

Modelling and Verification of Relay Interlocking Systems

Morten Aanæs
Hoang Phuong Thai

Kongens Lyngby 2012
IMM-MSC-2012-14

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

Summary

Electrical relay based interlocking systems are widely used by Banedanmark to ensure safe operation of trains at stations in Denmark. These systems are documented by diagrams showing the electrical circuitry, physical track layout of the stations and train route tables.

The safety of each station is currently verified by inspecting the diagrams by hand. This process is time consuming and possibly error-prone. Therefore Banedanmark wishes to automate the process.

Our goal is to develop a method for automated model and safety property generation of the external part of a particular type of relay interlocking system, the DSB type 1954. This is then to be combined with the already developed model of the internal system and verified as a whole.

We specified a data model for interlocking plans, developed a behavioural model of external events and formalised safety properties which, among other things, assert that trains do not collide and do not derail. We then developed an executable specification of a generator that, given an interlocking plan, can generate a model of the external events. This model can then be combined with the model of the internal events and model checked.

The entire process was applied to the small Danish railway station Stenstrup, which uses the DSB type 1954 relay interlocking system. The result is that all generated safety properties are satisfied for the combined model of the interlocking system at Stenstrup.

Resumé

Elektriske relæbaserede sikringsanlæg er bredt anvendt af Banedanmark til at sikre togdriften på stationer i Danmark. Disse systemer er dokumenteret med diagrammer der viser det elektriske kredsløb, den fysiske sporgeografi af stationen og togvejstabelen.

Sikkerheden for hver station verificeres i dag i hånden, ved at inspicere diagrammerne. Denne proces er tidskrævende og potentielt fejlbehæftet. Derfor ønsker Banedanmark at automatisere processen.

Vores mål er, at udvikle en fremgangsmåde til automatiseret model- og sikkerhedsegenskabsgenerering af den eksterne del af en bestemt type relæsikringsanlæg, DSB type 1954. Dette skal derefter kombineres med den allerede udviklede model af det interne system og verificeres som helhed.

Vi angav en datamodel for sikringplaner, udviklede en model af eksterne begivenheder og formaliserede sikkerhedsegenskaber, som blandt andet forsikrer, at togene ikke kolliderer og ikke afsporer. Vi udviklede en eksekverbar specifikation af en generator, der, givet en sikringsplan, kan generere en model af de eksterne hændelser. Denne model kan derefter kombineres med modellen af interne hændelser og modeltjekkes.

Hele processen blev anvendt på den lille danske station Stenstrup, der anvender et DSB type 1954 relæsikringsanlæg. Resultatet er, at alle genererede sikkerhedsegenskaber er opfyldt for den kombinerede model af sikringsanlægget på Stenstrup.

Preface

This master's thesis was prepared at the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU) in partial fulfilment of the requirements for acquiring the M.Sc. degree in Computer Science and Engineering.

The work was carried out in the period September 5th 2011 to February 29th 2012 and is worth 30 ECTS points. The project was supervised by Associate Professor Anne Elisabeth Haxthausen, DTU IMM and co-supervised by Kirsten Mark Hansen, Banedanmark.

The thesis deals with automating the verification process of relay based interlocking systems, in particular the generation of behavioural models from static diagrams and safety property generation.

The authors claim equal responsibility of the thesis.

Kgs. Lyngby, February 2012

Morten Aanæs
Hoang Phuong Thai

Versions

1.3 May 2013 Fixed error in enter station rule.

1.2 June 2012 First major revision. Changes include:

- Restructured the specification of the generator. This should make the specification easier to read and understand.
- Major rewrite of chapter [11](#) to reflect the changes to the specification.
- Changed and added well-formedness requirements in sections [4.3.5](#) and [4.3.9](#).
- Replaced the term “reserved route” with the correct term “locked route.”
- Fixed various layout issues.

1.1 March 2012 First publicly released version. Changes include:

- Replaced formula of general form of RSL-SAL assertions in section [11.4.1](#), with the correct formula and rephrased the text slightly for accuracy.
- Fixed various layout issues.

1.0 February 2012 Initial version. Not publicly released.

Acknowledgements

We would like to thank

Anne E. Haxthausen for showing great interest in the project and spending a lot of time discussing the project with us. We really appreciate it.

Alex Landex for allowing us to attend parts of the course Signalling Systems for Railways at DTU.

Kirsten M. Hansen for taking time out of her busy schedule to answer some questions of ours.

Kasper B. Nielsen for providing feedback on the thesis.

And our other close friends for their feedback.

Contents

Summary	i
Resumé	iii
Preface	v
Versions	vii
Acknowledgements	ix
1 Introduction	1
1.1 Motivation	2
1.2 Related Work	3
1.3 Goal	5
1.4 Prerequisites	6
1.5 Chapter Overview	6
2 Domain Description	9
2.1 Railway Network	10
2.2 Trains	13
2.3 Signalling	14
2.4 Safety	15
2.5 Interlocking System	19
2.6 Interlocking Plan	24
3 Method Description	33
3.1 Approach	33
3.2 Suggested Method	34

4	Data Models	37
4.1	Interlocking Plan	37
4.2	Station Layout Diagram	38
4.3	Train Route Table	52
4.4	Transition System	71
5	Behavioural Model of the Internal System	81
5.1	State Space	82
5.2	Transition Rules	84
5.3	The Timing Issue	84
6	Behavioural Model of Train Movements	87
6.1	Approaches to Modelling Train Movement	87
6.2	Model of Trains	89
6.3	Model of Train Movements	93
6.4	Rubber Band Trains	106
7	Behavioural Model of Buttons and Points	109
7.1	Button Behaviour	109
7.2	Points Behaviour	110
8	Associations	115
8.1	Object Relay Associations	116
8.2	Train Movement Associations	120
9	Inter-Model Consistency	129
9.1	Variables	130
9.2	Track Occupation	131
9.3	Track Free	131
9.4	Train Direction	132
9.5	Points Configuration	133
9.6	Train Connection	134
10	Safety Properties	135
10.1	Variables	135
10.2	No Collision	137
10.3	No Derailing	137
10.4	Points Position	139
10.5	Signal	139
10.6	Signal Release	141
10.7	Conflicting Routes	141
10.8	Train Route Release	142

11 Generator	145
11.1 Overview	146
11.2 State Generator	147
11.3 Transition Generator	155
11.4 Assertion Generator	167
11.5 Output	175
12 Test	181
12.1 Station Layout	181
12.2 Results	182
13 Case Study - Stenstrup	187
13.1 Defining Stenstrup	187
13.2 Generating Model of External Events	193
13.3 Combining the Models	196
13.4 Results Stenstrup	197
14 Conclusion	199
14.1 Future Work	201
A Word List	205
B Documentation of Stenstrup Station	207
C User Guide	211
C.1 Requirements	211
C.2 Stenstrup	212
D Behavioural Model of Train Movements Using Explicitly Defined Trains	215
D.1 Enter	216
D.2 Leave	218
E Specifications	221
E.1 Data Models	221
E.2 Generators	238
E.3 Associations	273
E.4 Misc.	280
E.5 Test	290
F Output	333
F.1 Output of Generator	333
F.2 Output of Model Checker	347
F.3 Output of Tests	356

CHAPTER 1

Introduction

This chapter gives an introduction to the overarching problem we seek to solve, present the related work and precisely define what we are trying to achieve.

Section 1.1 will introduce and justify the problem we are trying to solve.

Section 1.2 will explain what has been done before and what to choose to build upon and what we wish to improve.

Section 1.3 will explain the problem in greater detail, followed by a brief sketch of how we plan to solve it.

Section 1.4 will state some assumptions about the reader.

Lastly, section 1.5 will shortly describe each chapter in the thesis.

Throughout this chapter, terms and notions, that have not yet been explained, are used. Please refer to the domain description in chapter 2 for definitions and explanations.

1.1 Motivation

More than 170 million passengers and about 15 million tonnes of freight are transported on Banedanmarks railway network on a yearly basis. With about 40.000 train movements per day, reliability and safety are top priorities¹.

The safety at the stations is ensured by interlocking systems, which control the track side equipment like signals, points and level crossings. Many stations on the Danish railway are still secured by old relay based electrical interlocking systems.

The relay based interlocking systems are documented by the station documentation, which is a collection of diagrams of the layout of the track side equipment, the electrical circuitry and the train route table.

The interlocking systems are verified by manually inspecting the diagrams and drawing conclusions about the safety. Due to the high number of diagrams and their mutual correlation, this process is complex, time consuming (and thus expensive) and possibly error prone, which is not satisfactory for a safety critical system.

Furthermore, the process has to be repeated whenever changes are made to the system, e.g. changes to the physical layout of the station or direct changes to the relay circuits.

1.1.1 Vision

Banedanmark is interested in a tool that automates this process. A solution to the problem is illustrated on figure 1.1.

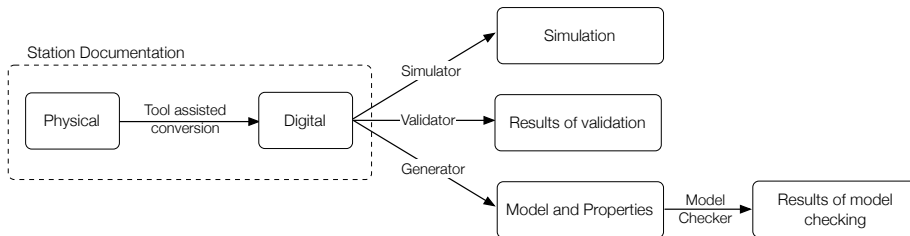


Figure 1.1: Station interlocking editing, simulation and verification tool.

¹Key figures from Banedanmark at <http://www.bane.dk/visArtikel.asp?artikelID=136>

The physical station documentation is digitalised using a graphical user interface. Here the diagrams of the circuitry and station layout are drawn and the train route table is created. When all the documentation is created, the system can be simulated, so the engineer can observe how changes affect the system.

Once the engineer is satisfied with the system, the tool should then be able to automatically verify that the system he (or she) designed, satisfies the safety properties required of such a system. This is done by generating a model of the interlocking system and the safety properties.

1.2 Related Work

Relay based interlocking system has been a subject of research at DTU. As described by Haxthausen [9], it was proposed to chose model checking as the verification approach to allow full automation. This automated verification process can be divided into two steps, as illustrated on figure 1.2. The first step involves generators to automatically create a RSL-SAL representation of:

- a behavioural state transition model of the relay interlocking system,
- a behavioural state transition model of the environment (train movement, points, and operator buttons),
- conditions about the behaviour of the system expressed in the form of formal assertions.

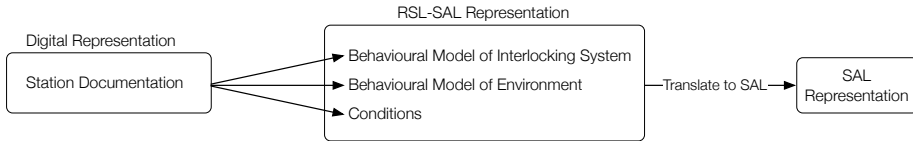


Figure 1.2: The generation step can be divided into distinct subproblems: Generating the behavioural model of the interlocking system (relay circuitry), the behavioural model of the environment (train movements) and conditions.

This generated RSL-SAL [11] representation is an extension of the RAISE specification language [8] allowing the construction of the state transition systems of the behavioural model of the relay interlocking system, and the model of the environment. Additionally, the desired properties about the behaviour of the system can be defined in the form of assertions using temporal logic LTL.

The RSL-SAL representation can then, as part of the second step, be translated to the SAL language [6] using the RAISE tool set [1], which allows the SAL model checker to check the validity of the two behavioural state transition systems with the desired conditions.

In the next section (1.2.1) we will introduce the current state of the tool that has been presented, and discuss which parts of the automatic verification process that is complete. Also, parts of the automatic verification process is, however, not complete which will be discussed.

1.2.1 Current State of the Tool

Over time several contributors have participated to create parts of the tool to fulfil the vision stated in 1.1.1. The contributed work consists of:

Graphical Editor and Simulator: Eriksen and Pedersen [7] developed a graphical relay circuitry editor and simulator. It allows the system designer to reproduce the static physical circuit diagrams, and thereafter simulate the propagation of a current through the circuit, and observe as the state of the components change dynamically. Furthermore, they implemented an editor to create a simple operator's panel as well as support for occupying and freeing track sections through the simulator. The rest of the system would then react to these events. Finally, they implemented an export feature to store the system in a XML file.

Verification Method of Relay Interlocking System: A part of the proposed tool has been developed by Kjær and Le Bliguet [2]. Specified in RSL and implemented in Java, they are able to take relay diagrams in XML format as input and automatically generate a RSL-SAL model of the internal behaviour of the interlocking system and associated confidence conditions. Furthermore, they have suggested a model of the environment for Stenstrup station and safety properties, and shown that the system satisfies the properties they stated.

Unfortunately, there is a missing link between the two parts since the exported XML file created by the editor is not of same format required by the verification of the interlocking system component.

Figure 1.3 illustrates the work that has been done (in grey) and which components needed to be automated.

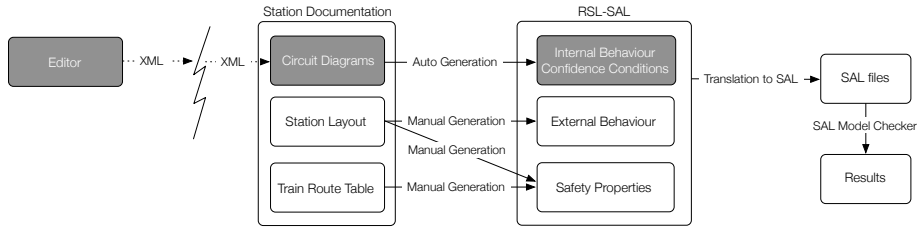


Figure 1.3: Current state of the tool. Automatic generator of external behaviour and safety properties is needed.

Internal behaviour	This is a model of the behaviour of the internal workings of the relay based interlocking system. Requires the circuit diagrams.
External behaviour	This is a model of the behaviour of the external events, to which the internal part of the interlocking system responds. Requires the station layout diagram.
Safety Properties	This is the conditions that is used to verify that the system behave as desired. Requires the station layout and the train route table.

1.3 Goal

Our goal is to investigate a method for automated model and safety property generation of the external part of a particular type of relay interlocking system, the DSB type 1954. This is then to be combined with the already developed model of the internal system and verified as a whole.

This project can be seen as an extension of the work mentioned in section 1.2 and we will therefore continue the research on the DSB type 1954 relay interlocking system and use Stenstrup as case study.

If successful, this would allow Banedanmark to save time on verifying and eliminate the possibility of human error.

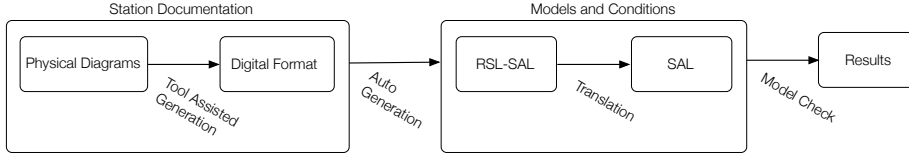


Figure 1.4: Overview of the intended validation process.

1.4 Prerequisites

The reader is expected to have knowledge of the following.

- Common features in the RAISE specification language (RSL).
- Common expressions in Linear Temporal Logic (LTL).
- State space based model checking in general.

1.5 Chapter Overview

This thesis contains following chapters.

Chapter 2 will introduce the domain, describing concepts and terms used throughout this thesis.

Chapter 3 will give an overview of the method used to approach the stated problem.

Chapter 4 will introduce the data models.

