrakePoint : T.TrainID × D.State ×
                                        S.Configuration → Bool,

      /* Processes */
      tccMsgReceiver : T.ComMsg × TCCState → TCCState
      tccMsgReceiver(comMsg,tcc) ≡
        let
          resp = T.getMsg(comMsg)
        in
          case resp of
            T.segResp(resGranted) →
            (
              let
                tcc = setTCCRequesting(false,tcc)
              in
                if(resGranted)
                then
                  setTCCRes(true,tcc)
                else
                  tcc
                end
              end
            ),
            _ → tcc
          end
        end,

      tccProcess : T.TrainID × T.Tick × TCCState × D.State ×
                          S.Configuration ~→ out COM.comChannel
                                        TCCState × D.State
      tccProcess(t,tick,cs,ds,con) ≡
        let
          (cs,ds) = checkSpeed(t,tick,cs,ds,con),
          cs = clearRes(t,cs,ds),
```

```
        (cs,ds) = handleRes(t,cs,ds,con)
    in
        (cs,ds)
    end,

checkSpeed : T.TrainID × T.Tick × TCCState × D.State ×
                              S.Configuration ~→ TCCState × D.State
checkSpeed(t,tick,cs,ds,con) ≡
    let
        dts = S.getTrainMaxAcc(t,con) ∗ tick,
        ts = D.getTrainSpeed(t,ds) + dts,
        trainMaxSpeed = S.getTrainMaxSpeed(t,con)
    in
        /∗ If train is entirely in an ESA ∗/
        if(D.trainInESA(t,ds))
        then
            if(ts > trainMaxSpeed)
            then
                decelerateTrain(t,cs,ds,con)
            else
                checkDeceleration(t,cs,ds,con)
            end
        else /∗ Train on segment and perhaps an ESA ∗/
            let
                tp = D.getTrainPosition(t,ds),

                /∗ Will always be an IsSeg ∗/
                isSeg = D.getTrainLoc(t,ds),

                frontSeg = T.getSeg(isSeg),
                segMaxSpeed = S.getSegMaxSpeed(frontSeg,con)
            in
                if (ts > segMaxSpeed ∨ ts > trainMaxSpeed)
                then
                    decelerateTrain(t,cs,ds,con)
                else
                    checkDeceleration(t,cs,ds,con)
                end
            end
        end
    end, /∗ let ∗/

checkDeceleration : T.TrainID × TCCState × D.State ×
                                        S.Configuration ~→
                                          TCCState × D.State
checkDeceleration(t,cs,ds,con) ≡
    if (isTrainDecelerating(cs))
    then
        let
            ds = D.setTrainAcc(t,0.0,ds,con),
            cs = setTrainDecelerating(false,cs)
        in
            (cs,ds)
        end
    else
        (cs,ds)
    end,
```

```
decelerateTrain : T.TrainID × TCCState × D.State ×
                                        S.Configuration →
                                          TCCState × D.State
decelerateTrain(t,tcc,ds,con) ≡
    let
        tcc = setTrainDecelerating(true,tcc),
        ds = D.decelerateTrain(t,con,ds)
    in
        (tcc,ds)
    end,

accelerateTrain : T.TrainID × TCCState × D.State ×
                                        S.Configuration →
                                          TCCState × D.State
accelerateTrain(t,tcc,ds,con) ≡
    let
        tcc = setTrainDecelerating(false,tcc),
        ds = D.accelerateTrain(t,con,ds)
    in
        (tcc,ds)
    end,

clearRes : T.TrainID × TCCState × D.State ~→ TCCState
clearRes(t,cs,ds) ≡
    if(∼T.oneLoc(D.getTrainPosition(t,ds)))
    then
        setTCCRes(false,cs)
    else
        cs
    end,

handleRes : T.TrainID × TCCState × D.State ×
                            S.Configuration → out COM.comChannel
                            TCCState × D.State
handleRes(t,cs,ds,con) ≡
    if(hasPassedResPoint(t,ds,con))
    then
        if(∼hasTCCRes(cs))
        then
            (cs,ds)
        else
            let
                (cs,ds) = if(hasPassedBrakePoint(t,ds,con))
                            then decelerateTrain(t,cs,ds,con)
                            else (cs,ds) end
            in
                if(∼isTCCRequesting(cs))
                then
                    tccRequestRes(t,cs,ds,con)
                else
                    (cs,ds)
                end
            end
        end
    else
        (cs,ds)
```

```
    end,

tccRequestRes : T.TrainID × TCCState × D.State ×
                                S.Configuration ~→ out COM.comChannel
                                TCCState × D.State
tccRequestRes(t,cs,ds,con) ≡
    let
        loc = T.frontLoc(D.getTrainPosition(t,ds)),
        dir = D.getTrainDirection(t,ds)
    in
        case loc of
            T.isESA(esa) →
            (
                /∗ If not facing line, then do nothing.
                     Will brake at brakepoint ∗/
                if (dir = T.end2Dir(esa))
                then
                    (cs,ds)
                else
                    (sendTCCReq(t,S.getESASB(esa,con),dir,cs),ds)
                end
            ),

            T.isSeg(seg) →
            (
                 (sendTCCReq(t,S.getSegSB(seg,dir,con),dir,cs),ds)
            )
        end /∗ case ∗/
    end, /∗ let ∗/

sendTCCReq : T.TrainID × T.SBID × T.Direction ×
                                TCCState ~→ out COM.comChannel TCCState
sendTCCReq(t,sb,dir,cs) ≡
    let
        sender = T.isTrain(t),
        receiver = T.isSB(sb),
        res = T.mk_res(t,dir),
        msg = T.segReq(res),
        comMsg = T.mk_comMsg(sender,receiver,msg),
        cs = setTCCRequesting(true,cs)
    in
        COM.sendMsg(comMsg); cs
    end,

/∗ Invariants ∗/
initReq : TCCState → Bool
initReq(tcc) ≡
    no_tcc_res(tcc) ∧
    tcc_not_requesting(tcc) ∧
    tcc_not_decelerating(tcc),

/∗ tcc may not have a reservation ∗/
no_tcc_res : TCCState → Bool
no_tcc_res(tcc) ≡
(
    ∼hasTCCRes(tcc)
),
```

```
/* No TCC is requesting segment access */
tcc_not_requesting : TCCState → Bool
tcc_not_requesting(tcc) ≡
(
   ~isTCCRequesting(tcc)
),

/* No TCC is requesting segment access */
tcc_not_decelerating : TCCState → Bool
tcc_not_decelerating(tcc) ≡
(
   ~isTrainDecelerating(tcc)
)

axiom

/* obs_gen */

/* hasTCCRes_gen */

[ hasTCCRes_setTCCRes ]
   ∀ b : Bool, tcc : TCCState •
      hasTCCRes(setTCCRes(b,tcc)) ≡
         b,

[ hasTCCRes_setTCCRequesting ]
   ∀ b : Bool, tcc : TCCState •
      hasTCCRes(setTCCRequesting(b,tcc)) ≡
         hasTCCRes(tcc),

[ hasTCCRes_setTrainDecelerating ]
   ∀ b : Bool, tcc : TCCState •
      hasTCCRes(setTrainDecelerating(b,tcc)) ≡
         hasTCCRes(tcc),

/* isTCCRequesting_gen */

[ isTCCRequesting_setTCCRes ]
   ∀ b : Bool, tcc : TCCState •
      isTCCRequesting(setTCCRes(b,tcc)) ≡
         isTCCRequesting(tcc),

[ isTCCRequesting_setTCCRequesting ]
   ∀ b : Bool, tcc : TCCState •
      isTCCRequesting(setTCCRequesting(b,tcc)) ≡
         b,

[ isTCCRequesting_setTrainDecelerating ]
   ∀ b : Bool, tcc : TCCState •
      isTCCRequesting(setTrainDecelerating(b,tcc)) ≡
         isTCCRequesting(tcc),

/* isTrainDecelerating_gen */

[ isTrainDecelerating_setTCCRes ]
   ∀ b : Bool, tcc : TCCState •
```

```
                isTrainDecelerating(setTCCRes(b,tcc)) ≡
                    isTrainDecelerating(tcc),

        [ isTrainDecelerating_setTCCRequesting ]
            ∀ b : Bool, tcc : TCCState •
                isTrainDecelerating(setTCCRequesting(b,tcc)) ≡
                    isTrainDecelerating(tcc),

        [ isTrainDecelerating_setTrainDecelerating ]
            ∀ b : Bool, tcc : TCCState •
                isTrainDecelerating(setTrainDecelerating(b,tcc)) ≡
                    b

    end
```

SBCC

```
context: AA_Types1, AA_Statics1, AA_Dynamics1, AA_ComService1
scheme AA_SBCC1(T : AA_Types1, S : AA_Statics1(T),
                D : AA_Dynamics1(T,S), COM : AA_ComService1(T)) =
    class
        type
            SBCCState

        value
            initSBCCState : SBCCState,

            getLineRes : SBCCState → T.HasRes,
            getBranchRes : SBCCState → T.HasRes,

            setLineRes : T.HasRes × SBCCState → SBCCState,
            setBranchRes : T.HasRes × SBCCState → SBCCState,

            getLastSensorStatus : SBCCState → T.SensorStatus,
            setLastSensorStatus : T.SensorStatus × SBCCState → SBCCState,

            getNextMsg : SBCCState → T.HasComMsg × SBCCState,
            storeMsg : T.ComMsg × SBCCState → SBCCState,

            setPrepRes : T.HasRes × SBCCState → SBCCState,
            getPrepRes : SBCCState → T.HasRes,

            removeLineRes : SBCCState → SBCCState
            removeLineRes(sbcc) ≡
                setLineRes(T.noRes,sbcc),

            removeBranchRes : SBCCState → SBCCState
            removeBranchRes(sbcc) ≡
                setBranchRes(T.noRes,sbcc),

            removePrepRes : SBCCState → SBCCState
            removePrepRes(cs) ≡
                setPrepRes(T.noRes,cs),

            isPreparing : SBCCState → Bool
```

```
isPreparing(cs) ≡
    case getPrepRes(cs) of
        T.noRes → false,
        _ → true
    end,

msgReceiver : T.ComMsg × SBCCState → SBCCState
msgReceiver(comMsg,sbcc) ≡
    storeMsg(comMsg,sbcc),

/∗ Processes ∗/
sbccProcess : T.SBID × T.Tick × SBCCState × D.State ×
                                        S.Configuration →
                                        out COM.comChannel SBCCState
sbccProcess(sb,tick,cs,ds,con) ≡
    let
        cs = sensorProcess(sb,cs,ds,con)
    in
        if(isPreparing(cs))
        then
            prepareProcess(sb,cs,ds,con)
        else
            sbccMsgProcess(sb,cs,ds,con)
        end
    end,

/∗ Waits for the preparation of a segment
    before a train is allowed to enter ∗/
prepareProcess : T.SBID × SBCCState × D.State ×
                                        S.Configuration →
                                        out COM.comChannel SBCCState
prepareProcess(sb,cs,ds,con) ≡
    case S.getSBType(sb,con) of
        /∗ case POINTSB → wait for !moving ∗/
        T.POINTSB →
        (
            if(D.getPointPosition(sb,ds,con) ∈ {T.UP, T.DOWN})
            then
                let
                    T.res(res) = getPrepRes(cs),
                    train = T.getTrain(res),
                    cs = removePrepRes(cs)
                in
                    sendSBCCMsg(sb,T.isTrain(train),T.segResp(true));cs
                end
            else
                cs
            end
        ),

        /∗ case crossingsb → wait for DOWN ∗/
        T.CROSSINGSB →
        (
            if(D.getBarrierPosition(sb,ds,con) = T.DOWN)
            then
                let
                    T.res(res) = getPrepRes(cs),
```

```
                train = T.getTrain(res),
                cs = removePrepRes(cs)
            in
                sendSBCCMsg(sb,T.isTrain(train),T.segResp(true));cs
            end
        else
            cs
        end
    ),

    _ → cs
end,

sensorProcess : T.SBID × SBCCState × D.State ×
                                    S.Configuration →
                                    out COM.comChannel SBCCState
sensorProcess(sb,sbcc,ds,con) ≡
    let
        sState = D.getSensorStatus(sb,ds),
        lastState = getLastSensorStatus(sbcc),
        sbcc = setLastSensorStatus(sState,sbcc)
    in
        if((lastState = T.ACTIVE) ∧ (sState = T.INACTIVE))
        then
            let
                ds = dePrepareSeg(sb,sbcc,ds,con)
            in
                if(S.isLineGuard(sb,con))
                then
                    makeDeRes(sb,sbcc,ds,con)
                else
                    sbcc
                end
            end
        else
            sbcc
        end
    end,

dePrepareSeg : T.SBID × SBCCState × D.State ×
                                        S.Configuration →
                                        SBCCState × D.State
dePrepareSeg(sb,cs,ds,con) ≡
    let
        cs = removePrepRes(cs)
    in
        case S.getSBType(sb,con) of
            T.CROSSINGSB →
            (
                (cs,D.setBarrierPosition(sb,T.MOVINGUP,ds,con))
            ),

            _ → (cs,ds)
        end
    end,

prepareSeg : T.SBID × T.Reservation × SBCCState ×
```

```
                                         D.State × S.Configuration →
                                         SBCCState × D.State
prepareSeg(sb,res,cs,ds,con) ≡
  let
    cs = setPrepRes(T.res(res),cs)
  in
    case S.getSBType(sb,con) of
      T.CROSSINGSB →
      (
        let
          ds = D.setSignalStatus(sb,T.ON,ds,con)
        in
          (cs,ds)
        end
      ),

      T.POINTSB →
      (
        case T.getDir(res) of
          T.UP →
          (
            let
              ds = D.setPointPosition(sb,T.MOVINGUP,ds,con)
            in
              (cs,ds)
            end
          ),

          T.DOWN →
          (
            let
              ds = D.setPointPosition(sb,T.MOVINGDOWN,ds,con)
            in
              (cs,ds)
            end
          )
        end
      ),

      _ → (cs,ds)
    end
  end,

makeDeRes : T.SBID × SBCCState × D.State ×
                                          S.Configuration ~→
                                          out COM.comChannel SBCCState
makeDeRes(sb,sbcc,ds,con) ≡
  case S.getSBType(sb,con) of
    T.ENDSB →
    (
      let
        T.res(lineRes) = getLineRes(sbcc),
        endDir = S.getEndDir(sb,con)
      in
        if(T.getDir(lineRes) = endDir)
        then
          let
```

```
                sbcc = removeLineRes(sbcc)
              in
                sendLDeResMsg(sb,S.getOppositeGuard(sb,con));sbcc
              end
            else
              sbcc
            end /* if */
          end /* let */
        ),

        T.POINTSB →
        (
          let
            T.res(lineRes) = getLineRes(sbcc),
            pointDir = S.getPointDir(sb,con)
          in
            if(T.getDir(lineRes) = pointDir)
            then
              let
                sbcc = removeLineRes(sbcc)
              in
                sendLDeResMsg(sb,S.getOppositeGuard(sb,con)); sbcc
              end
            else
              sendBDeResMsg(sb,S.getOppositeGuard(sb,con)); sbcc
            end /* if */
          end /* let */
        )
      end /* case */
  pre S.isLineGuard(sb,con),

  sendLBDeResMsg : T.SBID × T.SBID → out COM.comChannel Unit
  sendLBDeResMsg(thisSB,remoteSB) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineBranchDeRes),

  sendLDeResMsg : T.SBID × T.SBID → out COM.comChannel Unit
  sendLDeResMsg(thisSB,remoteSB) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineDeRes),

  sendBDeResMsg : T.SBID × T.SBID → out COM.comChannel Unit
  sendBDeResMsg(thisSB,remoteSB) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.branchDeRes),

  sendLBResMsg : T.SBID × T.SBID × T.Reservation →
                                    out COM.comChannel Unit
  sendLBResMsg(thisSB,remoteSB,aRes) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineBranchReq(aRes)),

  sendSBCCMsg : T.SBID × T.ComID × T.SBCCMsg →
                                        out COM.comChannel Unit
  sendSBCCMsg(sb,receiver,sbccMsg) ≡
    COM.sendMsg(T.mk_comMsg(T.isSB(sb),receiver,sbccMsg)),

  sbccMsgProcess : T.SBID × SBCCState × D.State ×
                                S.Configuration →
                                out COM.comChannel SBCCState
  sbccMsgProcess(sb,sbcc,ds,con) ≡
```

```
    let
        (hasComMsg,sbcc) = getNextMsg(sbcc)
    in
        case hasComMsg of
            T.comMsg(T.mk_comMsg(sender,receiver,msg)) →
            (
                case sender of
                    T.isTrain(_) →
                    (
                        let
                            (retMsg,sbcc,ds) =
                                    handleTCCMsg(sb,msg,sbcc,ds,con)
                        in
                            case retMsg of
                                T.hasMsg(aMsg) →
                                    sendSBCCMsg(sb,sender,aMsg); sbcc,
                                _ → sbcc
                            end
                        end
                    ),

                    T.isSB(_) →
                    (
                        let
                            (sbcc,retMsg) = handleSBCCMsg(sb,msg,sbcc)
                        in
                            case retMsg of
                                T.hasMsg(aMsg) →
                                    sendSBCCMsg(sb,sender,aMsg); sbcc,
                                _ → sbcc
                            end
                        end
                    )
                end /∗ case ∗/
            ),
            _ → sbcc /∗ no message to process ∗/
        end
    end, /∗ let ∗/

handleSBCCMsg : T.SBID × T.Message × SBCCState →
                                    out COM.comChannel
                                    SBCCState × T.ReturnSBCCMsg
handleSBCCMsg(sb,msg,sbcc) ≡
    case msg of
        /∗ Request ∗/
        T.lineBranchReq(_) → handleLBReq(msg,sbcc),

        /∗ Response ∗/
        T.lineBranchResp(_) → handleLBResp(sb,msg,sbcc),

        /∗ De reservation ∗/
        T.lineBranchDeRes → handleDeResMsg(msg,sbcc),
        T.lineDeRes → handleDeResMsg(msg,sbcc),
        T.branchDeRes → handleDeResMsg(msg,sbcc)
    end,

handleTCCMsg : T.SBID × T.Message × SBCCState ×
```

```
                                    D.State × S.Configuration →
                                    out COM.comChannel T.ReturnSBCCMsg ×
                                    SBCCState × D.State
handleTCCMsg(sb,msg,cs,ds,con) ≡
  let
    T.segReq(res) = msg
  in
    case S.getSBType(sb,con) of
      T.ENDSB →
      (
        /* if direction away from end → send msg
            else OK
        */
        if(T.getDir(res) = S.getEndDir(sb,con))
        then
          let
            (cs,ds) = prepareSeg(sb,res,cs,ds,con)
          in
            (T.noSBCCMsg,cs,ds)
          end
        else
          if(lineFree(cs))
          then
            let
              cs = setLineRes(T.res(res),cs),
              cs = sendLBResMsg(sb,
                          S.getOppositeGuard(sb,con),res)
            in
              (T.noSBCCMsg,cs,ds)
            end
          else
            (T.hasMsg(T.segResp(false)),cs,ds)
          end
        end
      ),

      /* If direction away from point → send msg
          else OK
      */
      T.POINTSB →
      (
        if(T.getDir(res) = S.getPointDir(sb,con))
        then
          let
            (cs,ds) = prepareSeg(sb,res,cs,ds,con)
          in
            (T.noSBCCMsg,cs,ds)
          end
        else
          if(lineFree(cs))
          then
            let
              cs = setLineRes(T.res(res),cs),
              cs = sendLBResMsg(sb,
                          S.getOppositeGuard(sb,con),res)
            in
              (T.noSBCCMsg,cs,ds)
```

```
                end
            else
                (T.hasMsg(T.segResp(false)),cs,ds)
            end
        end
    ),

    _ → /* PLAINSB, CROSSINGSB */
    (
        let
            (cs,ds) = prepareSeg(sb,res,cs,ds,con)
        in
            (T.noSBCCMsg,cs,ds)
        end
    )
  end /* case */
end, /* let */

/* if(!line free) then NO
   reserve line;
   if(end type) then YES
   if(!branch free) then deres line; NO
   res branch; YES;
*/
handleLBReq : T.Message × SBCCState →
                              SBCCState × T.ReturnSBCCMsg
handleLBReq(msg,sbcc) ≡
  let
    T.lineBranchReq(res) = msg
  in
    if(lineBranchFree(sbcc))
    then
      let
        sbcc = setLineRes(T.res(res),sbcc),
        sbcc = setBranchRes(T.res(res),sbcc)
      in
        (sbcc,T.hasMsg(T.lineBranchResp(res,true)))
      end
    else
      (sbcc,T.hasMsg(T.lineBranchResp(res,false)))
    end
  end,

lineBranchFree : SBCCState → Bool
lineBranchFree(sbcc) ≡
  (getLineRes(sbcc) = T.noRes) ∧
  (getBranchRes(sbcc) = T.noRes),

lineFree : SBCCState → Bool
lineFree(sbcc) ≡
  (getLineRes(sbcc) = T.noRes),

/* if(response = NO) deres; NO;
   if(!prepare_segment()) deres; NO;
   OK;
*/
handleLBResp : T.SBID × T.Message × SBCCState →
```

```
                                            out COM.comChannel
                                            SBCCState × T.ReturnSBCCMsg
handleLBResp(sb,msg,sbcc) ≡
    let
        T.lineBranchResp(res,granted) = msg
    in
        if(granted)
        then
            sendSBCCMsg(sb,T.isTrain(T.getTrain(res)),
                                            T.segResp(true));
            (sbcc,T.noSBCCMsg)
        else
            let
                sbcc = removeLineRes(sbcc)
            in
                sendSBCCMsg(sb,T.isTrain(T.getTrain(res)),
                                            T.segResp(false));
                (sbcc,T.noSBCCMsg)
            end
        end
    end,

/∗ case(msg)
            lb → deres line; deres branch
            l → deres line;
            b → deres branch;
∗/
handleDeResMsg : T.Message × SBCCState →
                                            SBCCState × T.ReturnSBCCMsg
handleDeResMsg(msg,sbcc) ≡
    case msg of
        T.lineBranchDeRes →
        (
            let
                sbcc = removeLineRes(sbcc),
                sbcc = removeBranchRes(sbcc)
            in
                (sbcc,T.noSBCCMsg)
            end
        ),

        T.lineDeRes →
        (
            let
                sbcc = removeLineRes(sbcc)
            in
                (sbcc,T.noSBCCMsg)
            end
        ),

        T.branchDeRes →
        (
            let
                sbcc = removeBranchRes(sbcc)
            in
                (sbcc,T.noSBCCMsg)
            end
```

```
        )
    end,

/∗ Invariants ∗/
initReq : SBCCState → Bool
initReq(sbcc) ≡
    no_sbcc_res(sbcc) ∧
    sbcc_not_preparing(sbcc),

no_sbcc_res : SBCCState → Bool
no_sbcc_res(sbcc) ≡
(
    ∀ branchRes,lineRes : T.HasRes •
                branchRes = getBranchRes(sbcc) ∧
                lineRes = getLineRes(sbcc)
                    ⇒
                        {branchRes} ∪ {lineRes} = {T.noRes}
),

/∗ No SBCC is currently preparing a segment ∗/
sbcc_not_preparing : SBCCState → Bool
sbcc_not_preparing(sbcc) ≡
(
    ∼isPreparing(sbcc)
)

axiom

/∗ getSBCCLineRes_gen∗/

[ getSBCCLineRes_setLineRes ]
    ∀ sbcc : SBCCState, sbRes : T.Reservation •
        getLineRes(setLineRes(T.res(sbRes),sbcc)) ≡
            T.res(sbRes),

[ getLineRes_setBranchRes ]
    ∀ sbcc : SBCCState, sbRes : T.Reservation •
        getLineRes(setBranchRes(T.res(sbRes),sbcc)) ≡
            getLineRes(sbcc),

[ getLineRes_setLastSensorStatus ]
    ∀ ss : T.SensorStatus, sbcc : SBCCState •
        getLineRes(setLastSensorStatus(ss,sbcc)) ≡
            getLineRes(sbcc),

[ getLineRes_storeMsg ]
    ∀ cm : T.ComMsg, sbcc : SBCCState •
        getLineRes(storeMsg(cm,sbcc)) ≡
            getLineRes(sbcc),

[ getLineRes_setPrepRes ]
    ∀ hr : T.HasRes, sbcc : SBCCState •
        getLineRes(setPrepRes(hr,sbcc)) ≡
            getLineRes(sbcc),

/∗ getSBCCBranchRes_gen ∗/
```

```
[ getBranchRes_setLineRes ]
    ∀ sbcc : SBCCState, sbRes : T.Reservation •
        getBranchRes(setLineRes(T.res(sbRes),sbcc)) ≡
            getBranchRes(sbcc),

[ getBranchRes_setBranchRes ]
    ∀ sbcc : SBCCState, sbRes : T.Reservation •
        getBranchRes(setBranchRes(T.res(sbRes),sbcc)) ≡
            T.res(sbRes),

[ getBranchRes_setLastSensorStatus ]
    ∀ ss : T.SensorStatus, sbcc : SBCCState •
        getBranchRes(setLastSensorStatus(ss,sbcc)) ≡
            getBranchRes(sbcc),

[ getBranchRes_storeMsg ]
    ∀ cm : T.ComMsg, sbcc : SBCCState •
        getBranchRes(storeMsg(cm,sbcc)) ≡
            getBranchRes(sbcc),

[ getBranchRes_setPrepRes ]
    ∀ hr : T.HasRes, sbcc : SBCCState •
        getBranchRes(setPrepRes(hr,sbcc)) ≡
            getBranchRes(sbcc),

/∗ getLastSensorStatus_gen ∗/

[ getLastSensorStatus_setLineRes ]
    ∀ sbcc : SBCCState, sbRes : T.Reservation •
        getLastSensorStatus(setLineRes(T.res(sbRes),sbcc)) ≡
            getLastSensorStatus(sbcc),

[ getLastSensorStatus_setBranchRes ]
    ∀ sbcc : SBCCState, sbRes : T.Reservation •
        getLastSensorStatus(setBranchRes(T.res(sbRes),sbcc)) ≡
            getLastSensorStatus(sbcc),

[ getLastSensorStatus_setLastSensorStatus ]
    ∀ ss : T.SensorStatus, sbcc : SBCCState •
        getLastSensorStatus(setLastSensorStatus(ss,sbcc)) ≡
            ss,

[ getLastSensorStatus_storeMsg ]
    ∀ cm : T.ComMsg, sbcc : SBCCState •
        getLastSensorStatus(storeMsg(cm,sbcc)) ≡
            getLastSensorStatus(sbcc),

[ getLastSensorStatus_setPrepRes ]
    ∀ hr : T.HasRes, sbcc : SBCCState •
        getLastSensorStatus(setPrepRes(hr,sbcc)) ≡
            getLastSensorStatus(sbcc),

/∗ getNextMsg_gen ∗/

[ getNextMsg_setLineRes ]
    ∀ sbcc : SBCCState, sbRes : T.Reservation •
        getNextMsg(setLineRes(T.res(sbRes),sbcc)) ≡
```

```
                getNextMsg(sbcc),

         [ getNextMsg_setBranchRes ]
            ∀ sbcc : SBCCState, sbRes : T.Reservation •
              getNextMsg(setBranchRes(T.res(sbRes),sbcc)) ≡
                getNextMsg(sbcc),

         [ getNextMsg_setLastSensorStatus ]
            ∀ ss : T.SensorStatus, sbcc : SBCCState •
              getNextMsg(setLastSensorStatus(ss,sbcc)) ≡
                getNextMsg(sbcc),

         [ getNextMsg_setPrepRes ]
            ∀ hr : T.HasRes, sbcc : SBCCState •
              getNextMsg(setPrepRes(hr,sbcc)) ≡
                getNextMsg(sbcc),

         /∗ getPrepRes_gen ∗/

         [ getPrepRes_setLineRes ]
            ∀ sbcc : SBCCState, sbRes : T.Reservation •
              getPrepRes(setLineRes(T.res(sbRes),sbcc)) ≡
                getPrepRes(sbcc),

         [ getPrepRes_setBranchRes ]
            ∀ sbcc : SBCCState, sbRes : T.Reservation •
              getPrepRes(setBranchRes(T.res(sbRes),sbcc)) ≡
                getPrepRes(sbcc),

         [ getPrepRes_setLastSensorStatus ]
            ∀ ss : T.SensorStatus, sbcc : SBCCState •
              getPrepRes(setLastSensorStatus(ss,sbcc)) ≡
                getPrepRes(sbcc),

         [ getPrepRes_storeMsg ]
            ∀ cm : T.ComMsg, sbcc : SBCCState •
              getPrepRes(storeMsg(cm,sbcc)) ≡
                getPrepRes(sbcc),

         [ getPrepRes_setPrepRes ]
            ∀ hr : T.HasRes, sbcc : SBCCState •
              getPrepRes(setPrepRes(hr,sbcc)) ≡
                hr

   end
```

F.3 Concrete model

F.3.1 Types

```
scheme CA_Types0 =
   class
      type
         /∗ Entity ID types ∗/
```

```
ID = Text,
ESAID = End,
SBID = {| sb : ID • sbIDLimit(sb) |},
SegmentID = {| seg : ID • segIDLimit(seg) |},
TrainID = {| t : ID • trainIDLimit(t) |},

/* Location of a train, on an esa or an segment */
Location == isESA(getESA : ESAID) | isSeg(getSeg : SegmentID),

/* The ends (esa ends) */
End == HIGH | LOW,
/* Driving direction on the line */
Direction == UP | DOWN,

/* Physical parameters */
Length = Real,
Speed = Real,
Acceleration = Real,

/* Neighbour of a SB in a direction */
SBSegment ==
          seg(getSeg : SegmentID) |
          point(getUpSeg : SegmentID, getDownSeg : SegmentID) |
          esa(getESA : ESAID),
/* The type of a SB */
SBType == POINTSB | ENDSB | CROSSINGSB | PLAINSB,
/* The segments etc around a point */
PointSegments == pointSegments(getStem : SegmentID,
                               getUpBranch : SegmentID,
                               getDownBranch : SegmentID,
                               getPointDir : Direction),

/* The state of different entities */
PointPosition == UP | DOWN | MOVINGUP | MOVINGDOWN,
BarrierPosition == UP | DOWN | MOVINGUP | MOVINGDOWN,
SignalStatus == ON | OFF,
SensorStatus == ACTIVE | INACTIVE,

HasPointPosition ==
            pointPos(getPos : PointPosition ↔ setPos) | none,
HasBarrierPosition ==
            barrierPos(getPos : BarrierPosition ↔ setPos) | none,
HasSignalStatus ==
      signalStatus(getStatus : SignalStatus ↔ setStatus) | none,

/* Location/position of a train */
TrainPosition :: frontPos : SegmentPosition ↔ setFrontPos
                 rearPos : SegmentPosition ↔ setRearPos,

/* Location / position of a train end */
SegmentPosition :: getLoc : Location
                   getLength : Length,

Tick = Real,
HasTicks == ticks(getTicks : Tick) | none,

/* From Control */
```

```
HasRes == res(Reservation) | noRes,
HasSeg == isSeg(SegmentID) | noSeg,

Message = TCCMsg | SBCCMsg,
TCCMsg == segReq(Reservation),
SBCCMsg = SBCCResMsg | SBCCDeResMsg | SBCCRespMsg,
SBCCResMsg == lineBranchReq(Reservation),
SBCCDeResMsg == lineBranchDeRes | lineDeRes
                        | branchDeRes,
SBCCRespMsg = LineBranchResp | SegmentResp,
LineBranchResp ==
          lineBranchResp(getRes : Reservation, isPos : Bool),
SegmentResp == segResp(isPos : Bool),

Reservation == mk_res(getTrain : TrainID, getDir : Direction),

ReturnSBCCMsg == hasMsg(SBCCMsg) | noSBCCMsg,

ComID == isSB(SBID) | isTrain(TrainID),
ComMsg == mk_comMsg(getSender : ComID,
                    getReceiver : ComID,
                    getMsg : Message),
HasComMsg == comMsg(ComMsg) | noComMsg

value
  /* The tick interval in seconds */
  tick_interval : Tick,

  /* Maximal ID numbers */
  sbIDSet : ID-set,
  segIDSet : ID-set,
  trainIDSet : ID-set,