Chapter 5 will introduce the behavioural model of the internal system.

Chapter 6 will discuss approaches to modelling a representation of train and train behaviour. Using a suitable approach a model of train movement is constructed.

Chapter 7 will present external behaviours of buttons and points.

Chapter 8 will introduce associations to create mappings of physical station elements with the internal model.

Chapter 9 will describe requirements to ensure that the interaction between the model of train movement and the internal relay system behaves as expected.

Chapter 10 will present the safety properties that are requirements to the system to ensure safe train operation.

Chapter 11 will introduce a generator, which facilitates the generation of the train movements discussed in chapter 6. The generator will also introduce the generation of the state space and assertions.

Chapter 12 will introduce a test strategy and show a thorough test of one of the data models.

Chapter 13 will show a case study of Stenstrup station.

Chapter 14 will conclude the work that have been conducted. And also presenting future work.

Appendix A contains a word list from english to danish.

Appendix B contains documentation of Stenstrup station.

Appendix C contains a user guide of the application.

Appendix D contains a modelling approach mentioned in chapter 6.

Appendix E contains the complete specifications introduced in chapter 4 and 11.

CHAPTER 2

Domain Description

In this chapter, concepts and terms of the Danish railway domain, which is deemed to have relevance for this project, will be introduced.

Section 2.1 gives a general description of the Danish Railways and introduces the components it consists of.

Section 2.2 defines what a train is.

Section 2.3 introduces both the signalling equipment and the meaning of the signals.

Section 2.4 describes the safety properties of train operation in the railway domain. Several approaches to enforcing the safety properties will also be introduced.

Section 2.5 introduces the different kinds of interlocking systems used in Denmark and explains the relay based interlocking systems in greater detail.

Section 2.6 introduces the interlocking plan, which is part of the station documentation. The diagrams shown are of Stenstrup station, which will be used as case study throughout this thesis.

2.1 Railway Network

The Danish railway network can be divided into two distinct parts, the open line¹ and stations. The stations are interconnected by the line. The line is defined as the part of the track network, which is outside the station limits [5].

The technical definition of a railway station, used by Banedanmark is a railway stop, which participates in the safety operation of the train service [5]. Railway stops that does not have its own interlocking system, is secured as part of the line block interlocking system, which is a separate system from the station interlocking system.

In this work, only station interlocking systems are considered.

2.1.1 Track Sections

A railway network consists of rails that can be divided into sections. These sections are of different types, such as *linears*, *points* and *level crossings*.

Linear track sections are connected to either one or two other sections, while points sections are able to direct traffic in one of two direction. Level crossings are intersections between the railway and a road, but are not considered for this project.

Figure 2.1 describes the correlation of the track elements.

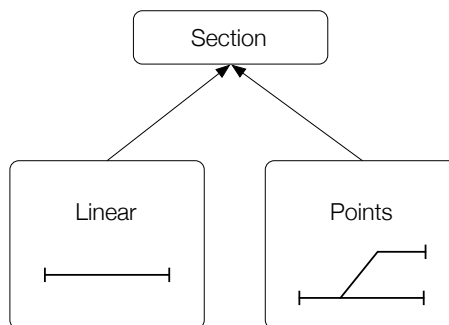


Figure 2.1: Definition of track sections.

¹Just “line” for short.

Linear sections must have at least one neighbour and at most two neighbours, where each neighbour of that section must be unique. A section can only have one neighbour at each end. Hence illegal configurations includes cases such as a linear section being neighbour with itself or two distinct linear sections are neighbours with each other on both ends (fig. 2.2).

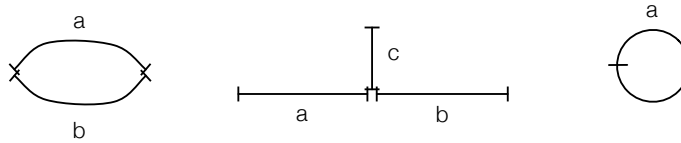


Figure 2.2: Illegal configurations.

2.1.1.1 Points Sections

A points section consists of a stem end and a branch end. The stem end has one neighbour, while the branch end has two. The least curved track on the branch end is called the *plus* branch and the most curved track is called the *minus* branch² (fig. 2.3).

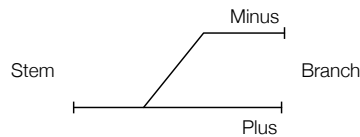


Figure 2.3: A points section has a stem end and a branch end. The least curved track on the branch end is called plus, while the most curved is called minus.

Movement from the stem end to the branch end is called a facing move, while movement from the branch end to the stem end is called a trailing move [12] as illustrated on figure 2.4. When performing a facing move, the destination depends on the position of the points section. If the points section is locked into the minus position, the train will travel onto the minus branch and vice versa. If the points have not been locked, the train runs the risk of derailing.

An illegal train movement on a points section is travelling from the plus branch to the minus branch or vice versa on the same points section. Figure 2.5 illustrates this movement.

²This convention is no longer used by Banedanmark on newer documentation. The current convention states that the right branch, as seen from the stem, is plus, while the left branch is minus. Since the documentation of Stenstrup uses the old definition, so will we to avoid confusion.

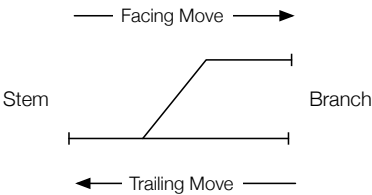


Figure 2.4: A facing move is a move on a points section from the stem side to the branch side. A trailing move is a move from the branch side to the stem side.

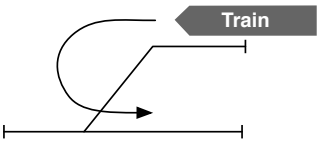


Figure 2.5: Illegal train movement on a points section.

Points must have three unique neighbours. Figure 2.6 shows some illegal connections of points.

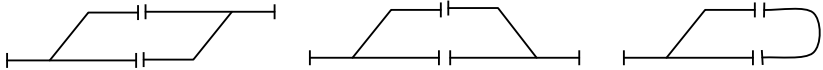


Figure 2.6: Illegal points section connections

A points section can be in one of three positions, *plus*, *minus* or *intermediate* (fig. 2.7). While in the plus position, the points section allow trains to traverse the section through the track on the plus branch and correspondingly with the minus position. The points section is in the intermediate position, when it is in the act of changing from plus to minus or vice versa.

Point Machine The position of the points section is controlled by a point machine. A point machine has three functions:

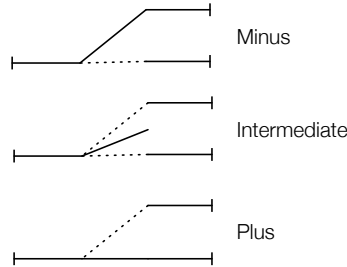


Figure 2.7: The different states of a points section. When in the plus position, it allows movements by the plus branch and likewise with the minus position. The points section is in the intermediate position when it is not locked in either plus or minus.

Switching points	The point machine is responsible for physically switching the points.
Locking points	The point machine is responsible for physically locking the points, once they are in position.
Supervising points	The point machine must communicate the status of the points, i.e. the position of the points and whether they are locked or not.

2.2 Trains

The technical definition of a train is rolling stock that performs a train movement, i.e. travels from one station to another³ [10]. Rolling stock is any vehicle that drives on a railway, which corresponds to the everyday notion of a train.

Trains have a long braking distance due to the low friction between wheel and rail. Combined with the weight of a freight train or the speed of a passenger train, trains often cannot stop within the sighting distance of the driver. Therefore railway operations rely on interlocking systems to ensure that the track within braking distance in front of the train is clear.

A shunting movement, on the other hand, is movement of rolling stock within the limits of a station. Due to the nature of shunting movements, other (less restrictive) safety rules apply to shunting movements.

³The definition of a train varies slightly around the world.

2.3 Signalling

Signalling is a method of communication used in the railway domain to indicate the state of the track ahead. The method of signalling varies from country to country, but the main objective is to notify the train driver whether to stop or proceed.

Regardless of the method of signalling (see section 2.3.2), a signal is a way of communicating a certain *indication*.

2.3.1 Indication

An indication is the meaning of a signal. Different signals can have the same indication. The following describes certain indications, which are used in Denmark and relevant for this project.

Stop	The train may not proceed past the signal.
Stop and proceed	The train must stop at the signal and may then carefully proceed past the signal to the next signal.
Proceed	The train may proceed past the signal and prepare to stop at the next signal.
Proceed through	The train may proceed past the signal and the next, which will show proceed or proceed through.

2.3.2 Signals

Signals are signalling devices. Banedanmark uses combinations of coloured lights to form *aspects*. Multiple aspects may have the same indication. Not every signal is able to show every aspect, since they have fewer lights. This explains why there are multiple aspects for certain indications.

Table 2.1 shows some of the aspects used in Denmark and their indication (meaning of which is given in section 2.3.1). R means the red lamp is lit, Y means the yellow lamp is lit and G means the green lamp is lit. “/” means over, e.g. Y/R means yellow over red. f means flashing and is always used as a prefix for a colour, e.g. fG for flashing green.






Aspect		Indication
R	Y/R	 Stop
fR	Y/fR	 Stop and proceed
	Y/G	 Proceed (at limited speed)
G		 Proceed
fG	G/G	 Proceed through

Table 2.1: Aspects of main signals and their indication

In this work, we are only interested in stopping and moving trains, ignoring speed limitations. Disregarding the signalling variants of stopping and proceeding the aspects has been reduced to the stop and proceed aspects. Table 2.2 denotes the aspects used throughout this thesis.

Aspect	Indication
R	Stop
G	Proceed

Table 2.2: Aspects

2.4 Safety

In this section the safety properties of the railway network are described as well as how it is enforced to ensure the safety of passengers and trains. These properties is a set of requirements to the system that needs to be maintained at all time. Furthermore, safety approaches are introduced, which are different kinds of implementations to maintain these safety properties.

2.4.1 Safety Properties

In the previous sections we have introduced the elements to construct a working railway network. The interaction between all these objects constitute the safety of the train operations. These interactions needs to follow a set of rules to fulfil the basic safety requirements, namely [4]:

- Trains/shunt movements must not collide.
- Trains/shunt movements must not derail.
- Train/shunt movements must not collide with vehicles or humans crossing the railway at authorised crossings.
- Protect railway employees from trains.

For this thesis we do not consider shunting movements, level crossings and railway employees. Therefore the basic safety requirements can be reduced to the following subset of the basic safety requirements:

- Trains must not collide.
- Trains must not derail.

Henceforth, any reference to the basic safety requirements will be to the above subset.

In the following sections it will be explained how the events of collision and derailling can occur, and later how to avoid them.

2.4.1.1 Collision

A collision is an unintended contact between trains. Combining trains is an example of intended contact between trains. High speed collisions, on the other hand, are likely to have catastrophic consequences.

We consider any situation where two or more trains occupy the same track section a collision. This situation will not necessarily result in a collision, but the potential is there and that is enough for it to be considered a collision.

2.4.1.2 Derailing

Derailment can be caused by many things. Broken rails, traversing curves at too high speed, as a result of collision, travelling beyond the end of line, hitting obstacles, travelling on unlocked points and so on.

Many of these situations are out of the scope for this project. In fact, they can be reduced to travelling on unlocked points. Figure 2.8 show situations that can result in a derailing on a points section.



Figure 2.8: Possibility of derailing

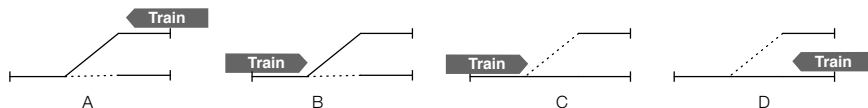


Figure 2.9: No derailing

2.4.2 Safety Approaches

Banedanmark uses two methods of securing train movements:

- Route based** The entire intended path of a train movement is locked for use by a single train only. The route is not released until the train has reached the end of the route. This principle is used by Banedanmark at stations.
- Block based** The railway is divided into blocks. When a train enters a block, no other train is allowed to enter that block, until the first train is out of the block again. This principle is used by Banedanmark on the open line.

2.4.2.1 Train Route

A train route is a secured path from one point⁴ (called the start point) to another (called the end point) in a railway network. These points are normally signals [10].

The following describes the concepts associated with train routes.

Safety overlap The safety overlap is an extension of a train route, intended to provide additional clear track in case the train fails to stop at the end point, e.g. the driver misjudges the braking distance.

Conflicting train routes Train routes are considered conflicting if they share points in their paths, incl. the safety overlap. Conflicting train routes are mutually exclusive and may not be locked at the same time. However, it is possible to extend a route with another by replacing the safety overlap with another route.

Flank protection Flank protection is protecting a train route from conflicting movements. The interlocking system achieves this first and foremost, by not allowing conflicting routes. This protects against train movements, but not against roll away vehicles. Therefore, further measures are taken, such as locking points, that are not part of the route, into positions that direct traffic away from the route in question

Locking train route A train route can be locked manually by pressing buttons on the operator's panel or automatically when the system detects an approaching train.

A train route can be locked if the track sections, that constitutes the route, are all clear, the points sections on the route are locked in proper position and the desired route is not in conflict with any other currently locked route.

A train route stays locked until it is released. The objects, which were part of the released train route, can then be used to form new train routes.

⁴This does not refer to points sections.

Release of train route A train route can be released in one of two ways, either automatically or manually.

Automatic release is performed by the interlocking system and happens once a train has completed its train movement.

Manual release is performed from the operator's panel. It can be performed even if a train is already on the route, but then the train no longer has the same measure of protection. Manual release of train routes is not considered.

2.5 Interlocking System

An interlocking system is a mechanical, electrical, electronic or hybrid system that controls physical objects, such as signals and points, in a limited area [10].

The Danish railway network is controlled by several different generations of interlocking systems, ranging from the very old mechanical, over the electrical relay based systems of the '50s and '60s to the newer computer based electronic systems.

This work considers the 54 type relay based interlocking system, since this is the type that is used at Stenstrup.

The interlocking system has three major fundamental tasks, which is to *control*, *supervise* and *ensure safety*.

Control	Manuel or automatic operation of points, signals etc.
Supervision	Continual supervision of the state of points, signals, isolation etc.
Ensuring Safety	Preventing points, signals etc. to be operated in such a way that it allows conflicting train movements.

2.5.1 Relay Based Interlocking

In a relay based interlocking system, the rules of the interlocking system is implemented in electrical circuits. A circuit consists of components, such as

relays, contacts and buttons, that are connected by wires and powered by a power supply.

2.5.1.1 Relays

A relay is an electrical component that can be in two states, drawn (symbolised with \uparrow) and dropped (\downarrow).

When power is supplied, the relay will be energised. This activates the electromagnet, which draws the armature so that the upper contacts become connected while the lower become disconnected (fig. 2.10a).

When the power is cut, the magnetic field will disappear and gravity will cause the armature to drop. This again switches the active connections (fig. 2.10b).

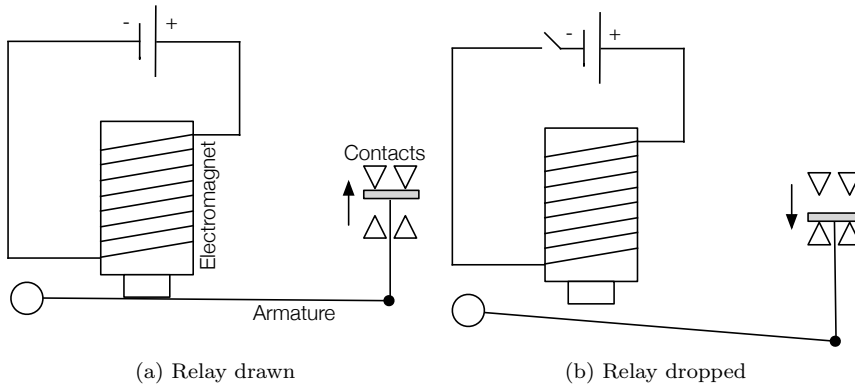


Figure 2.10: When a current energises a relay, its electromagnet is activated. This pulls the armature, which creates a connection with the upper contacts (a). Without power, the armature is dropped, which creates a connection with the lower contacts instead (b).

Other types of relays exist, for example steel core relays, which are able to retain their state even after losing power.

2.5.1.2 Logic of Relay Based Interlocking Systems

The state of the physical objects can be captured by relays. By assigning a relay (and sometimes multiple relays) to each physical object on the station, the

state of the entire station is captured. This can then be used to control the train movements in a safe manner.

Each track section has a relay associated (called a track relay), which is drawn when the track is considered vacant.

Each points section has two additional relays associated. One is drawn when the points section is locked in the plus position and the other is drawn when the points section is locked in the minus position.

In addition there are relays capturing the state of route locking, route releasing, each lamp on each signal etc.

Buttons can be pushed and released. When a button is pushed, the current is allowed to flow through, while the connection is cut when the button is released.

A relay based interlocking system consists of two parts, the *internal system* and the *external system*. The internal system is the *relay circuitry* that is controlled by the interlocking system, e.g. the signal relays. The external system on the other hand, consists of the circuitry that is controlled by the environment, e.g. points relays, and train detection relays.

The logic of the system is created by arranging the connections of the components in a certain way. Serial connections create conjunctions, while parallel connections create disjunctions, as the current will follow whichever path it is allowed (fig. 2.11).

The situation on fig. 2.11a can be translated to the following boolean expression $L = A \wedge B$, where L is true when the lamp is turned on and A and B are true when the respective button is pressed. Similarly the situation on fig. 2.11b can be translated to $L = A \vee B$, meaning either button A or B must be pressed for the lamp to turn on.

Circuit Diagrams The circuit system is documented in circuit diagrams as part of the station documentation. The diagrams are divided by functionality, such that one diagram may show the route locking, while another shows the route release. The diagrams always show the system in the *normal state*. The normal state is the state the system is in when every track section is vacant, no train routes are locked, the points are in their initial position and so on.

The system is operated from the operator's panel.

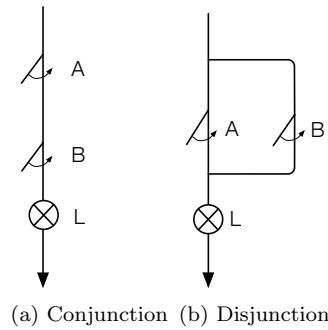


Figure 2.11: Serial connecting components create a conjunctive relationship between them, as seen in (a) where buttons A and B are serial connected to the lamp L. When both buttons are pressed, the current will be able to pass through and turn on the lamp. (b) shows a situation where buttons A and B are connected in parallel to the lamp L. The lamp will turn on if either A or B is pressed.

2.5.1.3 Operator's Panel

The operator's panel shows the layout of the station and is equipped with buttons and lamps. Some buttons are used to switch the position of the points, some are used to lock routes (fig. 2.12), while others again have different purposes. The lamps indicate the state of track sections and points.

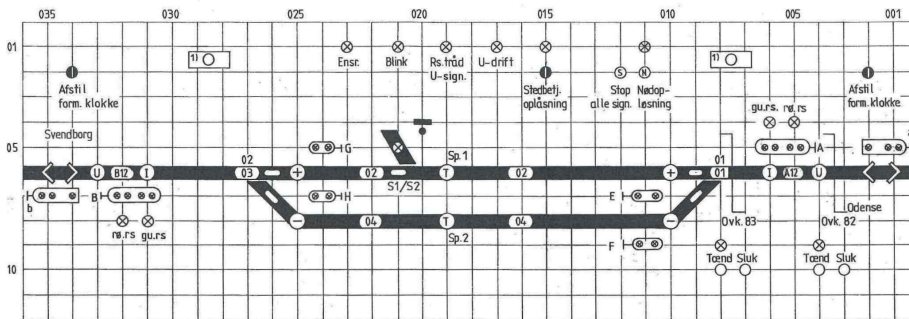


Figure 2.12: Diagram of operator's panel for Stenstrup station.

Switching Points The position of a points section is switched by pressing and holding the + or – button next to the points section. When the position has been

locked in either the plus or minus position a lamp will lit on the corresponding branch.

Locking Routes A route is locked by holding the button for the start point and the end point. The buttons are marked I, U and T. Button I is used to indicate the start of an entry route, while button U is used to indicate the start of an exit route. U or I must be pressed in combination with a T button, which indicates the end of the route.

2.5.2 Train Detection Equipment

Different methods of track detection are adopted throughout the world. Bane-danmark uses track circuits.

2.5.2.1 Track Circuit

A track circuit is a device that detects the absence of trains.

Each rail is connected to a power supply in one end and a relay in the other end. Since the rails are conductive, the relay will be energised by the current running through the rails (fig. 2.13a). Since train wheels and axles are conductive as well, a train on the track section will cause a short circuit, thus cutting the power to the relay. Without power, gravity will cause the relay to drop (fig. 2.13b).

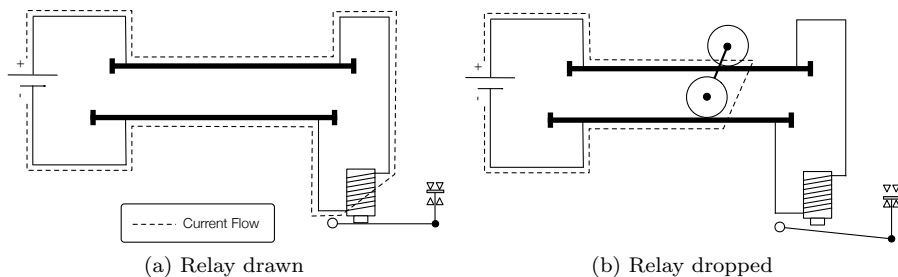


Figure 2.13: As long as the track section is clear, the current can travel through the rails and energise the relay (a). A train on tracks, on the other hand, will cause a short circuit, preventing power to reach the relay, which will drop (b).

As a consequence of this design, the relay will drop in case of power outage as well. This fail-safe feature contributes to the popularity of this method of train detection.

The track sections are isolated so that each section forms its own circuit. In this way the location of trains can be detected in discrete units.

2.6 Interlocking Plan

Part of the station documentation is the interlocking plan. It consists of a station layout diagram, which shows the geography of the station and a train route table, which shows the interlocking rules on a per route basis.

2.6.1 Station Layout

A station layout diagram is an illustration of the physical objects on a station and their relation.

Since it is easier to explain a diagram to someone who is looking at it, let us consider the station layout diagram of Stenstrup station as seen in [fig. 2.14](#).

2.6.1.1 Station Limit

Below signals A and B we see a circle with ST written in it. This marks the station limit. Everything between those marks are part of the station.

As seen there are physical objects outside the station limit which are under the station interlocking systems control. In this case it is the two distant signals a and b, Ovk 82 (level crossing 82). Neither distant signals nor level crossings are considered in this project.

2.6.1.2 Tracks

We see that the station has 6 named track sections, drawn by a thick line. These track sections can be identified as four linears (A12, 02, 04 and B12) and two points sections (01 and 03).

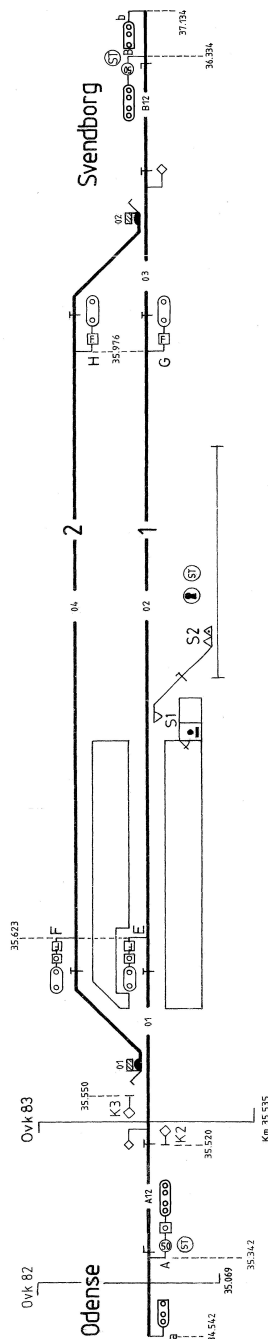


Figure 2.14: Station Layout Diagram of Stenstrup Station.

The thick line extends past the station limit. The part outside the station limits is the open line.

Furthermore there are unnamed two points sections (with point machines S1 and S2) and an unnamed linear section, which are connected to track section 02. These track are not used in the normal operation of the station and are therefore disregarded.

Point machines Note that the point machine on section 01 is called 01, but the point machine on section 03 is called 02.

2.6.1.3 Signals

Apart from the distant signals already discussed, the station has two entry signals (A and B) and four exit signals (E, F, G and H).

The direction the signal is facing can be read from the diagram. The signal can be read if travelling from the direction which has the foot of the signal.

Example: Signal A can be read when travelling from Odense to section A12.

2.6.1.4 Miscellaneous

The large 1 on track section 02 and the 2 on track section 04 are the station track numbers. These are used in the train route table, which is introduced in section [2.6.2](#).

The rest of the information is not relevant for this project.

2.6.2 Train Route Table

The train route table describes the required functionality of the train route based interlocking system.

Figure [2.15](#) shows the train route table of Stenstrup station. This is the original table in Danish. The English terms are used below, but have the Danish word

as reference at the first occurrence as well. The notes used in the train route table, are shown in figure 2.16, where they have been translated to English.

The following will describe how to read a train route table. Empty fields means that there are no requirement to the given object in that route.

2.6.2.1 Train Routes (Togveje)

This section identifies the train routes and their purpose.

no (nr) The identifier of the train route.

Track (spor) Which station track the train route uses (not used in this project).

Overlap (forløb) Describes the end point of the overlap. If the field is empty, there is no overlap for that route. On the train route table of Stenstrup some of the routes have “stræk” under overlap. This means that the overlap extends from the end point of the route all the way to the open line.

2.6.2.2 Signals (Signaler)

The Signals part of the table describes the aspect each signal must show when the route has been locked and all the track sections in the route are vacant. If other rules require it, the signals may show other aspects, when the track sections are occupied (see section 2.6.2.5).

gr means green, gu mean yellow and rø is red.

The entry may also contain a note, either 1, 4 or 5 from figure 2.16. Notes 4 and 5 are about proceed through, which was reduced away in section 2.3.2, so they can be ignored. However, note 1 is important. It allows a signal to show proceed if another route, with G as the start point, has been locked.

Consider train route 2 on Stenstrup. Signal A must indicate proceed, while F and G must show stop (F flank protects and the route stops at G). G may show proceed if route 9 has been locked. There is no requirement for signals B, E and H.

Togveje		Sporskifter		Sporisolationer		Ovk		Stop		Togvejsopl		Gensidige spæringer	
nr	forløb	a	b	A	B	E	F	G	H	I	J	K	L
2	fra Odense	Indk	1	stræk	gr ¹	gr ²	gr ³	gr ⁴	gr ⁵	gr ⁶	gr ⁷	gr ⁸	gr ⁹
3		Indk	2	stræk	gr ¹	gr ²	gr ³	gr ⁴	gr ⁵	gr ⁶	gr ⁷	gr ⁸	gr ⁹
5	fra Svendborg	Indk	1	stræk	gr ¹	gr ²	gr ³	gr ⁴	gr ⁵	gr ⁶	gr ⁷	gr ⁸	gr ⁹
6		Indk	2	stræk	gr ¹	gr ²	gr ³	gr ⁴	gr ⁵	gr ⁶	gr ⁷	gr ⁸	gr ⁹
7	til Odense	Udk	1										
8		Udk	2										
9	til Svendborg	Udk	1										
10		Udk	2										

Figure 2.15: Train Route Table for Stenstrup Station

- 1) Displays *proceed* if an exit route has already been locked.
- 2) Prevented from switching for 44 sec after track circuit ↓ 03.
- 3) Prevented from switching for 43 sec after track circuit ↓ 01.
- 4) Displays *proceed through* if an exit route, from track 1 in the same direction, has already been locked.
- 5) Displays *proceed through* if the entry signal displays *proceed through*.

Figure 2.16: The translated notes of the train route table for Stenstrup (fig. 2.15).

2.6.2.3 Points (Sporskifter)

The column marked Points, explains which position the points must be locked in (if any) before the train route can be locked (The points must remain locked while the train route is locked).

The + symbol means that the given points section must be locked in the plus position, while a – would mean that the points section must be locked in the minus position.

Consider train route 2. Both points 01 and 02 (notice that it refers to the point machines and not the track sections) must be locked in the plus position. Points section 03 (with point machine 02) is not on the route, but is used as part of the safety overlap.

2.6.2.4 Track Sections (Sporisolationer)

The track sections column lists every track section on the station. The field shows which state the track section relay must be in before the route can be locked. Recall that the track section relay will be in the drawn position (↑) when the track is clear.

Consider train route 2. The route can only be locked when all the track sections are vacant. Track sections 03 and B12 are used as part of the safety overlap.

2.6.2.5 Level Crossings (Ovk)

If the field is filled with “Ja” (Yes), then the level crossing must be secured.

2.6.2.6 Signal Release (Stop fald)

When the track section given in the lower part of the field is in the state given at the same location, the signal given at the upper part of the field should change to the stop aspect.

The fields must not be empty.

Consider train route 2. When track section A12 becomes occupied, signal A must change to the stop aspect.

2.6.2.7 Train Route Release (Togvejsopl.)

This describes the sequence that must occur before the train route can be released. “Indl” (Initiation) states the sequence of positions the track relays must have to begin the train route release sequence. “Opl” (Release) gives the sequence of track relay states that most occur for the train route release sequence to end.

The fields must not be empty.

Consider train route 2. The train route can be released when track section 01 is occupied and track section 02 is vacant followed by 02 being occupied and 01 being vacant. This means that the back end of the train has fully left 01 and is fully on 02.

Example Let us consider table 2.3 that specifies the train route release for a route on figure 2.17. The table denotes a release sequence that has to be satisfied to release the train route.

Train Route Release (Togvejsopl.)	
Indl	Opl
↓ A	↓ B
↑ B	↑ A

Table 2.3: Train route release for a route on figure 2.17.

The figure shows three linear track sections, A, B, and C, where the up arrow (\uparrow) beneath the tracks denotes that the track section relay is drawn, and the down arrow (\downarrow) denotes that the track section relay is dropped.

At the first state the train route covering track section A and B is locked. At the second state a train enters section A changing the state of the relay associated with A to drop. At this point the first condition of initiation (Ind1) is satisfied (A is dropped and B is drawn). The train then continues occupying both A and B and then finally stops at B. The second train route release condition (Opl) has been satisfied, by drawing A and dropping B, resulting in the train route is released at the fifth state.