  /* Limits the ID of SBs Segments and Trains */
  sbIDLimit : ID → Bool
  sbIDLimit(id) ≡
    id ∈ sbIDSet,

  segIDLimit : ID → Bool
  segIDLimit(id) ≡
    id ∈ segIDSet,

  trainIDLimit : ID → Bool
  trainIDLimit(id) ≡
    id ∈ trainIDSet,

  /* Inverse the direction */
  inverseDir : Direction → Direction
  inverseDir(dir) ≡
    case dir of
      UP → DOWN,
      DOWN → UP
    end,

  /* Determines if a certain location is
     included in a SBSegment */
  segPosInSBSeg : SegmentPosition × SBSegment → Bool
```

```
segPosInSBSeg(loc,sbSeg) ≡
    case sbSeg of
        seg(seg) → isSeg(seg) = getLoc(loc),
        point(upSeg,downSeg) → isSeg(upSeg) = getLoc(loc) ∨
                                       isSeg(downSeg) = getLoc(loc),
        esa(esa) → isESA(esa) = getLoc(loc)
    end,

/∗ Returns all the segments in a 'SBSegment' ∗/
sbSegToSet : SBSegment → SegmentID-set
sbSegToSet(sbSeg) ≡
    case sbSeg of
        seg(seg1) → { seg1 },
        point(seg1,seg2) → { seg1,seg2 },
        _ → {}
    end,

/∗ Returns the end to reach when
        following a certain direction ∗/
dir2End : Direction → End
dir2End(dir) ≡
    case dir of
        DOWN → LOW,
        UP → HIGH
    end,

/∗ Returns the direction against
     an ESA from the ESA ∗/
end2Dir : End → Direction
end2Dir(end1) ≡
    case end1 of
        LOW → DOWN,
        HIGH → UP
    end,

/∗ Determines if a location is an ESA ∗/
segPosIsESA : SegmentPosition → Bool
segPosIsESA(tEndPos) ≡
    case getLoc(tEndPos) of
        isESA(esa) → true,
        _ → false
    end,

/∗ Determines if an end position is a segment ∗/
segPosIsSeg : SegmentPosition → Bool
segPosIsSeg(tEndPos) ≡
    case getLoc(tEndPos) of
        isSeg(_) → true
    end,

/∗ Determines if a TrainPosition is
     located on one segment ∗/
trainOnlyOnESA : TrainPosition → Bool
trainOnlyOnESA(pos) ≡
    case getLoc(frontPos(pos)) of
        isESA(_) →
        (
```

```
            case getLoc(rearPos(pos)) of
              isESA(_) → true,
              _ → false
            end
          ),

          _ → false
        end,

      /∗ Returns a set containing a segment if the
         position is on a segment else an empty set ∗/
      segPosSeg : SegmentPosition → SegmentID-set
      segPosSeg(tep) ≡
        case getLoc(tep) of
          isSeg(seg) → {seg},
          _ → {}
        end,

      trainPosSegs : TrainPosition → SegmentID-set
      trainPosSegs(tp) ≡
        segPosSeg(frontPos(tp)) ∪ segPosSeg(rearPos(tp)),

      frontLoc : TrainPosition → Location
      frontLoc(tp) ≡
        getLoc(frontPos(tp)),

      rearLoc : TrainPosition → Location
      rearLoc(tp) ≡
        getLoc(rearPos(tp)),

      oneLoc : TrainPosition → Bool
      oneLoc(tp) ≡
        frontLoc(tp) = rearLoc(tp)

    axiom
      /∗ Two ID′s cannot be identical ∗/
      [ is_wf_id_sets ]
      sbIDSet ∪ segIDSet ∪ trainIDSet = {}

  end
```

F.3.2 Statics

```
context: CA_Types0, CA_SBs0, CA_Segs0, CA_Trains0, CA_ESAs0
scheme CA_Statics0(T : CA_Types0) =
  class
    object
      SBs : CA_SBs0(T),
      ESAs : CA_ESAs0(T),
      Segs : CA_Segs0(T),
      Trains : CA_Trains0(T)

    type
      /∗ Main railway line configuration type ∗/
      Configuration = SBs.SBs × Segs.Segs ×
```

```
                                        ESAs.ESAs × Trains.Trains

value
  conf : Configuration = (SBs.sbsConf, Segs.segsConf,
                                        ESAs.esasConf, Trains.trainsConf),

  /* Observers */

  /* ESA observers */
  getESASB : T.ESAID × Configuration → T.SBID
  getESASB(esa,(sbs,segs,esas,ts)) ≡
    ESAs.getESASB(esa,esas),

  getESALength : T.ESAID × Configuration → T.Length
  getESALength(esa,(sbs,segs,esas,ts)) ≡
    ESAs.getESALength(esa,esas),

  /* SB observers */
  getSBSeg : T.SBID × T.Direction ×
                                      Configuration → T.SBSegment
  getSBSeg(sb,dir,(sbs,segs,esas,ts)) ≡
    SBs.getSBSeg(sb,dir,sbs),

  getSBType : T.SBID × Configuration → T.SBType
  getSBType(sb,(sbs,segs,esas,ts)) ≡
    SBs.getSBType(sb,sbs),

  getPointTicks : T.SBID × Configuration ~→ T.Tick
  getPointTicks(sb,(sbs,segs,esas,ts)) ≡
    SBs.getPointTicks(sb,sbs)
  pre getSBType(sb,(sbs,segs,esas,ts)) = T.POINTSB,

  getBarrierTicks : T.SBID × Configuration ~→ T.Tick
  getBarrierTicks(sb,(sbs,segs,esas,ts)) ≡
    SBs.getBarrierTicks(sb,sbs)
  pre getSBType(sb,(sbs,segs,esas,ts)) = T.CROSSINGSB,

  getSignalTicks : T.SBID × Configuration ~→ T.Tick
  getSignalTicks(sb,(sbs,segs,esas,ts)) ≡
    SBs.getSignalTicks(sb,sbs)
  pre getSBType(sb,(sbs,segs,esas,ts)) = T.CROSSINGSB,

  /* Segment observers */
  getSegSB : T.SegmentID × T.Direction ×
                                          Configuration → T.SBID
  getSegSB(seg,dir,(sbs,segs,esas,ts)) ≡
    Segs.getSegSB(seg,dir,segs),

  getSegLength : T.SegmentID × Configuration → T.Length
  getSegLength(segID,(sbs,segs,esas,trains)) ≡
    Segs.getSegLength(segID,segs),

  getSegMaxSpeed : T.SegmentID × Configuration → T.Speed
  getSegMaxSpeed(segID,(sbs,segs,esas,trains)) ≡
    Segs.getSegMaxSpeed(segID,segs),

  /* Train observers */
```

```
getTrainLength : T.TrainID × Configuration → T.Length
getTrainLength(tID,(sbs,segs,esas,trains)) ≡
    Trains.getTrainLength(tID,trains),

getTrainMaxSpeed : T.TrainID × Configuration → T.Acceleration
getTrainMaxSpeed(t,(sbs,segs,esas,ts)) ≡
    Trains.getTrainMaxSpeed(t,ts),

getTrainMaxAcc : T.TrainID × Configuration → T.Acceleration
getTrainMaxAcc(t,(sbs,segs,esas,ts)) ≡
    Trains.getTrainMaxAcc(t,ts),

getTrainMaxDec : T.TrainID × Configuration → T.Acceleration
getTrainMaxDec(t,(sbs,segs,esas,ts)) ≡
    Trains.getTrainMaxDec(t,ts),

/* Reservation− and brake−point observers */
getResPoint : Configuration → T.Length
getResPoint((sbs,segs,esas,ts)) ≡
    Segs.getResPoint(segs),

getBrakePoint : Configuration → T.Length
getBrakePoint((sbs,segs,esas,ts)) ≡
    Segs.getBrakePoint(segs),

/* Auxiliary functions */

/* Determines if a SB is a Single Line Guard */
isLineGuard : T.SBID × Configuration → Bool
isLineGuard(sbID,con) ≡
    getSBType(sbID,con) ∈ {T.POINTSB, T.ENDSB},

/* Determines if a SB is a PointSB */
isPointSB : T.SBID × Configuration → Bool
isPointSB(sbID,con) ≡
    getSBType(sbID,con) = T.POINTSB,

/* Determines if a segment is a branch segment */
segIsBranch : T.SegmentID × Configuration → Bool
segIsBranch(seg,con) ≡
    getSBType(getSegSB(seg,T.UP,con),con) = T.POINTSB ∧
    getSBType(getSegSB(seg,T.DOWN,con),con) = T.POINTSB,

/* Determines if a segment is a line segment,
     i.e. a segment in a single line */
segIsLineSegment : T.SegmentID × Configuration → Bool
segIsLineSegment(seg,con) ≡
    ∼segIsBranch(seg,con),

/* If two T.Location′s are neighbours in configuration.
     Note that a T.Location is not neighbour to itself */
neighbours : T.Location × T.Location × Configuration → Bool
neighbours(loc1,loc2,con) ≡
    (getLocSBs(loc1,con) ∪ getLocSBs(loc2,con)) ≠ {},


/* Finds the distance (T.Length)
```

```
     between two T.SegmentPosition */
distance : T.SegmentPosition × T.SegmentPosition ×
                                         Configuration → T.Length
distance(segPos1,segPos2,con) ≡
    if (T.getLoc(segPos1) = T.getLoc(segPos2))
    then
        if (T.getLength(segPos1) < T.getLength(segPos2))
        then
            T.getLength(segPos2) − T.getLength(segPos1)
        else
            T.getLength(segPos1) − T.getLength(segPos2)
        end
    else
        if (segPosLower(segPos1,segPos2,con))
        then
            getLocLength(T.getLoc(segPos1),con) −
                    T.getLength(segPos1) + T.getLength(segPos2)
        else
            getLocLength(T.getLoc(segPos2),con) −
                    T.getLength(segPos2) + T.getLength(segPos1)
        end
    end
pre neighbours(T.getLoc(segPos1),T.getLoc(segPos2),con) ∨
        T.getLoc(segPos1) = T.getLoc(segPos2),

segPosLower : T.SegmentPosition × T.SegmentPosition ×
                                         Configuration → Bool
segPosLower(segPos1,segPos2,con) ≡
    if (T.getLoc(segPos1) = T.getLoc(segPos2)) then
        T.getLength(segPos1) < T.getLength(segPos2)
    else
        locLower(T.getLoc(segPos1),T.getLoc(segPos2),con)
    end,

/* If a location is immediatedly lower than
        another location. If one location is an ESA,
        the result will depend on the orientation
     of the ESA. */
locLower : T.Location × T.Location × Configuration → Bool
locLower(loc1,loc2,con) ≡
    case loc1 of
        T.isESA(esa1) → (esa1 = T.LOW),
        T.isSeg(seg1) →
        (
            case loc2 of
                T.isESA(esa2) → (esa2 = T.HIGH),
                T.isSeg(seg2) →
                (
                    seg2 ∈ getNextSegSet(seg1,T.UP,con)
                )
            end
        )
    end,

getLocLength : T.Location × Configuration → T.Length
getLocLength(loc,con) ≡
    case loc of
```

```
        T.isESA(esa) → getESALength(esa,con),
        T.isSeg(seg) → getSegLength(seg,con)
    end,

/* Returns the SB′s (up and down) of a
     location  (segment / esa) */
getLocSBs : T.Location × Configuration → T.SBID-set
getLocSBs(loc,con) ≡
    case loc of
        T.isESA(esa) → {getESASB(esa,con)},
        T.isSeg(seg) → {getSegSB(seg,T.UP,con),
                                getSegSB(seg,T.DOWN,con)}
    end,

/* Returns the segments around a point */
getSBPointSegs : T.SBID × Configuration ~→ T.PointSegments
getSBPointSegs(sb,(sbs,segs,esas,ts)) ≡
    SBs.getSBPointSegs(sb,sbs)
pre getSBType(sb,(sbs,segs,esas,ts)) = T.POINTSB,

/* Returns the driving direction of a branch */
branchDir : T.SegmentID × Configuration → T.Direction
branchDir(seg,con) ≡
    let
        T.point(up,down) = getSBSeg(
                        getSegSB(seg,T.UP,con),T.DOWN,con)
    in
        if (seg = up)
        then
            T.UP
        else
            T.DOWN
        end
    end
pre segIsBranch(seg,con),

/* Given a single line guard, it returns the single
     line guard at the opposite end of the single line */
getOppositeGuard : T.SBID × Configuration ~→ T.SBID
getOppositeGuard(sb,con) ≡
    let
        sbType = getSBType(sb,con),
        dir = if(sbType = T.POINTSB) then getPointDir(sb,con)
                else getEndDir(sb,con) end,
        lineDir = T.inverseDir(dir)
    in
        getSingleLineGuard(getNextSB(sb,lineDir,con),lineDir,con)
    end
pre isLineGuard(sb,con),

/* Given a point SB, it returns the point SB
     at the opposite end of the branches */
getOppositePointSB : T.SBID × Configuration ~→ T.SBID
getOppositePointSB(sb,con) ≡
    let
        dir = getPointDir(sb,con)
    in
```

```
        getNextSB(sb,dir,con)
    end
pre getSBType(sb,con) = T.POINTSB,

/∗ Given an SB, it returns the next SB ∗/
getNextSB : T.SBID × T.Direction ×
                                        Configuration ∼→ T.SBID
getNextSB(sb,dir,con) ≡
    let
        nextSeg = getSBSeg(sb,dir,con)
    in
        case nextSeg of
            T.seg(segID) → getSegSB(segID,dir,con),
            T.point(upSeg,downSeg) → getSegSB(upSeg,dir,con)
        end
    end
pre getSBType(sb,con) ≠ T.ENDSB ∨ getEndDir(sb,con) ≠ dir,

getNextSegSet : T.SegmentID × T.Direction ×
                                        Configuration → T.SegmentID-set
getNextSegSet(seg,dir,con) ≡
    T.sbSegToSet(getSBSeg(getSegSB(seg,dir,con),dir,con)),

/∗ Returns the first single line guard in a direction ∗/
getSingleLineGuard : T.SBID × T.Direction ×
                                        Configuration ∼→ T.SBID
getSingleLineGuard(sb,dir,con) ≡
    if(isLineGuard(sb,con))
    then
        sb
    else
        getSingleLineGuard(getNextSB(sb,dir,con),dir,con)
    end,

/∗ Returns the two single line guards of a line−segment ∗/
getSingleLineGuards : T.SegmentID ×
                                        Configuration ∼→ T.SBID-set
getSingleLineGuards(seg,con) ≡
    let
        sb = getSegSB(seg,T.UP,con)
    in
        { getSingleLineGuard(sb,T.UP,con),
            getSingleLineGuard(sb,T.DOWN,con) }
    end
pre ∼segIsBranch(seg,con),

/∗ Returns the direction of a point SB
    from the stem towards the branches ∗/
getPointDir : T.SBID × Configuration ∼→ T.Direction
getPointDir(sbID,(sbs,segs,esas,trains)) ≡
    SBs.getPointDir(sbID,sbs)
pre getSBType(sbID,(sbs,segs,esas,trains)) = T.POINTSB,

/∗ Returns the direction against an ESA from an END SB ∗/
getEndDir : T.SBID × Configuration ∼→ T.Direction
getEndDir(sbID,(sbs,segs,esas,trains)) ≡
```

```
    SBs.getEndDir(sbID,sbs)
pre getSBType(sbID,(sbs,segs,esas,trains)) = T.ENDSB,

sbsAreCrossings : T.SBID-set × Configuration → Bool
sbsAreCrossings(sbs,con) ≡
(
    ∀ sb : T.SBID •
        sb ∈ sbs ⇒
            getSBType(sb,con) = T.CROSSINGSB
),

sbsArePoints : T.SBID-set × Configuration → Bool
sbsArePoints(sbs,con) ≡
(
    ∀ sb : T.SBID •
        sb ∈ sbs ⇒
            getSBType(sb,con) = T.POINTSB
),

/∗ Invariants ∗/
is_wf : Configuration → Bool
is_wf((sbs,segs,esas,ts)) ≡
    SBs.is_wf(sbs) ∧
    Segs.is_wf(segs) ∧
    ESAs.is_wf(esas) ∧
    Trains.is_wf(ts) ∧
    composed_is_wf((sbs,segs,esas,ts)),

composed_is_wf : Configuration → Bool
composed_is_wf(con) ≡
    pointSegs_wf(con) ∧
    getESASBSeg_wf(con) ∧
    getSBSeg_getSegSB_wf(con)∧
    seg_train_length_wf(con) ∧
    esa_train_length_wf(con) ∧
    brakePoint_wf(con) ∧
    resPoint_wf(con) ∧
    collisions_detectable(con),

/∗ All associated point (points next to each other)
     must have same up and down branches ∗/
pointSegs_wf : Configuration → Bool
pointSegs_wf(con) ≡
(
    ∀ sb : T.SBID •
        getSBType(sb,con) = T.POINTSB
            ⇒
                let
                    pSegs1 = getSBPointSegs(sb,con),
                    dir1 = T.getPointDir(pSegs1),

                    sb2 = getNextSB(sb,dir1,con),
                    pSegs2 = getSBPointSegs(sb2,con)
                in
                    T.getUpBranch(pSegs1) = T.getUpBranch(pSegs2) ∧
                    T.getDownBranch(pSegs1) = T.getDownBranch(pSegs2)
                end
```

```
 ),

/* Given an ESA. From the coherent END SB
     the next SBSegment directed against the
     ESA must be the ESA */
getESASBSeg_wf : Configuration → Bool
getESASBSeg_wf(con) ≡
(
    ∀ esa : T.ESAID •
        getSBSeg(getESASB(esa,con),T.end2Dir(esa),con) = T.esa(esa)
),

/* Calculating the SB in a direction from each segment
     in the SBSegment calculated from a SB in the opposite
     direction must give the original SB */
getSBSeg_getSegSB_wf : Configuration → Bool
getSBSeg_getSegSB_wf(con) ≡
(
    ∀ sb : T.SBID, dir : T.Direction, seg : T.SegmentID •
        seg ∈ T.sbSegToSet(getSBSeg(sb,dir,con)) ⇒
            getSegSB(seg,T.inverseDir(dir),con) = sb
),

/* All segments must be longer than any train */
seg_train_length_wf : Configuration → Bool
seg_train_length_wf(con) ≡
(
    ∀ seg : T.SegmentID, t : T.TrainID •
        getSegLength(seg,con) >
                getBrakePoint(con) + getTrainLength(t,con)
),

/* An ESA must be longer than a brake point.
     This ensures that all the axioms above
     (concerning braking) also applies to the ESAs
*/
esa_train_length_wf : Configuration → Bool
esa_train_length_wf(con) ≡
(
    ∀ esa : T.ESAID, t : T.TrainID •
        getESALength(esa,con) > getTrainLength(t,con)
),

/* If a train starts to brake at the brakepoint
     it must be able to stop entirely before
     entering the next segment
*/
brakePoint_wf : Configuration → Bool
brakePoint_wf(con) ≡
(
    ∀ t : T.TrainID, tAcc : T.Acceleration,
            brakeP, brakeL, s_err : T.Length,
            tSpeed : T.Speed •
                tAcc = getTrainMaxDec(t,con) ∧
                brakeP = getBrakePoint(con) ∧
                tSpeed = getTrainMaxSpeed(t,con) ∧
                s_err = tSpeed * T.tick_interval ∧
```

```
                    brakeL = −0.5 ∗ tSpeed ∗ tSpeed / tAcc
                        ⇒
                            brakeP > brakeL + s_err
        ),

        /∗ ensures that resPoint > brakePoint and that
             a train is entirely on a single segment when
             resPoint is exceeded.
             also that brakePoint < segment length ∗/
        resPoint_wf : Configuration → Bool
        resPoint_wf(con) ≡
        (
            ∀ t : T.TrainID, seg : T.SegmentID,
                    tlen, slen, resPoint, brakePoint : T.Length •
                        tlen = getTrainLength(t,con) ∧
                        slen = getSegLength(seg,con) ∧
                        resPoint = getResPoint(con) ∧
                        brakePoint = getBrakePoint(con)
                            ⇒
                                slen  > (resPoint + tlen) ∧
                                brakePoint < slen
        ),

        /∗ Ensures that collisions can be
             detected before trains passes
             through each other ∗/
        collisions_detectable : Configuration → Bool
        collisions_detectable(con) ≡
        (
            ∀ t1, t2 : T.TrainID, sp1, sp2 : T.Speed,
                    s_err1, s_err2, s_col : T.Length •
                        sp1 = getTrainMaxSpeed(t1,con) ∧
                        sp2 = getTrainMaxSpeed(t2,con) ∧
                        s_err1 = sp1 ∗ T.tick_interval ∧
                        s_err2 = sp2 ∗ T.tick_interval ∧
                        s_col = s_err1 + s_err2
                            ⇒
                                s_col < getTrainLength(t1,con)
        )

    axiom
        [is_wf]
            is_wf(conf)

end
```

SBs

```
context: CA_Types0
scheme CA_SBs0(T : CA_Types0) =
    class
        type
            /∗ Type of interest ∗/
            SBs = T.SBID →m SBData,
```

```
    /* Data for each SB */
    SBData == mk_sb(getUpSeg : T.SBSegment,
                    getDownSeg : T.SBSegment,
                    getType : T.SBType,
                    getPointTicks : T.HasTicks,
                    getBarrierTicks : T.HasTicks,
                    getSignalTicks : T.HasTicks)

value
    sbsConf : SBs,

    /* SB observers */
    getSBSeg : T.SBID × T.Direction × SBs ~→ T.SBSegment
    getSBSeg(sb,dir,sbs) ≡
        if(dir = T.UP)
        then
            getUpSeg(sbs(sb))
        else
            getDownSeg(sbs(sb))
        end
    pre sbExistsInConf(sb,sbs),

    getSBType : T.SBID × SBs ~→ T.SBType
    getSBType(sb,sbs) ≡
        getType(sbs(sb))
    pre sbExistsInConf(sb,sbs),

    getPointTicks : T.SBID × SBs ~→ T.Tick
    getPointTicks(sb,sbs) ≡
        T.getTicks(getPointTicks(sbs(sb)))
    pre getSBType(sb,sbs) = T.POINTSB ∧
            sbExistsInConf(sb,sbs),

    getBarrierTicks : T.SBID × SBs ~→ T.Tick
    getBarrierTicks(sb,sbs) ≡
        T.getTicks(getBarrierTicks(sbs(sb)))
    pre getSBType(sb,sbs) = T.CROSSINGSB ∧
            sbExistsInConf(sb,sbs),

    getSignalTicks : T.SBID × SBs ~→ T.Tick
    getSignalTicks(sb,sbs) ≡
        T.getTicks(getSignalTicks(sbs(sb)))
    pre getSBType(sb,sbs) = T.CROSSINGSB ∧
            sbExistsInConf(sb,sbs),

    sbExistsInConf : T.SBID × SBs → Bool
    sbExistsInConf(sb,sbs) ≡
        sb ∈ dom sbs,

    /* Auxiliary functions */

    /* Returns the direction of a point SB
       from the stem towards the branches */
    getPointDir : T.SBID × SBs ~→ T.Direction
    getPointDir(sb,sbs) ≡
        let sbSeg = getSBSeg(sb,T.UP,sbs)
```

```
    in
        case sbSeg of
            T.point(_,_) → T.UP,
            _ → T.DOWN
        end
    end
pre getSBType(sb,sbs) = T.POINTSB,

/* Returns the segments around a point */
getSBPointSegs : T.SBID × SBs ~→ T.PointSegments
getSBPointSegs(sbID,sbs) ≡
    let
        dir = getPointDir(sbID,sbs),
        pointSegs = getSBSeg(sbID,dir,sbs),
        T.seg(stemSeg) = getSBSeg(sbID,T.inverseDir(dir),sbs)
    in
        T.pointSegments(stemSeg,
                              T.getUpSeg(pointSegs),
                              T.getDownSeg(pointSegs),
                              dir)
    end
pre getSBType(sbID,sbs) = T.POINTSB,

/* Returns the direction against an ESA from an END SB */
getEndDir : T.SBID × SBs ~→ T.Direction
getEndDir(sb,sbs) ≡
    case getSBSeg(sb,T.UP,sbs) of
        T.esa(_) → T.UP,
        _ → T.DOWN
    end
pre getSBType(sb,sbs) = T.ENDSB,

/* Invariants */
is_wf : SBs → Bool
is_wf(sbs) ≡
    sbsHaveConf(sbs) ∧
    getSBSeg_diff(sbs) ∧
    getSBSeg_point_wf(sbs) ∧
    getSBSeg_injective(sbs) ∧
    getSBSegType_wf(sbs),

/* A configuration for each SB must exists */
sbsHaveConf : SBs → Bool
sbsHaveConf(sbs) ≡
(
    ∀ sb : T.SBID •
        sbExistsInConf(sb,sbs)
),

/* The segments next to a SB are different
   in the UP and the DOWN direction.
   I.e. the line is not circular */
getSBSeg_diff : SBs → Bool
getSBSeg_diff(sbs) ≡
(
    ∀ sb : T.SBID •
        getSBSeg(sb,T.UP,sbs) ≠ getSBSeg(sb,T.DOWN,sbs)
```

```
),

/* The two branches of a junction are different */
getSBSeg_point_wf : SBs → Bool
getSBSeg_point_wf(sbs) ≡
(
    ∀ sb : T.SBID,
            seg1,seg2 : T.SegmentID,
            dir : T.Direction •
                T.point(seg1,seg2) = getSBSeg(sb,dir,sbs) ⇒
                    seg1 ≠ seg2
),

/* Two different SBs have different SBSegments
     in the same direction */
getSBSeg_injective : SBs → Bool
getSBSeg_injective(sbs) ≡
(
    ∀ sb1, sb2 : T.SBID,
            dir : T.Direction •
                sb1 ≠ sb2 ⇒
                    getSBSeg(sb1,dir,sbs) ≠ getSBSeg(sb2,dir,sbs)
),

/* The type of a SB must conform
     with the result of getSBSeg */
getSBSegType_wf : SBs → Bool
getSBSegType_wf(sbs) ≡
(
    ∀ sb : T.SBID •
        case getSBType(sb,sbs) of
            T.ENDSB →
                (∃! dir : T.Direction, esaID : T.ESAID •
                    esaID = T.dir2End(dir) ∧
                                    getSBSeg(sb,dir,sbs) = T.esa(esaID)),
            T.POINTSB →
                    (∃! dir : T.Direction, seg1,seg2 : T.SegmentID •
                    getSBSeg(sb,dir,sbs) = T.point(seg1,seg2)),
            T.CROSSINGSB →
                                (∀ dir : T.Direction •
                    ∃ seg : T.SegmentID •
                                getSBSeg(sb,dir,sbs) = T.seg(seg)),
            T.PLAINSB →
                                (∀ dir : T.Direction •
                    ∃ seg : T.SegmentID •
                                getSBSeg(sb,dir,sbs) = T.seg(seg))
        end
)

end
```

Segs

context: CA_Types0
scheme CA_Segs0(T : CA_Types0) =

```
class
  type
    /∗ Type of interest ∗/
    Segs = SegsData × SegPoints,

    SegsData = T.SegmentID →m SegData,
    SegPoints :: getRP : T.Length
                 getBP : T.Length,

    /∗ Data for each Segment ∗/
    SegData == mk_seg(getUpSB : T.SBID,
                      getDownSB : T.SBID,
                      getLength : T.Length,
                      getMaxSpeed : T.Speed)

  value
    segsConf : Segs,

    /∗ Segment observers ∗/
    getSegSB : T.SegmentID × T.Direction × Segs ~→ T.SBID
    getSegSB(seg,dir,(segs,sp)) ≡
      if(dir = T.UP)
      then
        getUpSB(segs(seg))
      else
        getDownSB(segs(seg))
      end
    pre segExistsInConf(seg,(segs,sp)),

    getSegLength : T.SegmentID × Segs ~→ T.Length
    getSegLength(seg,(segs,sp)) ≡
      getLength(segs(seg))
    pre segExistsInConf(seg,(segs,sp)),

    getSegMaxSpeed : T.SegmentID × Segs ~→ T.Speed
    getSegMaxSpeed(seg,(segs,sp)) ≡
      getMaxSpeed(segs(seg))
    pre segExistsInConf(seg,(segs,sp)),

    segExistsInConf : T.SegmentID × Segs → Bool
    segExistsInConf(seg,(segs,sp)) ≡
      seg ∈ dom segs,

    /∗ Reservation− and brake−point observers ∗/
    getResPoint : Segs → T.Length
    getResPoint((segs,sp)) ≡
      getRP(sp),

    getBrakePoint : Segs → T.Length
    getBrakePoint((segs,sp)) ≡
      getBP(sp),

    /∗ Invariant ∗/
    is_wf : Segs → Bool
    is_wf(segs) ≡
      segsHaveConf(segs) ∧
      getSegSB_injective(segs) ∧
```

```
            brakeResPoint_wf(segs),

        /* A configuration for each Segment must exists */
        segsHaveConf : Segs → Bool
        segsHaveConf(segs) ≡
        (
            (∀ seg : T.SegmentID •
                segExistsInConf(seg,segs)) ∧
             getResPoint(segs) > 0.0 ∧
             getBrakePoint(segs) > 0.0
        ),

        /**
        * The SB in the end of a segment is different
        * for two different segments or they are the
        * same in both direction (being branches)
        **/
        getSegSB_injective : Segs → Bool
        getSegSB_injective(segs) ≡
        (
            ∀ seg1, seg2 : T.SegmentID,
                  dir : T.Direction •
                      seg1 ≠ seg2 ⇒
                      (
                          getSegSB(seg1,dir,segs) ≠ getSegSB(seg2,dir,segs)
                      )
                          ∨
                      (
                      getSegSB(seg1,T.UP,segs) = getSegSB(seg2,T.UP,segs) ∧
                      getSegSB(seg1,T.DOWN,segs) = getSegSB(seg2,T.DOWN,segs)
                      )
        ),

        /* The reservation−point should be placed before
             the brake−point, i.e. there is a greater
             distance from the end of a segment to the
             reservation−point than to the brake−point */
        brakeResPoint_wf : Segs → Bool
        brakeResPoint_wf(segs) ≡
            getResPoint(segs) > getBrakePoint(segs)

  end
```

ESAs

```
context: CA_Types0
scheme CA_ESAs0(T : CA_Types0) =
  class
    type
        /* Type of interest */
        ESAs == mk_esa(getLowSB : T.SBID,
                                    getHighSB : T.SBID,
                                    getLowLength : T.Length,
                                    getHighLength : T.Length)
```

```
value
    esasConf : ESAs,

    /∗ ESA observers ∗/
    getESASB : T.ESAID × ESAs ~→ T.SBID
    getESASB(esa,esas) ≡
        if(esa = T.LOW)
        then
            getLowSB(esas)
        else
            getHighSB(esas)
        end
    pre esaExistsInConf(esa,esas),

    getESALength : T.ESAID × ESAs ~→ T.Length
    getESALength(esa,esas) ≡
        if(esa = T.LOW)
        then
            getLowLength(esas)
        else
            getHighLength(esas)
        end
    pre esaExistsInConf(esa,esas),

    esaExistsInConf : T.ESAID × ESAs → Bool
    esaExistsInConf(esa,esas) ≡
        getLowLength(esas) > 0.0 ∧
        getHighLength(esas) > 0.0,

    /∗ Invariants ∗/
    is_wf : ESAs → Bool
    is_wf(esas) ≡
        esasHaveConf(esas),

    /∗ A configuration for each ESA must exists ∗/
    esasHaveConf : ESAs → Bool
    esasHaveConf(esas) ≡
    (
        ∀ esa : T.ESAID •
            esaExistsInConf(esa,esas)
    )

end
```

Trains

```
context: CA_Types0
scheme CA_Trains0(T : CA_Types0) =
    class
        type
            /∗ Type of interest ∗/
            Trains = T.TrainID →m TData,

            /∗ Data for each Train ∗/
            TData == mk_train(getLength : T.Length,
```

```
                                          getMaxSpeed : T.Speed,
                                          getMaxAcc : T.Acceleration,
                                          getMaxDec : T.Acceleration)
    value
      trainsConf : Trains,

      /* Train observers */
      getTrainLength : T.TrainID × Trains ~→ T.Length
      getTrainLength(t,trains) ≡
        getLength(trains(t))
      pre trainExistsInConf(t,trains),

      getTrainMaxSpeed : T.TrainID × Trains ~→ T.Speed
      getTrainMaxSpeed(t,trains) ≡
        getMaxSpeed(trains(t))
      pre trainExistsInConf(t,trains),

      getTrainMaxAcc : T.TrainID × Trains ~→ T.Acceleration
      getTrainMaxAcc(t,trains) ≡
        getMaxAcc(trains(t))
      pre trainExistsInConf(t,trains),

      getTrainMaxDec : T.TrainID × Trains ~→ T.Acceleration
      getTrainMaxDec(t,trains) ≡
        getMaxDec(trains(t))
      pre trainExistsInConf(t,trains),

      trainExistsInConf : T.TrainID × Trains → Bool
      trainExistsInConf(t,trains) ≡
        t ∈ dom trains,

      /* Invariants */
      is_wf : Trains → Bool
      is_wf(trains) ≡
        trainsHaveConf(trains),

      /* A configuration for each Train must exists */
      trainsHaveConf : Trains → Bool
      trainsHaveConf(trains) ≡
      (
        ∀ t : T.TrainID •
          trainExistsInConf(t,trains)
      )

  end
```

F.3.3 Dynamics

```
context: CA_Statics0, CA_TrainDyn0, CA_SBDyn0, CA_Types0
scheme CA_Dynamics0(T : CA_Types0, S : CA_Statics0(T)) =
  class
    object
      TD : CA_TrainDyn0(T,S),
      SD : CA_SBDyn0(T,S)
```

```
type
   /* Type of interest */
   State = TD.TrainStates × SD.SBStates

value
   initState : State = (TD.initTrainStates,SD.initSBStates),

   /* Point observer */
   getPointPosition : T.SBID × State × S.Configuration ~→ T.PointPosition
   getPointPosition(sbID,(ts,sbs),con) ≡
      SD.getPointPosition(sbID,sbs,con)
   pre S.getSBType(sbID,con) = T.POINTSB,

   getPointTicks : T.SBID × State × S.Configuration ~→ T.Tick
   getPointTicks(sbID,(ts,ss),con) ≡
      SD.getPointTicks(sbID,ss,con)
   pre S.getSBType(sbID,con) = T.POINTSB,

   /* Point generator */
   setPointPosition : T.SBID × T.PointPosition × State ×
                                          S.Configuration ~→ State
   setPointPosition(sbID,ppos,(ts,ss),con) ≡
      (ts,SD.setPointPosition(sbID,ppos,ss,con))
   pre S.getSBType(sbID,con) = T.POINTSB ∧
         ∼trainOnJunction(sbID,con,(ts,ss)),

   setPointTicks : T.SBID × T.Tick × State × S.Configuration ~→ State
   setPointTicks(sbID,tick,(ts,ss),con) ≡
      (ts, SD.setPointTicks(sbID,tick,ss,con))
   pre S.getSBType(sbID,con) = T.POINTSB,

   /* Crossing observer */
   getBarrierPosition : T.SBID × State × S.Configuration ~→ T.BarrierPosition
   getBarrierPosition(sbID,(ts,sbs),con) ≡
      SD.getBarrierPosition(sbID,sbs,con)
   pre S.getSBType(sbID,con) = T.CROSSINGSB,

   getSignalStatus : T.SBID × State × S.Configuration ~→ T.SignalStatus
   getSignalStatus(sbID,(ts,sbs),con) ≡
      SD.getSignalStatus(sbID,sbs,con)
   pre S.getSBType(sbID,con) = T.CROSSINGSB,

   getBarrierTicks : T.SBID × State × S.Configuration ~→ T.Tick
   getBarrierTicks(sbID,(ts,ss),con) ≡
      SD.getBarrierTicks(sbID,ss,con)
   pre S.getSBType(sbID,con) = T.CROSSINGSB,

   getSignalTicks : T.SBID × State × S.Configuration ~→ T.Tick
   getSignalTicks(sbID,(ts,ss),con) ≡
      SD.getSignalTicks(sbID,ss,con)
   pre S.getSBType(sbID,con) = T.CROSSINGSB,

   /* Crossing generator */
   setBarrierPosition : T.SBID × T.BarrierPosition × State ×
                                          S.Configuration ~→ State
```

```
setBarrierPosition(sbID,bPos,(ts,sbs),con) ≡
    (ts,SD.setBarrierPosition(sbID,bPos,sbs,con))
pre S.getSBType(sbID,con) = T.CROSSINGSB,

setSignalStatus : T.SBID × T.SignalStatus × State × S.Configuration ~→ State
setSignalStatus(sbID,sigStat,(ts,sbs),con) ≡
    (ts,SD.setSignalStatus(sbID,sigStat,sbs,con))
pre S.getSBType(sbID,con) = T.CROSSINGSB,

setBarrierTicks : T.SBID × T.Tick × State × S.Configuration ~→ State
setBarrierTicks(sbID,tick,(ts,ss),con) ≡
    (ts, SD.setBarrierTicks(sbID,tick,ss,con))
pre S.getSBType(sbID,con) = T.CROSSINGSB,

setSignalTicks : T.SBID × T.Tick × State × S.Configuration ~→ State
setSignalTicks(sbID,tick,(ts,ss),con) ≡
    (ts, SD.setSignalTicks(sbID,tick,ss,con))
pre S.getSBType(sbID,con) = T.CROSSINGSB,

/∗ Sensor observer ∗/
getSensorStatus : T.SBID × State → T.SensorStatus
getSensorStatus(sbID,(ts,sbs)) ≡
    SD.getSensorStatus(sbID,sbs),

/∗ Sensor generator ∗/
setSensorStatus : T.SBID × T.SensorStatus × State × S.Configuration ~→ State
setSensorStatus(sbID,senStat,(ts,ss),con) ≡
    (ts,SD.setSensorStatus(sbID,senStat,ss))
pre sensor_guard(sbID,senStat,con,(ts,ss)),

/∗ Train observer ∗/
getTrainAcc : T.TrainID × State → T.Acceleration
getTrainAcc(tID,(ts,sbs)) ≡
    TD.getTrainAcc(tID,ts),

getTrainSpeed : T.TrainID × State → T.Speed
getTrainSpeed(tID,(ts,sbs)) ≡
    TD.getTrainSpeed(tID,ts),

getTrainPosition : T.TrainID × State → T.TrainPosition
getTrainPosition(tID,(ts,sbs)) ≡
    TD.getTrainPosition(tID,ts),

getTrainDirection : T.TrainID × State → T.Direction
getTrainDirection(tID,(ts,sbs)) ≡
    TD.getTrainDirection(tID,ts),

/∗ Train generator ∗/
setTrainAcc : T.TrainID × T.Acceleration × State × S.Configuration ~→ State
setTrainAcc(tID,acc,(ts,ss),con) ≡
    (TD.setTrainAcc(tID,acc,ts,con),ss)
pre acc ≤ S.getTrainMaxAcc(tID,con) ∧
        S.getTrainMaxDec(tID,con) ≤ acc,

setTrainSpeed : T.TrainID × T.Speed × State × S.Configuration ~→ State
setTrainSpeed(tID,speed,(ts,ss),con) ≡
```

```
    (TD.setTrainSpeed(tID,speed,ts,con),ss)
pre speed ≤ S.getTrainMaxSpeed(tID,con),

setTrainPosition : T.TrainID × T.TrainPosition × State ×
                                                    S.Configuration ~→ State
setTrainPosition(tID,pos,(ts,ss),con) ≡
    (TD.setTrainPosition(tID,pos,ts,con),ss)
pre ∼trainPositionOccupied(tID,pos,(ts,ss),con) ∧
        ∼tpDerailed(pos,getTrainDirection(tID,(ts,ss)),(ts,ss),con),

setTrainDirection : T.TrainID × T.Direction × State ~→ State
setTrainDirection(tID,dir,(ts,ss)) ≡
    (TD.setTrainDirection(tID,dir,ts),ss)
pre getTrainSpeed(tID,(ts,ss)) = 0.0 ∨
        getTrainDirection(tID,(ts,ss)) = dir,

changeTrainDirection : T.TrainID × State × S.Configuration → State
changeTrainDirection(t,s,con) ≡
    let
        dir = T.inverseDir(getTrainDirection(t,s)),
        tp = getTrainPosition(t,s),

        front = T.frontPos(tp),
        rear = T.rearPos(tp),

        tp = T.mk_TrainPosition(rear,front),

        s = setTrainDirection(t,dir,s)
    in
        setTrainPosition(t,tp,s,con)
    end,

/∗ Processes ∗/
tick : T.Tick × S.Configuration × State ~→ State
tick(tick,con,s) ≡
    let
        s = tickPoints(tick,con,s),
        s = tickCrossings(tick,con,s),
        s = tickTrains(tick,con,s)
    in
        s
    end,

tickPoints : T.Tick × S.Configuration × State ~→ State
tickPoints(tick,con,s) ≡
    let
        points = { p | p : T.SBID • S.getSBType(p,con) = T.POINTSB }
    in
        pointProcess(points,tick,con,s)
    end,

pointProcess : T.SBID-set × T.Tick × S.Configuration × State ~→ State
pointProcess(points,tick,con,s) ≡
    if(points = {})
    then
        s
```

```
    else
        let
            p : T.SBID • p ∈ points,
            points = points \ {p},
            s = updatePoint(p,tick,con,s)
        in
            pointProcess(points,tick,con,s)
        end
    end
pre S.sbsArePoints(points,con),

updatePoint : T.SBID × T.Tick × S.Configuration × State ~→ State
updatePoint(p,tick,con,s) ≡
    let
        pp = getPointPosition(p,s,con)
    in
        case pp of
            T.MOVINGDOWN → movePoint(p,tick,T.DOWN,s,con),
            T.MOVINGUP → movePoint(p,tick,T.UP,s,con),
            _ → s
        end
    end
pre S.getSBType(p,con) = T.POINTSB,

movePoint : T.SBID × T.Tick × T.PointPosition × State ×
                                            S.Configuration ~→ State
movePoint(p,tick,pp,s,con) ≡
    let
        ticks = S.getPointTicks(p,con),
        curTick = getPointTicks(p,s,con)
    in
        if(curTick ≥ ticks)
        then
            let
                s = setPointPosition(p,pp,s,con)
            in
                setPointTicks(p,0.0,s,con)
            end
        else
            setPointTicks(p,curTick+tick,s,con)
        end
    end,

tickCrossings : T.Tick × S.Configuration × State ~→ State
tickCrossings(tick,con,s) ≡
    let
        crossings = { c | c : T.SBID • S.getSBType(c,con) = T.CROSSINGSB }
    in
        crossingProcess(crossings,tick,con,s)
    end,

crossingProcess : T.SBID-set × T.Tick × S.Configuration × State ~→ State
crossingProcess(crossings,tick,con,s) ≡
    if(crossings = {})
    then
        s
    else
```

```
            let
                c : T.SBID • c ∈ crossings,
                crossings = crossings \ {c},
                s = updateCrossing(c,tick,con,s)
            in
                crossingProcess(crossings,tick,con,s)
            end
        end
    pre S.sbsAreCrossings(crossings,con),

    updateCrossing : T.SBID × T.Tick × S.Configuration × State ~→ State
    updateCrossing(cr,tick,con,s) ≡
        let
            bp = getBarrierPosition(cr,s,con),
            ss = getSignalStatus(cr,s,con),

            bTicks = S.getBarrierTicks(cr,con),
            newBTicks = tick + getBarrierTicks(cr,s,con),

            sTicks = S.getSignalTicks(cr,con),
            newSTicks = tick + getSignalTicks(cr,s,con)

        in
            case bp of
                T.UP →
                (
                    if(ss = T.ON)
                    then
                        if(newSTicks > sTicks)
                        then
                            let
                                s = setSignalTicks(cr,0.0,s,con),
                                s = setBarrierPosition(cr,T.MOVINGDOWN,s,con)
                            in
                                s
                            end
                        else
                            setSignalTicks(cr,newSTicks,s,con)
                        end
                    else
                        s
                    end
                ),
                T.MOVINGDOWN →
                (
                    if(newBTicks > bTicks)
                    then
                        let
                            s = setBarrierTicks(cr,0.0,s,con),
                            s = setBarrierPosition(cr,T.DOWN,s,con),
                            s = setSignalStatus(cr,T.OFF,s,con)
                        in
                            s
                        end
                    else
                        setBarrierTicks(cr,newBTicks,s,con)
                    end
```

```
            ),
            T.DOWN → s,
            T.MOVINGUP →
            (
                if(newBTicks > bTicks)
                then
                    let
                        s = setBarrierTicks(cr,0.0,s,con),
                        s = setBarrierPosition(cr,T.UP,s,con)
                    in
                        s
                    end
                else
                    setBarrierTicks(cr,newBTicks,s,con)
                end
            )
        end
    end
pre S.getSBType(cr,con) = T.CROSSINGSB,

tickTrains : T.Tick × S.Configuration × State ~→ State
tickTrains(tick,con,s) ≡
let
    trains = { t | t : T.TrainID}
in
    trainProcess(trains,tick,con,s)
end,

trainProcess : T.TrainID-set × T.Tick × S.Configuration × State ~→ State
trainProcess(trains,tick,con,s) ≡
    if(trains = {})
    then
        s
    else
        let
            t : T.TrainID • t ∈ trains,
            trains = trains \ {t},
            s = if (trainInESA(t,s)) then s else updateTrain(t,tick,con,s) end
        in
            trainProcess(trains,tick,con,s)
        end
    end,

updateTrain : T.TrainID × T.Tick × S.Configuration × State ~→ State
updateTrain(t,tick,con,s) ≡
    let
        dir = getTrainDirection(t,s),
        acc = getTrainAcc(t,s),
        curSpeed = getTrainSpeed(t,s),
        newSpeed = curSpeed + acc∗tick,

        /∗ To avoid negative speed when braking while standing still∗/
        newSpeed = if (newSpeed < 0.0) then 0.0 else newSpeed end,
        s = setTrainSpeed(t,newSpeed,s,con),

        tp = getTrainPosition(t,s),
        deltaProg = curSpeed∗tick + 0.5∗acc∗tick∗tick,
```

```
        deltaProg = if (dir = T.UP) then deltaProg else deltaProg ∗ −1.0 end,

        tp2 = if (deltaProg < 0.0) then tp
                 else updateTrainPosition(tp,deltaProg,s,con) end,
        s = setTrainPosition(t,tp,s,con)
    in
        /∗ updating sensor ∗/
        if (T.oneLoc(tp) ≠ T.oneLoc(tp2))
        then
            if (∼T.oneLoc(tp2))
            then
                let
                    status = T.ACTIVE,
                    sensorPos = tp2
                in
                    updateSensor(sensorPos,dir,status,s,con)
                end
            else
                let
                    status = T.INACTIVE,
                    sensorPos = tp
                in
                    updateSensor(sensorPos,dir,status,s,con)
                end
            end
        else
            s
        end
    end,

updateSensor : T.TrainPosition × T.Direction × T.SensorStatus ×
                                              State × S.Configuration → State
updateSensor(tp,dir,status,s,con) ≡
    let
        /∗ if on two segments sensor must be set active ∗/
        loc = T.frontLoc(tp)
    in
        case loc of
            T.isSeg(seg) →
            (
                let
                    sb = S.getSegSB(seg,T.inverseDir(dir),con)
                in
                    setSensorStatus(sb,status,s,con)
                end
            ),

            T.isESA(esa) →
            (
                let
                    sb = S.getESASB(esa,con)
                in
                    setSensorStatus(sb,status,s,con)
                end
            )
        end
    end,
```

```
updateTrainPosition : T.TrainPosition × T.Length × State ×
                                    S.Configuration → T.TrainPosition
updateTrainPosition(tp,delta,ds,con) ≡
    let
        curFrontPos = T.frontPos(tp),
        curFrontPos = updateSegPos(curFrontPos,delta,ds,con),

        curRearPos = T.rearPos(tp),
        curRearPos = updateSegPos(curRearPos,delta,ds,con)
    in
        T.mk_TrainPosition(curFrontPos, curRearPos)
    end,

updateSegPos : T.SegmentPosition × T.Length × State ×
                                    S.Configuration → T.SegmentPosition
updateSegPos(segPos,delta,ds,con) ≡
    let
        loc = T.getLoc(segPos),
        curProg = T.getLength(segPos) + delta,
        locLength = S.getLocLength(T.getLoc(segPos),con)
    in
        if (curProg < 0.0 )
        then
            let
                nextLoc = nextLoc(loc,T.DOWN,ds,con),
                nextPos = S.getLocLength(nextLoc,con) + curProg
            in
                T.mk_SegmentPosition(nextLoc,nextPos)
            end
        else
            if (curProg > locLength)
            then
                let
                    nextLoc = nextLoc(loc,T.UP,ds,con),
                    nextPos = curProg − locLength
                in
                    T.mk_SegmentPosition(nextLoc,nextPos)
                end
            else
                T.mk_SegmentPosition(loc,curProg)
            end
        end
    end,

nextLoc : T.Location × T.Direction × State × S.Configuration → T.Location
nextLoc(loc,dir,ds,con) ≡
    case loc of
        T.isESA(esa) →
        (
            let
                sb = S.getESASB(esa,con),
                T.seg(aSeg) = S.getSBSeg(sb,dir,con)
            in
                T.isSeg(aSeg)
            end
        ),
```

```
          T.isSeg(aSeg) →
          (
             let
                sb = S.getSegSB(aSeg,dir,con)
             in
                case S.getSBSeg(sb,dir,con) of
                   T.seg(nextSeg) → T.isSeg(nextSeg),
                   T.esa(aESA) → T.isESA(aESA),
                   T.point(up,down) → T.isSeg(getSegOfPoint(sb,ds,con))
                end
             end
          )
       end,

   /∗ Returns the front segment of a train. If front is on ESA then
      the rear segment is returned. This is used for speed checking ∗/
   getTrainLoc : T.TrainID × State → T.Location
   getTrainLoc(t,ds) ≡
      let
         tp = getTrainPosition(t,ds),
         frontLoc = T.getLoc(T.frontPos(tp)),
         rearLoc = T.getLoc(T.rearPos(tp))
      in
         case frontLoc of
            T.isESA(esa) → rearLoc,
            _ → frontLoc
         end
      end
   pre ∼trainInESA(t,ds),

   /∗ Get the point branch according
      to the point position ∗/
   getSegOfPoint : T.SBID × State × S.Configuration ~→ T.SegmentID
   getSegOfPoint(sb,ds,con) ≡
      let
         pp = getPointPosition(sb,ds,con),
         pointSegs = S.getSBPointSegs(sb,con)
      in
         case pp of
            T.UP → T.getUpBranch(pointSegs),
            T.DOWN → T.getDownBranch(pointSegs)
         end
      end
   pre (getPointPosition(sb,ds,con) = T.UP ∨
         getPointPosition(sb,ds,con) = T.DOWN),

tpDerailed : T.TrainPosition × T.Direction × State × S.Configuration → Bool
tpDerailed(tp,dir,s,con) ≡
   if (∼T.oneLoc(tp) ∧ ∼T.segPosIsESA(T.frontPos(tp))) then
      let
         seg = T.getSeg(T.frontLoc(tp)),
         sb = S.getSegSB(seg,T.inverseDir(dir),con)
      in
         case S.getSBType(sb,con) of
               T.POINTSB →
               (
```

```
                if (dir = S.getPointDir(sb,con)) then
                    pointConnected(sb,T.getSeg(T.frontLoc(tp)),s,con)
                else
                    pointConnected(sb,T.getSeg(T.rearLoc(tp)),s,con)
                end
            ),

            T.CROSSINGSB →
            (
                getBarrierPosition(sb,s,con) = T.DOWN
            ),

            _ → false
        end
    end
else
    false
end,

getESATrains : T.ESAID × State → T.TrainID-set
getESATrains(esa,s) ≡
    { t | t : T.TrainID • T.trainOnlyOnESA(getTrainPosition(t,s)) },

getTrainSegments : T.TrainID × State → T.SegmentID-set
getTrainSegments(t,(ts,ss)) ≡
    TD.getTrainSegments(t,ts),

getTrainBranch : T.TrainID × State × S.Configuration ~→ T.SegmentID
getTrainBranch(t,s,con) ≡
(
    let
        seg : T.SegmentID • seg ∈ getTrainSegments(t,s) ∧
                                      S.segIsBranch(seg,con)
    in
        seg
    end
)
pre (∃ sb : T.SBID • trainOnJunction(t,sb,con,s)),

trainOnSegment : T.TrainID × T.SegmentID × S.Configuration × State → Bool
trainOnSegment(tID,seg,con,ds) ≡
    seg ∈ getTrainSegments(tID,ds),

trainOnJunction : T.TrainID × T.SBID × S.Configuration × State → Bool
trainOnJunction(t,sb,con,ds) ≡
(
    S.getSBType(sb,con) = T.POINTSB ∧
    trainOnSensor(t,sb,con,ds)
),

trainOnJunction : T.SBID × S.Configuration × State → Bool
trainOnJunction(sb,con,s) ≡
    S.getSBType(sb,con) = T.POINTSB ∧
    trainOnSensor(sb,con,s),

trainOnSensor : T.TrainID × T.SBID × S.Configuration × State → Bool
trainOnSensor(t,sb,con,ds) ≡
```

```
(
  ∃ dir : T.Direction, tPos : T.TrainPosition,
          sp1,sp2 : T.SegmentPosition •
            tPos = getTrainPosition(t,ds) ∧
    T.segPosInSBSeg(sp1, S.getSBSeg(sb,dir,con)) ∧
    T.segPosInSBSeg(sp2, S.getSBSeg(sb,T.inverseDir(dir),con))
),

trainOnSensor : T.SBID × S.Configuration × State → Bool
trainOnSensor(sb,con,s) ≡
(
  ∃ t : T.TrainID, dir : T.Direction, tPos : T.TrainPosition,
          sp1,sp2 : T.SegmentPosition •
            tPos = getTrainPosition(t,s) ∧
    T.segPosInSBSeg(sp1, S.getSBSeg(sb,dir,con)) ∧
    T.segPosInSBSeg(sp2, S.getSBSeg(sb,T.inverseDir(dir),con))
),

trainInESA : T.TrainID × State → Bool
trainInESA(t,(ts,ss)) ≡
  TD.trainInESA(t,ts),

trainInESADrivingOut : T.TrainID × State ~→ Bool
trainInESADrivingOut(t,(ts,ss)) ≡
  TD.trainInESADrivingOut(t,ts),

trainFrontInESA : T.TrainID × State → Bool
trainFrontInESA(t,(ts,ss)) ≡
  TD.trainFrontInESA(t,ts),

/∗ Telling if a train is (partly) on a single line ∗/
trainOnSingleLine : T.TrainID × S.Configuration × State → Bool
trainOnSingleLine(t,con,s) ≡
  let
    tPos = getTrainPosition(t,s),
    segSet = T.trainPosSegs(tPos)
  in
    if (segSet ≠ {}) then
      (
        ∃ s : T.SegmentID •
          s ∈ segSet ⇒
            ∼S.segIsBranch(s,con)
      )
    else
      false
    end
  end,

/∗ Telling if a train is (partly) on a branch ∗/
trainOnBranch : T.TrainID × S.Configuration × State → Bool
trainOnBranch(t,con,s) ≡
  let
    tPos = getTrainPosition(t,s),
    segSet = T.trainPosSegs(tPos)
  in
    if (segSet ≠ {}) then
      (
```

```
            ∃ s : T.SegmentID •
                s ∈ segSet ⇒
                    S.segIsBranch(s,con)
        )
    else
        false
    end
end,

/∗ Telling if a train is only on a branch ∗/
trainOnlyOnBranch : T.TrainID × S.Configuration × State → Bool
trainOnlyOnBranch(t,con,s) ≡
    let
        tPos = getTrainPosition(t,s),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∀ s : T.SegmentID •
                    s ∈ segSet ⇒
                        S.segIsBranch(s,con)
            )
        else
            false
        end
    end,

pointConnected : T.SBID × T.SegmentID × State × S.Configuration → Bool
pointConnected(sbID,seg,ds,con) ≡
    let
        pointSegs = S.getSBPointSegs(sbID,con)
    in
        case getPointPosition(sbID,ds,con) of
            T.UP → (seg = T.getUpBranch(pointSegs)),
            T.DOWN → (seg = T.getDownBranch(pointSegs)),
            _ → false
        end
    end
pre S.getSBType(sbID,con) = T.POINTSB,

trainFrontLoc : T.TrainID × State → T.Location
trainFrontLoc(t,(ts,ss)) ≡
    TD.trainFrontLoc(t,ts),


sensor_guard : T.SBID × T.SensorStatus × S.Configuration × State → Bool
sensor_guard(sen,ss,con,s) ≡
    (ss = T.ACTIVE ∧ trainOnSensor(sen,con,s)) ∨
    (ss = T.INACTIVE ∧ ∼trainOnSensor(sen,con,s)),

decelerateTrain : T.TrainID × S.Configuration × State → State
decelerateTrain(t,con,s) ≡
    if(getTrainSpeed(t,s) ≠ 0.0)
    then
        let
            maxDec = S.getTrainMaxDec(t,con),
            curDec = getTrainAcc(t,s)
```

```
    in
        if(maxDec ≠ curDec)
        then
            setTrainAcc(t,maxDec,s,con)
        else
            s
        end
    end
else
    setTrainAcc(t,0.0,s,con)
end,

accelerateTrain : T.TrainID × S.Configuration × State ~→ State
accelerateTrain(tID,con,s) ≡
    setTrainAcc(tID,S.getTrainMaxAcc(tID,con),s,con),


commonSegs : T.TrainPosition × T.TrainID × State → T.SegmentID-set
commonSegs(tp1,t2,ds) ≡
    T.trainPosSegs(tp1) ∩ getTrainSegments(t2,ds),

commonSegs : T.TrainID × T.TrainID × State → T.SegmentID-set
commonSegs(t1,t2,ds) ≡
    getTrainSegments(t1,ds) ∩ getTrainSegments(t2,ds),

trainPositionOccupied : T.TrainID × T.TrainPosition × State ×
                                                S.Configuration → Bool
trainPositionOccupied(t1,tp1,ds,con) ≡
(
    ∀ segs : T.SegmentID-set, dir1,dir2 : T.Direction,
            tp1,tp2 : T.TrainPosition •
                ∃ t2 : T.TrainID •
                    t2 ≠ t1 ∧
                    segs = commonSegs(tp1,t2,ds) ∧
                    segs ≠ {} ∧
                    (dir1,dir2) = (getTrainDirection(t1,ds),getTrainDirection(t2,ds)) ∧
                    tp2 = getTrainPosition(t2,ds) ∧

                    case dir1 of
                        T.UP →
                        (
                            if (dir1 = dir2)
                            then
                                S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con) ⇒
                                    ∼S.segPosLower(T.frontPos(tp1),T.rearPos(tp2),con)
                            else
                                ∼S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
                            end
                        ),

                        T.DOWN →
                        (
                            if (dir1 = dir2) then
                                ∼S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con) ⇒
                                    S.segPosLower(T.frontPos(tp1),T.rearPos(tp2),con)
                            else
                                S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
```

```
                        end
                    )
                end
),

/* Invariants etc. */

/* Telling if the railway line is safe */
safe : State × S.Configuration → Bool
safe(s,con) ≡
    is_wf(s,con) ∧
    noCollisions(con,s) ∧
    trainPosPossible(con,s) ∧
    pointsSafe(con,s) ∧
    crossingsSafe(con,s),

/**
*   The position of a train may not overlap
*   with the position of other trains
**/
noCollisions : S.Configuration × State → Bool
noCollisions(con,s) ≡
(
    ∀ t : T.TrainID •
        ∼trainPositionOccupied(t,getTrainPosition(t,s),s,con)
),

/**
*   Trains cannot end up on same segment
*   driving in opposite directions away from each other.
*
*   If two train are on same segment driving in opposite
*   directions then the train driving up must be lower
*   on the line than the train driving down.
**/
trainPosPossible : S.Configuration × State → Bool
trainPosPossible(con,ds) ≡
(
    ∀ t1,t2 : T.TrainID, segs : T.SegmentID-set,
            tp1,tp2 : T.TrainPosition, seg : T.SegmentID •
                commonSegs(t1,t2,ds) ≠ {} ∧
                (tp1,tp2) = (getTrainPosition(t1,ds),getTrainPosition(t1,ds)) ∧
                getTrainDirection(t1,ds) ≠ getTrainDirection(t2,ds) ∧
                getTrainDirection(t1,ds) = T.UP
                    ⇒
                        S.segPosLower(T.frontPos(tp1),T.frontPos(tp2),con)
),

/**
*   If the train is located upon a junction,
*   the point must be connected to the
*   branch, on which the train is located
**/
pointsSafe : S.Configuration × State → Bool
pointsSafe(con,ds) ≡
(
    ∀ sb : T.SBID, t : T.TrainID, seg : T.SegmentID •
```

```
            trainOnJunction(t,sb,con,ds) ∧
            trainOnSegment(t,seg,con,ds) ∧
            S.segIsBranch(seg,con) ⇒
                pointConnected(sb,seg,ds,con)
    ),

    /∗ When a train is located on a crossing
         the barriers must be down ∗/
    crossingsSafe : S.Configuration × State → Bool
    crossingsSafe(con,s) ≡
    (
        ∀ sb : T.SBID •
            S.getSBType(sb,con) = T.CROSSINGSB ∧
            trainOnSensor(sb,con,s) ⇒
                getBarrierPosition(sb,s,con) = T.DOWN
    ),

    /∗ Wellformedness ∗/
    is_wf : State × S.Configuration → Bool
    is_wf((ts,ss),con) ≡
        TD.is_wf(ts,con) ∧
        SD.is_wf(ss,con),

    init_req : State × S.Configuration → Bool
    init_req((ts,ss),con) ≡
        is_wf((ts,ss),con) ∧
        TD.init_req(ts) ∧
        SD.init_req(ss,con)


  axiom
    [ wellformedness ]
        init_req(initState,S.conf)

end
```

TrainDyn

```
context: CA_Types0, CA_Statics0
scheme CA_TrainDyn0(T : CA_Types0, S : CA_Statics0(T)) =
  class
    type
      TrainStates = T.TrainID →m TrainState,

      TrainState == mk_tState(getTAcc : T.Acceleration ↔ setTAcc,
                              getTSpeed : T.Speed ↔ setTSpeed,
                              getTPos : T.TrainPosition ↔ setTPos,
                              getTDir : T.Direction ↔ setTDir)

    value
      initTrainStates : TrainStates,

      /∗ Train observer ∗/
      getTrainAcc : T.TrainID × TrainStates ~→ T.Acceleration
      getTrainAcc(tID,ts) ≡
```

```
    getTAcc(ts(tID))
pre trainStateExists(tID,ts),

getTrainSpeed : T.TrainID × TrainStates ~→ T.Speed
getTrainSpeed(tID,ts) ≡
    getTSpeed(ts(tID))
pre trainStateExists(tID,ts),

getTrainPosition : T.TrainID × TrainStates ~→
                                        T.TrainPosition
getTrainPosition(tID,ts) ≡
    getTPos(ts(tID))
pre trainStateExists(tID,ts),

getTrainDirection : T.TrainID × TrainStates ~→ T.Direction
getTrainDirection (tID,ts) ≡
    getTDir(ts(tID))
pre trainStateExists(tID,ts),

/* Train generator */
setTrainAcc : T.TrainID × T.Acceleration × TrainStates ×
                                S.Configuration ~→ TrainStates
setTrainAcc(tID,acc,ts,con) ≡
    ts † [ tID ↦ setTAcc(acc,ts(tID)) ]
pre acc ≤ S.getTrainMaxAcc(tID,con) ∧
        S.getTrainMaxDec(tID,con) ≤ acc ∧
        trainStateExists(tID,ts),

setTrainSpeed : T.TrainID × T.Speed × TrainStates ×
                                S.Configuration ~→ TrainStates
setTrainSpeed(tID,speed,ts,con) ≡
    ts † [ tID ↦ setTSpeed(speed,ts(tID)) ]
pre speed ≤ S.getTrainMaxSpeed(tID,con) ∧
        trainStateExists(tID,ts),

setTrainPosition : T.TrainID × T.TrainPosition ×
                TrainStates × S.Configuration ~→ TrainStates
setTrainPosition(tID,tPos,ts,con) ≡
    ts † [ tID ↦ setTPos(tPos,ts(tID)) ]
pre train_pos_ok(tID,tPos,ts,con) ∧
        trainStateExists(tID,ts),

setTrainDirection : T.TrainID × T.Direction ×
                                TrainStates ~→ TrainStates
setTrainDirection(tID,dir,ts) ≡
    ts † [ tID ↦ setTDir(dir,ts(tID)) ]
pre (getTrainSpeed(tID,ts) = 0.0 ∨
        getTrainDirection(tID,ts) = dir) ∧
        trainStateExists(tID,ts),

/* Tells if a train has a state in the system */
trainStateExists : T.TrainID × TrainStates → Bool
trainStateExists(t,ts) ≡
    t ∈ dom(ts),