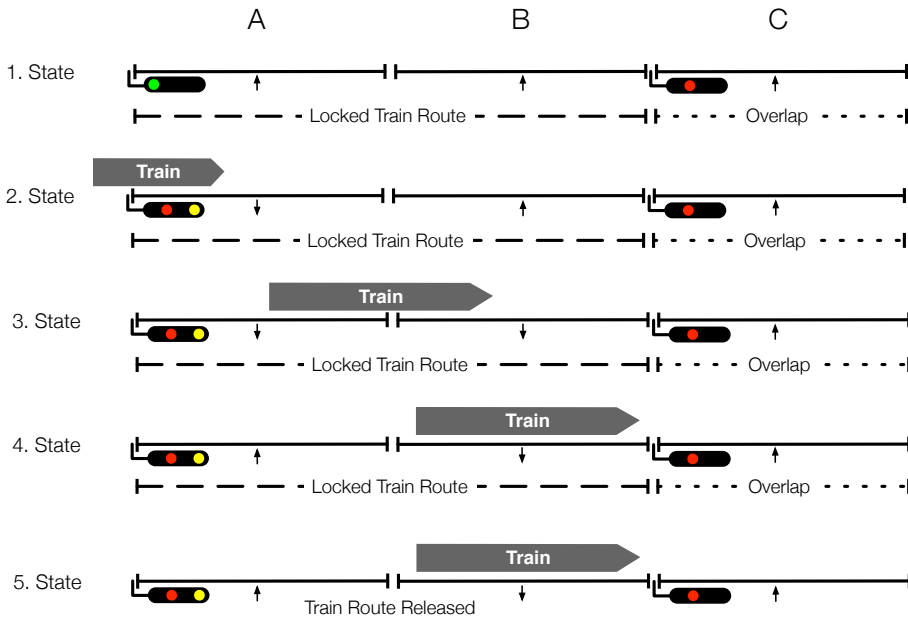


Figure 2.17: Release of train route after the train route release sequence.

2.6.2.8 Mutual Exclusions (Gensidige spærringer)

If a circle is present, the routes must not be locked at the same time.

Consider train route 2. Only train route 9 can be locked at the same time as train route 2. Train route 9 happens to be the exit route from station track 1 in the same direction as train route 2 is the entry route to station track 1. This

means that a combined route through the station can be locked for a train from Odense.

CHAPTER 3

Method Description

This chapter will describe the method used to reach the goal and explain why this method was used.

Section [3.2](#) introduces the steps we made in solving the problem.

3.1 Approach

Our goal is develop a method of generating a model of external events, from station documentation, which can be combined with the already developed model of internal events. The model of internal events is generated as a RSL-SAL transition system. Targeting the model of external events to RSL-SAL as well, means the models would be compatible. With the RSL to SAL translator, developed by Perna [\[11\]](#), the model of the entire interlocking system, acquired by combining the two models, can then be translated to a SAL and model checked (fig. [3.1](#)).

The RSL-SAL model of the interlocking system is to be generated from the station documentation. This means a generator tool is to be developed. By specifying both the station documentation and the generator in RSL, the entire product is



Figure 3.1: A model in RSL-SAL can be translated to SAL automatically.

kept in a single language, by specifying the generator in the translatable subset of RSL. (fig. 3.2).

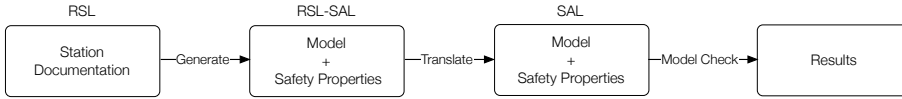


Figure 3.2: An RSL-SAL model can be auto-generated from station documentation, using an executable specification of the generator.

The physical representation of the station documentation is manually converted to the RSL representation and we supply verification checks that helps ensuring the integrity of the documentation.



Figure 3.3: The physical representation of the station documentation is manually converted to the RSL representation.

3.2 Suggested Method

First a suitable data representation of the station documentation needs to be developed. Since we decided to use RSL, data models of *station layout diagrams*, *train route tables* and the *RSL-SAL transition system* are specified in chapter 4. Part of the specification is the well-formedness constraints. These eliminate certain illegal configurations, by specifying not what type of content is allowed, but which specific values or combinations of values.

Next the behavioural model of the external events are modelled. This is divided into developing a model of train movements (chapter 6) and other external events, i.e. the behaviour of points sections and buttons on the operator's panel 7.

Some relations are not captured by either the interlocking plan or the circuit

diagrams. Chapter 8 present a structure that associates variables in the state space of the models with the physical objects of the station.

In order to verify that the model of train movements interact correctly with the model of the interlocking system, consistency conditions have been formulated in chapter 9.

The main goals of this thesis is to develop a method for automated verification of interlocking systems. A big part of this is to identify and formalise the safety properties of an interlocking system. This is done in chapter 10.

Up until this point, everything needed to create a model of the external events have been presented. Chapter 11 shows how the process can be automated by presenting an executable specification of a generation tool.

Chapter 12 will present a test suite that verify that the well-formedness expressions capture the intended properties.

Finally the developed method is applied to a model of Stenstrup (chapter 13). The chapter goes through the entire process of validation, from creating a digital representation of the interlocking plan, to combining the generated model of external events with the model of the internal events and model checking it. It will also present the result of the model check.

Data Models

This chapter will describe the chosen data model of interlocking plans, which was introduced in section [2.6](#) and the chosen data model of RSL-SAL transition systems and assertions. Furthermore, well-formedness expressions are defined, which are used to constrain the model so that certain configurations are disallowed.

First a formal RSL specification is presented and then the data model is applied to Stenstrup as a concrete example of how the data model is intended to be used. The entire specification of all the data models can be found in appendix [E.1](#). The specification introduced in this chapter is slightly simplified in order to better present the important information.

The data model of interlocking plans is introduced in section [4.1](#). Then data models of station layout diagrams and train route tables are given in sections [4.2](#) and [4.3](#) respectively. Lastly, the specification of transition systems is presented in section [4.4](#).

4.1 Interlocking Plan

An interlocking plan consists of a station layout diagram and a train route table. See section [E.1.2](#) for complete specification.

```

type
  InterlockingPlan ::
    sld : Diagram
    trt : TrainRouteTable

```

The interlocking plan is well-formed if both the station layout diagram and the train route table are well-formed.

```

value
  isWfInterlockingPlan : InterlockingPlan → Bool
  isWfInterlockingPlan(ip) ≡
    let d = sld(ip),
        trt = trt(ip)
    in isWfDiagram(d) ∧ isWfTrainRouteTable(trt, d)
  end

```

The specification of station layout diagrams and train route tables are presented next.

4.2 Station Layout Diagram

A data model of station layout diagrams is introduced in this section. The complete specification is written in the file *StationLayout.rsl*, which can be found in the appendix section [E.1.3](#).

4.2.1 Identifiers

As described in section [2.6.1](#) a station layout diagram shows how a collection of physical objects are connected in a certain configuration. Each physical object has its own unique identifier, which is represented as text. The physical objects considered in this project are track sections, point machines and signals, identifiers of which are called *TrackId*, *PointMachineId* and *SignalId* respectively.

```

type
  TrackId = Text,
  PointMachineId = Text,
  SignalId = Text

```

Each object in each category must have an unique identifier. This is ensured by storing them in a set structure as described in sections 4.2.4, 4.2.5 and 4.2.7.

4.2.2 Diagram

The station layout diagram is represented as a record type, which consist of sets of identifiers for the different physical objects and structures that captures their mutual relations. Each of the constituents will be explained in the following sections.

type

```
Diagram ::
  line : TrackId
  allLines : TrackId-set
  allPoints : TrackId-set
  allSignals : SignalId-set
  neighbours : (TrackId × TrackId)-set
  branchNeighbours : (TrackId × TrackId)  $\overline{\mapsto}$  Branch
  pointMachineTrack : PointMachineId  $\overline{\mapsto}$  TrackId
  trackPointMachine : TrackId  $\overline{\mapsto}$  PointMachineId
  signalLocations : (TrackId × TrackId)  $\overline{\mapsto}$  SignalId
```

where Branch is a subtype of PointsState. Recall that a points section can be in either the plus, minus or intermediate position:

type

```
PointsState == plus | minus | arbitrary,
Branch = {| ps : PointsState • ps ≠ arbitrary |},
```

Not every value contained in the types described above forms a valid station. The well-formedness expressions that constrains the data type to disallow certain invalid configurations are shown below:

value

```
isWfDiagram : Diagram → Bool
isWfDiagram(d) ≡
  isWfIdentifiers(d) ∧
  isWfNeighbours(d) ∧
```

```

isWfBranchNeighbours(d) ∧
isWfPointMachines(d) ∧
isWfSignalLocations(d)

```

A track section cannot be both a linear section and a points section.

value

```

isWfIdentifiers : Diagram → Bool
isWfIdentifiers(d) ≡
  allLinears(d) ∩ allPoints(d) = {}

```

The rest of the well-formedness expressions will be explained in the following sections, when the structure they are related to, is explained.

4.2.3 Line

The line is modelled as a single track section.

value

```

line : TrackId = "line"

```

The line is not part of a station as such. However, entry signals are placed on the line at the station limits. As will be explained in further detail in section 4.2.7, signals are placed between pairs of track sections, thus the line is needed as a track section to model signal location is this way.

Alternately, one could choose not to include it at all and model signal placement differently or make a more elaborate solution where the line is unique for each station exit, e.g. "Odense" and "Svendborg" for Stenstrup.

4.2.4 Track Sections

Each track section on a station has a unique identifier. The identifier of each linear section is stored in one set and the identifier of each points section is stored in another set. This is not strictly necessary, but it makes it easy to find, for example, all the points.

value

```
allLinears : TrackId-set,
allPoints  : TrackId-set
```

From the station layout diagram of Stenstrup (fig. 2.14), we see a total of six track sections, whereof two are points (01 and 03) and four are linears (A12, 02, 04 and B12):

value

```
allLinears : TrackId-set = {"A12", "02", "04", "B12"},
allPoints  : TrackId-set = {"01", "03"}
```

The railway track of the station is modelled as a directed graph, where the track sections are nodes and the edges denote which track sections are connected (fig. 4.1).

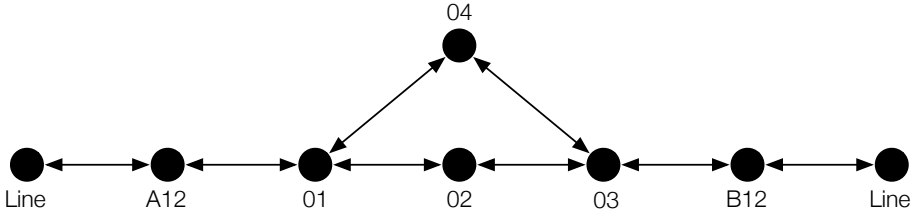


Figure 4.1: The track layout is modelled as a graph, where the track sections are nodes and the edges denote neighbour relations.

The graph structure is captured by the neighbours relation, which is modelled as a set of pairs of track identifiers.

value

```
neighbours : (TrackId × TrackId)-set
```

This set representation is preferred over, for example, an adjacency matrix due to the sparse number of edges.

For Stenstrup station the neighbour relation is:

value

```

neighbours : (TrackId × TrackId)-set =
  { ("line", "A12"), ("A12", "01"),
    ("01", "02"), ("01", "04"),
    ("02", "03"), ("03", "04"),
    ("03", "B12"), ("B12", "line") }

```

Note that the neighbours tuple contains the relationship between (tId1, tId2), but not (tId2, tId1). Doing so is to avoid having both (tId1, tId2) and (tId2, tId1) as entries, which is redundant. Instead the areNeighbours function is introduced to maintain the symmetrical entries by looking up a neighbouring relationship both ways.

value

```

areNeighbours : TrackId × TrackId × Diagram → Bool
areNeighbours(tId1, tId2, d) ≡
  (tId1, tId2) ∈ neighbours(d) ∨
  (tId2, tId1) ∈ neighbours(d)

```

If areNeighbours does not find the neighbour pair (tId1, tId2) it simply looks for the pair (tId2, tId1) instead. If neither pair is in the set, then the two given track sections are not neighbours.

4.2.4.1 Well-formedness of neighbour relation

The following describes the validation checks performed on the data to ensure the data model is correct.

value

```

isWfNeighbours : Diagram → Bool
isWfNeighbours(d) ≡
  neighbours_OnlySections(d) ∧
  neighbours_EverySection(d) ∧
  neighbours_Irreflexive(d) ∧
  neighbours_Antisymmetrical(d) ∧
  linears_1or2Neighbours(d) ∧
  points_3Neighbours(d) ∧
  pointsNotNeighbours(d)

```

1. Contains Only Track Sections The neighbour relation should contain only track sections.

value

```

neighbours_OnlySections : Diagram → Bool
neighbours_OnlySections(d) ≡
  ( ∀ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ neighbours(d) ⇒
      isSection(tId1, d) ∧ isSection(tId2, d) )

```

2. Contains Every Track Section The neighbour relation should contain every track section on the station.

value

```

neighbours_EverySection : Diagram → Bool
neighbours_EverySection(d) ≡
  ( ∀ tId : TrackId • tId ∈ allSections(d) ⇒
    ( ∃ nb : TrackId • nb ∈ allSections(d) ∧
      areNeighbours(tId, nb, d) ) ),

```

where `isSection` is a function that returns true if the given track section is in the given diagram, i.e. the track section is either the line, in `allLinears` or in `allPoints`.

value

```

isSection : TrackId × Diagram → Bool
isSection(tId, d) ≡ tId ∈ allSections(d),

allSections : Diagram → TrackId-set
allSections(d) ≡
  allLinears(d) ∪ allPoints(d) ∪ {line(d)},

```

3. Irreflexive No track section may be neighbour with itself.

value

```

neighbours_Irreflexive : Diagram → Bool
neighbours_Irreflexive(d) ≡
  ( ∀ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ neighbours(d) ⇒
      tId1 ≠ tId2 )

```

4. Anti-symmetric Symmetrical entries are not allowed as they are not required and simply clutter the set, which decreases the readability.

value

```

neighbours_Antisymmetrical : Diagram → Bool
neighbours_Antisymmetrical(d) ≡
  ( ∀ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ neighbours(d) ⇒
      (tId2, tId1) ∉ neighbours(d) )

```

5. Number of Neighbours - Linears Linear track sections must have at least one neighbour and at most two.

value

```

linears_1or2Neighbours : Diagram → Bool
linears_1or2Neighbours(d) ≡
  (∀ tId : TrackId •
    tId ∈ allLinears(d) ⇒
      let n = card getNeighboursOf(tId, d)
      in n ≥ 1 ∧ n ≤ 2
    end)

```

where getNeighboursOf is a function that returns the set of all the given track sections neighbours in the given diagram.

6. Number of Neighbours - Points Points sections must have three neighbours.

value

```

points_3Neighbours : Diagram → Bool
points_3Neighbours(d) ≡
  (∀ tId : TrackId • tId ∈ allPoints(d) ⇒
    card getNeighboursOf(tId, d) = 3)

```

7. Points sections may not be neighbours Two points sections may not be neighbours.

value

```

pointsNotNeighbours : Diagram → Bool
pointsNotNeighbours(d) ≡
  ~( ∃ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ neighbours(d) ∧
    isPoints(tId1, d) ∧ isPoints(tId2, d) )

```

4.2.5 Points

The neighbour relation models the neighbour relationship of all track sections, but it does not give any information about which branch of a point section a given neighbour is connected to. Therefore, an additional data structure, called `branchNeighbours`, is introduced.

value

```

branchNeighbours : (TrackId × TrackId)  $\overrightarrow{m}$  Branch

```

The idea is to label the edges between points sections and their neighbours on the branch end. The label denotes which branch the points section and the neighbour is connected on (fig. 4.2).