/* Front and rear position of a train must be exactly
```

```
     'train length' apart */
train_pos_ok : T.TrainID × T.TrainPosition ×
                              TrainStates × S.Configuration ~→ Bool
train_pos_ok(t,tp,s,con) ≡
(
    let
        T.mk_TrainPosition(posFront,posRear) = tp
    in
        (S.distance(posFront,posRear,con) =
                              S.getTrainLength(t,con)) ∧
        train_pos_dir_ok(getTrainDirection(t,s),tp,s,con)
    end
),

/* If train drives UP then rear pos
     must be lower than front pos
     and vice versa */
train_pos_dir_ok : T.Direction × T.TrainPosition ×
                              TrainStates × S.Configuration → Bool
train_pos_dir_ok(dir,tp,s,con) ≡
(
    case dir of
        T.UP →
        (
            S.segPosLower(T.rearPos(tp),T.frontPos(tp),con)
        ),

        T.DOWN →
        (
            S.segPosLower(T.frontPos(tp),T.rearPos(tp),con)
        )
    end
),

getTrainSegments : T.TrainID × TrainStates ~→
                                        T.SegmentID-set
getTrainSegments(t,s) ≡
    T.trainPosSegs(getTrainPosition(t,s)),

trainInESA : T.TrainID × TrainStates ~→ Bool
trainInESA(t,s) ≡
    T.trainOnlyOnESA(getTrainPosition(t,s)),

trainInESADrivingOut : T.TrainID × TrainStates ~→ Bool
trainInESADrivingOut(t,s) ≡
    if(∼T.trainOnlyOnESA(getTrainPosition(t,s)))
    then
        false
    else
        let
            segPos = T.frontPos(getTrainPosition(t,s)),
            esa = T.getESA(T.getLoc(segPos)),
            dir = getTrainDirection(t,s)
        in
            T.end2Dir(esa) ≠ dir
        end
    end,
```

```
trainFrontInESA : T.TrainID × TrainStates ~→ Bool
trainFrontInESA(t,s) ≡
    let
        tPos = getTrainPosition(t,s)
    in
        T.segPosIsESA(T.frontPos(tPos))
    end,

trainFrontLoc : T.TrainID × TrainStates ~→ T.Location
trainFrontLoc(t,ds) ≡
    case T.frontLoc(getTrainPosition(t,ds)) of
        T.isESA(_) → T.rearLoc(getTrainPosition(t,ds)),
        T.isSeg(seg) → T.isSeg(seg)
    end,

is_wf : TrainStates × S.Configuration ~→ Bool
is_wf(s,con) ≡
    allTrainStatesExist(s) ∧
    train_pos_wf(con,s),

/* All trains must have a state */
allTrainStatesExist : TrainStates → Bool
allTrainStatesExist(s) ≡
(
    ∀ trainID : T.TrainID •
        trainStateExists(trainID,s)
),

/* Front and rear position of a train must be exactly
     'train length' apart */
train_pos_wf : S.Configuration × TrainStates ~→ Bool
train_pos_wf(con,s) ≡
(
    ∀ t : T.TrainID •
        train_pos_ok(t,getTrainPosition(t,s),s,con)
),

init_req : TrainStates ~→ Bool
init_req(s) ≡
    allTrainsInESA(s) ∧
    allTrainsStopped(s) ∧
    allTrainsFacingLine(s),

allTrainsInESA : TrainStates ~→ Bool
allTrainsInESA(s) ≡
(
    ∀ t : T.TrainID •
        trainInESA(t,s)
),

allTrainsStopped : TrainStates ~→ Bool
allTrainsStopped(s) ≡
(
    ∀ t : T.TrainID •
        getTrainSpeed(t,s) = 0.0 ∧
```

```
            getTrainAcc(t,s) = 0.0
        ),

        allTrainsFacingLine : TrainStates ~→ Bool
        allTrainsFacingLine(s) ≡
        (
            ∀ t : T.TrainID, esa : T.ESAID •
                T.isESA(esa) = T.getLoc(T.frontPos(
                                        getTrainPosition(t,s))) ∧
                (esa = T.LOW ⇒ getTrainDirection(t,s) = T.UP) ∧
                (esa = T.HIGH ⇒ getTrainDirection(t,s) = T.DOWN)
        )

    end
```

SBDyn

```
context: CA_Types0, CA_Statics0
scheme CA_SBDyn0(T : CA_Types0, S : CA_Statics0(T)) =
    class
        type
            SBStates = T.SBID →m SBState,

            SBState == mk_sbState(
                            getPP : T.HasPointPosition ↔ setPP,
                            getPTicks : T.HasTicks ↔ setPTicks,
                            getBP : T.HasBarrierPosition ↔ setBP,
                            getSignal : T.HasSignalStatus ↔ setSignal,
                            getBTicks : T.HasTicks ↔ setBTicks,
                            getSTicks : T.HasTicks ↔ setSTicks,
                            getSensor : T.SensorStatus ↔ setSensor)

        value
            initSBStates : SBStates,

            /* Point observer */
            getPointPosition : T.SBID × SBStates × S.Configuration ~→
                                                    T.PointPosition
            getPointPosition(p,sbs,con) ≡
                T.getPos(getPP(sbs(p)))
            pre S.getSBType(p,con) = T.POINTSB ∧
                    pointStateExists(p,sbs,con),

            getPointTicks : T.SBID × SBStates × S.Configuration ~→
                                                    T.Tick
            getPointTicks(p,sbs,con) ≡
                T.getTicks(getPTicks(sbs(p)))
            pre S.getSBType(p,con) = T.POINTSB ∧
                    pointStateExists(p,sbs,con),

            /* Point generator */
            setPointPosition : T.SBID × T.PointPosition × SBStates ×
                                        S.Configuration ~→ SBStates
            setPointPosition(p,pp,sbs,con) ≡
                sbs † [ p ↦ setPP(T.pointPos(pp),sbs(p)) ]
```

```
pre S.getSBType(p,con) = T.POINTSB ∧
      pointStateExists(p,sbs,con),

setPointTicks : T.SBID × T.Tick × SBStates ×
                                    S.Configuration ~→ SBStates
setPointTicks(p,tick,sbs,con) ≡
   sbs † [ p ↦ setPTicks(T.ticks(tick),sbs(p)) ]
pre S.getSBType(p,con) = T.POINTSB ∧
      pointStateExists(p,sbs,con),

/∗ Crossing observer ∗/
getBarrierPosition : T.SBID × SBStates ×
                              S.Configuration ~→ T.BarrierPosition
getBarrierPosition(cr,sbs,con) ≡
   T.getPos(getBP(sbs(cr)))
pre S.getSBType(cr,con) = T.CROSSINGSB ∧
      crossingStateExists(cr,sbs,con),

getSignalStatus : T.SBID × SBStates ×
                               S.Configuration ~→ T.SignalStatus
getSignalStatus(cr,sbs,con) ≡
   T.getStatus(getSignal(sbs(cr)))
pre S.getSBType(cr,con) = T.CROSSINGSB ∧
      crossingStateExists(cr,sbs,con),

getBarrierTicks : T.SBID × SBStates ×
                                     S.Configuration ~→ T.Tick
getBarrierTicks(cr,sbs,con) ≡
   T.getTicks(getBTicks(sbs(cr)))
pre S.getSBType(cr,con) = T.CROSSINGSB ∧
      crossingStateExists(cr,sbs,con),

getSignalTicks : T.SBID × SBStates ×
                                     S.Configuration ~→ T.Tick
getSignalTicks(cr,sbs,con) ≡
   T.getTicks(getSTicks(sbs(cr)))
pre S.getSBType(cr,con) = T.CROSSINGSB ∧
      crossingStateExists(cr,sbs,con),

/∗ Crossing generator ∗/
setBarrierPosition : T.SBID × T.BarrierPosition ×
                          SBStates × S.Configuration ~→ SBStates
setBarrierPosition(cr,bp,sbs,con) ≡
   sbs † [ cr ↦ setBP(T.barrierPos(bp),sbs(cr)) ]
pre S.getSBType(cr,con) = T.CROSSINGSB ∧
      crossingStateExists(cr,sbs,con),

setSignalStatus : T.SBID × T.SignalStatus × SBStates ×
                                    S.Configuration ~→ SBStates
setSignalStatus(cr,ss,sbs,con) ≡
   sbs † [ cr ↦ setSignal(T.signalStatus(ss),sbs(cr)) ]
pre S.getSBType(cr,con) = T.CROSSINGSB ∧
      crossingStateExists(cr,sbs,con),

setBarrierTicks : T.SBID × T.Tick × SBStates ×
                                    S.Configuration ~→ SBStates
```

```
setBarrierTicks(cr,tick,sbs,con) ≡
    sbs † [ cr ↦ setBTicks(T.ticks(tick),sbs(cr)) ]
pre S.getSBType(cr,con) = T.CROSSINGSB ∧
        crossingStateExists(cr,sbs,con),

setSignalTicks : T.SBID × T.Tick × SBStates ×
                                        S.Configuration ~→ SBStates
setSignalTicks(cr,tick,sbs,con) ≡
    sbs † [ cr ↦ setSTicks(T.ticks(tick),sbs(cr)) ]
pre S.getSBType(cr,con) = T.CROSSINGSB ∧
        crossingStateExists(cr,sbs,con),

/∗ Sensor observer ∗/
getSensorStatus : T.SBID × SBStates ~→ T.SensorStatus
getSensorStatus(sen,sbs) ≡
    getSensor(sbs(sen))
pre sensorStateExists(sen,sbs),

/∗ Sensor generator ∗/
setSensorStatus : T.SBID × T.SensorStatus ×
                                        SBStates ~→ SBStates
setSensorStatus(sen,ss,sbs) ≡
    sbs † [ sen ↦ setSensor(ss,sbs(sen)) ]
pre sensorStateExists(sen,sbs),

/∗ Tells if a sensor has a state in the system ∗/
sensorStateExists : T.SBID × SBStates → Bool
sensorStateExists(sb,s) ≡
    sb ∈ dom(s),

/∗ Tells if a crossing has a state in the system ∗/
crossingStateExists : T.SBID × SBStates ×
                                        S.Configuration ~→ Bool
crossingStateExists(sb,s,con) ≡
    let state = s(sb) in
        getBP(state) ≠ T.none ∧
        getBTicks(state) ≠ T.none ∧
        getSignal(state) ≠ T.none ∧
        getSTicks(state) ≠ T.none
    end
pre sensorStateExists(sb,s) ∧
        S.getSBType(sb,con) = T.CROSSINGSB,

/∗ Tells if a point has a state in the system ∗/
pointStateExists : T.SBID × SBStates ×
                                        S.Configuration ~→ Bool
pointStateExists(sb,s,con) ≡
    let state = s(sb) in
        getPP(state) ≠ T.none ∧
        getPTicks(state) ≠ T.none
    end
pre sensorStateExists(sb,s) ∧
        S.getSBType(sb,con) = T.POINTSB,

/∗ Invariants ∗/
```

```
    is_wf : SBStates × S.Configuration ~→ Bool
    is_wf(s,con) ≡
        allCrossingStatesExist(con,s) ∧
        allPointStatesExist(con,s) ∧
        allSensorStatesExist(s),

    /∗ All crossings must have a state ∗/
    allCrossingStatesExist : S.Configuration ×
                                                    SBStates ~→ Bool
    allCrossingStatesExist(con,s) ≡
    (
        ∀ cr : T.SBID •
            S.getSBType(cr,con) = T.CROSSINGSB ⇒
                crossingStateExists(cr,s,con)
    ),

    /∗ All points must have a state ∗/
    allPointStatesExist : S.Configuration × SBStates ~→ Bool
    allPointStatesExist(con,s) ≡
    (
        ∀ p : T.SBID •
            S.getSBType(p,con) = T.POINTSB ⇒
                pointStateExists(p,s,con)
    ),

    /∗ All sensors must have a state ∗/
    allSensorStatesExist : SBStates → Bool
    allSensorStatesExist(s) ≡
    (
        ∀ sen : T.SBID •
            sensorStateExists(sen,s)
    ),

    init_req : SBStates × S.Configuration ~→ Bool
    init_req(s,con) ≡
        allBarriersUp(con,s) ∧
        allPointsNotShifting(con,s),

    allBarriersUp : S.Configuration × SBStates ~→ Bool
    allBarriersUp(con,s) ≡
    (
        ∀ sb : T.SBID •
            S.getSBType(sb,con) = T.CROSSINGSB ⇒
                getBarrierPosition(sb,s,con) = T.UP
    ),

    allPointsNotShifting : S.Configuration × SBStates ~→ Bool
    allPointsNotShifting(con,s) ≡
    (
        ∀ sb : T.SBID •
            S.getSBType(sb,con) = T.POINTSB ⇒
                getPointPosition(sb,s,con) ∈ { T.UP, T.DOWN }
    )

end
```

F.3.4 Control

```
context: CA_Dynamics0, CA_ComService0, CA_SBCC0, CA_TCC0
scheme CA_Control0(T : CA_Types0, S : CA_Statics0(T),
                                        D : CA_Dynamics0(T,S)) =
  class
    object
      COM : CA_ComService0(T),
      SBCC : CA_SBCC0(T,S,D,COM),
      TCC : CA_TCC0(T,S,D,COM)

    type
      ControlState = SBCCStates × TCCStates,

      SBCCStates = T.SBID →m SBCC.SBCCState,
      TCCStates = T.SBID →m TCC.TCCState

    value
      initControlState : ControlState,

      updateSBCCState : T.SBID × SBCC.SBCCState ×
                                        ControlState ~→ ControlState
      updateSBCCState(sb,sbcc,(sbccs,tccs)) ≡
        (sbccs † [sb ↦ sbcc],tccs)
      pre sbccStateExists(sb,(sbccs,tccs)),

      getSBCCState : T.SBID × ControlState ~→ SBCC.SBCCState
      getSBCCState(sb,(sbccs,tccs)) ≡
        sbccs(sb)
      pre sbccStateExists(sb,(sbccs,tccs)),

      updateTCCState : T.TrainID × TCC.TCCState ×
                                        ControlState ~→ ControlState
      updateTCCState(t,tcc,(sbccs,tccs)) ≡
        (sbccs,tccs † [t ↦ tcc])
      pre tccStateExists(t,(sbccs,tccs)),

      getTCCState : T.TrainID × ControlState ~→ TCC.TCCState
      getTCCState(t,(sbccs,tccs)) ≡
        tccs(t)
      pre tccStateExists(t,(sbccs,tccs)),

      sbccStateExists : T.SBID × ControlState → Bool
      sbccStateExists(sb,(sbccs,tccs)) ≡
        sb ∈ dom sbccs,

      tccStateExists : T.TrainID × ControlState → Bool
      tccStateExists(t,(sbccs,tccs)) ≡
        t ∈ dom tccs,

      /* Processes */
      tick : T.Tick × ControlState × D.State ×
                                S.Configuration ~→ out COM.comChannel
                                        (ControlState × D.State)
      tick(tick,cs,ds,con) ≡
      (
```

```
    let
        tSet = T.trainIDSet,
        sbSet = T.sbIDSet,
        (cs,ds) = tickTCCs(tSet,tick,cs,ds,con),
        cs = tickSBCCs(sbSet,tick,cs,ds,con)
    in
        (cs,ds)
    end
),

tickSBCCs : T.SBID-set × T.Tick × ControlState × D.State ×
                                S.Configuration ~→ out COM.comChannel
                                                ControlState
tickSBCCs(sbSet,tick,cs,ds,con) ≡
    if (sbSet = {}) then
        cs
    else
        let
            sbcc : T.SBID • sbcc ∈ sbSet,
            sbSet = sbSet \ {sbcc},
            sbccState = getSBCCState(sbcc,cs),
            sbccState = SBCC.sbccProcess(sbcc,tick,sbccState,ds,con),
            cs = updateSBCCState(sbcc,sbccState,cs)
        in
            tickSBCCs(sbSet,tick,cs,ds,con)
        end
    end,

tickTCCs : T.TrainID-set × T.Tick × ControlState × D.State ×
                                S.Configuration ~→ out COM.comChannel
                                        ControlState × D.State
tickTCCs(tccSet,tick,cs,ds,con) ≡
    if (tccSet = {}) then
        (cs,ds)
    else
        let
            tcc : T.TrainID • tcc ∈ tccSet,
            tccSet = tccSet \ {tcc},
            tccState = getTCCState(tcc,cs),
            tccState = TCC.tccProcess(tcc,tick,tccState,ds,con),
            cs = updateTCCState(tcc,tccState,cs)
        in
            tickTCCs(tccSet,tick,cs,ds,con)
        end
    end,

/∗ Communication ∗/
comService : ControlState ~→ in COM.comChannel ControlState
comService(cs) ≡
    let
        comMsg = COM.getMsg()
    in
        case T.getReceiver(comMsg) of
            T.isSB(sb) →
            (
                let
                    sbcc = getSBCCState(sb,cs),
```

```
                    sbcc = SBCC.msgReceiver(comMsg, sbcc)
                in
                    updateSBCCState(sb,sbcc,cs)
                end
            ),
            T.isTrain(t) →
            (
                let
                    tcc = getTCCState(t,cs),
                    tcc = TCC.tccMsgReceiver(comMsg,tcc)
                in
                    updateTCCState(t,tcc,cs)
                end
            )
        end
    end,

/* Invariants */
is_wf : ControlState × D.State × S.Configuration → Bool
is_wf(cs,ds,con) ≡
    D.is_wf(ds,con) ∧
    tcc_has_state(cs) ∧
    sbcc_has_state(cs),

tcc_has_state : ControlState → Bool
tcc_has_state(cs) ≡
(
    ∀ t : T.TrainID •
        tccStateExists(t,cs)
),

sbcc_has_state : ControlState → Bool
sbcc_has_state(cs) ≡
(
    ∀ sb : T.SBID •
        sbccStateExists(sb,cs)
),

/**
* Defines that the control system and all its components
* must be consistent e.g. the information stored in the
* control system must reflect the physical world and
* unintended states must not occur.
*
* Also the physical world must abide
* by the rules of the control system.
**/
consistent : ControlState × D.State × S.Configuration → Bool
consistent(cs,ds,con) ≡
    is_wf(cs,ds,con) ∧
    train_on_branch_dir(ds,con) ∧
    tcc_hasRes_passedResPoint(cs,ds,con) ∧
    sbcc_res_wf(cs,con) ∧
    position_branch_sbcc_res_wf(cs,ds,con) ∧
    tcc_res_branch_wf(cs,ds,con) ∧
    position_sl_sbcc_res_wf(cs,ds,con),
```

```
/* When a train is on a branch segment it is consitent
     with the driving direction of the train */
train_on_branch_dir : D.State × S.Configuration → Bool
train_on_branch_dir(ds,con) ≡
(
    ∀ t : T.TrainID, seg : T.SegmentID •
        D.trainOnBranch(t,con,ds) ∧
        D.trainOnSegment(t,seg,con,ds) ∧
        S.segIsBranch(seg,con) ⇒
            S.branchDir(seg,con) = D.getTrainDirection(t,ds)
),

/* If a train has a reservation then it has passed
     the reservation point on the given segment */
tcc_hasRes_passedResPoint : ControlState × D.State ×
                                                        S.Configuration → Bool
tcc_hasRes_passedResPoint(cs,ds,con) ≡
(
    ∀ t : T.TrainID,
            tccState : TCC.TCCState •
                tccState = getTCCState(t,cs) ∧
                TCC.hasTCCRes(tccState) ⇒
                    TCC.hasPassedResPoint(t,ds,con)
),

/* Only POINTSB and ENDSB may have line reservations
     Only POINTSB may have branch reservations */
sbcc_res_wf : ControlState × S.Configuration → Bool
sbcc_res_wf(cs,con) ≡
(
    ∀ sb : T.SBID,
            sbcc : SBCC.SBCCState,
            lineRes, branchRes : T.HasRes •
        (
            S.getSBType(sb,con) ∈ {T.PLAINSB, T.CROSSINGSB} ∧
            sbcc = getSBCCState(sb,cs) ∧
            lineRes = SBCC.getLineRes(sbcc) ∧
            branchRes = SBCC.getBranchRes(sbcc)
                ⇒
                    {lineRes} ∪ {branchRes} = {T.noRes}
        )
            ∧
        (
            S.getSBType(sb,con) = T.ENDSB ∧
            sbcc = getSBCCState(sb,cs)
                ⇒
                    SBCC.getBranchRes(sbcc) = T.noRes
        )
),

/* When a train is on a branch segment it must have
     a branch reservation in the SB behind */
position_branch_sbcc_res_wf : ControlState × D.State ×
                                                        S.Configuration → Bool
position_branch_sbcc_res_wf(cs,ds,con) ≡
(
```

```
    ∀ t : T.TrainID,
          sb : T.SBID,
          tDir : T.Direction,
          seg : T.SegmentID,
          sbcc : SBCC.SBCCState •
             tDir = D.getTrainDirection(t,ds) ∧
             D.trainOnSegment(t,seg,con,ds) ∧
             D.trainOnBranch(t,con,ds) ∧
             S.segIsBranch(seg,con) ∧
             sb = S.getSegSB(seg,T.inverseDir(tDir),con) ∧
             sbcc = getSBCCState(sb,cs)
                ⇒
                   SBCC.getBranchRes(sbcc) = T.res(T.mk_res(t,tDir))
 ),

/* If a train is (only) on a branch and has reservation
   then the SB in front of it and the other guard has a
   reservation for that train in that direction */
 tcc_res_branch_wf : ControlState × D.State ×
                                          S.Configuration → Bool
 tcc_res_branch_wf(cs,ds,con) ≡
 (
    ∀ t : T.TrainID,
          seg : T.SegmentID,
          trainDir : T.Direction,
          guard1,guard2 : T.SBID,
          sbcc1,sbcc2 : SBCC.SBCCState,
          res : T.Reservation •
             D.trainOnSegment(t,seg,con,ds) ∧
             D.trainOnlyOnBranch(t,con,ds) ∧
             TCC.hasTCCRes(getTCCState(t,cs)) ∧
             trainDir = D.getTrainDirection(t,ds) ∧
             guard1 = S.getSegSB(seg,T.inverseDir(trainDir),con) ∧
             guard2 = S.getSingleLineGuard(guard1,trainDir,con) ∧
             sbcc1 = getSBCCState(guard1,cs) ∧
             sbcc2 = getSBCCState(guard2,cs) ∧
             res = T.mk_res(t,trainDir)
                ⇒
                   T.res(res) = SBCC.getLineRes(sbcc1) ∧
                   T.res(res) = SBCC.getLineRes(sbcc2) ∧
                   T.res(res) = SBCC.getBranchRes(sbcc2)
 ),

 /* When a train is on a single line it must have a
 reservation in both guards with the appropriate direction
  + a branch reservation if driving to a point */
 position_sl_sbcc_res_wf : ControlState × D.State ×
                                          S.Configuration → Bool
 position_sl_sbcc_res_wf(cs,ds,con) ≡
 (
    ∀ t : T.TrainID,
          seg : T.SegmentID,
          sb1,sb2 : T.SBID,
          sbcc1,sbcc2 : SBCC.SBCCState,
          dir : T.Direction,
          res : T.Reservation •
             dir = D.getTrainDirection(t,ds) ∧
```

```
                D.trainOnSegment(t,seg,con,ds) ∧
                S.segIsLineSegment(seg,con) ∧
                sb1 = S.getSingleLineGuard(seg,T.inverseDir(dir),con) ∧
                sb2 = S.getSingleLineGuard(seg,dir,con) ∧
                sbcc1 = getSBCCState(sb1,cs) ∧
                sbcc2 = getSBCCState(sb2,cs) ∧
                res = T.mk_res(t,dir) ⇒
                    T.res(res) = SBCC.getLineRes(sbcc1) ∧
                    T.res(res) = SBCC.getLineRes(sbcc2) ∧
                    (S.getSBType(sb2,con) = T.POINTSB ⇒
                              T.res(res) = SBCC.getBranchRes(sbcc2))
),

initReq : ControlState × D.State × S.Configuration → Bool
initReq(cs,ds,con) ≡
    is_wf(cs,ds,con) ∧
    is_tcc_init(cs) ∧
    is_sbcc_init(cs) ∧
    all_tcc_initReq(cs) ∧
    all_sbcc_initReq(cs),

/∗ Requires that the init constants in TCC
     scheme is used for initialization ∗/
is_tcc_init : ControlState → Bool
is_tcc_init((sbccs,tccs)) ≡
(
    ∀ tcc : TCC.TCCState •
        tcc ∈ rng(tccs) ⇒ tcc = TCC.initTCCState
),

/∗ Requires that the init constants in SBCC
     scheme is used for initialization ∗/
is_sbcc_init : ControlState → Bool
is_sbcc_init((sbccs,tccs)) ≡
(
    ∀ sbcc : SBCC.SBCCState •
        sbcc ∈ rng(sbccs) ⇒ sbcc = SBCC.initSBCCState
),

all_tcc_initReq : ControlState → Bool
all_tcc_initReq(cs) ≡
(
    ∀ t : T.TrainID, tcc : TCC.TCCState •
        tcc = getTCCState(t,cs) ⇒
            TCC.initReq(tcc)
),

all_sbcc_initReq : ControlState → Bool
all_sbcc_initReq(cs) ≡
(
    ∀ sb : T.SBID, sbcc : SBCC.SBCCState •
        sbcc = getSBCCState(sb,cs) ⇒
            SBCC.initReq(sbcc)
)

axiom
/∗ The initial state has to be wellformed and
```

```
          fullfill the initial state requirements */
      [initial_state]
         initReq(initControlState,D.initState,S.conf)

  end
```

TCC

```
context: CA_Types0,
CA_Statics0,
CA_Dynamics0,
CA_ComService0

scheme CA_TCC0(T : CA_Types0, S : CA_Statics0(T),
               D : CA_Dynamics0(T,S), COM : CA_ComService0(T)) =
  class
    type
      TCCState ::
            hasTCCRes : Bool ↔ setTCCRes
            isTCCRequesting : Bool ↔ setTCCRequesting
            isTrainDecelerating : Bool ↔ setTrainDecelerating


    value
      initTCCState : TCCState,

      hasPassedResPoint : T.TrainID × D.State ×
                                        S.Configuration → Bool
      hasPassedResPoint(t,ds,con) ≡
        let
          front = T.frontPos(D.getTrainPosition(t,ds))
        in
          case T.getLoc(front) of
            T.isESA(esa) →
            (
              let
                resPoint = S.getResPoint(con),
                posFront = T.getLength(front),
                esaLength = S.getESALength(esa,con),
                dir = D.getTrainDirection(t,ds)
              in
                passedPoint(posFront,resPoint,esaLength,dir)
              end
            ),

            T.isSeg(seg) →
            (
              let
                resPoint = S.getResPoint(con),
                posFront = T.getLength(front),
                segLength = S.getSegLength(seg,con),
                dir = D.getTrainDirection(t,ds)
              in
                passedPoint(posFront,resPoint,segLength,dir)
              end
```

```
        )
      end
    end,

  hasPassedBrakePoint : T.TrainID × D.State ×
                                          S.Configuration → Bool
  hasPassedBrakePoint(t,ds,con) ≡
    let
      front = T.frontPos(D.getTrainPosition(t,ds))
    in
      case T.getLoc(front) of
        T.isESA(esa) →
        (
          let
            brkPoint = S.getBrakePoint(con),
            posFront = T.getLength(front),
            esaLength = S.getESALength(esa,con),
            dir = D.getTrainDirection(t,ds)
          in
            passedPoint(posFront,brkPoint,esaLength,dir)
          end
        ),

        T.isSeg(seg) →
        (
          let
            brkPoint = S.getBrakePoint(con),
            posFront = T.getLength(front),
            segLength = S.getSegLength(seg,con),
            dir = D.getTrainDirection(t,ds)
          in
            passedPoint(posFront,brkPoint,segLength,dir)
          end
        )
      end
    end,

  passedPoint : T.Length × T.Length × T.Length ×
                                          T.Direction → Bool
  passedPoint(posFront,passPoint,segLength,dir) ≡
    case dir of
      T.UP → posFront > (segLength − passPoint),
      T.DOWN → posFront < passPoint
    end,

  /∗ Processes ∗/
  tccMsgReceiver : T.ComMsg × TCCState → TCCState
  tccMsgReceiver(comMsg,tcc) ≡
    let
      resp = T.getMsg(comMsg)
    in
      case resp of
        T.segResp(resGranted) →
        (
          let
            tcc = setTCCRequesting(false,tcc)
          in
```

```
                    if(resGranted)
                    then
                        setTCCRes(true,tcc)
                    else
                        tcc
                    end
                end
            ),
            _ → tcc
        end
    end,

tccProcess : T.TrainID × T.Tick × TCCState × D.State ×
                                S.Configuration ~→ out COM.comChannel
                                                TCCState × D.State
tccProcess(t,tick,cs,ds,con) ≡
    let
        (cs,ds) = checkSpeed(t,tick,cs,ds,con),
        cs = clearRes(t,cs,ds),
        (cs,ds) = handleRes(t,cs,ds,con)
    in
        (cs,ds)
    end,

checkSpeed : T.TrainID × T.Tick × TCCState × D.State ×
                                S.Configuration ~→ TCCState × D.State
checkSpeed(t,tick,cs,ds,con) ≡
    let
        dts = S.getTrainMaxAcc(t,con) ∗ tick,
        ts = D.getTrainSpeed(t,ds) + dts,
        trainMaxSpeed = S.getTrainMaxSpeed(t,con)
    in
        /∗ If train is entirely in an ESA ∗/
        if(D.trainInESA(t,ds))
        then
            if(ts > trainMaxSpeed)
            then
                decelerateTrain(t,cs,ds,con)
            else
                checkDeceleration(t,cs,ds,con)
            end
        else /∗ Train on segment and perhaps an ESA ∗/
            let
                tp = D.getTrainPosition(t,ds),

                /∗ Will always be an IsSeg ∗/
                isSeg = D.getTrainLoc(t,ds),

                frontSeg = T.getSeg(isSeg),
                segMaxSpeed = S.getSegMaxSpeed(frontSeg,con)
            in
                if (ts > segMaxSpeed ∨ ts > trainMaxSpeed)
                then
                    decelerateTrain(t,cs,ds,con)
                else
                    checkDeceleration(t,cs,ds,con)
                end
```

```
            end
        end
    end, /* let */

checkDeceleration : T.TrainID × TCCState × D.State ×
                              S.Configuration ~→ TCCState × D.State
checkDeceleration(t,cs,ds,con) ≡
    if (isTrainDecelerating(cs))
    then
        let
            ds = D.setTrainAcc(t,0.0,ds,con),
            cs = setTrainDecelerating(false,cs)
        in
            (cs,ds)
        end
    else
        (cs,ds)
    end,

decelerateTrain : T.TrainID × TCCState × D.State ×
                              S.Configuration → TCCState × D.State
decelerateTrain(t,tcc,ds,con) ≡
    let
        tcc = setTrainDecelerating(true,tcc),
        ds = D.decelerateTrain(t,con,ds)
    in
        (tcc,ds)
    end,

accelerateTrain : T.TrainID × TCCState × D.State ×
                              S.Configuration → TCCState × D.State
accelerateTrain(t,tcc,ds,con) ≡
    let
        tcc = setTrainDecelerating(false,tcc),
        ds = D.accelerateTrain(t,con,ds)
    in
        (tcc,ds)
    end,

clearRes : T.TrainID × TCCState × D.State ~→ TCCState
clearRes(t,cs,ds) ≡
    if(∼T.oneLoc(D.getTrainPosition(t,ds)))
    then
        setTCCRes(false,cs)
    else
        cs
    end,

handleRes : T.TrainID × TCCState × D.State ×
                              S.Configuration → out COM.comChannel
                                        TCCState × D.State
handleRes(t,cs,ds,con) ≡
    if(hasPassedResPoint(t,ds,con))
    then
        if(∼hasTCCRes(cs))
        then
            (cs,ds)
```

```
        else
            let
                (cs,ds) = if(hasPassedBrakePoint(t,ds,con))
                                then decelerateTrain(t,cs,ds,con)
                                else (cs,ds) end
            in
                if(∼isTCCRequesting(cs))
                then
                    tccRequestRes(t,cs,ds,con)
                else
                    (cs,ds)
                end
            end
        end
    else
        (cs,ds)
    end,

tccRequestRes : T.TrainID × TCCState × D.State ×
                        S.Configuration ~→ out COM.comChannel
                                    TCCState × D.State
tccRequestRes(t,cs,ds,con) ≡
    let
        loc = T.frontLoc(D.getTrainPosition(t,ds)),
        dir = D.getTrainDirection(t,ds)
    in
        case loc of
            T.isESA(esa) →
            (
                /∗ If not facing line, then do nothing.
                   Will brake at brakepoint ∗/
                if (dir = T.end2Dir(esa))
                then
                    (cs,ds)
                else
                    (sendTCCReq(t,S.getESASB(esa,con),dir,cs),ds)
                end
            ),