The map `branchNeighbours` takes a pair of track ids as key and holds the branch that they are neighbours on as value. The first track id of the key of the pair is the id of the points section and the other is one of its neighbours on the branch side.

value

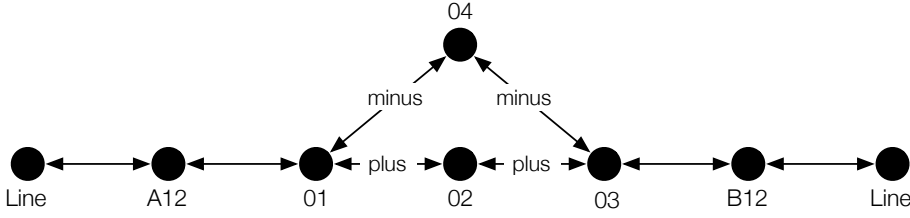


Figure 4.2: The branchNeighbours data structure adds information to the edges of the neighbours relation about which branch the neighbours of a points section are on.

branchNeighbours : (TrackId × TrackId) \overrightarrow{m} Branch =
 [("01", "02") \mapsto plus,
 ("01", "04") \mapsto minus,
 ("03", "02") \mapsto plus,
 ("03", "04") \mapsto minus]

For example, points section 01 has linear section 02 as neighbour on the plus branch.

4.2.5.1 Well-formedness of branchNeighbours

The branchNeighbours map is well-formed if:

1. Each pair of track sections in the domain are neighbours.
2. The first track section of the pair, is a points section.
3. There are two entries for each points section on the station.
4. The points section is not neighbour with the same track section on both branches.

value

isWfBranchNeighbours : Diagram \rightarrow **Bool**
 isWfBranchNeighbours(d) \equiv
 branchNeighbours_areNeighbours(d) \wedge
 branchNeighbours_FirstIsPoints(d) \wedge
 branchNeighbours_everyPoints_2branchNbs(d) \wedge
 branchNeighbours_diffNbsOnBranch(d),

1. Pairs are neighbours Since `branchNeighbours` is supposed to label edges that denote neighbour-ship between the connected nodes, that edge must exist, i.e. all pairs of track sections in domain of the map must be neighbours.

value

```
branchNeighbours_areNeighbours : Diagram → Bool
branchNeighbours_areNeighbours(d) ≡
  ( ∀ (tId1, tId2) : (TrackId × TrackId) •
    (tId1, tId2) ∈ dom branchNeighbours(d) ⇒
      areNeighbours(tId1, tId2, d) )
```

2. First track section is a points section The first track section in each pair of the domain of the map, must be a points section.

value

```
branchNeighbours_FirstIsPoints : Diagram → Bool
branchNeighbours_FirstIsPoints(d) ≡
  ( ∀ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ dom branchNeighbours(d) ⇒
      isPoints(tId1, d) )
```

3. Each points section have two branch side neighbours A points section must have three neighbours, one on the stem side and two on the branch side. Therefore, each points section must have a neighbour connected to each branch.

value

```
branchNeighbours_everyPoints_2branchNbs : Diagram → Bool
branchNeighbours_everyPoints_2branchNbs(d) ≡
  (∀ p : TrackId • p ∈ allPoints(d) ⇒
    (∃ (tId1, tId2) : TrackId × TrackId •
      (tId1, tId2) ∈ dom branchNeighbours(d) ∧
      p = tId1 ∧
      (∃ (tId1', tId2') : TrackId × TrackId •
        (tId1', tId2') ∈ dom branchNeighbours(d) ∧
        p = tId1' ∧ tId2 ≠ tId2')
    )
  )
```

4. The branch side neighbours are unique The neighbour on each branch must be unique.

value

```
branchNeighbours_diffNbsOnBranch : Diagram → Bool
branchNeighbours_diffNbsOnBranch(d) ≡
  (∀ (p, tId1) : TrackId × TrackId •
    (p, tId1) ∈ branchNeighbours(d) ⇒
      (∀ (p', tId2) : TrackId × TrackId •
        (p', tId2) ∈ branchNeighbours(d) ⇒
          (p = p' ∧ tId1 ≠ tId2 ⇒
            (branchNeighbours(d)(p, tId1) = plus ∧
              branchNeighbours(d)(p, tId2) = minus) ∨
            (branchNeighbours(d)(p, tId1) = minus ∧
              branchNeighbours(d)(p, tId2) = plus))))))
```

4.2.6 Point Machines

Each points section has a point machine that control and supervise the points section. The point machine is a distinct entity from the points section. It has its own identifier and has its own relays associated.

The relation of points sections and point machines are modelled with two mappings. One from point machine ids to track ids and one the other way.

type

```
pointMachineTrack : PointMachineId → TrackId
trackPointMachine : TrackId → PointMachineId
```

Strictly, just one of the maps is sufficient, but look ups, in either directions, are performed frequently enough that this data redundancy is acceptable.

Stenstrup has two points sections 01 and 03 with a point machine each, 01 and 02 respectively.

value

```
pointMachineTrack : PointMachineId → TrackId =
  [ "01" ↦ "01",
```



```

    "02"  $\mapsto$  "03"],

    trackPointMachine : TrackId  $\xrightarrow{m}$  PointMachineId =
    [ "01"  $\mapsto$  "01",
      "03"  $\mapsto$  "02"],

```

4.2.6.1 Well-formedness of Point Machines

The maps of point machines are well-formed if

1. The maps are symmetrical.
2. The track sections referred to in the maps are points sections and every points section is referred to.

value

```

isWfPointMachines : Diagram  $\rightarrow$  Bool
isWfPointMachines(d)  $\equiv$ 
  areSymmetrical(pointMachineTrack(d), trackPointMachine(d))  $\wedge$ 
  pointsSectionsExists(d)

```

1. Symmetrical Any mapping $\text{val1} \mapsto \text{val2}$ in one map, must appear as the mapping $\text{val2} \mapsto \text{val1}$ in the other map.

value

```

areSymmetrical : (PointMachineId  $\xrightarrow{m}$  TrackId)  $\times$ 
  (TrackId  $\xrightarrow{m}$  PointMachineId)  $\rightarrow$  Bool
areSymmetrical(pmt, tpm)  $\equiv$ 
  dom pmt = rng tpm  $\wedge$ 
  dom tpm = rng pmt  $\wedge$ 
  (  $\forall$  pmId : PointMachineId • pmId  $\in$  dom pmt  $\Rightarrow$ 
    let tId = pmt(pmId)
    in pmId = tpm(tId)
    end )  $\wedge$ 
  (  $\forall$  tId : TrackId • tId  $\in$  dom tpm  $\Rightarrow$ 
    let pmId = tpm(tId)
    in tId = pmt(pmId)
    end )

```

Notice the lines:

```
dom pmt = rng tpm ∧
dom tpm = rng pmt
```

They provide graceful error handling in our specific implementation in SML. Without them SML will throw an exception, instead of returning false, if a value exist in one map, but not the other.

2. Existing track sections. The ids of track sections referred to in the maps, must be ids of points sections. Furthermore all points sections must have a point machine associated.

```
value
  pointsSectionsExists : Diagram → Bool
  pointsSectionsExists(d) ≡
    dom trackPointMachine(d) = allPoints(d)
```

4.2.7 Signals

Each signal has a unique identifier, the specification of which has already been given at the start of this chapter. Since the range of aspects have been reduced to just proceed and stop (section 2.3.2), there is no need to distinguish between the signal types.

```
value
  allSignals : SignalId-set,
```

The signals are placed between track sections¹, i.e. on the edges of the graph (fig. 4.3). Therefore, the location of signals is represented as a map from a neighbour pair to a signal id. This explains the need for the line construction. Without the line, the track sections on the station border would only have one neighbour. As a result there would be no edge to put the entrance signal on.

¹Actually, the signals are placed just before the separation to the signals facing side, such that a train will have to pass the signal in order to enter the following section.

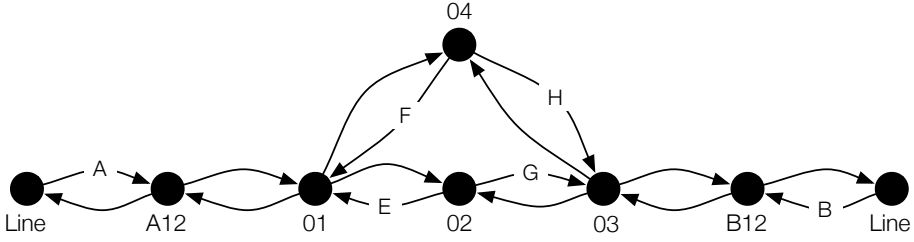


Figure 4.3: The signals are located at the directed edges such that they can be read when travelling in the direction indicated by the arrow.

value

$\text{signalLocations} : (\text{TrackId} \times \text{TrackId}) \xrightarrow{\text{m}} \text{SignalId}$

From the station layout diagram of Stenstrup (fig. 2.14), we see that the station has eight signals in total. The two distant signals (a and b) are not to be modelled, but the two entrance signals (A and B) and four exit signals (E, F, G and H) are. The entrance and exit signals are added to the set of all signals.

value

$\text{allSignals} : \text{SignalId-set} = \{ "A", "B", "E", "F", "G", "H" \},$

Signals are located between track sections and is only readable from one direction. Therefore they can be placed on the appropriate edge on the graph. The signal can be read when travelling in the direction indicated by the arrow.

value

$\text{signalLocations} : (\text{TrackId} \times \text{TrackId}) \xrightarrow{\text{m}} \text{SignalId} =$
 $[("line", "A12") \mapsto "A", ("line", "B12") \mapsto "B",$
 $("02", "01") \mapsto "E", ("04", "01") \mapsto "F",$
 $("02", "03") \mapsto "G", ("04", "03") \mapsto "H"],$

In the example above, signal A is located between the line and A12 and readable when travelling from line to A12.

4.2.7.1 Signal Location Well-formedness

The signal location map is well-formed if the signals used in the map are the same as the ones on the station and they are placed on edges between neighbours.

value

```
isWfSignalLocations : Diagram → Bool
isWfSignalLocations(d) ≡
  signalLocationUsesExistingSignals(d) ∧
  signalsArePlacedAtValidNeighbours(d)
```

1. Every signal is given a location. Every signal on the station is given a location and only existing signals are given a location.

value

```
signalLocationUsesExistingSignals : Diagram → Bool
signalLocationUsesExistingSignals(d) ≡
  rng signalLocations(d) = allSignals(d)
```

2. Signals are placed on valid edges. The signals must be placed between track sections, i.e. on edges that is given by the neighbours relation.

value

```
signalsArePlacedAtValidNeighbours : Diagram → Bool
signalsArePlacedAtValidNeighbours(d) ≡
  ( ∀ (id1, id2) : (TrackId × TrackId) •
    (id1, id2) ∈ dom signalLocations(d) ⇒
      areNeighbours(id1, id2, d) )
```

4.3 Train Route Table

A data model of the train route table is introduced in this section. The complete specification is given in the file `TrainRouteTable.rsl`, which can be found in the appendix section [E.1.4](#).

The train route table is modelled as a mapping between train routes and the data the row contains.

```
type
  TrainRouteTable = TrainRouteId  $\overrightarrow{m}$  DataRow
```

The train route identifier is specified in the usual way:

```
type
  TrainRouteId = Text
```

and DataRow is specified in section 4.3.2.

The train route table of Stenstrup (fig. 2.15) shows eight routes, numbered 2, 3, 5, 6, 7, 8, 9 and 10. For each train route a DataRow is created. An example of this is given after the specification in section 4.3.2.

```
value
  stenstrupTRT : TrainRouteTable =
    [ "2"  $\mapsto$  dataRow2, "3"  $\mapsto$  dataRow3,
      "5"  $\mapsto$  dataRow5, "6"  $\mapsto$  dataRow6,
      "7"  $\mapsto$  dataRow7, "8"  $\mapsto$  dataRow8,
      "9"  $\mapsto$  dataRow9, "10"  $\mapsto$  dataRow10 ]
```

4.3.1 Well-formedness of Train Route Table

A train route table is well-formed if every data row it contains is well-formed. Also, the train route ids in the domain should be mutually distinct, but the map structure ensures this property.

```
value
  isWfTrainRouteTable : TrainRouteTable  $\times$  Diagram  $\rightarrow$  Bool
  isWfTrainRouteTable(trt, d)  $\equiv$ 
    ( $\forall$  trId : TrainRouteId  $\bullet$  trId  $\in$  dom trt  $\Rightarrow$ 
      isWfDataRow(trId, d, trt)),
```

4.3.2 DataRow

The data in each row is stored in a record type called `DataRow`. Each part of the train route table gets its own entry and are described in detail in the following sections.

type

```
DataRow ::
  overlap : TrackId-set
  signals : SignalId  $\overrightarrow{m}$  Aspect
  points : PointMachineId  $\overrightarrow{m}$  PointsState
  trainDetection : TrackId  $\overrightarrow{m}$  Bool
  signalRelease : SignalId  $\times$  TrackId
  trainRouteRelease : ReleaseSequence
  mutuallyBlocking : TrainRouteId  $\overrightarrow{m}$  Bool,
```

As an example, consider train route 2 on Stenstrup. The creation of each of the constituents are shown separately in the following sections to improve the readability.

value

```
dataRow2 : DataRow =
  mk_DataRow(
    overlap2,
    signals2,
    points2,
    trainDetection2,
    signalRelease2,
    trainRouteRelease2,
    mutuallyBlocking2
  ),
```

4.3.2.1 Well-formedness of DataRow

A data row is well-formed if each of its constituents are well-formed.

value

```

isWfDataRow : TrainRouteId × Diagram × TrainRouteTable → Bool
isWfDataRow(trId, d, trt) ≡
  let dr = trt(trId)
  in isWfOverlap(dr, d) ∧
    isWfSignals(dr, d) ∧
    isWfPoints(dr, d) ∧
    isWfTrainDetection(trainDetection(dr), d) ∧
    isWfSignalRelease(signalRelease(dr), d) ∧
    isWfTrainRouteRelease(trainRouteRelease(dr), d) ∧
    isWfMutuallyBlocking(trId, trt)
end

```

4.3.3 Overlap

Overlap is modelled as a set of track ids.

```

value
  overlap : TrackId-set

```

If there is no overlap, then the set is empty. Otherwise the set consist of the ids of the track sections that are part of the overlap. This varies slightly from the paper table, where it is either “stræk” or empty. It is changed for a few reasons.

First of all, the information is static. It only changes if the layout of the station or the routes are changed. If something is changed, then the interlocking plan should be updated accordingly anyway, so the details about the overlap could be changed at that point as well.

Secondly, since the data is static, it seems needless to compute it every single time it is used. Even if computed only once and saved in an appropriate structure, “stræk” would then redundant. Besides computing the overlap is difficult, since the train route table does not explicitly state the start and end points of routes.

Consider the overlap of route 2 on Stenstrup. The stretch from the end point of route 2 to the line is track sections 03 and B12 and hence they are added as a set to the overlap entry.

```

value

```

$\text{overlap2} : \text{TrackId-set} = \{''03'', ''B12''\}$

4.3.3.1 Well-formedness of Overlap

If the following hold, the overlap is considered well-formed.

1. If there is an overlap for a given route, then the track sections of the overlap must be vacant when the route is locked.
2. If there is an overlap, the track sections of the overlap must form a connected path.
3. If there is an overlap, then it must be an extension of the route, i.e. only one section of the overlap may be neighbour with a section from the route.

value

```

isWfOverlap : DataRow × Diagram → Bool
isWfOverlap(dr, d) ≡
  let overlap = overlap(dr)
  in card overlap > 0 ⇒
    overlapCovered(overlap, trainDetection(dr)) ∧
    overlapIsConnected(overlap, d) ∧
    endOfRoute(dr, d)
end

```

1. Overlap Sections in Train Detection Track sections which are part of the overlap, must be covered by the route.

value

```

overlapCovered : TrackId-set × (TrackId  $\overline{\Rightarrow}$  Bool) → Bool
overlapCovered(overlap, trainDetection) ≡
  overlap ⊆ coveredSections(trainDetection)

```

where the covered track sections, are the sections that are required to be vacant, when the route is locked.

value

```
coveredSections : (TrackId  $\xrightarrow{m}$  Bool)  $\rightarrow$  TrackId-set
coveredSections(trainDetection)  $\equiv$ 
  { tId | tId : TrackId • tId  $\in$  dom trainDetection  $\wedge$ 
    trainDetection(tId)},
```

2. Overlap is connected Any non-empty overlap, must form a connected path.

value

```
overlapIsConnected : TrackId-set  $\times$  Diagram  $\rightarrow$  Bool
overlapIsConnected(overlap, d)  $\equiv$ 
  areConnected(overlap, d)
```

where is a function that attempts to create a connected path through the set of track sections it is given, using the neighbour relation from the station layout diagram it is given as input.

value

```
areConnected : TrackId-set  $\times$  Diagram  $\rightarrow$  Bool
areConnected(sections, d)  $\equiv$ 
  if sections = {} then true else
    let s = hd sections, sections' = sections \ {s}
    in findConnected({s}, {s}, sections, d) = sections
    end
  end,

findConnected : TrackId-set  $\times$  TrackId-set  $\times$ 
  TrackId-set  $\times$  Diagram  $\rightarrow$  TrackId-set
findConnected(queue, visited, sectionsInRoute, d)  $\equiv$ 
  if queue = {} then visited else
    let q = hd queue,
        nbs = getNeighboursOf(q, d)  $\cap$  sectionsInRoute,
        queue' = (queue \ {q})  $\cup$  (nbs \ visited),
        visited' = visited  $\cup$  {q}
    in findConnected(queue', visited', sectionsInRoute, d)
    end
  end
```

3. Overlap is extension The overlap must be an extension of the route, i.e. there is only a single track section from the overlap, which is neighbour with a track section from the route.

```

value
  endOfRoute : DataRow × Diagram → Bool
  endOfRoute(dr, d) ≡
    let overlap = overlap(dr),
        coveredSections = coveredSections(trainDetection(dr)),
        route = coveredSections \ overlap
    in ( ∃! tId : TrackId • tId ∈ overlap ∧
        let nbs = getNeighboursOf(tId, d),
            nbsInOverlap = nbs ∩ route
        in card nbsInOverlap ≥ 1
        end
    )
end

```

4.3.4 Signals

The signals section is specified as a mapping from each signal on the station to required aspect.

```

value
  signals : SignalId  $\overline{\mapsto}$  Aspect

```

Recall from section 2.3.2 that the possible aspects has been reduced to *proceed* and *stop*. These are modelled as *gr* and *re* respectively. If there is no requirement to the aspect, *arbitrary* can written.

```

type
  Aspect == gr | re | arbitrary,

```

Consider the signals part for train route 2 on Stenstrup. We see that signal A is required to show proceed, while F and G should show stop. There are no requirements about the aspect of the rest of the signals.

value

```
signals2 : SignalId  $\overrightarrow{m}$  Aspect =
  [ "A"  $\mapsto$  gr, "B"  $\mapsto$  arbitrary, "E"  $\mapsto$  arbitrary,
    "F"  $\mapsto$  re, "G"  $\mapsto$  re, "H"  $\mapsto$  arbitrary ]
```

4.3.4.1 Well-formedness of Signals

Every signal on the station must be present as keys to the map structure. Furthermore the entry signal must show proceed aspect.

value

```
isWfSignals : DataRow  $\times$  Diagram  $\rightarrow$  Bool
isWfSignals(dr, d)  $\equiv$ 
  let signals = signals(dr)
  in existingSignals(signals, d)  $\wedge$ 
    proceedEntry(getEntrySignal(dr), signals)
end
```

where the entry signal is extracted from the signal release entry.

value

```
getEntrySignal : DataRow  $\rightarrow$  SignalId
getEntrySignal(dr)  $\equiv$ 
  let (sId, tId) = signalRelease(dr) in sId end
```

1. Existing Signals

value

```
existingSignals : (SignalId  $\overrightarrow{m}$  Aspect)  $\times$  Diagram  $\rightarrow$  Bool
existingSignals(signals, d)  $\equiv$ 
  dom signals = allSignals(d)
```

2. Entry Signal Shows Proceed

value

```

proceedEntry : SignalId × (SignalId  $\xrightarrow{m}$  Aspect) → Bool
proceedEntry(entrySignal, signals) ≡
  signals(entrySignal) = gr

```

4.3.5 Points

The part of the train route table called Points is modelled as a map from the id of each point machine to the required position of the points section, if any.

value

```

points : PointMachineId  $\xrightarrow{m}$  PointsState

```

where PointsState is defined as

type

```

PointsState == plus | minus | arbitrary

```

Consider train route 2 on Stenstrup. Both points must be set to plus.

value

```

points : PointMachineId  $\xrightarrow{m}$  PointsState = [ "01" ↦ plus, "02" ↦ plus ]

```

4.3.5.1 Well-formedness of Points

The points part of the train route table is well-formed if:

1. Any track section associated with a point machine is part of the station.
2. For every points section, its associated point machine is in points.
3. Every points section on the route has a required position.

4. The point position matches the route.

value

```

isWfPoints : DataRow × Diagram → Bool
isWfPoints(dr, d) ≡
  let points = points(dr)
  in pointMachinesExist(dom points, d) ∧
    pointMachinesInTRT(dr, d) ∧
    pointsInRouteLocked(dr, d) ∧
    correctPos(dr, d)
end

```

- 1. Point machines exist** The point machines in the train route table are also defined in the station layout diagram.

value

```

pointMachinesExist : PointMachineId-set × Diagram → Bool
pointMachinesExist(points, d) ≡
  ( ∀ pId : PointMachineId • pId ∈ points ⇒
    pId ∈ dom pointMachineTrack(d) ∧
    let tId = pointMachineTrack(d)(pId)
    in tId ∈ allPoints(d)
  end )

```

- 2. Point machine given** The point machine associated with each points section of the station, must be in the points section of the train route table.

value

```

pointMachinesInTRT : DataRow × Diagram → Bool
pointMachinesInTRT(dr, d) ≡
  ( ∀ tId : TrackId •
    tId ∈ dom trainDetection(dr) ⇒
    isPoints(tId, d) ⇒
    let pmId = trackPointMachine(d)(tId)
    in pmId ∈ dom points(dr)
  end ),

```

3. Covered points sections are required to be locked Each points section, which is part of a route, must be required to be locked in either plus or minus.

value

```

pointsInRouteLocked : DataRow × Diagram → Bool
pointsInRouteLocked(dr, d) ≡
  ( ∀ pId : PointMachineId • pId ∈ points(dr) ⇒
    let pos = points(dr)(pId),
        tId = pointMachineTrack(d)(pId)
    in tId ∈ coveredSections(trainDetection(dr)) ⇒
      (pos = plus ∨ pos = minus)
    end ),

```

4. The point position matches the route Every points section, which is part of the route and has a neighbour on the branch side, which is also part of the route, must be locked into the position that allows movement to that neighbour.

value

```

correctPos : DataRow × Diagram → Bool
correctPos(dr, d) ≡
  ( ∀ pId : TrackId •
    pId ∈ coveredSections(trainDetection(dr)) ∩
    allPoints(d) ⇒
    ( ∀ nb : TrackId •
      nb ∈ getNeighboursOf(pId, d) ∩
      coveredSections(trainDetection(dr)) ⇒
      isBranchNb(pId, nb, d) ⇒
      points(dr)(trackPointMachine(d)(pId)) =
      branchNeighbours(d)(pId, nb) ))
  )

```

4.3.6 Train Detection

Train detection is a mapping from track ids to a boolean value. The value is true if the track is required to be vacant and false if not.

value

```

trainDetection : TrackId  $\overrightarrow{m}$  Bool

```

In order to lock train route 2 on Stenstrup, every track section, except 04, must be vacant.

value

```
trainDetection : TrackId  $\overrightarrow{\mapsto}$  Bool =
  [ "A12"  $\mapsto$  true, "01"  $\mapsto$  true, "02"  $\mapsto$  true,
    "04"  $\mapsto$  false, "03"  $\mapsto$  true, "B12"  $\mapsto$  true ]
```

4.3.6.1 Well-formedness of Train Detection

1. The domain of the map must contain every track section on the station
2. At least one track section must be part of the route.
3. The track sections that are part of the route, must form a connected path.
4. The path may not cover both branches of a points section.

value

```
isWfTrainDetection : (TrackId  $\overrightarrow{\mapsto}$  Bool)  $\times$  Diagram  $\rightarrow$  Bool
isWfTrainDetection(trainDetection, d)  $\equiv$ 
  existingTrackSections(dom trainDetection, d)  $\wedge$ 
  atLeastOneSectionInRoute(trainDetection)  $\wedge$ 
  connectedRoute(trainDetection, d)  $\wedge$ 
  justOneBranchNeighbour(trainDetection, d)
```

1. Existing Track Sections The track sections in train detection must also be defined in the station layout diagram.

value

```
existingTrackSections : TrackId-set  $\times$  Diagram  $\rightarrow$  Bool
existingTrackSections(sections, d)  $\equiv$ 
  sections = allLinears(d)  $\cup$  allPoints(d),
```

2. At Least One Section In Route At least one of the track sections in train detection must be part of the route.

value

```
atLeastOneSectionInRoute : (TrackId  $\xrightarrow{m}$  Bool)  $\rightarrow$  Bool
atLeastOneSectionInRoute(trainDetection)  $\equiv$ 
  ( $\exists$  b : Bool • b  $\in$  rng trainDetection  $\wedge$  b)
```

3. Connected The track sections in the route, must form a connected path.

value

```
connectedRoute : (TrackId  $\xrightarrow{m}$  Bool)  $\times$  Diagram  $\rightarrow$  Bool
connectedRoute(trainDetection, d)  $\equiv$ 
  areConnected(coveredSections(trainDetection), d)
```

4. No path over both branches A train cannot travel from one branch to the other on a points section. Therefore routes may not have this configuration either.

value

```
justOneBranchNeighbour :
  (TrackId  $\xrightarrow{m}$  Bool)  $\times$  Diagram  $\rightarrow$  Bool
justOneBranchNeighbour(trainDetection, d)  $\equiv$ 
  (  $\forall$  tId : TrackId •
    tId  $\in$  coveredSections(trainDetection)  $\Rightarrow$ 
      /* Only interested in points sections */
      isPoints(tId, d)  $\Rightarrow$ 
        /* Does not have two branch side neighbour */
         $\sim$ (  $\exists$  bNb1 : TrackId •
          bNb1  $\in$  coveredSections(trainDetection)  $\wedge$ 
          isBranchNb(tId, bNb1, d)  $\wedge$ 
          (  $\exists$  bNb2 : TrackId •
            bNb2  $\in$  coveredSections(trainDetection)  $\wedge$ 
            isBranchNb(tId, bNb2, d)  $\wedge$ 
            bNb1  $\neq$  bNb2 )
        )
  )
```


4.3.7 Signal Release

Signal release is modelled as a pair of a signal id and a track id.

value

$\text{signalRelease} : \text{SignalId} \times \text{TrackId}$

When the given track becomes occupied, the given signal will be forced to release back to stop. The down arrow shown in the train route table is not modelled, as it is always the case that it is dropped when the track section is occupied.

In train route 2 on Stenstrup, signal A should release back to stop when track section A12 becomes occupied.

value

$\text{signalRelease} : \text{SignalId} \times \text{TrackId} = (\text{"A"}, \text{"A12"})$

4.3.7.1 Well-formedness of Signal Release

The given signal id must match a signal on the station and the track section must be part of the station. Furthermore, that signal must be on an edge between that track section and another and must be readable when travelling from the track section to its neighbour.

value

$\text{isWfSignalRelease} : (\text{SignalId} \times \text{TrackId}) \times \text{Diagram} \rightarrow \text{Bool}$
 $\text{isWfSignalRelease}((\text{sId}, \text{tId}), \text{d}) \equiv$
 $\text{sId} \in \text{allSignals}(\text{d}) \wedge$
 $\text{tId} \in \text{allLinears}(\text{d}) \cup \text{allPoints}(\text{d}) \wedge$
 $\text{signalFollowedByTrackSection}(\text{sId}, \text{tId}, \text{d}),$

where the given track section ts is directly after the signal if there exist another track section ts' such that the given signal is located on the edge going from ts' to ts .

value

```

signalFollowedByTrackSection : SignalId × TrackId × Diagram → Bool
signalFollowedByTrackSection(sId, tId, d) ≡
  (∃ tId2 : TrackId • tId2 ∈ allSections(d) ∧
    (tId2, tId) ∈ dom signalLocations(d) ∧
    signalLocations(d)(tId2, tId) = sId
  )

```

4.3.8 Train Route Release

Train Route Release is modelled as a pair of release condition pairs.

value

```

trainRouteRelease : ReleaseSequence

```

type

```

ReleaseSequence = ReleaseCond × ReleaseCond,
ReleaseCond = (TrackId × RelayState) × (TrackId × RelayState)

```

A release condition pair consists of a track id and a relay state, which can be either drawn or dropped.

type

```

RelayState == drawn | dropped,

```

Consider train route 2 on Stenstrup. The route is released when track section 01 is occupied, while track section 02 is vacant, followed by 01 being vacant and 02 occupied.

value

```

trainRouteRelease : ReleaseSequence =
  ( ("01", dropped), ("02", drawn)),
  (("02", dropped), ("01", drawn)) )

```

4.3.8.1 Well-formedness of Train Route Release

The release sequence follow a strict pattern. First the first track section of the release sequence must occupied, while the second is vacant. Then the second track section must be occupied, while the first is vacant.

```

value
  isWfTrainRouteRelease : DataRow × Diagram → Bool
  isWfTrainRouteRelease(dr, d) ≡
    let (indl, opl) = trainRouteRelease(dr),
      ((tId1, rs1), (tId2, rs2)) = indl,
      ((tId3, rs3), (tId4, rs4)) = opl
    in reverseTrackSections(tId1, tId2, tId3, tId4) ∧
      reverseRelayState(rs1, rs2, rs3, rs4) ∧
      trackSectionPartOfRoute(tId1, dr) ∧
      trackSectionPartOfRoute(tId2, dr) ∧
      properRelayState(rs1, rs2) ∧
      areNeighbours(tId1, tId2, d)
    end

```

Therefore, for the sequence to be well-formed, the track sections in the first release condition, must be same as in the second condition.

```

value
  reverseTrackSections : TrackId × TrackId ×
    TrackId × TrackId → Bool
  reverseTrackSections(tId1, tId2, tId3, tId4) ≡
    tId1 = tId4 ∧ tId2 = tId3

```

The track sections must be part of the route.

```

value
  trackSectionPartOfRoute : TrackId × DataRow → Bool
  trackSectionPartOfRoute(tId, dr) ≡
    tId ∈ sectionsInRoute(trainDetection(dr)),

```

The state of the relays must follow the pattern dropped, drawn, dropped, drawn.

value

```
reverseRelayState : RelayState × RelayState ×
  RelayState × RelayState → Bool
reverseRelayState(rs1, rs2, rs3, rs4) ≡
  rs1 = rs3 ∧ rs2 = rs4,

properRelayState : RelayState × RelayState → Bool
properRelayState(rs1, rs2) ≡
  rs1 = dropped ∧ rs2 = drawn
```