            T.isSeg(seg) →
            (
                (sendTCCReq(t,S.getSegSB(seg,dir,con),dir,cs),ds)
            )
        end /∗ case ∗/
    end, /∗ let ∗/

sendTCCReq : T.TrainID × T.SBID × T.Direction ×
                        TCCState ~→ out COM.comChannel
                                    TCCState
sendTCCReq(t,sb,dir,cs) ≡
    let
        sender = T.isTrain(t),
        receiver = T.isSB(sb),
        res = T.mk_res(t,dir),
        msg = T.segReq(res),
        comMsg = T.mk_comMsg(sender,receiver,msg),
        cs = setTCCRequesting(true,cs)
```

```
        in
          COM.sendMsg(comMsg); cs
        end,

      /∗ Invariants ∗/
      initReq : TCCState → Bool
      initReq(tcc) ≡
        no_tcc_res(tcc) ∧
        tcc_not_requesting(tcc) ∧
        tcc_not_decelerating(tcc),

      /∗ tcc may not have a reservation ∗/
      no_tcc_res : TCCState → Bool
      no_tcc_res(tcc) ≡
      (
        ∼hasTCCRes(tcc)
      ),

      /∗ No TCC is requesting segment access ∗/
      tcc_not_requesting : TCCState → Bool
      tcc_not_requesting(tcc) ≡
      (
        ∼isTCCRequesting(tcc)
      ),

      /∗ No TCC is requesting segment access ∗/
      tcc_not_decelerating : TCCState → Bool
      tcc_not_decelerating(tcc) ≡
      (
        ∼isTrainDecelerating(tcc)
      )

  end
```

SBCC

```
context: CA_Types0, CA_Statics0, CA_Dynamics0, CA_ComService0
scheme CA_SBCC0(T : CA_Types0, S : CA_Statics0(T),
                D : CA_Dynamics0(T,S), COM : CA_ComService0(T)) =
  class
    type
      SBCCState :: getLineRes : T.HasRes ↔ setLineRes
                   getBranchRes : T.HasRes ↔ setBranchRes
                   getLastSensorStatus : T.SensorStatus ↔
                                         setLastSensorStatus
                   getMsgs : T.ComMsg* ↔ setMsgs
                   getPrepRes : T.HasRes ↔ setPrepRes

    value
      initSBCCState : SBCCState,

      getNextMsg : SBCCState → T.HasComMsg × SBCCState
      getNextMsg(sbcc) ≡
        let
          msgList = getMsgs(sbcc)
```

```
    in
        if(msgList = ⟨ ⟩)
        then
            (T.noComMsg,sbcc)
        else
            (T.comMsg(hd msgList), setMsgs(tl msgList,sbcc))
        end
    end,

storeMsg : T.ComMsg × SBCCState → SBCCState
storeMsg(msg,sbcc) ≡
    setMsgs(getMsgs(sbcc) ⁀ ⟨ msg ⟩,sbcc),

removeLineRes : SBCCState → SBCCState
removeLineRes(sbcc) ≡
    setLineRes(T.noRes,sbcc),

removeBranchRes : SBCCState → SBCCState
removeBranchRes(sbcc) ≡
    setBranchRes(T.noRes,sbcc),

msgReceiver : T.ComMsg × SBCCState → SBCCState
msgReceiver(comMsg,sbcc) ≡
    storeMsg(comMsg,sbcc),

removePrepRes : SBCCState → SBCCState
removePrepRes(cs) ≡
    setPrepRes(T.noRes,cs),

isPreparing : SBCCState → Bool
isPreparing(cs) ≡
    case getPrepRes(cs) of
        T.noRes → false,
        _ → true
    end,

/* Processes */
sbccProcess : T.SBID × T.Tick × SBCCState × D.State ×
                                S.Configuration → out COM.comChannel SBCCState
sbccProcess(sb,tick,cs,ds,con) ≡
    let
        cs = sensorProcess(sb,cs,ds,con)
    in
        if(isPreparing(cs))
        then
            prepareProcess(sb,cs,ds,con)
        else
            sbccMsgProcess(sb,cs,ds,con)
        end
    end,

/* Waits for the preparation of a segment
     before a train is allowed to enter */
prepareProcess : T.SBID × SBCCState × D.State ×
                                S.Configuration → out COM.comChannel SBCCState
prepareProcess(sb,cs,ds,con) ≡
    case S.getSBType(sb,con) of
```

```
        /∗ case POINTSB → wait for !moving ∗/
        T.POINTSB →
        (
            if(D.getPointPosition(sb,ds,con) ∈ {T.UP, T.DOWN})
            then
                let
                    T.res(res) = getPrepRes(cs),
                    train = T.getTrain(res),
                    cs = removePrepRes(cs)
                in
                    sendSBCCMsg(sb,T.isTrain(train),T.segResp(true)); cs
                end
            else
                cs
            end
        ),

        /∗ case crossingsb → wait for DOWN ∗/
        T.CROSSINGSB →
        (
            if(D.getBarrierPosition(sb,ds,con) = T.DOWN)
            then
                let
                    T.res(res) = getPrepRes(cs),
                    train = T.getTrain(res),
                    cs = removePrepRes(cs)
                in
                    sendSBCCMsg(sb,T.isTrain(train),T.segResp(true)); cs
                end
            else
                cs
            end
        ),

        _ → cs
    end,

sensorProcess : T.SBID × SBCCState × D.State ×
                                S.Configuration → out COM.comChannel
                                            SBCCState
sensorProcess(sb,sbcc,ds,con) ≡
    let
        sState = D.getSensorStatus(sb,ds),
        lastState = getLastSensorStatus(sbcc),
        sbcc = setLastSensorStatus(sState,sbcc)
    in
        if((lastState = T.ACTIVE) ∧ (sState = T.INACTIVE))
        then
            let
                ds = dePrepareSeg(sb,sbcc,ds,con)
            in
                if(S.isLineGuard(sb,con))
                then
                    makeDeRes(sb,sbcc,ds,con)
                else
                    sbcc
                end
```

```
        end
      else
        sbcc
      end
    end,

dePrepareSeg : T.SBID × SBCCState × D.State ×
                          S.Configuration → SBCCState × D.State
dePrepareSeg(sb,cs,ds,con) ≡
  let
    cs = removePrepRes(cs)
  in
    case S.getSBType(sb,con) of
      T.CROSSINGSB →
      (
        (cs,D.setBarrierPosition(sb,T.MOVINGUP,ds,con))
      ),

      _ → (cs,ds)
    end
  end,

prepareSeg : T.SBID × T.Reservation × SBCCState ×
                          D.State × S.Configuration →
                                    SBCCState × D.State
prepareSeg(sb,res,cs,ds,con) ≡
  let
    cs = setPrepRes(T.res(res),cs)
  in
    case S.getSBType(sb,con) of
      T.CROSSINGSB →
      (
        let
          ds = D.setSignalStatus(sb,T.ON,ds,con)
        in
          (cs,ds)
        end
      ),

      T.POINTSB →
      (
        case T.getDir(res) of
          T.UP →
          (
            let
              ds = D.setPointPosition(sb,T.MOVINGUP,ds,con)
            in
              (cs,ds)
            end
          ),

          T.DOWN →
          (
            let
              ds = D.setPointPosition(sb,T.MOVINGDOWN,ds,con)
            in
              (cs,ds)
```

```
                end
            )
        end
    ),

    _ → (cs,ds)
  end
end,

makeDeRes : T.SBID × SBCCState × D.State ×
                              S.Configuration → out COM.comChannel SBCCState
makeDeRes(sb,sbcc,ds,con) ≡
  case S.getSBType(sb,con) of
    T.ENDSB →
    (
      let
        T.res(lineRes) = getLineRes(sbcc),
        endDir = S.getEndDir(sb,con)
      in
        if(T.getDir(lineRes) = endDir)
        then
          let
            sbcc = removeLineRes(sbcc)
          in
            sendLDeResMsg(sb,S.getOppositeGuard(sb,con)); sbcc
          end
        else
          sbcc
        end /∗ if ∗/
      end /∗ let ∗/
    ),

    T.POINTSB →
    (
      let
        T.res(lineRes) = getLineRes(sbcc),
        pointDir = S.getPointDir(sb,con)
      in
        if(T.getDir(lineRes) = pointDir)
        then
          let
            sbcc = removeLineRes(sbcc)
          in
            sendLDeResMsg(sb,S.getOppositeGuard(sb,con)); sbcc
          end
        else
          sendBDeResMsg(sb,S.getOppositeGuard(sb,con)); sbcc
        end /∗ if ∗/
      end /∗ let ∗/
    )
  end /∗ case ∗/
pre S.isLineGuard(sb,con),

sendLBDeResMsg : T.SBID × T.SBID → out COM.comChannel Unit
sendLBDeResMsg(thisSB,remoteSB) ≡
  sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineBranchDeRes),
```

```
sendLDeResMsg : T.SBID × T.SBID → out COM.comChannel Unit
sendLDeResMsg(thisSB,remoteSB) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineDeRes),

sendBDeResMsg : T.SBID × T.SBID → out COM.comChannel Unit
sendBDeResMsg(thisSB,remoteSB) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.branchDeRes),

sendLBResMsg : T.SBID × T.SBID × T.Reservation →
                                        out COM.comChannel Unit
sendLBResMsg(thisSB,remoteSB,aRes) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineBranchReq(aRes)),

sendSBCCMsg : T.SBID × T.ComID × T.SBCCMsg →
                                        out COM.comChannel Unit
sendSBCCMsg(sb,receiver,sbccMsg) ≡
    COM.sendMsg(T.mk_comMsg(T.isSB(sb),receiver,sbccMsg)),

sbccMsgProcess : T.SBID × SBCCState × D.State ×
                    S.Configuration → out COM.comChannel SBCCState
sbccMsgProcess(sb,sbcc,ds,con) ≡
    let
        (hasComMsg,sbcc) = getNextMsg(sbcc)
    in
        case hasComMsg of
            T.comMsg(T.mk_comMsg(sender,receiver,msg)) →
            (
                case sender of
                    T.isTrain(_) →
                    (
                        let
                            (retMsg,sbcc,ds) =handleTCCMsg(sb,msg,sbcc,ds,con)
                        in
                            case retMsg of
                                T.hasMsg(aMsg) → sendSBCCMsg(sb,sender,aMsg);
                                                sbcc,
                                _ → sbcc
                            end
                        end
                    ),

                    T.isSB(_) →
                    (
                        let
                            (sbcc,retMsg) = handleSBCCMsg(sb,msg,sbcc)
                        in
                            case retMsg of
                                T.hasMsg(aMsg) → sendSBCCMsg(sb,sender,aMsg);
                                                sbcc,
                                _ → sbcc
                            end
                        end
                    )
                end /* case */
            ),
            _ → sbcc /* no message to process */
        end
```

```
    end, /* let */

handleSBCCMsg : T.SBID × T.Message × SBCCState →
                                        out COM.comChannel
                                         SBCCState × T.ReturnSBCCMsg
handleSBCCMsg(sb,msg,sbcc) ≡
    case msg of
        /* Request */
        T.lineBranchReq(_) → handleLBReq(msg,sbcc),

        /* Response */
        T.lineBranchResp(_) → handleLBResp(sb,msg,sbcc),

        /* De reservation */
        T.lineBranchDeRes → handleDeResMsg(msg,sbcc),
        T.lineDeRes → handleDeResMsg(msg,sbcc),
        T.branchDeRes → handleDeResMsg(msg,sbcc)
    end,

handleTCCMsg : T.SBID × T.Message × SBCCState × D.State ×
                                S.Configuration → out COM.comChannel
                                T.ReturnSBCCMsg × SBCCState × D.State
handleTCCMsg(sb,msg,cs,ds,con) ≡
    let
        T.segReq(res) = msg
    in
        case S.getSBType(sb,con) of
            T.ENDSB →
            (
                /* if direction away from end → send msg
                     else OK
                */
                if(T.getDir(res) = S.getEndDir(sb,con))
                then
                    let
                        (cs,ds) = prepareSeg(sb,res,cs,ds,con)
                    in
                        (T.noSBCCMsg,cs,ds)
                    end
                else
                    if(lineFree(cs))
                    then
                        let
                            cs = setLineRes(T.res(res),cs),
                            cs = sendLBResMsg(sb,
                                        S.getOppositeGuard(sb,con),res)
                        in
                            (T.noSBCCMsg,cs,ds)
                        end
                    else
                        (T.hasMsg(T.segResp(false)),cs,ds)
                    end
                end
            ),

            /* If direction away from point → send msg
                 else OK
```

```
            */
            T.POINTSB →
            (
                if(T.getDir(res) = S.getPointDir(sb,con))
                then
                    let
                        (cs,ds) = prepareSeg(sb,res,cs,ds,con)
                    in
                        (T.noSBCCMsg,cs,ds)
                    end
                else
                    if(lineFree(cs))
                    then
                        let
                            cs = setLineRes(T.res(res),cs),
                            cs = sendLBResMsg(sb,
                                        S.getOppositeGuard(sb,con),res)
                        in
                            (T.noSBCCMsg,cs,ds)
                        end
                    else
                        (T.hasMsg(T.segResp(false)),cs,ds)
                    end
                end
            ),

            _ → /* PLAINSB, CROSSINGSB */
            (
                let
                    (cs,ds) = prepareSeg(sb,res,cs,ds,con)
                in
                    (T.noSBCCMsg,cs,ds)
                end
            )
        end /* case */
    end, /* let */

/* if(!line free) then NO
    reserve line;
    if(end type) then YES
    if(!branch free) then deres line; NO
    res branch; YES;
*/
handleLBReq : T.Message × SBCCState →
                                        SBCCState × T.ReturnSBCCMsg
handleLBReq(msg,sbcc) ≡
    let
        T.lineBranchReq(res) = msg
    in
        if(lineBranchFree(sbcc))
        then
            let
                sbcc = setLineRes(T.res(res),sbcc),
                sbcc = setBranchRes(T.res(res),sbcc)
            in
                (sbcc,T.hasMsg(T.lineBranchResp(res,true)))
            end
```

```
        else
            (sbcc,T.hasMsg(T.lineBranchResp(res,false)))
        end
    end,

lineBranchFree : SBCCState → Bool
lineBranchFree(sbcc) ≡
    (getLineRes(sbcc) = T.noRes) ∧
    (getBranchRes(sbcc) = T.noRes),

lineFree : SBCCState → Bool
lineFree(sbcc) ≡
    (getLineRes(sbcc) = T.noRes),

/∗ if(response = NO) deres; NO;
     if(!prepare_segment()) deres; NO;
     OK;
∗/
handleLBResp : T.SBID × T.Message × SBCCState →
                                                    out COM.comChannel
                                                    SBCCState × T.ReturnSBCCMsg
handleLBResp(sb,msg,sbcc) ≡
    let
        T.lineBranchResp(res,granted) = msg
    in
        if(granted)
        then
            sendSBCCMsg(sb,T.isTrain(T.getTrain(res)),
                                                    T.segResp(true));
            (sbcc,T.noSBCCMsg)
        else
            let
                sbcc = removeLineRes(sbcc)
            in
                sendSBCCMsg(sb,T.isTrain(T.getTrain(res)),
                                                        T.segResp(false));
                (sbcc,T.noSBCCMsg)
            end
        end
    end,

/∗ case(msg)
            lb → deres line; deres branch
            l → deres line;
            b → deres branch;
∗/
handleDeResMsg : T.Message × SBCCState →
                                                    SBCCState × T.ReturnSBCCMsg
handleDeResMsg(msg,sbcc) ≡
    case msg of
        T.lineBranchDeRes →
        (
            let
                sbcc = removeLineRes(sbcc),
                sbcc = removeBranchRes(sbcc)
            in
                (sbcc,T.noSBCCMsg)
```

```
            end
          ),

          T.lineDeRes →
          (
            let
              sbcc = removeLineRes(sbcc)
            in
              (sbcc,T.noSBCCMsg)
            end
          ),

          T.branchDeRes →
          (
            let
              sbcc = removeBranchRes(sbcc)
            in
              (sbcc,T.noSBCCMsg)
            end
          )
        end,

      /* Invariants */
      initReq : SBCCState → Bool
      initReq(sbcc) ≡
        no_sbcc_res(sbcc) ∧
        sbcc_not_preparing(sbcc),

      no_sbcc_res : SBCCState → Bool
      no_sbcc_res(sbcc) ≡
      (
        ∀ branchRes,lineRes : T.HasRes •
              branchRes = getBranchRes(sbcc) ∧
              lineRes = getLineRes(sbcc)
                ⇒
                  {branchRes} ∪ {lineRes} = {T.noRes}
      ),

      /* No SBCC is currently preparing a segment */
      sbcc_not_preparing : SBCCState → Bool
      sbcc_not_preparing(sbcc) ≡
      (
        ∼isPreparing(sbcc)
      )

  end
```

F.4 Imperative model

F.4.1 Types

```
scheme I_Types0 =
  class
    type
```

```
/* Entity ID types */
ID = Text,
ESAID = End,
SBID = {| sb : ID • sbIDLimit(sb) |},
SegmentID = {| seg : ID • segIDLimit(seg) |},
TrainID = {| t : ID • trainIDLimit(t) |},

/* Location of a train, on an esa or an segment */
Location == isESA(getESA : ESAID) |
                  isSeg(getSeg : SegmentID),

/* The ends (esa ends) */
End == HIGH | LOW,
/* Driving direction on the line */
Direction == UP | DOWN,

/* Physical parameters */
Length = Real,
Speed = Real,
Acceleration = Real,

/* Neighbour of a SB in a direction */
SBSegment ==
      seg(getSeg : SegmentID) |
      point(getUpSeg : SegmentID, getDownSeg : SegmentID) |
      esa(getESA : ESAID),
/* The type of a SB */
SBType == POINTSB | ENDSB | CROSSINGSB | PLAINSB,
/* The segments etc around a point */
PointSegments == pointSegments(getStem : SegmentID,
                               getUpBranch : SegmentID,
                               getDownBranch : SegmentID,
                               getPointDir : Direction),

/* The state of different entities */
PointPosition == UP | DOWN | MOVINGUP | MOVINGDOWN,
BarrierPosition == UP | DOWN | MOVINGUP | MOVINGDOWN,
SignalStatus == ON | OFF,
SensorStatus == ACTIVE | INACTIVE,

HasPointPosition ==
   pointPos(getPos : PointPosition ↔ setPos) | none,
HasBarrierPosition ==
   barrierPos(getPos : BarrierPosition ↔ setPos) | none,
HasSignalStatus ==
   signalStatus(getStatus : SignalStatus ↔ setStatus) | none,

/* Location/position of a train */
TrainPosition :: frontPos : SegmentPosition ↔ setFrontPos
                 rearPos : SegmentPosition ↔ setRearPos,

/* Location / position of a train end */
SegmentPosition :: getLoc : Location
                   getLength : Length,

Tick = Real,
HasTicks == ticks(getTicks : Tick) | none,
```

```
    /∗ From Control ∗/
    HasRes == res(Reservation) | noRes,
    HasSeg == isSeg(SegmentID) | noSeg,

    Message = TCCMsg | SBCCMsg,
    TCCMsg == segReq(Reservation),
    SBCCMsg = SBCCResMsg | SBCCDeResMsg | SBCCRespMsg,
    SBCCResMsg == lineBranchReq(Reservation),
    SBCCDeResMsg == lineBranchDeRes | lineDeRes
                              | branchDeRes,
    SBCCRespMsg = LineBranchResp | SegmentResp,
    LineBranchResp ==
                lineBranchResp(getRes : Reservation, isPos : Bool),
    SegmentResp == segResp(isPos : Bool),

    Reservation ==
                mk_res(getTrain : TrainID, getDir : Direction),

    ReturnSBCCMsg == hasMsg(SBCCMsg) | noSBCCMsg,

    ComID == isSB(SBID) | isTrain(TrainID),
    ComMsg == mk_comMsg(getSender : ComID,
                              getReceiver : ComID,
                              getMsg : Message),
    HasComMsg == comMsg(ComMsg) | noComMsg

value
    /∗ The tick interval in seconds ∗/
    tick_interval : Tick,

    /∗ Maximal ID numbers ∗/
    sbIDSet : ID-set,
    segIDSet : ID-set,
    trainIDSet : ID-set,

    /∗ Limits the ID of SBs Segments and Trains ∗/
    sbIDLimit : ID → Bool
    sbIDLimit(id) ≡
        id ∈ sbIDSet,

    segIDLimit : ID → Bool
    segIDLimit(id) ≡
        id ∈ segIDSet,

    trainIDLimit : ID → Bool
    trainIDLimit(id) ≡
        id ∈ trainIDSet,

    /∗ Inverse the direction ∗/
    inverseDir : Direction → Direction
    inverseDir(dir) ≡
        case dir of
            UP → DOWN,
            DOWN → UP
        end,
```

```
/∗ Determines if a certain location is
     included in a SBSegment ∗/
segPosInSBSeg : SegmentPosition × SBSegment → Bool
segPosInSBSeg(loc,sbSeg) ≡
    case sbSeg of
        seg(seg) → isSeg(seg) = getLoc(loc),
        point(upSeg,downSeg) → isSeg(upSeg) = getLoc(loc) ∨
                                     isSeg(downSeg) = getLoc(loc),
        esa(esa) → isESA(esa) = getLoc(loc)
    end,

/∗ Returns all the segments in a 'SBSegment′ ∗/
sbSegToSet : SBSegment → SegmentID-set
sbSegToSet(sbSeg) ≡
    case sbSeg of
        seg(seg1) → { seg1 },
        point(seg1,seg2) → { seg1,seg2 },
        _ → {}
    end,

/∗ Returns the end to reach when
        following a certain direction ∗/
dir2End : Direction → End
dir2End(dir) ≡
    case dir of
        DOWN → LOW,
        UP → HIGH
    end,

/∗ Returns the direction against
     an ESA from the ESA ∗/
end2Dir : End → Direction
end2Dir(end1) ≡
    case end1 of
        LOW → DOWN,
        HIGH → UP
    end,

/∗ Determines if a location is an ESA ∗/
segPosIsESA : SegmentPosition → Bool
segPosIsESA(tEndPos) ≡
    case getLoc(tEndPos) of
        isESA(esa) → true,
        _ → false
    end,

/∗ Determines if an end position is a segment ∗/
segPosIsSeg : SegmentPosition → Bool
segPosIsSeg(tEndPos) ≡
    case getLoc(tEndPos) of
        isSeg(_) → true
    end,

/∗ Determines if a TrainPosition is
     located on one segment ∗/
trainOnlyOnESA : TrainPosition → Bool
trainOnlyOnESA(pos) ≡
```

```
        case getLoc(frontPos(pos)) of
          isESA(_) →
          (
            case getLoc(rearPos(pos)) of
              isESA(_) → true,
              _ → false
            end
          ),

          _ → false
        end,

      /* Returns a set containing a segment if the
         position is on a segment else an empty set */
      segPosSeg : SegmentPosition → SegmentID-set
      segPosSeg(tep) ≡
        case getLoc(tep) of
          isSeg(seg) → {seg},
          _ → {}
        end,

      trainPosSegs : TrainPosition → SegmentID-set
      trainPosSegs(tp) ≡
        segPosSeg(frontPos(tp)) ∪ segPosSeg(rearPos(tp)),

      frontLoc : TrainPosition → Location
      frontLoc(tp) ≡
        getLoc(frontPos(tp)),

      rearLoc : TrainPosition → Location
      rearLoc(tp) ≡
        getLoc(rearPos(tp)),

      oneLoc : TrainPosition → Bool
      oneLoc(tp) ≡
        frontLoc(tp) = rearLoc(tp)

    axiom
      /* Two ID′s cannot be identical */
      [is_wf_id_sets]
      sbIDSet ∪ segIDSet ∪ trainIDSet = {}

  end
```

F.4.2 Statics

```
context: I_Types0, I_SBs0, I_Segs0, I_Trains0, I_ESAs0
scheme I_Statics0(T : I_Types0) =
  class
    object
      SBs : I_SBs0(T),
      ESAs : I_ESAs0(T),
      Segs : I_Segs0(T),
      Trains : I_Trains0(T)
```

```
type
    /∗ Main railway line configuration type ∗/
    Configuration = SBs.SBs × Segs.Segs ×
                                        ESAs.ESAs × Trains.Trains

value
    conf : Configuration = (SBs.sbsConf, Segs.segsConf,
                                        ESAs.esasConf, Trains.trainsConf),

    /∗ Observers ∗/

    /∗ ESA observers ∗/
    getESASB : T.ESAID → read ESAs.v_ESAs T.SBID
    getESASB(esa) ≡
        ESAs.getESASB(esa),

    getESALength : T.ESAID → read ESAs.v_ESAs T.Length
    getESALength(esa) ≡
        ESAs.getESALength(esa),

    /∗ SB observers ∗/
    getSBSeg : T.SBID × T.Direction → read SBs.v_SBs T.SBSegment
    getSBSeg(sb,dir) ≡
        SBs.getSBSeg(sb,dir),

    getSBType : T.SBID → read SBs.v_SBs T.SBType
    getSBType(sb) ≡
        SBs.getSBType(sb),

    getPointTicks : T.SBID ~→ read SBs.v_SBs T.Tick
    getPointTicks(sb) ≡
        SBs.getPointTicks(sb)
    pre getSBType(sb) = T.POINTSB,

    getBarrierTicks : T.SBID ~→ read SBs.v_SBs T.Tick
    getBarrierTicks(sb) ≡
        SBs.getBarrierTicks(sb)
    pre getSBType(sb) = T.CROSSINGSB,

    getSignalTicks : T.SBID ~→ read SBs.v_SBs T.Tick
    getSignalTicks(sb) ≡
        SBs.getSignalTicks(sb)
    pre getSBType(sb) = T.CROSSINGSB,

    /∗ Segment observers ∗/
    getSegSB : T.SegmentID × T.Direction →
                                        read Segs.v_segs T.SBID
    getSegSB(seg,dir) ≡
        Segs.getSegSB(seg,dir),

    getSegLength : T.SegmentID → read Segs.v_segs T.Length
    getSegLength(segID) ≡
        Segs.getSegLength(segID),

    getSegMaxSpeed : T.SegmentID → read Segs.v_segs T.Speed
    getSegMaxSpeed(segID) ≡
        Segs.getSegMaxSpeed(segID),
```

```
/* Train observers */
getTrainLength : T.TrainID → read Trains.v_trains T.Length
getTrainLength(tID) ≡
    Trains.getTrainLength(tID),

getTrainMaxSpeed : T.TrainID →
                                read Trains.v_trains T.Acceleration
getTrainMaxSpeed(t) ≡
    Trains.getTrainMaxSpeed(t),

getTrainMaxAcc : T.TrainID →
                                read Trains.v_trains T.Acceleration
getTrainMaxAcc(t) ≡
    Trains.getTrainMaxAcc(t),

getTrainMaxDec : T.TrainID →
                                read Trains.v_trains T.Acceleration
getTrainMaxDec(t) ≡
    Trains.getTrainMaxDec(t),

/* Reservation− and brake−point observers */
getResPoint : Unit → read Segs.v_points T.Length
getResPoint() ≡
    Segs.getResPoint(),

getBrakePoint : Unit → read Segs.v_points T.Length
getBrakePoint() ≡
    Segs.getBrakePoint(),

/* Auxiliary functions */

/* Determines if a SB is a Single Line Guard */
isLineGuard : T.SBID → read SBs.v_SBs Bool
isLineGuard(sbID) ≡
    getSBType(sbID) ∈ {T.POINTSB, T.ENDSB},

/* Determines if a SB is a PointSB */
isPointSB : T.SBID → read SBs.v_SBs Bool
isPointSB(sbID) ≡
    getSBType(sbID) = T.POINTSB,

/* Determines if a segment is a branch segment */
segIsBranch : T.SegmentID → read Segs.v_segs, SBs.v_SBs Bool
segIsBranch(seg) ≡
    getSBType(getSegSB(seg,T.UP)) = T.POINTSB ∧
    getSBType(getSegSB(seg,T.DOWN)) = T.POINTSB,

/* Determines if a segment is a line segment,
     i.e. a segment in a single line */
segIsLineSegment : T.SegmentID →
                                read Segs.v_segs, SBs.v_SBs Bool
segIsLineSegment(seg) ≡
    ∼segIsBranch(seg),

/* If two T.Location′s are neighbours in configuration.
     Note that a T.Location is not neighbour to itself */
```

```
neighbours : T.Location × T.Location →
                                    read Segs.v_segs, ESAs.v_ESAs Bool
neighbours(loc1,loc2) ≡
    (getLocSBs(loc1) ∪ getLocSBs(loc2)) ≠ {},

/∗ Finds the distance (T.Length)
     between two T.SegmentPosition ∗/
distance : T.SegmentPosition × T.SegmentPosition →
                                    read Segs.v_segs, SBs.v_SBs,
                                       ESAs.v_ESAs T.Length
distance(segPos1,segPos2) ≡
    if (T.getLoc(segPos1) = T.getLoc(segPos2))
    then
        if (T.getLength(segPos1) < T.getLength(segPos2))
        then
            T.getLength(segPos2) − T.getLength(segPos1)
        else
            T.getLength(segPos1) − T.getLength(segPos2)
        end
    else
        if (segPosLower(segPos1,segPos2))
        then
            getLocLength(T.getLoc(segPos1)) −
                        T.getLength(segPos1) + T.getLength(segPos2)
        else
            getLocLength(T.getLoc(segPos2)) −
                        T.getLength(segPos2) + T.getLength(segPos1)
        end
    end
pre neighbours(T.getLoc(segPos1),T.getLoc(segPos2)) ∨
        T.getLoc(segPos1) = T.getLoc(segPos2),

segPosLower : T.SegmentPosition × T.SegmentPosition →
                                    read Segs.v_segs, SBs.v_SBs Bool
segPosLower(segPos1,segPos2) ≡
    if (T.getLoc(segPos1) = T.getLoc(segPos2)) then
        T.getLength(segPos1) < T.getLength(segPos2)
    else
        locLower(T.getLoc(segPos1),T.getLoc(segPos2))
    end,


/∗ If a location is immediatedly lower
     than another location. If one location
     is an ESA, the result will depend on
     the orientation of the ESA. ∗/
locLower : T.Location × T.Location →
                                    read Segs.v_segs, SBs.v_SBs Bool
locLower(loc1,loc2) ≡
    case loc1 of
        T.isESA(esa1) → (esa1 = T.LOW),
        T.isSeg(seg1) →
        (
            case loc2 of
                T.isESA(esa2) → (esa2 = T.HIGH),
                T.isSeg(seg2) →
                (
```

```
                seg2 ∈ getNextSegSet(seg1,T.UP)
            )
          end
      )
   end,

getLocLength : T.Location →
                          read ESAs.v_ESAs, Segs.v_segs T.Length
getLocLength(loc) ≡
   case loc of
      T.isESA(esa) → getESALength(esa),
      T.isSeg(seg) → getSegLength(seg)
   end,

/∗ Returns the SB′s (up and down)
    of a location  (segment / esa) ∗/
getLocSBs : T.Location → read ESAs.v_ESAs,
                             Segs.v_segs T.SBID-set
getLocSBs(loc) ≡
   case loc of
      T.isESA(esa) → {getESASB(esa)},
      T.isSeg(seg) → {getSegSB(seg,T.UP),getSegSB(seg,T.DOWN)}
   end,

/∗ Returns the segments around a point ∗/
getSBPointSegs : T.SBID ~→ read SBs.v_SBs T.PointSegments
getSBPointSegs(sb) ≡
   SBs.getSBPointSegs(sb)
pre getSBType(sb) = T.POINTSB,

/∗ Returns the driving direction of a branch ∗/
branchDir : T.SegmentID → read Segs.v_segs,
                             SBs.v_SBs T.Direction
branchDir(seg) ≡
   let
      T.point(up,down) = getSBSeg(getSegSB(seg,T.UP),T.DOWN)
   in
      if (seg = up)
      then
         T.UP
      else
         T.DOWN
      end
   end
pre segIsBranch(seg),

/∗ Given a single line guard, it returns
    the single line guard at the opposite
    end of the single line ∗/
getOppositeGuard : T.SBID ~→ read Segs.v_segs,
                                SBs.v_SBs T.SBID
getOppositeGuard(sb) ≡
   let
      sbType = getSBType(sb),
      dir = if(sbType = T.POINTSB) then getPointDir(sb)
                else getEndDir(sb) end,
      lineDir = T.inverseDir(dir)
```

```
    in
        getSingleLineGuard(getNextSB(sb,lineDir),lineDir)
    end
pre isLineGuard(sb),

/* Given a point SB, it returns the point SB
     at the opposite end of the branches */
getOppositePointSB : T.SBID ~→
                                  read Segs.v_segs, SBs.v_SBs T.SBID
getOppositePointSB(sb) ≡
    let
        dir = getPointDir(sb)
    in
         getNextSB(sb,dir)
    end
pre getSBType(sb) = T.POINTSB,

/* Given an SB, it returns the next SB */
getNextSB : T.SBID × T.Direction ~→
                                  read Segs.v_segs, SBs.v_SBs T.SBID
getNextSB(sb,dir) ≡
    let
        nextSeg = getSBSeg(sb,dir)
    in
        case nextSeg of
            T.seg(segID) → getSegSB(segID,dir),
            T.point(upSeg,downSeg) → getSegSB(upSeg,dir)
        end
    end
pre getSBType(sb) ≠ T.ENDSB ∨ getEndDir(sb) ≠ dir,

getNextSegSet : T.SegmentID × T.Direction →
                                  read Segs.v_segs, SBs.v_SBs
                                          T.SegmentID-set
getNextSegSet(seg,dir) ≡
    T.sbSegToSet(getSBSeg(getSegSB(seg,dir),dir)),

/* Returns the first single line
     guard in a direction */
getSingleLineGuard : T.SBID × T.Direction ~→
                                  read Segs.v_segs, SBs.v_SBs T.SBID
getSingleLineGuard(sb,dir) ≡
    if(isLineGuard(sb))
    then
        sb
    else
        getSingleLineGuard(getNextSB(sb,dir),dir)
    end,

/* Returns the two single line
     guards of a line−segment */
getSingleLineGuards : T.SegmentID ~→
                              read Segs.v_segs, SBs.v_SBs T.SBID-set
getSingleLineGuards(seg) ≡
    let
        sb = getSegSB(seg,T.UP)
    in
```

```
        { getSingleLineGuard(sb,T.UP),
            getSingleLineGuard(sb,T.DOWN) }
    end
pre ~segIsBranch(seg),

/* Returns the direction of a point SB
     from the stem towards the branches */
getPointDir : T.SBID ~→ read SBs.v_SBs T.Direction
getPointDir(sb) ≡
    SBs.getPointDir(sb)
pre getSBType(sb) = T.POINTSB,

/* Returns the direction against an ESA from an END SB */
getEndDir : T.SBID ~→ read SBs.v_SBs T.Direction
getEndDir(sb) ≡
    SBs.getEndDir(sb)
pre getSBType(sb) = T.ENDSB,

sbsAreCrossings : T.SBID-set → read SBs.v_SBs Bool
sbsAreCrossings(sbs) ≡
(
    ∀ sb : T.SBID •
        sb ∈ sbs ⇒
            getSBType(sb) = T.CROSSINGSB
),

sbsArePoints : T.SBID-set → read SBs.v_SBs Bool
sbsArePoints(sbs) ≡
(
    ∀ sb : T.SBID •
        sb ∈ sbs ⇒
            getSBType(sb) = T.POINTSB
),

/* Invariants */
is_wf : Unit → read Segs.any, Segs.v_points, SBs.v_SBs,
                                    Trains.v_trains, ESAs.v_ESAs Bool
is_wf() ≡
    SBs.is_wf() ∧
    Segs.is_wf() ∧
    ESAs.is_wf() ∧
    Trains.is_wf() ∧
    composed_is_wf(),

composed_is_wf : Unit → read Segs.any, SBs.v_SBs,
                                    Trains.v_trains, ESAs.v_ESAs Bool
composed_is_wf() ≡
    pointSegs_wf() ∧
    getESASBSeg_wf() ∧
    getSBSeg_getSegSB_wf()∧
    seg_train_length_wf() ∧
    esa_train_length_wf() ∧
    brakePoint_wf() ∧
    resPoint_wf() ∧
    collisions_detectable(),

/* All associated point (points next
```

```
     to each other) must have
     same up and down branches */
pointSegs_wf : Unit → read SBs.v_SBs, Segs.v_segs Bool
pointSegs_wf() ≡
(
    ∀ sb : T.SBID •
        getSBType(sb) = T.POINTSB
            ⇒
                let
                    pSegs1 = getSBPointSegs(sb),
                    dir1 = T.getPointDir(pSegs1),

                    sb2 = getNextSB(sb,dir1),
                    pSegs2 = getSBPointSegs(sb2)
                in
                    T.getUpBranch(pSegs1) = T.getUpBranch(pSegs2) ∧
                    T.getDownBranch(pSegs1) = T.getDownBranch(pSegs2)
                end
),

/* Given an ESA. From the coherent END SB
     the next SBSegment directed against the
     ESA must be the ESA */
getESASBSeg_wf : Unit → read ESAs.v_ESAs, SBs.v_SBs Bool
getESASBSeg_wf() ≡
(
    ∀ esa : T.ESAID •
        getSBSeg(getESASB(esa),T.end2Dir(esa)) = T.esa(esa)
),

/* Calculating the SB in a direction from each segment
     in the SBSegment calculated from a SB in the opposite
     direction must give the original SB */
getSBSeg_getSegSB_wf : Unit →
                                    read Segs.v_segs, SBs.v_SBs Bool
getSBSeg_getSegSB_wf() ≡
(
    ∀ sb : T.SBID, dir : T.Direction, seg : T.SegmentID •
        seg ∈ T.sbSegToSet(getSBSeg(sb,dir)) ⇒
            getSegSB(seg,T.inverseDir(dir)) = sb
),

/* All segments must be longer than any train */
seg_train_length_wf : Unit →
                                    read Segs.v_segs, Trains.v_trains Bool
seg_train_length_wf() ≡
(
    ∀ seg : T.SegmentID, t : T.TrainID •
        getSegLength(seg) > getTrainLength(t)
),

/* All ESA′s must be longer than any train */
esa_train_length_wf : Unit → read ESAs.v_ESAs,
                                    Segs.v_points, Trains.v_trains Bool
esa_train_length_wf() ≡
(
    ∀ esa : T.ESAID, t : T.TrainID •
```

```
        getESALength(esa) > getBrakePoint() + getTrainLength(t)
),

/* If a train starts to brake at the brakepoint
     it must be able to stop entirely before
     entering the next segment
*/
brakePoint_wf : Unit → read Trains.v_trains,
                                          Segs.v_points Bool
brakePoint_wf() ≡
(
    ∀ t : T.TrainID, tAcc : T.Acceleration,
            brakeP, brakeL, s_err : T.Length,
            tSpeed : T.Speed •
                tAcc = getTrainMaxDec(t) ∧
                brakeP = getBrakePoint() ∧
                tSpeed = getTrainMaxSpeed(t) ∧
                s_err = tSpeed ∗ T.tick_interval ∧
                brakeL = −0.5 ∗ tSpeed ∗ tSpeed / tAcc
                    ⇒
                        brakeP > brakeL + s_err
),

/* ensures that resPoint > brakePoint and
     that a train is entirely on a single
     segment when resPoint is exceeded
     also that brakePoint < segment length */
resPoint_wf : Unit → read Trains.v_trains, Segs.any Bool
resPoint_wf() ≡
(
    ∀ t : T.TrainID, seg : T.SegmentID,
            tlen, slen, resPoint, brakePoint : T.Length •
                tlen = getTrainLength(t) ∧
                slen = getSegLength(seg) ∧
                resPoint = getResPoint() ∧
                brakePoint = getBrakePoint()
                    ⇒
                        slen  > (resPoint + tlen) ∧
                        brakePoint < slen
),

/* Ensures that collisions can be
     detected before trains passes
     through each other */
collisions_detectable : Unit → read Trains.v_trains Bool
collisions_detectable() ≡
(
    ∀ t1, t2 : T.TrainID, sp1, sp2 : T.Speed,
            s_err1, s_err2, s_col : T.Length •
                sp1 = getTrainMaxSpeed(t1) ∧
                sp2 = getTrainMaxSpeed(t2) ∧
                s_err1 = sp1 ∗ T.tick_interval ∧
                s_err2 = sp2 ∗ T.tick_interval ∧
                s_col = s_err1 + s_err2
                    ⇒
                        s_col < getTrainLength(t1)
)
```

```
    axiom
      [initial]
        initialise post is_wf()

  end
```

SBs

```
context: I_Types0
scheme I_SBs0(T : I_Types0) =
  class
    type
      /* Type of interest */
      SBs = T.SBID -m-> SBData,

      /* Data for each SB */
      SBData == mk_sb(getUpSeg : T.SBSegment,
                      getDownSeg : T.SBSegment,
                      getType : T.SBType,
                      getPointTicks : T.HasTicks,
                      getBarrierTicks : T.HasTicks,
                      getSignalTicks : T.HasTicks)

    variable
      v_SBs : SBs := sbsConf

    value
      sbsConf : SBs,

      /* SB observers */
      getSBSeg : T.SBID × T.Direction -~-> read v_SBs T.SBSegment
      getSBSeg(sb,dir) ≡
        if(dir = T.UP)
        then
          getUpSeg(v_SBs(sb))
        else
          getDownSeg(v_SBs(sb))
        end
      pre sbExistsInConf(sb),

      getSBType : T.SBID -~-> read v_SBs T.SBType
      getSBType(sb) ≡
        getType(v_SBs(sb))
      pre sbExistsInConf(sb),

      getPointTicks : T.SBID -~-> read v_SBs T.Tick
      getPointTicks(sb) ≡
        T.getTicks(getPointTicks(v_SBs(sb)))
      pre getSBType(sb) = T.POINTSB ∧
          sbExistsInConf(sb),

      getBarrierTicks : T.SBID -~-> read v_SBs T.Tick
      getBarrierTicks(sb) ≡
        T.getTicks(getBarrierTicks(v_SBs(sb)))
```

```
pre getSBType(sb) = T.CROSSINGSB ∧
        sbExistsInConf(sb),

getSignalTicks : T.SBID -~-> read v_SBs T.Tick
getSignalTicks(sb) ≡
    T.getTicks(getSignalTicks(v_SBs(sb)))
pre getSBType(sb) = T.CROSSINGSB ∧
        sbExistsInConf(sb),

sbExistsInConf : T.SBID → read v_SBs Bool
sbExistsInConf(sb) ≡
    sb ∈ dom v_SBs,

/∗ Auxiliary functions ∗/

/∗ Returns the direction of a point SB
     from the stem towards the branches ∗/
getPointDir : T.SBID -~-> read v_SBs T.Direction
getPointDir(sb) ≡
    let sbSeg = getSBSeg(sb,T.UP)
    in
        case sbSeg of
            T.point(_,_) → T.UP,
            _ → T.DOWN
        end
    end
pre getSBType(sb) = T.POINTSB,

/∗ Returns the segments around a point ∗/
getSBPointSegs : T.SBID -~-> read v_SBs T.PointSegments
getSBPointSegs(sb) ≡
    let
        dir = getPointDir(sb),
        pointSegs = getSBSeg(sb,dir),
        T.seg(stemSeg) = getSBSeg(sb,T.inverseDir(dir))
    in
        T.pointSegments(stemSeg,
                                      T.getUpSeg(pointSegs),
                                      T.getDownSeg(pointSegs),
                                      dir)
    end
pre getSBType(sb) = T.POINTSB,

/∗ Returns the direction against an ESA from an END SB ∗/
getEndDir : T.SBID -~-> read v_SBs T.Direction
getEndDir(sb) ≡
    case getSBSeg(sb,T.UP) of
        T.esa(_) → T.UP,
        _ → T.DOWN
    end
pre getSBType(sb) = T.ENDSB,

/∗ Invariants ∗/
is_wf : Unit → read v_SBs Bool
is_wf() ≡
    sbsHaveConf() ∧
    getSBSeg_diff() ∧
```

```
    getSBSeg_point_wf() ∧
    getSBSeg_injective() ∧
    getSBSegType_wf(),

/∗ A configuration for each SB must exists ∗/
sbsHaveConf : Unit → read v_SBs Bool
sbsHaveConf() ≡
(
    ∀ sb : T.SBID •
        sbExistsInConf(sb)
),

/∗ The segments next to a SB are different
     in the UP and the DOWN direction.
     I.e. the line is not circular ∗/
getSBSeg_diff : Unit → read v_SBs Bool
getSBSeg_diff() ≡
(
    ∀ sb : T.SBID •
        getSBSeg(sb,T.UP) ≠ getSBSeg(sb,T.DOWN)
),

/∗ The two branches of a junction are different ∗/
getSBSeg_point_wf : Unit → read v_SBs Bool
getSBSeg_point_wf() ≡
(
    ∀ sb : T.SBID,
            seg1,seg2 : T.SegmentID,
            dir : T.Direction •
                T.point(seg1,seg2) = getSBSeg(sb,dir) ⇒
                    seg1 ≠ seg2
),

/∗ Two different SBs have different
     SBSegments in the same direction ∗/
getSBSeg_injective : Unit → read v_SBs Bool
getSBSeg_injective() ≡
(
    ∀ sb1, sb2 : T.SBID,
            dir : T.Direction •
                sb1 ≠ sb2 ⇒
                    getSBSeg(sb1,dir) ≠ getSBSeg(sb2,dir)
),

/∗ The type of a SB must conform
     with the result of getSBSeg ∗/
getSBSegType_wf : Unit → read v_SBs Bool
getSBSegType_wf() ≡
(
    ∀ sb : T.SBID •
        case getSBType(sb) of
            T.ENDSB →
                (∃! dir : T.Direction, esa : T.ESAID •
                    esa = T.dir2End(dir) ∧
                    getSBSeg(sb,dir) = T.esa(esa) ∧
                    (
                        getBarrierTicks(v_SBs(sb)) = T.none ∧
```

```
                    getSignalTicks(v_SBs(sb)) = T.none ∧
                    getPointTicks(v_SBs(sb)) = T.none
                 )),
            T.POINTSB →
               (∃! dir : T.Direction, seg1,seg2 : T.SegmentID •
                    getSBSeg(sb,dir) = T.point(seg1,seg2) ∧
                    (
                       getBarrierTicks(v_SBs(sb)) = T.none ∧
                       getSignalTicks(v_SBs(sb)) = T.none ∧
                       getPointTicks(v_SBs(sb)) ≠ T.none
                    )),
            T.CROSSINGSB →
               ((∀ dir : T.Direction •
                    ∃! seg : T.SegmentID •
                       getSBSeg(sb,dir) = T.seg(seg)
                 ) ∧
                 (
                    getBarrierTicks(v_SBs(sb)) ≠ T.none ∧
                    getSignalTicks(v_SBs(sb)) ≠ T.none ∧
                    getPointTicks(v_SBs(sb)) = T.none
                 )),
            T.PLAINSB →
               ((∀ dir : T.Direction •
                       ∃! seg : T.SegmentID •
                          getSBSeg(sb,dir) = T.seg(seg)
                 ) ∧
                 (
                       getBarrierTicks(v_SBs(sb)) = T.none ∧
                       getSignalTicks(v_SBs(sb)) = T.none ∧
                       getPointTicks(v_SBs(sb)) = T.none
                 ))
         end
      )

   end
```

Segs

```
context: I_Types0
scheme I_Segs0(T : I_Types0) =
   class
      type
         /* Type of interest */
         Segs = SegsData × SegPoints,

         SegsData = T.SegmentID →m SegData,
         SegPoints :: getRP : T.Length
                      getBP : T.Length,

         /* Data for each Segment */
         SegData == mk_seg(getUpSB : T.SBID,
                           getDownSB : T.SBID,
                           getLength : T.Length,
                           getMaxSpeed : T.Speed)
```

```
variable
    v_segs : SegsData := segsDataConf,
    v_points : SegPoints := segPointsConf

value
    segsConf : Segs = (segsDataConf,segPointsConf),
    segsDataConf : SegsData,
    segPointsConf : SegPoints,

    /∗ Segment observers ∗/
    getSegSB : T.SegmentID × T.Direction ~→ read v_segs T.SBID
    getSegSB(seg,dir) ≡
        if(dir = T.UP)
        then
            getUpSB(v_segs(seg))
        else
            getDownSB(v_segs(seg))
        end
    pre segExistsInConf(seg),

    getSegLength : T.SegmentID ~→ read v_segs T.Length
    getSegLength(seg) ≡
        getLength(v_segs(seg))
    pre segExistsInConf(seg),

    getSegMaxSpeed : T.SegmentID ~→ read v_segs T.Speed
    getSegMaxSpeed(seg) ≡
        getMaxSpeed(v_segs(seg))
    pre segExistsInConf(seg),

    segExistsInConf : T.SegmentID → read v_segs Bool
    segExistsInConf(seg) ≡
        seg ∈ dom v_segs,

    /∗ Reservation− and brake−point observers ∗/
    getResPoint : Unit → read v_points T.Length
    getResPoint() ≡
        getRP(v_points),

    getBrakePoint : Unit → read v_points T.Length
    getBrakePoint() ≡
        getBP(v_points),

    /∗ Invariant ∗/
    is_wf : Unit → read v_segs, v_points Bool
    is_wf() ≡
        segsHaveConf() ∧
        getSegSB_injective() ∧
        brakeResPoint_wf(),

    /∗ A configuration for each Segment must exists ∗/
    segsHaveConf : Unit → read v_segs, v_points Bool
    segsHaveConf() ≡
    (
        (∀ seg : T.SegmentID •
            segExistsInConf(seg)) ∧
         getResPoint() > 0.0 ∧
```

```
          getBrakePoint() > 0.0
      ),

      /**
      * The SB in the end of a segment is different
      * for two different segments or they are the
      * same in both direction (being branches)
      **/
      getSegSB_injective : Unit → read v_segs Bool
      getSegSB_injective() ≡
      (
         ∀ seg1, seg2 : T.SegmentID,
               dir : T.Direction •
                  seg1 ≠ seg2 ⇒
                  (
                     getSegSB(seg1,dir) ≠ getSegSB(seg2,dir)
                  )
                     ∨
                  (
                     getSegSB(seg1,T.UP) = getSegSB(seg2,T.UP) ∧
                     getSegSB(seg1,T.DOWN) = getSegSB(seg2,T.DOWN)
                  )
      ),

      /* The reservation−point should be placed before
           the brake−point, i.e. there is a greater
           distance from the end of a segment to the
           reservation−point than to the brake−point */
      brakeResPoint_wf : Unit → read v_points Bool
      brakeResPoint_wf() ≡
         getResPoint() > getBrakePoint()

   end
```

ESAs

```
context: I_Types0
scheme I_ESAs0(T : I_Types0) =
   class
      type
         /* Type of interest */
         ESAs == mk_esa(getLowSB : T.SBID,
                                    getHighSB : T.SBID,
                                    getLowLength : T.Length,
                                    getHighLength : T.Length)

      variable
         v_ESAs : ESAs := esasConf

      value
         esasConf : ESAs,

         /* ESA observers */
         getESASB : T.ESAID ~→ read v_ESAs T.SBID
         getESASB(esa) ≡
```

```
        if(esa = T.LOW)
        then
            getLowSB(v_ESAs)
        else
            getHighSB(v_ESAs)
        end
    pre esaExistsInConf(esa),

    getESALength : T.ESAID ~→ read v_ESAs T.Length
    getESALength(esa) ≡
        if(esa = T.LOW)
        then
            getLowLength(v_ESAs)
        else
            getHighLength(v_ESAs)
        end
    pre esaExistsInConf(esa),

    esaExistsInConf : T.ESAID → read v_ESAs Bool
    esaExistsInConf(esa) ≡
        if(esa = T.LOW)
        then
            getLowLength(v_ESAs) > 0.0
        else
            getHighLength(v_ESAs) > 0.0
        end,

    /* Invariants */
    is_wf : Unit → read v_ESAs Bool
    is_wf() ≡
        esasHaveConf(),

    /* A configuration for each ESA must exists */
    esasHaveConf : Unit → read v_ESAs Bool
    esasHaveConf() ≡
    (
        ∀ esa : T.ESAID •
            esaExistsInConf(esa)
    )

end
```

Trains

```
context: I_Types0
scheme I_Trains0(T : I_Types0) =
    class
        type
            /* Type of interest */
            Trains = T.TrainID →m TData,

            /* Data for each Train */
            TData == mk_train(getLength : T.Length,
                              getMaxSpeed : T.Speed,
                              getMaxAcc : T.Acceleration,
```

```
                                    getMaxDec : T.Acceleration)

    variable
        v_trains : Trains := trainsConf

    value
        trainsConf : Trains,

        /* Train observers */
        getTrainLength : T.TrainID ~→ read v_trains T.Length
        getTrainLength(t) ≡
            getLength(v_trains(t))
        pre trainExistsInConf(t),

        getTrainMaxSpeed : T.TrainID ~→ read v_trains T.Speed
        getTrainMaxSpeed(t) ≡
            getMaxSpeed(v_trains(t))
        pre trainExistsInConf(t),

        getTrainMaxAcc : T.TrainID ~→ read v_trains T.Acceleration
        getTrainMaxAcc(t) ≡
            getMaxAcc(v_trains(t))
        pre trainExistsInConf(t),

        getTrainMaxDec : T.TrainID ~→ read v_trains T.Acceleration
        getTrainMaxDec(t) ≡
            getMaxDec(v_trains(t))
        pre trainExistsInConf(t),

        trainExistsInConf : T.TrainID → read v_trains Bool
        trainExistsInConf(t) ≡
            t ∈ dom v_trains,

        /* Invariants */
        is_wf : Unit → read v_trains Bool
        is_wf() ≡
            trainsHaveConf(),

        /* A configuration for each Train must exists */
        trainsHaveConf : Unit → read v_trains Bool
        trainsHaveConf() ≡
        (
            ∀ train : T.TrainID •
                train ∈ dom v_trains
        )

  end
```

F.4.3 Dynamics

```
context: I_Statics0, I_TrainDyn0, I_SBDyn0, I_Types0
scheme I_Dynamics0(T : I_Types0, S : I_Statics0(T)) =
  class
    object
```

```
    TD : I_TrainDyn0(T,S),
    SD : I_SBDyn0(T,S)

value
    /* Point observer */
    getPointPosition : T.SBID ~→ read S.SBs.v_SBs,
                                    SD.v_SBStates T.PointPosition
    getPointPosition(sbID) ≡
        SD.getPointPosition(sbID)
    pre S.getSBType(sbID) = T.POINTSB,

    getPointTicks : T.SBID ~→ read S.SBs.v_SBs, SD.v_SBStates T.Tick
    getPointTicks(sbID) ≡
        SD.getPointTicks(sbID)
    pre S.getSBType(sbID) = T.POINTSB,

    /* Point generator */
    setPointPosition : T.SBID × T.PointPosition ~→ read S.SBs.v_SBs
                                    write SD.v_SBStates Unit
    setPointPosition(sbID,ppos) ≡
        SD.setPointPosition(sbID,ppos)
    pre S.getSBType(sbID) = T.POINTSB ∧
            ∼trainOnJunction(sbID),

    setPointTicks : T.SBID × T.Tick ~→ read S.SBs.v_SBs
                                    write SD.v_SBStates Unit
    setPointTicks(sbID,tick) ≡
        SD.setPointTicks(sbID,tick)
    pre S.getSBType(sbID) = T.POINTSB,

    /* Crossing observer */
    getBarrierPosition : T.SBID ~→ read SD.v_SBStates T.BarrierPosition
    getBarrierPosition(sbID) ≡
        SD.getBarrierPosition(sbID)
    pre S.getSBType(sbID) = T.CROSSINGSB,

    getSignalStatus : T.SBID ~→ read SD.v_SBStates T.SignalStatus
    getSignalStatus(sbID) ≡
        SD.getSignalStatus(sbID)
    pre S.getSBType(sbID) = T.CROSSINGSB,

    getBarrierTicks : T.SBID ~→ read SD.v_SBStates T.Tick
    getBarrierTicks(sbID) ≡
        SD.getBarrierTicks(sbID)
    pre S.getSBType(sbID) = T.CROSSINGSB,

    getSignalTicks : T.SBID ~→ read SD.v_SBStates T.Tick
    getSignalTicks(sbID) ≡
        SD.getSignalTicks(sbID)
    pre S.getSBType(sbID) = T.CROSSINGSB,

    /* Crossing generator */
    setBarrierPosition : T.SBID × T.BarrierPosition ~→ read S.SBs.v_SBs
                                    write SD.v_SBStates Unit
    setBarrierPosition(sbID,bPos) ≡
        SD.setBarrierPosition(sbID,bPos)
```

```
pre S.getSBType(sbID) = T.CROSSINGSB,

setSignalStatus : T.SBID × T.SignalStatus ~→ read S.SBs.v_SBs
                                             write SD.v_SBStates Unit
setSignalStatus(sbID,sigStat) ≡
   SD.setSignalStatus(sbID,sigStat)
pre S.getSBType(sbID) = T.CROSSINGSB,

setBarrierTicks : T.SBID × T.Tick ~→ read S.SBs.v_SBs
                                     write SD.v_SBStates Unit
setBarrierTicks(sbID,tick) ≡
   SD.setBarrierTicks(sbID,tick)
pre S.getSBType(sbID) = T.CROSSINGSB,

setSignalTicks : T.SBID × T.Tick ~→ read S.SBs.v_SBs
                                    write SD.v_SBStates Unit
setSignalTicks(sbID,tick) ≡
   SD.setSignalTicks(sbID,tick)
pre S.getSBType(sbID) = T.CROSSINGSB,

/∗ Sensor observer ∗/
getSensorStatus : T.SBID ~→ read SD.v_SBStates T.SensorStatus
getSensorStatus(sbID) ≡
   SD.getSensorStatus(sbID),

/∗ Sensor generator ∗/
setSensorStatus : T.SBID × T.SensorStatus ~→ write SD.v_SBStates Unit
setSensorStatus(sbID,senStat) ≡
   SD.setSensorStatus(sbID,senStat)
pre sensor_guard(sbID,senStat),

/∗ Train observer ∗/
getTrainAcc : T.TrainID ~→ read TD.v_TrainStates T.Acceleration
getTrainAcc(tID) ≡
   TD.getTrainAcc(tID),

getTrainSpeed : T.TrainID ~→ read TD.v_TrainStates T.Speed
getTrainSpeed(tID) ≡
   TD.getTrainSpeed(tID),

getTrainPosition : T.TrainID ~→ read TD.v_TrainStates T.TrainPosition
getTrainPosition(tID) ≡
   TD.getTrainPosition(tID),

getTrainDirection : T.TrainID ~→ read TD.v_TrainStates T.Direction
getTrainDirection(tID) ≡
   TD.getTrainDirection(tID),

changeTrainDirection : T.TrainID → write TD.v_TrainStates Unit
changeTrainDirection(t) ≡
   let
      dir = T.inverseDir(getTrainDirection(t)),
      tp = getTrainPosition(t),

      front = T.frontPos(tp),
      rear = T.rearPos(tp),
```

```
        tp = T.mk_TrainPosition(rear,front),

        s = setTrainDirection(t,dir)
    in
        setTrainPosition(t,tp)
    end,

/* Train generator */
setTrainAcc : T.TrainID × T.Acceleration ~→ write TD.v_TrainStates Unit
setTrainAcc(tID,acc) ≡
    (TD.setTrainAcc(tID,acc))
pre acc ≤ S.getTrainMaxAcc(tID) ∧
        S.getTrainMaxDec(tID) ≤ acc,

setTrainSpeed : T.TrainID × T.Speed ~→ write TD.v_TrainStates Unit
setTrainSpeed(tID,speed) ≡
    (TD.setTrainSpeed(tID,speed))
pre speed ≤ S.getTrainMaxSpeed(tID),

setTrainPosition : T.TrainID × T.TrainPosition ~→
                                        write TD.v_TrainStates Unit
setTrainPosition(tID,pos) ≡
    (TD.setTrainPosition(tID,pos))
pre ~trainPositionOccupied(tID,pos) ∧
        ~tpDerailed(pos,getTrainDirection(tID)),

setTrainDirection : T.TrainID × T.Direction ~→ write TD.v_TrainStates Unit
setTrainDirection(tID,dir) ≡
    (TD.setTrainDirection(tID,dir))
pre getTrainSpeed(tID) = 0.0 ∨
        getTrainDirection(tID) = dir,

/* Processes */
tick : T.Tick ~→ read S.any
                             write TD.v_TrainStates, SD.v_SBStates Unit
tick(tick) ≡
        tickPoints(tick);
        tickCrossings(tick);
        tickTrains(tick),

tickPoints : T.Tick ~→ read S.SBs.v_SBs
                               write SD.v_SBStates Unit
tickPoints(tick) ≡
    let
        points = { p | p : T.SBID • S.getSBType(p) = T.POINTSB }
    in
        pointProcess(points,tick)
    end,

pointProcess : T.SBID-set × T.Tick ~→ read S.SBs.v_SBs
                                          write SD.v_SBStates Unit
pointProcess(points,tick) ≡
    while (points ≠ {})
    do
        let
```

```
            p : T.SBID • p ∈ points,
            points = points \ {p},
            s = updatePoint(p,tick)
        in
            pointProcess(points,tick)
        end
    end
pre S.sbsArePoints(points),

updatePoint : T.SBID × T.Tick ~→ read S.SBs.v_SBs
                                    write SD.v_SBStates Unit
updatePoint(p,tick) ≡
    let
        pp = getPointPosition(p)
    in
        case pp of
            T.MOVINGDOWN → movePoint(p,tick,T.DOWN),
            T.MOVINGUP → movePoint(p,tick,T.UP),
            _ → ()
        end
    end
pre S.getSBType(p) = T.POINTSB,

movePoint : T.SBID × T.Tick × T.PointPosition ~→ read S.SBs.v_SBs
                                    write SD.v_SBStates Unit
movePoint(p,tick,pp) ≡
    let
        ticks = S.getPointTicks(p),
        curTick = getPointTicks(p)
    in
        if(curTick ≥ ticks)
        then
            setPointPosition(p,pp);
            setPointTicks(p,0.0)
        else
            setPointTicks(p,curTick+tick)
        end
    end,

tickCrossings : T.Tick ~→ read S.SBs.v_SBs write SD.v_SBStates Unit
tickCrossings(tick) ≡
    let
        crossings = { c | c : T.SBID • S.getSBType(c) = T.CROSSINGSB }
    in
        crossingProcess(crossings,tick)
    end,

crossingProcess : T.SBID-set × T.Tick ~→ read S.SBs.v_SBs
                                    write SD.v_SBStates Unit
crossingProcess(crossings,tick) ≡
    while (crossings ≠ {})
    do
        let
            c : T.SBID • c ∈ crossings,
            crossings = crossings \ {c},
            s = updateCrossing(c,tick)
        in
```

```
            crossingProcess(crossings,tick)
        end
    end
pre S.sbsAreCrossings(crossings),

updateCrossing : T.SBID × T.Tick ~→ read S.SBs.v_SBs
                                    write SD.v_SBStates Unit
updateCrossing(cr,tick) ≡
    let
        bp = getBarrierPosition(cr),
        ss = getSignalStatus(cr),

        bTicks = S.getBarrierTicks(cr),
        newBTicks = tick + getBarrierTicks(cr),

        sTicks = S.getSignalTicks(cr),
        newSTicks = tick + getSignalTicks(cr)

    in
        case bp of
            T.UP →
            (
                if(ss = T.ON)
                then
                    if(newSTicks > sTicks)
                    then
                        setSignalTicks(cr,0.0);
                        setBarrierPosition(cr,T.MOVINGDOWN)
                    else
                        setSignalTicks(cr,newSTicks)
                    end
                end
            ),
            T.MOVINGDOWN →
            (
                if(newBTicks > bTicks)
                then
                    setBarrierTicks(cr,0.0);
                    setBarrierPosition(cr,T.DOWN);
                    setSignalStatus(cr,T.OFF)
                else
                    setBarrierTicks(cr,newBTicks)
                end
            ),
            T.DOWN → (),
            T.MOVINGUP →
            (
                if(newBTicks > bTicks)
                then
                    setBarrierTicks(cr,0.0);
                    setBarrierPosition(cr,T.UP)
                else
                    setBarrierTicks(cr,newBTicks)
                end
            )
        end
    end
```