4.3.9 Mutually Blocking

Mutually blocking is a mapping from train route ids to a boolean value. If the value is true, then the routes are conflicting with each other.

value

```
mutuallyBlocking : TrainRouteId  $\overline{\mapsto}$  Bool
```

Train route 2 on Stenstrup conflicts with every other train route except for route 9.

value

```
mutuallyBlocking : TrainRouteId  $\overline{\mapsto}$  Bool =
  [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
    "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ true ]
```

Well-formedness of Mutually Blocking Mutually Blocking is well-formed if:

1. Must have an entry for every train route and no other.
2. If two routes share at least one track section, which is in neither routes overlap, then the two routes are conflicting.
3. If two routes require different positioning of a points section, then the two routes are conflicting.

value

```

isWfMutuallyBlocking : TrainRouteId × TrainRouteTable → Bool
isWfMutuallyBlocking(trId, trt) ≡
  let dr = trt(trId)
  in entryForEveryRoute(mutuallyBlocking(dr), trt) ∧
    conflictingWhenShareSection(trId, trt) ∧
    conflictingWhenDifferentPos(trId, trt)
end

```

1. Every train route has an entry The domain of the mutually blocking mapping, must be exactly the set of train routes in the train route table.

value

```

entryForEveryRoute : (TrainRouteId  $\overline{\mapsto}$  Bool) ×
  TrainRouteTable → Bool
entryForEveryRoute(mutuallyBlocking, trt) ≡
  dom mutuallyBlocking ⊆ dom trt ∧
  dom mutuallyBlocking ⊇ dom trt

```

2. Conflicting Routes When Sharing Sections If two routes share a track section, then they must be conflicting.

value

```

conflictingWhenShareSection : TrainRouteId × TrainRouteId ×
  TrainRouteTable → Bool
conflictingWhenShareSection(route1, route2, trt) ≡
  let -- Route 1
    dr1 = trt(route1),
    sections1 = coveredSections(trainDetection(dr1)),
    overlap1 = overlap(dr1),
    sectionsNoOverlap1 = sections1 \ overlap1,
    -- Route 2
    dr2 = trt(route2),
    sections2 = coveredSections(trainDetection(dr2)),
    overlap2 = overlap(dr2),
    sectionsNoOverlap2 = sections2 \ overlap2
  in card (sectionsNoOverlap1 ∩ sectionsNoOverlap2) > 0 ⇒
    mutuallyBlocking(dr1)(route2)
end

```

This must hold for every combination of routes:

```

value
  conflictingWhenShareSection : TrainRouteId ×
    TrainRouteTable → Bool
  conflictingWhenShareSection(route1, trt) ≡
    let routes = dom trt
    in conflictingWhenShareSection(route1, routes, trt)
  end,

  conflictingWhenShareSection : TrainRouteId ×
    TrainRouteId-set × TrainRouteTable → Bool
  conflictingWhenShareSection(route1, routes, trt) ≡
    if routes = {} then true else
      let route2 = hd routes, routes' = routes \ {route2}
      in conflictingWhenShareSection(route1, route2, trt) ∧
        conflictingWhenShareSection(route1, routes', trt) -- Recursive Call
    end
  end

```

3. Conflicting if different position Two routes which require different positioning of a points sections are conflicting.

```

value
  conflictingWhenDifferentPos : TrainRouteId × TrainRouteId ×
    TrainRouteTable → Bool
  conflictingWhenDifferentPos(route1, route2, trt) ≡
    let -- Route 1
      dr1 = trt(route1),
      points1 = points(trt(route1)),
      -- Route 2
      dr2 = trt(route2),
      points2 = points(dr2)
    in ( ∃ pmId : SL.PointMachineId •
      pmId ∈ dom points1 ∧
      pmId ∈ dom points2 ∧
      points1(pmId) ≠ T.arbitrary ∧
      points2(pmId) ≠ T.arbitrary ∧
      points1(pmId) ≠ points2(pmId)
    ) ⇒ mutuallyBlocking(dr1)(route2)
  end

```

This must hold for every combination of routes:

value

```

conflictingWhenDifferentPos : TrainRouteId × TrainRouteTable → Bool
conflictingWhenDifferentPos(route1, trt) ≡
  let routes = dom trt
  in conflictingWhenDifferentPos(route1, routes, trt)
end,

conflictingWhenDifferentPos : TrainRouteId ×
  TrainRouteId-set × TrainRouteTable → Bool
conflictingWhenDifferentPos(route1, routes, trt) ≡
  if routes = {} then true else
    let route2 = hd routes,
      routes' = routes \ {route2}
    in conflictingWhenDifferentPos(route1, route2, trt) ∧
      conflictingWhenDifferentPos(route1, routes', trt) -- Recursive Call
  end
end

```

4.4 Transition System

The data model of the RSL-SAL transition system is introduced in this section. It is an extension of the structure introduced by Kjær and Le Bliguet [2]. The complete specification is written in the file `TransitionSystem.rsl`, which can be found in appendix section E.1.5.

A transition system has a name and consists of a state space, a set of transitions and a set of assertions.

type

```

TransitionSystem ::
  name : Text
  state : Var-set
  transitionRules : TransitionRule-set
  assertions : Assertion-set

```

Assertions are not generally considered to be part of a transition system. Normally one would supply a model checker with a system model (the transition system)

and a property specification (assertions). However, we wanted to be able to generate both the transition system and the assertions with a single call to the generator. With assertions as part of the transition system, the generator can pass both the transition system and the assertions directly to the unparser. Alternatively, a wrapper structure that contains the transition system and the assertions could solve the same problem.

A transition system is well-formed if the name, the state space, the transitions and the assertions are all well-formed.

value

```
isWfTransitionSystem : TransitionSystem → Bool
isWfTransitionSystem(ts) ≡
  isWfName(name(ts)) ∧
  isWfState(state(ts)) ∧
  isWfTransitionRules(ts) ∧
  isWfAssertions(assertions(ts))
```

4.4.1 Transition System Name

The transition system may not have the empty string as name.

value

```
isWfName : Text → Bool
isWfName(name) ≡
  name ≠ ""
```

4.4.2 State Space

The state space is a set of variables. A variable have an identifier and a value.

type

```
Var ::
  id : VarId
  val : Val
```


where the id of the variable is modelled as text:

```
type
  VarId = Text
```

and the value can be either of type boolean or natural number.

$$\text{Val} == \text{mk_BoolVal}(b : \mathbf{Bool}) \mid \text{mk_NatVal}(n : \mathbf{Nat})$$

These are the value types used in the behavioural model, which is introduced in section 6.2. Additional could be added if needed.

4.4.2.1 Well-formedness of State Space

The state space is well-formed if each variable has a unique id and the id of each variable is valid.

```
value
  isWfState : Var-set → Bool
  isWfState(vars) ≡ uniqueIds(vars) ∧ validIds(vars)
```

1. Unique Ids An id is unique if there does not exist another variable in the same state space with the same id.

```
value
  uniqueIds : Var-set → Bool
  uniqueIds(vars) ≡
    ( ∀ var1 : Var • var1 ∈ vars ⇒
      ~ ( ∃ var2 : Var • var2 ∈ vars ∧
          id(var1) = id(var2) ∧ var1 ≠ var2 )
    )
```

2. Valid Ids The id is not allowed to begin with a number and may not be the empty string.

value

```
validIds : Var-set → Bool
validIds(vars) ≡
  ( ∀ var : Var • var ∈ vars ⇒
    id(var) ≠ "" ∧
    let char = hd id(var)
    in char ≠ '0' ∧ char ≠ '1' ∧
      char ≠ '2' ∧ char ≠ '3' ∧
      char ≠ '4' ∧ char ≠ '5' ∧
      char ≠ '6' ∧ char ≠ '7' ∧
      char ≠ '8' ∧ char ≠ '9'
    end )
```

4.4.3 Transition Rules

Transition rules are the transitions in the transition system. Each transition has a name, a guard and an assignment². The guard limits the states a given transition can be taken from, while the assignment changes the state by assigning different values to the variables in the state space.

type

```
TransitionRule ::
  name : Text
  guard : BooleanExp
  assignments : MultipleAssignment
```

where a boolean expression is defined in the following way:

type

```
BooleanExp ==
  and(a : BooleanExp-set) |
  or(o : BooleanExp-set) |
  neg(n : BooleanExp) |
```

²Sometimes called update.

```

lessthan(ArithmeticExp, ArithmeticExp) |
greaterthan(ArithmeticExp, ArithmeticExp) |
geq(ArithmeticExp, ArithmeticExp) | /* greater than equal */
equals(ArithmeticExp, ArithmeticExp) |
literal(id : VarId) |
bool(b : Bool)

```

using the following definition of arithmetic expressions:

```

type
  ArithmeticExp ==
    add(a : ArithmeticExp-set) |
    sub(s : ArithmeticExp-set) |
    literal(l : VarId),

```

The update part of a transition rule consists of a set of assignments as well:

```

type
  MultipleAssignment = Assignment-set,

  Assignment ::
    id : VarId
    assign : AssignExp,

  AssignExp == BoolAssign(ba : Bool) |
               NatAssign(na : Nat) |
               AddAssign(aa : Nat) |
               SubAssign(sa : Nat) ,

```

4.4.3.1 Well-formedness of Transition Rules

The following must hold for transitions.

1. Names must be valid.
2. Names must be unique.
3. Value assignments to existing variables.

4. Variables in guard must be part of the state space.
5. At least one assignment in update.
6. Each variable may only be updated once.

value

```

isWfTransitionRules : TransitionSystem → Bool
isWfTransitionRules(ts) ≡
  let state = state(ts),
      trs = transitionRules(ts)
  in validNames(trs) ∧
      uniqueNames(trs) ∧
      assignmentToExistingIds(ts) ∧
      idsInGuardMustBeInState(state, trs) ∧
      nonemptyAssignment(trs) ∧
      idUpdOnlyOnce(trs)
end

```

1. Names must be nonempty The empty string is not a valid name for a transition rule.

value

```

validNames : TransitionRule-set → Bool
validNames(trs) ≡
  ( ∀ tr : TransitionRule • tr ∈ trs ⇒
    ~isEmptyString(name(tr))
  )

```

2. Names must be unique Two transitions with the same name may not exist.

value

```

uniqueNames : TransitionRule-set → Bool
uniqueNames(trs) ≡
  ( ∀ tr1 : TransitionRule • tr1 ∈ trs ⇒
    ~( ∃ tr2 : TransitionRule • tr2 ∈ trs ∧
      tr1 ≠ tr2 ∧ name(tr1) = name(tr2) )
  )

```

3. Value assignments to existing variables Any value assignment in a transition rule, must be to a variable in the state space of the same transition system.

value

```

assignmentToExistingIds : TransitionSystem → Bool
assignmentToExistingIds(ts) ≡
  ( ∀ tr : TransitionRule • tr ∈ transitionRules(ts) ⇒
    ( ∀ assign : Assignment • assign ∈ assignments(tr) ⇒
      ( ∃ var : Var • var ∈ state(ts) ∧
        id(assign) = id(var) )
    )
  )

```

4. Variables in guard must be part of the state space The variables used in each guard, must be part of the state space.

value

```

idsInGuardMustBeInState : Var-set × TransitionRule-set → Bool
idsInGuardMustBeInState(state, trs) ≡
  ( ∀ tr : TransitionRule • tr ∈ trs ⇒
    ( ∀ id : VarId • id ∈ idsInBoolExp(guard(tr)) ⇒
      ( ∃ var : Var • var ∈ state ∧
        id = id(var) )
    )
  )

```

5. At least one assignment in update The empty assignment is not allowed.

value

```

nonemptyAssignment : TransitionRule-set → Bool
nonemptyAssignment(trs) ≡
  ( ∀ tr : TransitionRule • tr ∈ trs ⇒
    assignments(tr) ≠ {}
  )

```

6. Each variable may only be updated once Each variable may only be updated once in any transition rule.

value

```

idUpdOnlyOnce : TransitionRule-set → Bool
idUpdOnlyOnce(trs) ≡
  ( ∀ tr : TransitionRule • tr ∈ trs ⇒
    let ma = assignments(tr)
    in ( ∀ assign : Assignment • assign ∈ ma ⇒
      ~ ( ∃ assign2 : Assignment • assign2 ∈ ma ∧
        assign ≠ assign2 ∧ id(assign) = id(assign2) )
    )
  )
end
)

```

4.4.4 Assertions

An assertion consists of a name and a LTL formula.

type

Assertion = **Text** × LTLformula

where an LTL formula has been defined in the following way.

type

```

LTLformula ==
  b(bExp : BooleanExp) |
  ltrl(l : Text) | /* Literal */
  conj(LTLformula-set) | /* Conjunction */
  disj(LTLformula-set) | /* Disjunction */
  paren(LTLformula) | /* Parenthesis */
  neg(LTLformula) | /* Negation */
  impl(LTLformula, LTLformula) | /* Implication */
  x(LTLformula) | /* Next */
  g(LTLformula) | /* Globally */
  f(LTLformula) | /* Finally */
  u(LTLformula, LTLformula) /* Until */

```

This differs from the syntax used by Kjær and Le Bliguet [2] and thus some work is required here, if the specifications are to be merged. The reason for the change is that they only allow LTL operators at the outer levels. As will be seen in section 10.8, expressions like $\Diamond(A \wedge \bigcirc B)$ will be needed.

The definition of the LTL syntax consists of a subset of the above defined LTLformula of temporal and logical operators. Most of them are derived from the fundamental definitions described by Baier and Katoen [3]. The scope for this project is not to derive the grammar, but use the LTL formulae to generate the expressions. Thus, for convenience all the needed temporal and logical operators used in generating the assertions are defined in the LTLformula type.

4.4.4.1 Well-formedness of Assertions

An assertion is well-formed if its name is unique among the assertions in the transition system and the name is non-empty. Furthermore, any variables in the LTL formula must exist in the state space.

value

```

isWfAssertions : TransitionSystem → Bool
isWfAssertions(ts) ≡
  let state = state(ts),
      assertions = assertions(ts)
  in nonemptyNames(assertions) ∧
      uniqueNames(assertions) ∧
      existingVars(state, assertions)
end

```

1. Valid name Assertion name must be non-empty.

value

```

nonemptyNames : Assertion-set → Bool
nonemptyNames(assertions) ≡
  ( ∀ (name, ltl) : Assertion • (name, ltl) ∈ assertions ⇒
    name ≠ "" )

```

Assertion name must be unique.

value

```

uniqueNames : Assertion-set → Bool
uniqueNames(assertions) ≡
  ( ∀ (name, ltl) : Assertion • (name, ltl) ∈ assertions ⇒
    ~ ( ∃ (name2, ltl2) : Assertion •
      (name2, ltl2) ∈ assertions ∧
      (name, ltl) ≠ (name2, ltl2) ∧
      name = name2 )
  )

```

2. Existing variables The variables used in the LTL expression must be part of the state space of the transition system.

value

```

existingVars : Var-set × Assertion-set → Bool
existingVars(state, assertions) ≡
  ( ∀ (name, ltl) : Assertion • (name, ltl) ∈ assertions ⇒
    ( ∀ id : T.VarId • id ∈ idsInLTL(ltl) ⇒
      isIdInState(id, state) )
  )

```


CHAPTER 5

Behavioural Model of the Internal System

This chapter will describe the behavioural model of the internal part of the interlocking system.

This model was developed by Kjær and Le Bliguet [2]. The model will be described briefly, with an emphasis on the concrete model of Stenstrup that they created using their generator tool. For a deeper understanding of the model and choices made in creating it, we refer to their thesis.

Section 5.1 introduces the state space. Here the relevant variables will be explained and associated with entities from the station documentation.