```
pre S.getSBType(cr) = T.CROSSINGSB,

tickTrains : T.Tick ~→ read S.any write TD.v_TrainStates,
                                    SD.v_SBStates Unit
tickTrains(tick) ≡
let
    trains = { t | t : T.TrainID}
in
    trainProcess(trains,tick)
end,

trainProcess : T.TrainID-set × T.Tick ~→ read S.any
                                  write TD.v_TrainStates, SD.v_SBStates Unit
trainProcess(trains,tick) ≡
    while (trains ≠ {})
    do
        let
            t : T.TrainID • t ∈ trains,
            trains = trains \ {t},
            s = if (∼trainInESA(t)) then updateTrain(t,tick) end
        in
            trainProcess(trains,tick)
        end
    end,

updateTrain : T.TrainID × T.Tick ~→ read S.any
                                  write TD.v_TrainStates, SD.v_SBStates Unit
updateTrain(t,tick) ≡
    let
        dir = getTrainDirection(t),
        acc = getTrainAcc(t),
        curSpeed = getTrainSpeed(t),
        newSpeed = curSpeed + acc∗tick,

        /∗ To avoid negative speed when braking while standing still∗/
        newSpeed = if (newSpeed < 0.0) then 0.0 else newSpeed end,
        s = setTrainSpeed(t,newSpeed),

        tp = getTrainPosition(t),
        deltaProg = curSpeed∗tick + 0.5∗acc∗tick∗tick,
        deltaProg = if (dir = T.UP) then deltaProg else deltaProg ∗ −1.0 end,

        tp2 = if (deltaProg < 0.0) then tp
              else  updateTrainPosition(tp,deltaProg) end
    in
        setTrainPosition(t,tp2);

        /∗ updating sensor ∗/
        if (T.oneLoc(tp) ≠ T.oneLoc(tp2))
        then
            if (∼T.oneLoc(tp2))
            then
                let
                    status = T.ACTIVE,
                    sensorPos = tp2
                in
                    updateSensor(sensorPos,dir,status)
```

```
            end
        else
            let
                status = T.INACTIVE,
                sensorPos = tp
            in
                updateSensor(sensorPos,dir,status)
            end
        end
    end

end,

updateSensor : T.TrainPosition × T.Direction × T.SensorStatus →
                                        read  S.Segs.v_segs, S.ESAs.v_ESAs
                                        write SD.v_SBStates Unit
updateSensor(tp,dir,status) ≡
    let
        /* if on two segments sensor must be set active */
        loc = T.frontLoc(tp)
    in
        case loc of
            T.isSeg(seg) →
            (
                let
                    sb = S.getSegSB(seg,T.inverseDir(dir))
                in
                    setSensorStatus(sb,status)
                end
            ),

            T.isESA(esa) →
            (
                let
                    sb = S.getESASB(esa)
                in
                    setSensorStatus(sb,status)
                end
            )
        end
    end,

updateTrainPosition : T.TrainPosition × T.Length ~→
                                read  S.any, SD.v_SBStates T.TrainPosition
updateTrainPosition(tp,delta) ≡
    let
        curFrontPos = T.frontPos(tp),
        curFrontPos = updateSegPos(curFrontPos,delta),

        curRearPos = T.rearPos(tp),
        curReartPos = updateSegPos(curRearPos,delta)
    in
        T.mk_TrainPosition(curFrontPos, curRearPos)
    end,

updateSegPos : T.SegmentPosition × T.Length ~→
                                read S.any, SD.v_SBStates T.SegmentPosition
```

```
updateSegPos(segPos,delta) ≡
   let
      loc = T.getLoc(segPos),
      curProg = T.getLength(segPos) + delta,
      locLength = S.getLocLength(T.getLoc(segPos))
   in
      if (curProg < 0.0 )
      then
         let
            nextLoc = nextLoc(loc,T.DOWN),
            nextPos = S.getLocLength(nextLoc) + curProg
         in
            T.mk_SegmentPosition(nextLoc,nextPos)
         end
      else
         if (curProg > locLength)
         then
            let
               nextLoc = nextLoc(loc,T.UP),
               nextPos = curProg − locLength
            in
               T.mk_SegmentPosition(nextLoc,nextPos)
            end
         else
            T.mk_SegmentPosition(loc,curProg)
         end
      end
   end,

nextLoc : T.Location × T.Direction ~→ read S.any, SD.v_SBStates T.Location
nextLoc(loc,dir) ≡
   case loc of
      T.isESA(esa) →
      (
         let
            sb = S.getESASB(esa),
            T.seg(aSeg) = S.getSBSeg(sb,dir)
         in
            T.isSeg(aSeg)
         end
      ),

      T.isSeg(aSeg) →
      (
         let
            sb = S.getSegSB(aSeg,dir)
         in
            case S.getSBSeg(sb,dir) of
               T.seg(nextSeg) → T.isSeg(nextSeg),
               T.esa(aESA) → T.isESA(aESA),
               T.point(up,down) → T.isSeg(getSegOfPoint(sb))
            end
         end
      )
   end,

/∗ Returns the front segment of a train. If front is on ESA then
```

```
      the rear segment is returned. This is used for speed checking */
  getTrainLoc : T.TrainID ~→ read TD.v_TrainStates T.Location
  getTrainLoc(t) ≡
    let
      tp = getTrainPosition(t),
      frontLoc = T.getLoc(T.frontPos(tp)),
      rearLoc = T.getLoc(T.rearPos(tp))
    in
      case frontLoc of
        T.isESA(esa) → rearLoc,
        _ → frontLoc
      end
    end
  pre ∼trainInESA(t),

  /* Get the point branch according
      to the point position */
  getSegOfPoint : T.SBID ~→ read S.SBs.v_SBs, SD.v_SBStates T.SegmentID
  getSegOfPoint(sb) ≡
    let
      pp = getPointPosition(sb),
      pointSegs = S.getSBPointSegs(sb)
    in
      case pp of
        T.UP → T.getUpBranch(pointSegs),
        T.DOWN → T.getDownBranch(pointSegs)
      end
    end
  pre (getPointPosition(sb) = T.UP ∨
      getPointPosition(sb) = T.DOWN),

/* Auxiliary functions */
tpDerailed : T.TrainPosition × T.Direction → read S.Segs.v_segs,
                                                S.SBs.v_SBs, SD.v_SBStates Bool
tpDerailed(tp,dir) ≡
  if (∼T.oneLoc(tp) ∧ ∼T.segPosIsESA(T.frontPos(tp))) then
    let
      seg = T.getSeg(T.frontLoc(tp)),
      sb = S.getSegSB(seg,T.inverseDir(dir))
    in
      case S.getSBType(sb) of
          T.POINTSB →
          (
            if (dir = S.getPointDir(sb)) then
              pointConnected(sb,T.getSeg(T.frontLoc(tp)))
            else
              pointConnected(sb,T.getSeg(T.rearLoc(tp)))
            end
          ),

          T.CROSSINGSB →
          (
            getBarrierPosition(sb) = T.DOWN
          ),

          _ → false
      end
```

```
    end
  else
    false
  end,

getESATrains : T.ESAID ~→ read TD.v_TrainStates T.TrainID-set
getESATrains(esa) ≡
  { t | t : T.TrainID • T.trainOnlyOnESA(getTrainPosition(t)) },

getTrainSegments : T.TrainID ~→ read TD.v_TrainStates T.SegmentID-set
getTrainSegments(t) ≡
  TD.getTrainSegments(t),

getTrainBranch : T.TrainID ~→ read S.Segs.v_segs, S.SBs.v_SBs,
                                   TD.v_TrainStates  T.SegmentID
getTrainBranch(t) ≡
(
  let
    seg : T.SegmentID • seg ∈ getTrainSegments(t) ∧
                              S.segIsBranch(seg)
  in
    seg
  end
)
pre (∃ sb : T.SBID • trainOnJunction(t,sb)),

trainOnSegment : T.TrainID × T.SegmentID ~→ read TD.v_TrainStates Bool
trainOnSegment(tID,seg) ≡
  seg ∈ getTrainSegments(tID),

trainOnJunction : T.TrainID × T.SBID ~→ read S.SBs.v_SBs,
                                   TD.v_TrainStates Bool
trainOnJunction(t,sb) ≡
(
  S.getSBType(sb) = T.POINTSB ∧
  trainOnSensor(t,sb)
),

trainOnJunction : T.SBID ~→ read S.SBs.v_SBs, TD.v_TrainStates Bool
trainOnJunction(sb) ≡
  S.getSBType(sb) = T.POINTSB ∧
  trainOnSensor(sb),

trainOnSensor : T.TrainID × T.SBID ~→
                        read S.SBs.v_SBs, TD.v_TrainStates Bool
trainOnSensor(t,sb) ≡
(
  ∃ dir : T.Direction, tPos : T.TrainPosition,
        sp1,sp2 : T.SegmentPosition •
          tPos = getTrainPosition(t) ∧
    T.segPosInSBSeg(sp1, S.getSBSeg(sb,dir)) ∧
    T.segPosInSBSeg(sp2, S.getSBSeg(sb,T.inverseDir(dir)))
),

trainOnSensor : T.SBID ~→ read S.SBs.v_SBs,
                           TD.v_TrainStates Bool
```

```
trainOnSensor(sb) ≡
(
    ∃ t : T.TrainID, dir : T.Direction, tPos : T.TrainPosition,
                sp1,sp2 : T.SegmentPosition •
                    tPos = getTrainPosition(t) ∧
        T.segPosInSBSeg(sp1, S.getSBSeg(sb,dir)) ∧
        T.segPosInSBSeg(sp2, S.getSBSeg(sb,T.inverseDir(dir)))
),

trainInESA : T.TrainID ~→ read TD.v_TrainStates Bool
trainInESA(t) ≡
    TD.trainInESA(t),

trainInESADrivingOut : T.TrainID ~→ read TD.v_TrainStates Bool
trainInESADrivingOut(t) ≡
    TD.trainInESADrivingOut(t),

trainFrontInESA : T.TrainID ~→ read TD.v_TrainStates Bool
trainFrontInESA(t) ≡
    TD.trainFrontInESA(t),

/* Telling if a train is (partly) on a single line */
trainOnSingleLine : T.TrainID ~→ read S.Segs.v_segs, S.SBs.v_SBs,
                                        TD.v_TrainStates Bool
trainOnSingleLine(t) ≡
    let
        tPos = getTrainPosition(t),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∃ s : T.SegmentID •
                    s ∈ segSet ⇒
                        ∼S.segIsBranch(s)
            )
        else
            false
        end
    end,

/* Telling if a train is (partly) on a branch */
trainOnBranch : T.TrainID ~→ read S.Segs.v_segs, S.SBs.v_SBs,
                                    TD.v_TrainStates Bool
trainOnBranch(t) ≡
    let
        tPos = getTrainPosition(t),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∃ s : T.SegmentID •
                    s ∈ segSet ⇒
                        S.segIsBranch(s)
            )
        else
            false
```

```
        end
    end,

/* Telling if a train is only on a branch */
trainOnlyOnBranch : T.TrainID ~→ read S.Segs.v_segs, S.SBs.v_SBs,
                                               TD.v_TrainStates Bool
trainOnlyOnBranch(t) ≡
    let
        tPos = getTrainPosition(t),
        segSet = T.trainPosSegs(tPos)
    in
        if (segSet ≠ {}) then
            (
                ∀ s : T.SegmentID •
                    s ∈ segSet ⇒
                        S.segIsBranch(s)
            )
        else
            false
        end
    end,

pointConnected : T.SBID × T.SegmentID ~→
                                   read S.SBs.v_SBs, SD.v_SBStates Bool
pointConnected(sbID,seg) ≡
    let
        pointSegs = S.getSBPointSegs(sbID)
    in
        case getPointPosition(sbID) of
            T.UP → (seg = T.getUpBranch(pointSegs)),
            T.DOWN → (seg = T.getDownBranch(pointSegs)),
            _ → false
        end
    end
pre S.getSBType(sbID) = T.POINTSB,

trainFrontLoc : T.TrainID ~→ write TD.v_TrainStates T.Location
trainFrontLoc(t) ≡
    TD.trainFrontLoc(t),


sensor_guard : T.SBID × T.SensorStatus ~→ read S.SBs.v_SBs,
                                                   TD.v_TrainStates Bool
sensor_guard(sen,ss) ≡
    (ss = T.ACTIVE ∧ trainOnSensor(sen)) ∨
    (ss = T.INACTIVE ∧ ∼trainOnSensor(sen)),

decelerateTrain : T.TrainID ~→ read S.Trains.v_trains
                                        write  TD.v_TrainStates Unit
decelerateTrain(t) ≡
    if(getTrainSpeed(t) ≠ 0.0)
    then
        let
            maxDec = S.getTrainMaxDec(t),
            curDec = getTrainAcc(t)
        in
```

```
            if(maxDec ≠ curDec)
            then
               setTrainAcc(t,maxDec)
            end
         end
      else
         setTrainAcc(t,0.0)
      end,

accelerateTrain : T.TrainID ~→ read S.Trains.v_trains
                                    write  TD.v_TrainStates Unit
accelerateTrain(tID) ≡
   setTrainAcc(tID,S.getTrainMaxAcc(tID)),

commonSegs : T.TrainPosition × T.TrainID ~→ read TD.v_TrainStates
                                                 T.SegmentID-set
commonSegs(tp1,t2) ≡
   T.trainPosSegs(tp1) ∩ getTrainSegments(t2),

commonSegs : T.TrainID × T.TrainID ~→ read TD.v_TrainStates
                                          T.SegmentID-set
commonSegs(t1,t2) ≡
   getTrainSegments(t1) ∩ getTrainSegments(t2),

trainPositionOccupied : T.TrainID × T.TrainPosition ~→
                                    read S.Segs.v_segs, S.SBs.v_SBs,
                                         TD.v_TrainStates  Bool
trainPositionOccupied(t1,tp1) ≡
(
   ∀ segs : T.SegmentID-set, dir1,dir2 : T.Direction,
         tp1,tp2 : T.TrainPosition •
            ∃ t2 : T.TrainID •
               t2 ≠ t1 ∧
               segs = commonSegs(tp1,t2) ∧
               segs ≠ {} ∧
               (dir1,dir2) = (getTrainDirection(t1),getTrainDirection(t2)) ∧
               tp2 = getTrainPosition(t2) ∧

               case dir1 of
                  T.UP →
                  (
                     if (dir1 = dir2)
                     then
                        S.segPosLower(T.frontPos(tp1),T.frontPos(tp2)) ⇒
                           ∼S.segPosLower(T.frontPos(tp1),T.rearPos(tp2))
                     else
                        ∼S.segPosLower(T.frontPos(tp1),T.frontPos(tp2))
                     end
                  ),

                  T.DOWN →
                  (
                     if (dir1 = dir2) then
                        ∼S.segPosLower(T.frontPos(tp1),T.frontPos(tp2)) ⇒
                           S.segPosLower(T.frontPos(tp1),T.rearPos(tp2))
                     else
                        S.segPosLower(T.frontPos(tp1),T.frontPos(tp2))
```

```
                    end
                  )
                end
),

/* Invariants etc. */

/* Telling if the railway line is safe */
safe : Unit ~→ read S.any, TD.v_TrainStates, SD.v_SBStates Bool
safe() ≡
   is_wf() ∧
   noCollisions() ∧
   trainPosPossible() ∧
   pointsSafe() ∧
   crossingsSafe(),

/**
*  The position of a train may not overlap
*  with the position of other trains
**/
noCollisions : Unit ~→ read S.Segs.v_segs, S.SBs.v_SBs,
                              TD.v_TrainStates Bool
noCollisions() ≡
(
   ∀ t : T.TrainID •
      ~trainPositionOccupied(t,getTrainPosition(t))
),

/**
*  Trains cannot end up on same segment
*  driving in opposite directions away from each other.
*
*  If two train are on same segment driving in opposite
*  directions then the train driving up must be lower
*  on the line than the train driving down.
**/
trainPosPossible : Unit ~→ read S.Segs.v_segs, S.SBs.v_SBs,
                                          TD.v_TrainStates Bool
trainPosPossible() ≡
(
   ∀ t1,t2 : T.TrainID, segs : T.SegmentID-set,
         tp1,tp2 : T.TrainPosition, seg : T.SegmentID •
            commonSegs(t1,t2) ≠ {} ∧
            (tp1,tp2) = (getTrainPosition(t1),getTrainPosition(t1)) ∧
            getTrainDirection(t1) ≠ getTrainDirection(t2) ∧
            getTrainDirection(t1) = T.UP
               ⇒
                  S.segPosLower(T.frontPos(tp1),T.frontPos(tp2))
),

/**
*  If the train is located upon a junction,
*  the point must be connected to the
*  branch, on which the train is located
**/
pointsSafe : Unit ~→ read S.Segs.v_segs, S.SBs.v_SBs,
                              TD.v_TrainStates, SD.v_SBStates Bool
```

```
        pointsSafe() ≡
        (
            ∀ sb : T.SBID, t : T.TrainID, seg : T.SegmentID •
                trainOnJunction(t,sb) ∧
                trainOnSegment(t,seg) ∧
                S.segIsBranch(seg) ⇒
                    pointConnected(sb,seg)
        ),

        /∗ When a train is located on a crossing
             the barriers must be down ∗/
        crossingsSafe : Unit ~→ read S.SBs.v_SBs, TD.v_TrainStates,
                                                        SD.v_SBStates Bool
        crossingsSafe() ≡
        (
            ∀ sb : T.SBID •
                S.getSBType(sb) = T.CROSSINGSB ∧
                trainOnSensor(sb) ⇒
                    getBarrierPosition(sb) = T.DOWN
        ),

        /∗ Wellformedness ∗/
        is_wf : Unit ~→ read S.any, TD.v_TrainStates, SD.v_SBStates Bool
        is_wf() ≡
            S.is_wf() ∧
            TD.is_wf() ∧
            SD.is_wf(),

        init_req : Unit ~→ read S.any, TD.v_TrainStates, SD.v_SBStates Bool
        init_req() ≡
            is_wf() ∧
            TD.init_req() ∧
            SD.init_req()

        axiom
        [Initial_state]
            initialise post init_req()

    end
```

TrainDyn

```
context: I_Types0, I_Statics0
scheme I_TrainDyn0(T : I_Types0, S : I_Statics0(T)) =
    class
        type
            TrainStates = T.TrainID →m TrainState,

            TrainState == mk_tState(
                                    getTAcc : T.Acceleration ↔ setTAcc,
                                    getTSpeed : T.Speed ↔ setTSpeed,
                                    getTPos : T.TrainPosition ↔ setTPos,
                                    getTDir : T.Direction ↔ setTDir)
        variable
            v_TrainStates : TrainStates := initTrainStates
```

```
value
  initTrainStates : TrainStates,

  /* Train observer */
  getTrainAcc : T.TrainID → read v_TrainStates T.Acceleration
  getTrainAcc(t) ≡
    getTAcc(v_TrainStates(t)),

  getTrainSpeed : T.TrainID → read v_TrainStates T.Speed
  getTrainSpeed(t) ≡
    getTSpeed(v_TrainStates(t))
  pre trainStateExists(t),

  getTrainPosition : T.TrainID →
                                    read v_TrainStates T.TrainPosition
  getTrainPosition(t) ≡
    getTPos(v_TrainStates(t))
  pre trainStateExists(t),

  getTrainDirection : T.TrainID →
                                      read v_TrainStates T.Direction
  getTrainDirection (t) ≡
    getTDir(v_TrainStates(t))
  pre trainStateExists(t),

  /* Train generator */
  setTrainAcc : T.TrainID × T.Acceleration →
                                          write v_TrainStates Unit
  setTrainAcc(t,acc) ≡
    v_TrainStates := v_TrainStates †
                                    [ t ↦ setTAcc(acc,v_TrainStates(t)) ]
  pre acc ≤ S.getTrainMaxAcc(t) ∧
        S.getTrainMaxDec(t) ≤ acc ∧
        trainStateExists(t),

  setTrainSpeed : T.TrainID × T.Speed →
                                          write v_TrainStates Unit
  setTrainSpeed(t,speed) ≡
    v_TrainStates := v_TrainStates †
                                [ t ↦ setTSpeed(speed,v_TrainStates(t)) ]
  pre speed ≤ S.getTrainMaxSpeed(t) ∧
        trainStateExists(t),

  setTrainPosition : T.TrainID × T.TrainPosition →
                                          write v_TrainStates Unit
  setTrainPosition(t,tPos) ≡
    v_TrainStates := v_TrainStates †
                                  [ t ↦ setTPos(tPos,v_TrainStates(t)) ]
  pre train_pos_ok(t,tPos) ∧
        trainStateExists(t),

  setTrainDirection : T.TrainID × T.Direction ~→
                                          write v_TrainStates Unit
  setTrainDirection(t,dir) ≡
    v_TrainStates := v_TrainStates †
                                    [ t ↦ setTDir(dir,v_TrainStates(t)) ]
```

```
pre getTrainSpeed(t) = 0.0 ∨
      getTrainDirection(t) = dir,

/∗ Tells if a train has a state in the system ∗/
trainStateExists : T.TrainID → read v_TrainStates Bool
trainStateExists(t) ≡
   t ∈ dom(v_TrainStates),

/∗ Front and rear position of a train
    must be exactly 'train length' apart ∗/
train_pos_ok : T.TrainID × T.TrainPosition ~→
                              read S.any, v_TrainStates Bool
train_pos_ok(t,tp) ≡
(
   let
      T.mk_TrainPosition(posFront,posRear) = tp
   in
      (S.distance(posFront,posRear) = S.getTrainLength(t)) ∧
      train_pos_dir_ok(getTrainDirection(t),tp)
   end
),

/∗ If train drives UP then rear pos
    must be lower than front pos
    and vice versa ∗/
train_pos_dir_ok : T.Direction × T.TrainPosition →
                                       read S.any Bool
train_pos_dir_ok(dir,tp) ≡
(
   case dir of
      T.UP →
      (
         S.segPosLower(T.rearPos(tp),T.frontPos(tp))
      ),

      T.DOWN →
      (
         S.segPosLower(T.frontPos(tp),T.rearPos(tp))
      )
   end
),

getTrainSegments : T.TrainID →
                           read v_TrainStates T.SegmentID-set
getTrainSegments(t) ≡
   T.trainPosSegs(getTrainPosition(t)),

trainInESA : T.TrainID ~→ read v_TrainStates Bool
trainInESA(t) ≡
   T.trainOnlyOnESA(getTrainPosition(t)),

trainInESADrivingOut : T.TrainID ~→
                                read v_TrainStates Bool
trainInESADrivingOut(t) ≡
   if(∼T.trainOnlyOnESA(getTrainPosition(t)))
   then
      false
```

```
    else
        let
            segPos = T.frontPos(getTrainPosition(t)),
            esa = T.getESA(T.getLoc(segPos)),
            dir = getTrainDirection(t)
        in
            T.end2Dir(esa) ≠ dir
        end
    end,

trainFrontInESA : T.TrainID → read v_TrainStates Bool
trainFrontInESA(t) ≡
    let
        tPos = getTrainPosition(t)
    in
        T.segPosIsESA(T.frontPos(tPos))
    end,

trainFrontLoc : T.TrainID → write v_TrainStates T.Location
trainFrontLoc(t) ≡
    case T.frontLoc(getTrainPosition(t)) of
        T.isESA(_) → T.rearLoc(getTrainPosition(t)),
        T.isSeg(seg) → T.isSeg(seg)
    end,

is_wf : Unit → read S.any, v_TrainStates Bool
is_wf() ≡
    allTrainStatesExist() ∧
    train_pos_wf(),

/* All trains must have a state */
allTrainStatesExist : Unit → read v_TrainStates Bool
allTrainStatesExist() ≡
(
    ∀ trainID : T.TrainID •
        trainStateExists(trainID)
),

/* Front and rear position of a train must be exactly
    'train length' apart */
train_pos_wf : Unit ~→ read S.any, v_TrainStates Bool
train_pos_wf() ≡
(
    ∀ t : T.TrainID •
        train_pos_ok(t,getTrainPosition(t))
),

init_req : Unit → read v_TrainStates Bool
init_req() ≡
    allTrainsInESA() ∧
    allTrainsStopped() ∧
    allTrainsFacingLine(),

allTrainsInESA : Unit → read v_TrainStates Bool
allTrainsInESA() ≡
(
    ∀ t : T.TrainID •
```

```
            trainInESA(t)
        ),

        allTrainsStopped : Unit → read v_TrainStates Bool
        allTrainsStopped() ≡
        (
            ∀ t : T.TrainID •
                getTrainSpeed(t) = 0.0 ∧
                getTrainAcc(t) = 0.0
        ),

        allTrainsFacingLine : Unit → read v_TrainStates Bool
        allTrainsFacingLine() ≡
        (
            ∀ t : T.TrainID, esa : T.ESAID •
                T.isESA(esa) = T.getLoc(T.frontPos(
                                        getTrainPosition(t))) ∧
                (esa = T.LOW ⇒ getTrainDirection(t) = T.UP) ∧
                (esa = T.HIGH ⇒ getTrainDirection(t) = T.DOWN)
        )

    end
```

SBDyn

```
context: I_Types0, I_Statics0
scheme I_SBDyn0(T : I_Types0, S : I_Statics0(T)) =
    class
        type
            SBStates = T.SBID →m SBState,

            SBState == mk_sbState(
                            getPP : T.HasPointPosition ↔ setPP,
                            getPTicks : T.HasTicks ↔ setPTicks,
                            getBP : T.HasBarrierPosition ↔ setBP,
                            getSignal : T.HasSignalStatus ↔ setSignal,
                            getBTicks : T.HasTicks ↔ setBTicks,
                            getSTicks : T.HasTicks ↔ setSTicks,
                            getSensor : T.SensorStatus ↔ setSensor)
        variable
            v_SBStates : SBStates := initSBStates

        value
            initSBStates : SBStates,

            /* Point observer */
            getPointPosition : T.SBID ~→ read v_SBStates T.PointPosition
            getPointPosition(p) ≡
                T.getPos(getPP(v_SBStates(p)))
            pre S.getSBType(p) = T.POINTSB ∧
                    pointStateExists(p),

            getPointTicks : T.SBID ~→ read v_SBStates T.Tick
            getPointTicks(p) ≡
                T.getTicks(getPTicks(v_SBStates(p)))
```

```
pre S.getSBType(p) = T.POINTSB ∧
      pointStateExists(p),

/∗ Point generator ∗/
setPointPosition : T.SBID × T.PointPosition ~→
                                   write v_SBStates Unit
setPointPosition(p,pp) ≡
   v_SBStates := v_SBStates †
                [ p ↦ setPP(T.pointPos(pp),v_SBStates(p)) ]
pre S.getSBType(p) = T.POINTSB ∧
      pointStateExists(p),

setPointTicks : T.SBID × T.Tick ~→ write v_SBStates Unit
setPointTicks(p,tick) ≡
   v_SBStates := v_SBStates †
              [ p ↦ setPTicks(T.ticks(tick),v_SBStates(p)) ]
pre S.getSBType(p) = T.POINTSB ∧
      pointStateExists(p),

/∗ Crossing observer ∗/
getBarrierPosition : T.SBID ~→
                          read v_SBStates T.BarrierPosition
getBarrierPosition(cr) ≡
   T.getPos(getBP(v_SBStates(cr)))
pre S.getSBType(cr) = T.CROSSINGSB ∧
      crossingStateExists(cr),

getSignalStatus : T.SBID ~→ read v_SBStates T.SignalStatus
getSignalStatus(cr) ≡
   T.getStatus(getSignal(v_SBStates(cr)))
pre S.getSBType(cr) = T.CROSSINGSB ∧
      crossingStateExists(cr),

getBarrierTicks : T.SBID ~→ read v_SBStates T.Tick
getBarrierTicks(cr) ≡
   T.getTicks(getBTicks(v_SBStates(cr)))
pre S.getSBType(cr) = T.CROSSINGSB ∧
      crossingStateExists(cr),

getSignalTicks : T.SBID ~→ read v_SBStates T.Tick
getSignalTicks(cr) ≡
   T.getTicks(getSTicks(v_SBStates(cr)))
pre S.getSBType(cr) = T.CROSSINGSB ∧
      crossingStateExists(cr),

/∗ Crossing generator ∗/
setBarrierPosition : T.SBID × T.BarrierPosition ~→
                                   write v_SBStates Unit
setBarrierPosition(cr,bp) ≡
   v_SBStates := v_SBStates †
                [ cr ↦ setBP(T.barrierPos(bp),v_SBStates(cr)) ]
pre S.getSBType(cr) = T.CROSSINGSB ∧
      crossingStateExists(cr),

setSignalStatus : T.SBID × T.SignalStatus ~→
                                   write v_SBStates Unit
```

```
setSignalStatus(cr,ss) ≡
    v_SBStates := v_SBStates †
                [ cr ↦ setSignal(T.signalStatus(ss),v_SBStates(cr)) ]
pre S.getSBType(cr) = T.CROSSINGSB ∧
        crossingStateExists(cr),

setBarrierTicks : T.SBID × T.Tick ~→ write v_SBStates Unit
setBarrierTicks(cr,tick) ≡
    v_SBStates := v_SBStates †
                [ cr ↦ setBTicks(T.ticks(tick),v_SBStates(cr)) ]
pre S.getSBType(cr) = T.CROSSINGSB ∧
        crossingStateExists(cr),

setSignalTicks : T.SBID × T.Tick ~→ write v_SBStates Unit
setSignalTicks(cr,tick) ≡
    v_SBStates := v_SBStates †
                [ cr ↦ setSTicks(T.ticks(tick),v_SBStates(cr)) ]
pre S.getSBType(cr) = T.CROSSINGSB ∧
        crossingStateExists(cr),

/* Sensor observer */
getSensorStatus : T.SBID ~→ read v_SBStates T.SensorStatus
getSensorStatus(sen) ≡
    getSensor(v_SBStates(sen))
pre sensorStateExists(sen),

/* Sensor generator */
setSensorStatus : T.SBID × T.SensorStatus ~→
                                        write v_SBStates Unit
setSensorStatus(sen,ss) ≡
    v_SBStates := v_SBStates †
                        [ sen ↦ setSensor(ss,v_SBStates(sen)) ]
pre sensorStateExists(sen),

/* Tells if a sensor has a state in the system */
sensorStateExists : T.SBID → read v_SBStates Bool
sensorStateExists(sb) ≡
    sb ∈ dom(v_SBStates),

/* Tells if a crossing has a state in the system */
crossingStateExists : T.SBID ~→ read v_SBStates Bool
crossingStateExists(sb) ≡
    let state = v_SBStates(sb) in
        getBP(state) ≠ T.none ∧
        getBTicks(state) ≠ T.none ∧
        getSignal(state) ≠ T.none ∧
        getSTicks(state) ≠ T.none
    end
pre sensorStateExists(sb) ∧
        S.getSBType(sb) = T.CROSSINGSB,

/* Tells if a point has a state in the system */
pointStateExists : T.SBID ~→ read v_SBStates Bool
pointStateExists(sb) ≡
    let state = v_SBStates(sb) in
        getPP(state) ≠ T.none ∧
```

```
        getPTicks(state) ≠ T.none
    end
pre sensorStateExists(sb) ∧
        S.getSBType(sb) = T.POINTSB,

/* Invariants */

is_wf : Unit ~→ read S.SBs.v_SBs, v_SBStates Bool
is_wf() ≡
    allCrossingStatesExist() ∧
    allPointStatesExist() ∧
    allSensorStatesExist(),

/* All crossings must have a state */
allCrossingStatesExist : Unit ~→
                            read S.SBs.v_SBs, v_SBStates Bool
allCrossingStatesExist() ≡
(
    ∀ cr : T.SBID •
        S.getSBType(cr) = T.CROSSINGSB ⇒
            crossingStateExists(cr)
),

/* All points must have a state */
allPointStatesExist : Unit ~→
                            read S.SBs.v_SBs, v_SBStates Bool
allPointStatesExist() ≡
(
    ∀ p : T.SBID •
        S.getSBType(p) = T.POINTSB ⇒
            pointStateExists(p)
),

/* All sensors must have a state */
allSensorStatesExist : Unit ~→
                            read S.SBs.v_SBs, v_SBStates Bool
allSensorStatesExist(s) ≡
(
    ∀ sen : T.SBID •
        sensorStateExists(sen)
),

init_req : Unit ~→ read S.SBs.v_SBs, v_SBStates Bool
init_req() ≡
    allBarriersUp() ∧
    allPointsNotShifting(),

allBarriersUp : Unit ~→ read S.SBs.v_SBs, v_SBStates Bool
allBarriersUp() ≡
(
    ∀ sb : T.SBID •
        S.getSBType(sb) = T.CROSSINGSB ⇒
            getBarrierPosition(sb) = T.UP
),

allPointsNotShifting : Unit ~→
```

```
                                  read S.SBs.v_SBs, v_SBStates Bool
        allPointsNotShifting() ≡
        (
           ∀ sb : T.SBID •
              S.getSBType(sb) = T.POINTSB ⇒
                 getPointPosition(sb) ∈ { T.UP, T.DOWN }
        )

  end
```

F.4.4 Control

```
context: I_Dynamics0, I_ComService0, I_SBCC0, I_TCC0
scheme I_Control0(T : I_Types0, S : I_Statics0(T),
                          D : I_Dynamics0(T,S)) =
  class
     type
        SBCCIndex = {| n : Nat • n > 0 ∧ n ≤ card T.sbIDSet |},
        TCCIndex = {| n : Nat • n > 0 ∧ n ≤ card T.trainIDSet |}

     object
        COM : I_ComService0(T),
        SBCC[n : SBCCIndex] : I_SBCC0(T,S,D,COM),
        TCC[n : TCCIndex] : I_TCC0(T,S,D,COM)

     value
        sbIndex : T.SBID →m Nat,
        tIndex : T.TrainID →m Nat,

        sbccStateExists : T.SBID → Bool
        sbccStateExists(sb) ≡
           sb ∈ dom sbIndex,

        tccStateExists : T.TrainID → Bool
        tccStateExists(t) ≡
           t ∈ dom tIndex,

        /* Processes */
        tick : T.Tick ~→ read S.any, {TCC[t].v_tccRes | t : Nat}
                              write {SBCC[t].any | t : Nat}, D.any,
                                 {TCC[t].any | t : Nat}
                              out COM.comChannel Unit
        tick(tick) ≡
        (
           tickTCCs(T.trainIDSet,tick);
           tickSBCCs(T.sbIDSet,tick); ()
        ),

        tickSBCCs : T.SBID-set × T.Tick ~→
                           read S.any, {TCC[t].v_tccRes | t : Nat}
                           write {SBCC[t].any | t : Nat}, D.any
                           out COM.comChannel Unit
        tickSBCCs(sbSet,tick) ≡
           while(sbSet ≠ {})
           do
```

```
        let
            sb : T.SBID • sb ∈ sbSet,
            sbSet = sbSet \ {sb}
        in
            SBCC[ sbIndex(sb) ].sbccProcess(sb,tick)
        end
    end,

tickTCCs : T.TrainID-set × T.Tick ~→
                              read S.any, {TCC[ t ].v_tccRes | t : Nat}
                              write D.any, {TCC[ t ].any | t : Nat}
                              out COM.comChannel Unit
tickTCCs(tSet,tick) ≡
    while(tSet = {})
    do
        let
            t : T.TrainID • t ∈ tSet,
            tSet = tSet \ {t}
        in
            TCC[ tIndex(t) ].tccProcess(t,tick)
        end
    end,

/* Communication */
comService : Unit ~→ write {SBCC[ t ].any | t : Nat},
                           {TCC[ t ].any | t : Nat}
                        in COM.comChannel Unit
comService() ≡
    let
        comMsg = COM.getMsg()
    in
        case T.getReceiver(comMsg) of
            T.isSB(sb) →
            (
                SBCC[ sbIndex(sb) ].msgReceiver(comMsg); ()
            ),
            T.isTrain(t) →
            (
                TCC[ tIndex(t) ].tccMsgReceiver(comMsg); ()
            )
        end
    end,

/* Invariants */
is_wf : Unit → read S.any, D.any, {TCC[ t ].any | t : Nat},
                         {SBCC[ t ].any | t : Nat}
                          out COM.comChannel Bool
is_wf() ≡
    tcc_has_state() ∧
    sbcc_has_state(),

tcc_has_state : Unit → Bool
tcc_has_state() ≡
(
    ∀ t : T.TrainID •
        tccStateExists(t)
),
```

```
sbcc_has_state : Unit → Bool
sbcc_has_state() ≡
(
    ∀ sb : T.SBID •
        sbccStateExists(sb)
),

/**
* Defines that the control system and all its
* components must be consistent e.g. the information
* stored in the control system must reflect the
* physical world and unintended states must not occur.
*
* Also the physical world must abide by the
* rules of the control system.
**/
consistent : Unit → read S.any, D.any,
                                   {TCC[t].any | t : Nat},
                                   {SBCC[t].any | t : Nat}
                                  out COM.comChannel Bool
consistent() ≡
    is_wf() ∧
    train_on_branch_dir() ∧
    tcc_hasRes_passedResPoint() ∧
    sbcc_res_wf() ∧
    position_branch_sbcc_res_wf() ∧
    tcc_res_branch_wf() ∧
    position_sl_sbcc_res_wf(),

/* When a train is on a branch segment
     the driving direction equals
     the driving direction of the train */
train_on_branch_dir : Unit → read S.any, D.any,
                                   {TCC[t].any | t : Nat},
                                   {SBCC[t].any | t : Nat}
                                   out COM.comChannel Bool
train_on_branch_dir() ≡
(
    ∀ t : T.TrainID, seg : T.SegmentID •
        D.trainOnBranch(t) ∧
        D.trainOnSegment(t,seg) ∧
        S.segIsBranch(seg) ⇒
            S.branchDir(seg) = D.getTrainDirection(t)
),

/* If a train has a reservation then
     it has passed the reservation point
     on the given segment */
tcc_hasRes_passedResPoint : Unit → read S.any, D.any,
                                        {TCC[t].any | t : Nat},
                                        {SBCC[t].any | t : Nat}
                                        out COM.comChannel Bool
tcc_hasRes_passedResPoint() ≡
(
    ∀ t : T.TrainID •
            TCC[tIndex(t)].hasTCCRes() ⇒
```

```
                    TCC[ tIndex(t) ].hasPassedResPoint(t)
),

/∗ Only POINTSB and ENDSB may have line reservations
     Only POINTSB may have branch reservations ∗/
sbcc_res_wf : Unit → read S.any, D.any,
                                        {TCC[ t ].any | t : Nat},
                                        {SBCC[ t ].any | t : Nat}
                                        out COM.comChannel Bool
sbcc_res_wf() ≡
(
    ∀ sb : T.SBID,
            lineRes, branchRes : T.HasRes •
        (
            ∼S.isLineGuard(sb) ∧
            lineRes = SBCC[ sbIndex(sb) ].getLineRes() ∧
            branchRes = SBCC[ sbIndex(sb) ].getBranchRes()
                ⇒
                    {lineRes} ∪ {branchRes} = {T.noRes}
        )
            ∧
        (
            S.getSBType(sb) = T.ENDSB
                ⇒
                    SBCC[ sbIndex(sb) ].getBranchRes() = T.noRes
        )
),

/∗ When a train is on a branch segment
     it must have a branch reservation
     in the SB behind ∗/
position_branch_sbcc_res_wf : Unit →
                                read S.any, D.any, {TCC[ t ].any | t : Nat},
                                    {SBCC[ t ].any | t : Nat}
                                out COM.comChannel Bool
position_branch_sbcc_res_wf() ≡
(
    ∀ t : T.TrainID,
            sb : T.SBID,
            tDir : T.Direction,
            seg : T.SegmentID •
                tDir = D.getTrainDirection(t) ∧
                D.trainOnSegment(t,seg) ∧
                D.trainOnBranch(t) ∧
                S.segIsBranch(seg) ∧
                sb = S.getSegSB(seg,T.inverseDir(tDir))
                    ⇒
                        SBCC[ sbIndex(sb) ].getBranchRes() =
                                                T.res(T.mk_res(t,tDir))
),

/∗ If a train is (only) on a branch and has
     reservation then the SB in front of it
     and the other guard has a reservation
     for that train in that direction ∗/
tcc_res_branch_wf : Unit  → read S.any, D.any,
                                        {TCC[ t ].any | t : Nat},
```

```
                                        {SBCC[t].any | t : Nat}
                                        out COM.comChannel Bool
tcc_res_branch_wf() ≡
(
   ∀ tID : T.TrainID,
         seg : T.SegmentID,
         trainDir : T.Direction,
         guard1,guard2 : T.SBID,
         res : T.Reservation •
            D.trainOnSegment(tID,seg) ∧
            D.trainOnlyOnBranch(tID) ∧
            TCC[tIndex(tID)].hasTCCRes() ∧
            trainDir = D.getTrainDirection(tID) ∧
            guard1 = S.getSegSB(seg,T.inverseDir(trainDir)) ∧
            guard2 = S.getSingleLineGuard(guard1,trainDir) ∧
            res = T.mk_res(tID,trainDir)
               ⇒
               T.res(res) = SBCC[sbIndex(guard1)].getLineRes() ∧
               T.res(res) = SBCC[sbIndex(guard2)].getLineRes() ∧
               (S.getSBType(guard2) = T.POINTSB ⇒
                  T.res(res) = SBCC[sbIndex(guard2)].getBranchRes())
),

/* When a train is on a single line it must
    have a reservation in both guards with
    the appropriate direction + a branch
reservation if driving to a point */
position_sl_sbcc_res_wf : Unit →  read S.any, D.any,
                                        {TCC[t].any | t : Nat},
                                       {SBCC[t].any | t : Nat}
                                        out COM.comChannel Bool
position_sl_sbcc_res_wf() ≡
(
   ∀ t : T.TrainID,
         seg : T.SegmentID,
         sb1,sb2 : T.SBID,
         dir : T.Direction,
         res : T.Reservation •
            dir = D.getTrainDirection(t) ∧
            D.trainOnSegment(t,seg) ∧
            S.segIsLineSegment(seg) ∧
            sb1 = S.getSingleLineGuard(seg,T.inverseDir(dir)) ∧
            sb2 = S.getSingleLineGuard(seg,dir) ∧
            res = T.mk_res(t,dir) ⇒
                  T.res(res) = SBCC[sbIndex(sb1)].getLineRes() ∧
                  T.res(res) = SBCC[sbIndex(sb2)].getLineRes() ∧
                  (S.getSBType(sb2) = T.POINTSB ⇒
                     T.res(res) = SBCC[sbIndex(sb2)].getBranchRes())
),

initReq : Unit →  read S.any, D.any, {TCC[t].any | t : Nat},
                                       {SBCC[t].any | t : Nat}
                                        out COM.comChannel Bool
initReq() ≡
   is_wf()

end
```

TCC

```
context: I_Dynamics0, I_ComService0
scheme I_TCC0(T : I_Types0, S : I_Statics0(T),
                    D : I_Dynamics0(T,S), COM : I_ComService0(T)) =
  class
    type
      TCCState :: hasTCCRes : Bool
                  isTCCRequesting : Bool
                  isTrainDecelerating : Bool

    variable
      v_tccRes : Bool := hasTCCRes(initTCCState),
      v_isReq : Bool := isTCCRequesting(initTCCState),
      v_isDec : Bool := isTrainDecelerating(initTCCState)

    value
      initTCCState : TCCState,

      hasTCCRes : Unit → read v_tccRes Bool
      hasTCCRes() ≡
        v_tccRes,

      setTCCRes : Bool → write v_tccRes Unit
      setTCCRes(tccRes) ≡
        v_tccRes := tccRes,

      isTCCRequesting : Unit → read v_isReq Bool
      isTCCRequesting() ≡
        v_isReq,

      setTCCRequesting : Bool → write v_isReq Unit
      setTCCRequesting(isReq) ≡
        v_isReq := isReq,

      isTrainDecelerating : Unit → read v_isDec Bool
      isTrainDecelerating() ≡
        v_isDec,

      setTrainDecelerating : Bool → write v_isDec Unit
      setTrainDecelerating(isDec) ≡
        v_isDec := isDec,

      hasPassedResPoint : T.TrainID →
                    read D.TD.v_TrainStates, S.ESAs.v_ESAs,
                         S.Segs.v_points, S.Segs.v_segs Bool
      hasPassedResPoint(t) ≡
        let
          front = T.frontPos(D.getTrainPosition(t))
        in
          case T.getLoc(front) of
            T.isESA(esa) →
            (
              let
                resPoint = S.getResPoint(),
                posFront = T.getLength(front),
                esaLength = S.getESALength(esa),
```

```
            dir = D.getTrainDirection(t)
          in
            passedPoint(posFront,resPoint,esaLength,dir)
          end
        ),

        T.isSeg(seg) →
        (
          let
            resPoint = S.getResPoint(),
            posFront = T.getLength(front),
            segLength = S.getSegLength(seg),
            dir = D.getTrainDirection(t)
          in
            passedPoint(posFront,resPoint,segLength,dir)
          end
        )
      end
    end,

hasPassedBrakePoint : T.TrainID →
                      read D.TD.v_TrainStates, S.ESAs.v_ESAs,
                           S.Segs.v_points, S.Segs.v_segs Bool
hasPassedBrakePoint(t) ≡
  let
    front = T.frontPos(D.getTrainPosition(t))
  in
    case T.getLoc(front) of
      T.isESA(esa) →
      (
        let
          brkPoint = S.getBrakePoint(),
          posFront = T.getLength(front),
          esaLength = S.getESALength(esa),
          dir = D.getTrainDirection(t)
        in
          passedPoint(posFront,brkPoint,esaLength,dir)
        end
      ),

      T.isSeg(seg) →
      (
        let
          brkPoint = S.getBrakePoint(),
          posFront = T.getLength(front),
          segLength = S.getSegLength(seg),
          dir = D.getTrainDirection(t)
        in
          passedPoint(posFront,brkPoint,segLength,dir)
        end
      )
    end
  end,

passedPoint : T.Length × T.Length × T.Length ×
                                        T.Direction → Bool
passedPoint(posFront,passPoint,segLength,dir) ≡
```

```
    case dir of
        T.UP → posFront > (segLength − passPoint),
        T.DOWN → posFront < passPoint
    end,

/* Processes */
tccMsgReceiver : T.ComMsg → write v_isReq, v_tccRes Unit
tccMsgReceiver(comMsg) ≡
    let
        resp = T.getMsg(comMsg)
    in
        case resp of
            T.segResp(resGranted) →
            (
                setTCCRequesting(false);
                if(resGranted)
                then
                    setTCCRes(true)
                end
            ),
            _ → ()
        end
    end,

tccProcess : T.TrainID × T.Tick → read S.any
                                  write D.TD.v_TrainStates,
                                  v_isDec, v_isReq, v_tccRes
                                  out COM.comChannel Unit
tccProcess(t,tick) ≡
    checkSpeed(t,tick);
    clearRes(t);
    handleRes(t),

checkSpeed : T.TrainID × T.Tick → read S.Segs.v_segs,
                            S.Trains.v_trains, v_tccRes
                            write D.TD.v_TrainStates, v_isDec Unit
checkSpeed(t,tick) ≡
    let
        dts = S.getTrainMaxAcc(t) * tick,
        ts = D.getTrainSpeed(t) + dts,
        trainMaxSpeed = S.getTrainMaxSpeed(t)
    in
        /* If train is entirely in an ESA */
        if(D.trainInESA(t))
        then
            if(ts > trainMaxSpeed)
            then
                decelerateTrain(t)
            else
                checkDeceleration(t)
            end
        else /* Train on segment and perhaps an ESA */
            let
                tp = D.getTrainPosition(t),

                /* Will always be an IsSeg */
                isSeg = D.getTrainLoc(t),
```

```
                frontSeg = T.getSeg(isSeg),
                segMaxSpeed = S.getSegMaxSpeed(frontSeg)
            in
                if (ts > segMaxSpeed ∨ ts > trainMaxSpeed)
                then
                    decelerateTrain(t)
                else
                    checkDeceleration(t)
                end
            end
        end
    end, /* let */

checkDeceleration : T.TrainID ~→
                        write D.TD.v_TrainStates, v_isDec Unit
checkDeceleration(t) ≡
    if (isTrainDecelerating())
    then
        D.setTrainAcc(t,0.0);
        setTrainDecelerating(false)
    end,

decelerateTrain : T.TrainID → read S.Trains.v_trains
                              write D.TD.v_TrainStates, v_isDec Unit
decelerateTrain(t) ≡
    setTrainDecelerating(true);
    D.decelerateTrain(t),

accelerateTrain : T.TrainID → read S.Trains.v_trains
                              write D.TD.v_TrainStates, v_isDec Unit
accelerateTrain(t) ≡
    setTrainDecelerating(true);
    D.accelerateTrain(t),

clearRes : T.TrainID ~→ read D.TD.v_TrainStates
                              write v_tccRes Unit
clearRes(t) ≡
    if(∼T.oneLoc(D.getTrainPosition(t)))
    then
        setTCCRes(false)
    end,

handleRes : T.TrainID → read S.any, v_tccRes, v_isReq
                              write D.TD.v_TrainStates,
                              v_isReq, v_isDec
                              out COM.comChannel Unit
handleRes(t) ≡
    if(hasPassedResPoint(t))
    then
        if(∼hasTCCRes())
        then
            if(hasPassedBrakePoint(t))
            then
                decelerateTrain(t)
            end;
```

```
            if(∼isTCCRequesting())
            then
                tccRequestRes(t); ()
            end
        end
    end,

tccRequestRes : T.TrainID ~→
                                read S.ESAs.v_ESAs, S.Segs.v_segs,
                                D.TD.v_TrainStates
                                write v_isReq out COM.comChannel Unit
tccRequestRes(t) ≡
    let
        loc = T.frontLoc(D.getTrainPosition(t)),
        dir = D.getTrainDirection(t)
    in
        case loc of
            T.isESA(esa) →
            (
                /∗ If not facing line, then do nothing.
                     Will brake at brakepoint ∗/
                if (dir = T.end2Dir(esa))
                then
                    ()
                else
                    sendTCCReq(t,S.getESASB(esa),dir)
                end
            ),

            T.isSeg(seg) →
            (
                 sendTCCReq(t,S.getSegSB(seg,dir),dir)
            )
        end /∗ case ∗/
    end, /∗ let ∗/

sendTCCReq : T.TrainID × T.SBID × T.Direction ~→
                                write v_isReq out COM.comChannel Unit
sendTCCReq(t,sb,dir) ≡
    let
        sender = T.isTrain(t),
        receiver = T.isSB(sb),
        res = T.mk_res(t,dir),
        msg = T.segReq(res),
        comMsg = T.mk_comMsg(sender,receiver,msg),
        cs = setTCCRequesting(true)
    in
        COM.sendMsg(comMsg)
    end,

/∗ Invariants ∗/
initReq : Unit → read v_tccRes, v_isReq, v_isDec Bool
initReq() ≡
    no_tcc_res() ∧
    tcc_not_requesting() ∧
    tcc_not_decelerating(),
```

```
      /* tcc may not have a reservation */
      no_tcc_res : Unit → read v_tccRes Bool
      no_tcc_res() ≡
      (
          ~hasTCCRes()
      ),

      /* No TCC is requesting segment access */
      tcc_not_requesting : Unit → read v_isReq  Bool
      tcc_not_requesting() ≡
      (
          ~isTCCRequesting()
      ),

      /* No TCC is requesting segment access */
      tcc_not_decelerating : Unit → read v_isDec Bool
      tcc_not_decelerating() ≡
      (
          ~isTrainDecelerating()
      )

  axiom
      /* Initial state */
      [initial]
          initialise post initReq()

end
```

SBCC

```
context: I_Dynamics0, I_ComService0
scheme I_SBCC0(T : I_Types0, S : I_Statics0(T),
                       D : I_Dynamics0(T,S), COM : I_ComService0(T)) =
  class
    type
      SBCCState :: getLineRes : T.HasRes
                        getBranchRes : T.HasRes
                        getSensorStatus : T.SensorStatus
                        getMsgs : T.ComMsg*
                        getPrepRes : T.HasRes

    variable
      v_lineRes : T.HasRes := getLineRes(initSBCCState),
      v_branchRes : T.HasRes := getBranchRes(initSBCCState),
      v_sensorStatus : T.SensorStatus :=
                                 getSensorStatus(initSBCCState),
      v_msgs : T.ComMsg* := getMsgs(initSBCCState),
      v_prepRes : T.HasRes := getPrepRes(initSBCCState)

    value
      initSBCCState : SBCCState,

      getLineRes : Unit → read v_lineRes T.HasRes
      getLineRes() ≡
          v_lineRes,
```

```
getBranchRes : Unit → read v_branchRes T.HasRes
getBranchRes() ≡
    v_branchRes,

getSensorStatus : Unit → read v_sensorStatus T.SensorStatus
getSensorStatus() ≡
    v_sensorStatus,

getMsgs : Unit → read v_msgs T.ComMsg*
getMsgs() ≡
    v_msgs,

getPrepRes : Unit → read v_prepRes T.HasRes
getPrepRes() ≡
    v_prepRes,

getNextMsg : Unit → write any T.HasComMsg
getNextMsg() ≡
    if(v_msgs = ⟨ ⟩)
    then
        T.noComMsg
    else
        let
            firstMsg = hd v_msgs
        in
            v_msgs := tl v_msgs;
            T.comMsg(firstMsg)
        end
    end,

storeMsg : T.ComMsg → write any Unit
storeMsg(msg) ≡
    v_msgs := v_msgs ⁀ ⟨ msg ⟩,

removeLineRes : Unit → write any Unit
removeLineRes() ≡
    v_lineRes := T.noRes,

removeBranchRes : Unit → write any Unit
removeBranchRes() ≡
    v_branchRes := T.noRes,

msgReceiver : T.ComMsg → write any Unit
msgReceiver(comMsg) ≡
    storeMsg(comMsg),

removePrepRes : Unit → write any Unit
removePrepRes() ≡
    v_prepRes := T.noRes,

isPreparing : Unit → read any Bool
isPreparing() ≡
    case v_prepRes of
        T.noRes → false,
        _ → true
    end,
```

```
/∗ Processes ∗/
sbccProcess : T.SBID × T.Tick →
                                        read S.SBs.v_SBs, S.Segs.v_segs
                                        write any, D.SD.v_SBStates
                                        out COM.comChannel Unit
sbccProcess(sb,tick) ≡
        sensorProcess(sb);
        if(isPreparing())
        then
            prepareProcess(sb)
        else
            sbccMsgProcess(sb)
        end,

/∗ Waits for the preparation of a segment
     before a train is allowed to enter ∗/
prepareProcess : T.SBID → read S.SBs.v_SBs, D.SD.v_SBStates
                                              write any out COM.comChannel Unit
prepareProcess(sb) ≡
    case S.getSBType(sb) of
        /∗ case POINTSB → wait for !moving ∗/
        T.POINTSB →
        (
            if(D.getPointPosition(sb) ∈ {T.UP, T.DOWN})
            then
                let
                    T.res(res) = v_prepRes,
                    train = T.getTrain(res)
                in
                    removePrepRes();
                    sendSBCCMsg(sb,T.isTrain(train),T.segResp(true));()
                end
            end
        ),

        /∗ case crossingsb → wait for DOWN ∗/
        T.CROSSINGSB →
        (
            if(D.getBarrierPosition(sb) = T.DOWN)
            then
                let
                    T.res(res) = v_prepRes,
                    train = T.getTrain(res)
                in
                    removePrepRes();
                    sendSBCCMsg(sb,T.isTrain(train),T.segResp(true)); ()
                end
            end
        ),

        _ → ()
    end,

sensorProcess : T.SBID → read S.SBs.v_SBs, S.Segs.v_segs
                                            write any, D.SD.v_SBStates
                                            out COM.comChannel Unit
```

```
sensorProcess(sb) ≡
    let
        sState = D.getSensorStatus(sb),
        lastState = v_sensorStatus
    in
        v_sensorStatus := sState;

        if((lastState = T.ACTIVE) ∧ (sState = T.INACTIVE))
        then
            dePrepareSeg(sb);
            if(S.isLineGuard(sb))
            then
                makeDeRes(sb); ()
            end
        end
    end,

dePrepareSeg : T.SBID → read S.SBs.v_SBs
                             write D.SD.v_SBStates, any Unit
dePrepareSeg(sb) ≡
    removePrepRes();
    case S.getSBType(sb) of
        T.CROSSINGSB →
        (
            D.setBarrierPosition(sb,T.MOVINGUP); ()
        ),

        _ → ()
    end,

prepareSeg : T.SBID × T.Reservation → read S.SBs.v_SBs
                                      write D.SD.v_SBStates, any Unit
prepareSeg(sb,res) ≡
    v_prepRes := T.res(res);

    case S.getSBType(sb) of
        T.CROSSINGSB → D.setSignalStatus(sb,T.ON); (),
        T.POINTSB →
        (
            case T.getDir(res) of
                T.UP → D.setPointPosition(sb,T.MOVINGUP); (),
                T.DOWN → D.setPointPosition(sb,T.MOVINGDOWN); ()
            end
        ),

        _ → ()
    end,

makeDeRes : T.SBID → read S.SBs.v_SBs, D.SD.v_SBStates,
                          S.Segs.v_segs write any
                          out COM.comChannel Unit
makeDeRes(sb) ≡
    case S.getSBType(sb) of
        T.ENDSB →
        (
            let
                T.res(lineRes) = v_lineRes,
```

```
                endDir = S.getEndDir(sb)
            in
                if(T.getDir(lineRes) = endDir)
                then
                    removeLineRes();
                    sendLDeResMsg(sb,S.getOppositeGuard(sb)); ()
                end /∗ if ∗/
            end /∗ let ∗/
        ),

        T.POINTSB →
        (
            let
                T.res(lineRes) = v_lineRes,
                pointDir = S.getPointDir(sb)
            in
                if(T.getDir(lineRes) = pointDir)
                then
                    removeLineRes();
                    sendLDeResMsg(sb,S.getOppositeGuard(sb)); ()
                else
                    sendBDeResMsg(sb,S.getOppositeGuard(sb)); ()
                end /∗ if ∗/
            end /∗ let ∗/
        )
    end /∗ case ∗/
pre S.isLineGuard(sb),

sendLBDeResMsg : T.SBID × T.SBID → out COM.comChannel Unit
sendLBDeResMsg(thisSB,remoteSB) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineBranchDeRes),

sendLDeResMsg : T.SBID × T.SBID → out COM.comChannel Unit
sendLDeResMsg(thisSB,remoteSB) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineDeRes),

sendBDeResMsg : T.SBID × T.SBID → out COM.comChannel Unit
sendBDeResMsg(thisSB,remoteSB) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.branchDeRes),

sendLBResMsg : T.SBID × T.SBID × T.Reservation →
                                        out COM.comChannel Unit
sendLBResMsg(thisSB,remoteSB,aRes) ≡
    sendSBCCMsg(thisSB,T.isSB(remoteSB),T.lineBranchReq(aRes)),

sendSBCCMsg : T.SBID × T.ComID × T.SBCCMsg →
                                        out COM.comChannel Unit
sendSBCCMsg(sb,receiver,sbccMsg) ≡
    COM.sendMsg(T.mk_comMsg(T.isSB(sb),receiver,sbccMsg)),

sbccMsgProcess : T.SBID → read S.SBs.v_SBs, S.Segs.v_segs
                                write D.SD.v_SBStates, any
                                out COM.comChannel Unit
sbccMsgProcess(sb) ≡
    let
        hasComMsg = getNextMsg()
    in
```

```
        case hasComMsg of
            T.comMsg(T.mk_comMsg(sender,receiver,msg)) →
            (
                case sender of
                    T.isTrain(_) →
                    (
                        let
                            retMsg = handleTCCMsg(sb,msg)
                        in
                            case retMsg of
                                T.hasMsg(aMsg) →
                                        sendSBCCMsg(sb,sender,aMsg); (),
                                _ → ()
                            end
                        end
                    ),

                    T.isSB(_) →
                    (
                        let
                            retMsg = handleSBCCMsg(sb,msg)
                        in
                            case retMsg of
                                T.hasMsg(aMsg) →
                                          sendSBCCMsg(sb,sender,aMsg); (),
                                _ → ()
                            end
                        end
                    )
                end /* case */
            ),
            _ → () /* no message to process */
        end
    end, /* let */

handleSBCCMsg : T.SBID × T.Message → write any
                                      out COM.comChannel T.ReturnSBCCMsg
handleSBCCMsg(sb,msg) ≡
    case msg of
        /* Request */
        T.lineBranchReq(_) → handleLBReq(msg),

        /* Response */
        T.lineBranchResp(_) → handleLBResp(sb,msg),

        /* De reservation */
        T.lineBranchDeRes → handleDeResMsg(msg),
        T.lineDeRes → handleDeResMsg(msg),
        T.branchDeRes → handleDeResMsg(msg)
    end,

handleTCCMsg : T.SBID × T.Message →
                                      read S.SBs.v_SBs, S.Segs.v_segs
                                      write D.SD.v_SBStates, any
                                      out COM.comChannel T.ReturnSBCCMsg
handleTCCMsg(sb,msg) ≡
    let
```

```
        T.segReq(res) = msg
    in
        case S.getSBType(sb) of
            T.ENDSB →
            (
                /∗ if direction away from end → send msg
                     else OK
                ∗/
                if(T.getDir(res) = S.getEndDir(sb))
                then
                    prepareSeg(sb,res);
                    T.noSBCCMsg
                else
                    if(lineFree())
                    then
                        v_lineRes := T.res(res);
                        sendLBResMsg(sb,S.getOppositeGuard(sb),res);
                        T.noSBCCMsg
                    else
                        T.hasMsg(T.segResp(false))
                    end
                end
            ),

            /∗ If direction away from point → send msg
                 else OK
            ∗/
            T.POINTSB →
            (
                if(T.getDir(res) = S.getPointDir(sb))
                then
                    prepareSeg(sb,res);
                    T.noSBCCMsg
                else
                    if(lineFree())
                    then
                        v_lineRes := T.res(res);
                        sendLBResMsg(sb,S.getOppositeGuard(sb),res);
                        T.noSBCCMsg
                    else
                        T.hasMsg(T.segResp(false))
                    end
                end
            ),

            _ → /∗ PLAINSB, CROSSINGSB ∗/
            (
                prepareSeg(sb,res);
                T.noSBCCMsg
            )
        end /∗ case ∗/
    end, /∗ let ∗/

/∗ if(!line free) then NO
     reserve line;
     if(end type) then YES
     if(!branch free) then deres line; NO
```

```
     res branch; YES;
*/
handleLBReq : T.Message → write any T.ReturnSBCCMsg
handleLBReq(msg) ≡
    let
        T.lineBranchReq(res) = msg
    in
        if(lineBranchFree())
        then
            v_lineRes := T.res(res);
            v_branchRes := T.res(res);
            T.hasMsg(T.lineBranchResp(res,true))
        else
            T.hasMsg(T.lineBranchResp(res,false))
        end
    end,

lineBranchFree : Unit → write any Bool
lineBranchFree() ≡
    (v_lineRes = T.noRes) ∧
    (v_branchRes = T.noRes),

lineFree : Unit → write any Bool
lineFree() ≡
    (v_lineRes = T.noRes),

/* if(response = NO) deres; NO;
     if(!prepare_segment()) deres; NO;
     OK;
*/
handleLBResp : T.SBID × T.Message → write any
                                  out COM.comChannel T.ReturnSBCCMsg
handleLBResp(sb,msg) ≡
    let
        T.lineBranchResp(res,granted) = msg
    in
        if(granted)
        then
            sendSBCCMsg(sb,T.isTrain(T.getTrain(res)),
                                        T.segResp(true));
            T.noSBCCMsg
        else
            removeLineRes();
            sendSBCCMsg(sb,T.isTrain(T.getTrain(res)),
                                        T.segResp(false));
            T.noSBCCMsg
        end
    end,

/* case(msg)
            lb → deres line; deres branch
            l → deres line;
            b → deres branch;
*/
handleDeResMsg : T.Message → write any T.ReturnSBCCMsg
handleDeResMsg(msg) ≡
    case msg of
```

```
            T.lineBranchDeRes →
            (
               removeLineRes();
               removeBranchRes();
               T.noSBCCMsg
            ),

            T.lineDeRes →
            (
               removeLineRes();
               T.noSBCCMsg
            ),

            T.branchDeRes →
            (
               removeBranchRes();
               T.noSBCCMsg
            )
         end,

      /∗ Invariants ∗/
      initReq : Unit → read any Bool
      initReq() ≡
         no_sbcc_res() ∧
         sbcc_not_preparing(),

      no_sbcc_res : Unit → read any Bool
      no_sbcc_res() ≡
      (
         ∀ branchRes,lineRes : T.HasRes •
                  branchRes = v_branchRes ∧
                  lineRes = v_lineRes
                     ⇒
                        {branchRes} ∪ {lineRes} = {T.noRes}
      ),

      /∗ No SBCC is currently preparing a segment ∗/
      sbcc_not_preparing : Unit → read any Bool
      sbcc_not_preparing() ≡
      (
         ∼isPreparing()
      )

   axiom
      /∗ Initial state ∗/
      [initial]
         initialise post initReq()

end
```