Section 5.2 will briefly the transition rules of the internal system. These are not as relevant for this work, as the external environment does not interact with the transitions, but rather the variables of the state space.

Lastly, section 5.3 will describe how time is modelled. This section is highly relevant, as it affects every external event.

5.1 State Space

The state of the model of the internal interlocking system contains a boolean variable for each relay and each button. The variable is true when the relay is drawn or button is pushed respectively and false when the relay is dropped or button is released respectively.

The initial value of the variables are the state of the relay/button in the normal state.

5.1.1 State Space of Stenstrup

Table 5.1 shows the relevant subset of the 60 boolean variables created by the generator of the internal system, when Stenstrup station is given as input.

Type	Variable	State
Track	<i>a12, t01, t02, t03, t04, b12</i>	False when the related track section is occupied.
Points	<i>plus01, minus01, plus02, minus02</i>	True when the related points section is locked into the given position.
Signals	<i>aGreen, aRed, bGreen, bRed, eGreen, eRed, fGreen, fRed, gGreen, gRed, hGreen, hRed</i>	True when the given light, of the related signal, is on.
Locking	<i>ia, ib, ua, ub</i>	False when the related train route is locked.
Buttons	<i>b00406, b00606, b03106, b03306</i>	True when the related button on the operator's panel is pressed. Used to lock routes.

Table 5.1: Variables used in the internal model of the relay interlocking system at Stenstrup station.

Table 5.2 shows the relation between each track section on Stenstrup and the generated track relay variables, which models the corresponding track section's track relay.

Each point machine has two point detection relays, one for the plus position and one for the minus position. Table 5.3 shows the relation between the point machines on Stenstrup and the generated point detection relay variables.

Track	Variable
A12	a12
01	t01
02	t02
03	t03
04	t04
B12	b12

Table 5.2: Shows the relation between the track sections and the variables that models their associated track relay.

Point Machine	Plus	Minus
01	plus01	minus01
02	plus02	minus02

Table 5.3: Shows the relation between point machines and the point locking relay variables associated with them.

Each signals is controlled by as many signal lamp relays, as the signal has lamps. In this thesis only the red and green lamp are considered, so each signal needs to be associated with two signal lamp relays. Table 5.4 shows this relation for Stenstrup with the generated variables.

Signal	Green	Red
A	aGreen	aRed
B	bGreen	bRed
E	eGreen	eRed
F	fGreen	fRed
G	gGreen	gRed
H	hGreen	hRed

Table 5.4: Shows the names of the signal lamp relay variables for each signal on Stenstrup.

Table 5.5 shows which train routes the locking relays and buttons relate to. Note that several routes may share a locking relay¹. In those cases, both the state of the train route locking relay and the state of the point detection relays needs to be known in order to identify which of the shared routes are locked.

Consider train routes 2 and 3 on Stenstrup. Table 5.5 shows that they share the

¹This is done to reduce cost.

Train Routes	Locking Relay	Button
2, 3	<i>ia</i>	<i>b00606</i>
5, 6	<i>ib</i>	<i>b03106</i>
7, 8	<i>ua</i>	<i>b00406</i>
9, 10	<i>ub</i>	<i>b03306</i>

Table 5.5: Shows which locking relays and which buttons on the operators panel relate to which train routes.

same train route locking relay. However, train route 2 require the point machines 01 and 02 to be locking in the plus position, while train route 3 require the point machines to be locking in the minus position. Since the points must be locked into position before a train route can be locked, the state of the points determine which route is locked, when pressing button b00606.

5.2 Transition Rules

Two transition rules are generated for each relay. One allows the relay to be drawn, while the other allows the relay to be dropped.

The guards are inferred from the logic of the circuitry and can thus be very long. The update on the other hand is always quite simple. It just switches the state of the relay.

5.3 The Timing Issue

SAL uses LTL as its property specification language, but LTL does not offer an explicit notion of time. Time in LTL is abstracted to a series of discrete events that happens in a linear manner. This means that LTL can express properties about the relative order of events, but not the precise timing. Therefore, it can be expressed that a button is released after it has been pressed, but not that the button is released 1 second after being pressed.

In the electrical relay circuitry the current would (likely) propagate through the closed contact of the pressed button faster than the operator can release the button again. This means that the internal events that the current triggers, should occur prior to the external event that the button is released. This leads

to the assumption that internal events happen so quickly, in relation to external events, that any possible internal event should be allowed to happen before the next external event can occur.

5.3.1 Idle and Busy States

This issue is resolved by introducing the concept of idle and busy states of the system. The system is busy if internal events are possible and idle if not (fig. 5.1).

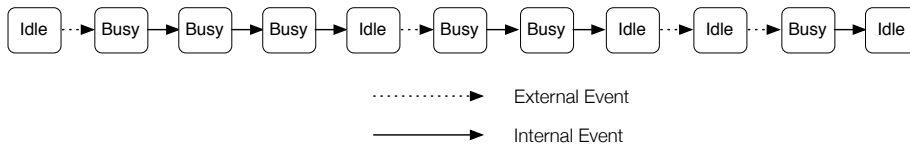


Figure 5.1: The initial state of the internal interlocking system is the idle state. Here the system waits for external events to occur. When an external event occurs, the system switches to the busy state, where only internal events may occur. When no further internal events can occur, the system returns to the idle state. [2]

Therefore, any external event must have the rule that:

- An external event can only occur if the interlocking system is in the idle state.

In order to allow the relays to react to the external event, the system must be set to being busy whenever an external event takes place:

- Any external event results in the system being busy.

The system is then busy as long as it is possible for an internal event to occur. When this is no longer the case, the system becomes idle again:

- The system becomes idle once no internal event can occur.

On a practical level, this leads to the introduction of the artificial boolean variable *idle*, which is true when the system is idle and false when the system is busy, and the transition *setIdle*, which sets *idle* to true if no internal transition is available.

Idle is initialised to true, since no internal event can occur in the normal state.

5.3.2 Returning To the Idle State

The system can return to the idle state, when no further internal events can occur. The generator of the internal system generates a transition rule called *setIdle*, which has the signature:

$$[\text{setIdle}] \sim \text{idle} \wedge \sim \langle \text{allInternalGuards} \rangle \longrightarrow \\ \text{idle}' = \mathbf{true}$$

where $\langle \text{allInternalGuards} \rangle$ is the conjunction of all the guards of the internal transitions.

The interlocking system includes a function that allows it to consider a pressed button as released even though its not. This functionality activates if a button is not released within a reasonable amount of time. This functionality is not modelled in the internal system, but is instead simulated by the following update to the *setIdle* transition, which is not generated by the generator of the internal model. Therefore, it is necessary to manually replace the *setIdle* update with the content shown below:

$$\begin{aligned} \text{idle}' &= \sim(\text{b00406} \vee \text{b00606} \vee \text{b03106} \vee \text{b03306}), \\ \text{b00406}' &= \mathbf{false}, \\ \text{b00606}' &= \mathbf{false}, \\ \text{b03106}' &= \mathbf{false}, \\ \text{b03306}' &= \mathbf{false} \end{aligned}$$

CHAPTER 6

Behavioural Model of Train Movements

The following chapter will briefly discuss approaches to modelling train movements, where section 6.1 will introduce two models and discuss their differences, and why one model is chosen over the other.

The chosen model of train movements consists of two parts. A model representation of *trains* discussed in section 6.2 and a model of the train *movement* rules in section 6.3.

6.1 Approaches to Modelling Train Movement

The idea behind introducing new approaches is to solve the limitation of the model of train movement used by Kjær and Le Bliguet [2].

Let us shortly recap their model. It uses a direction and location of trains by assigning a boolean variable for each track section. For instance, consider Stenstrup, if a train is at section a12 and heading towards Svendborg, the variable a12_fwd would be true. If the train proceeds to 01 and therefore occupies both a12 and 01, a12_t01_fwd would be true. This requires a variable for each permutation

of location, direction and train length. Avoiding a large state space they limited the trains to have a maximum length of two sections.

All in all their model is excellent in its simplicity, but with the major limitation that it is too specific, and therefore require a boolean variable for each possible situation.

During the modelling phase of train movements two approaches were considered. The first model can be found in appendix D. Recapping this approach, it uses a location variable containing all the track sections a train occupies in a set and a direction variable which points to a neighbouring track section the train is heading towards. When the train is moving the location set is updated as well as the direction variable.

Having this approach it eliminates the limitation of train length, and allows one to dictate movement behaviour specific for each train, since trains are explicitly created. Thus, creating several trains requires behaviours for each train. The downside of this approach is, that the model checker is limited to the number of defined trains.

In the early stages of modelling train movement there were conducted several tests to compare the model of fixed train movement¹ with the explicit model and with the implicit model, which will be introduced in the following section. The comparison was in terms of elapsed time and memory consumption when model checking. The test showed, as illustrated on figure 6.1, that the implicit model was slightly slower than the fixed train model, and faster than the two cases compared with the explicit train model.

Model	Condition	Measurements	
		Time	Memory (MB)
Fixed train length	Basic safety	15 sec	76
	TRT safety	2.36 min	121
Implicit train	Basic safety	16 sec	83
	TRT safety	3.00 min	140
Explicit train	Basic safety	21 sec	81
	TRT safety	4.25 min	138

Table 6.1: Model checked train models showing the time spend and memory consumption.

As mentioned earlier, the rest of this chapter will present the implicit model, that

¹The model provided by Kjær and Le Bliguet

eliminates the issues of train length, and introduces no upper limit of trains. This allows the model checker to spawn unlimited trains, that the explicit modelling approach cannot. It can be arguable how many trains are needed to verify the case of Stenstrup station, but having a larger station would require more trains making the following approach more elegant.

6.2 Model of Trains

Recall the definition of trains from section 2.2. A train is rolling stock that performs a train movement, i.e. travels from one station to another.

Some of the following rules may apply for shunting movements as well, while others will not.

The following properties are true about trains:

- Occupies a minimum of one track section, with no upper limit.
- Track sections occupied must form a path, i.e. a sequence of track sections, where each track section is neighbour with the next track section in the sequence.

Instead of the limitations of modelling train movement discussed in section 6.1 we will present a model to eliminate this.

A counter variable will be introduced to indicate how many trains currently occupies a given track section, thus allowing an arbitrary number of trains. A connection variable will be introduced, allowing a train to have a length with no upper bound, and a direction variable to indicate the travelling movement.

Using solely the state of the track circuit relays, one is able to tell on which track sections trains are located². One cannot tell how many trains are on each section, which direction they are heading or if several adjacent track sections are occupied, whether it is by a single long train or several smaller.

The variables rely on the name of the track sections. Considering Stenstrup station on figure 6.1, it can be of reference point when constructing the variable names.

²assuming only trains short the track circuits, which is reasonable when model checking

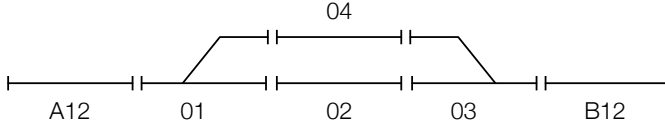


Figure 6.1: Track Layout of Stenstrup station.

6.2.1 Counter

A counter for each track section is introduced, which counts how many trains are located on the associated track section. Shown on figure 6.2 each variable is prefixed with *cnt_* followed by the name of the track section to denote a counter variable.

When a train enters a section, the counter is incremented and when a train leaves the section, the counter is decremented.

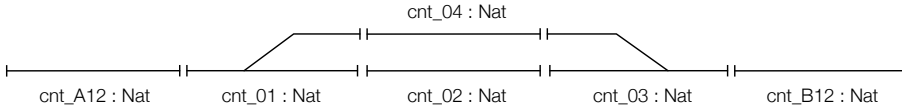


Figure 6.2: Relation between the track sections and the counter variables.

The initial state of Stenstrup has no trains located on the station, hence all counters are assigned with zero.

```

cnt_A12 : Nat := 0,
cnt_01  : Nat := 0,
cnt_02  : Nat := 0,
cnt_04  : Nat := 0,
cnt_03  : Nat := 0,
cnt_B12 : Nat := 0,

```

Example Let us consider a subset of Stenstrup station using track sections A12, 01, 02 and 04. Train A and B each occupy track section A12 and 01, and 01 and 02, respectively. Hence the counters of A12 and 02 are both one. Having both A and B occupy track section 02 the counter is two. Track section 04 is unoccupied leaving the counter to zero.

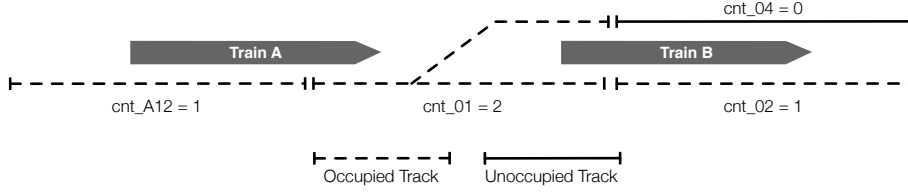


Figure 6.3: An example of trains A and B occupying track sections A12, 01 and 02 showing the counter variables.

6.2.2 Connection

A connection variable is introduced to distinguish different trains. Without a connection variable it would be hard to tell whether two occupied tracks are by two separate trains or a single train, that occupy both track sections.

This time a boolean variable is introduced for each border between track sections shown on figure 6.4. The variables are true when the same train is located on both the adjacent track sections.

Recall the neighbour relation from section 4.2.4. The variables are constructed such that each pair of neighbours, excluding the line, are prefixed with *con_* followed by a track section, followed by an underscore and the neighbouring track section.

```
con_A12_01 : Bool := false,
con_01_02 : Bool := false,
con_01_04 : Bool := false,
con_02_03 : Bool := false,
con_03_04 : Bool := false,
con_03_B12 : Bool := false,
```

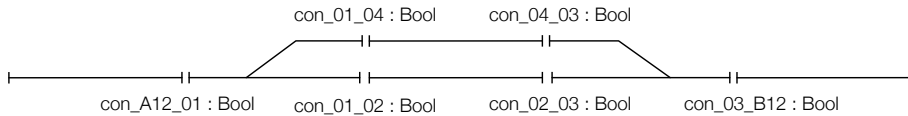


Figure 6.4: The connection variables are true if a single train occupy both neighbouring track sections.

Example Let us consider the same scenario on Stenstrup station in the example in section 6.2.1. Figure 6.5 shows that train A occupies section A12 and 01, hence `con_A12_01` is true. While train B occupies 01 and 02 hence `con_01_02` is true. There is no connection between 01 and 04 leaving `con_01_04` false.

Using the connection variable in conjunction with the counter variable it can be derived that there are two trains, since the counter for track section 01 is two. Thus, one train connected with A12 and 01 and another connected with 01 and 02.

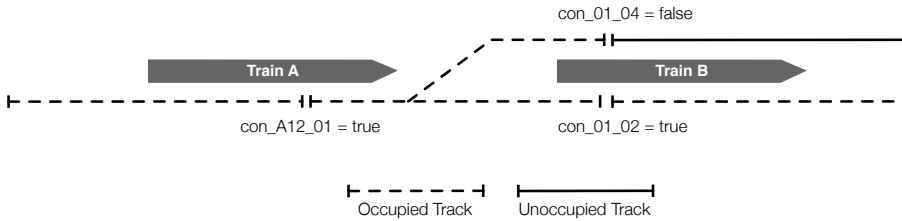


Figure 6.5: An example of train A and B occupying track sections A12, 01 and 02 showing the connection variables.

6.2.3 Direction

The direction variable denotes the travelling movement of trains. Each track section has two direction variables prefixed with *fwd_* and *bwd_* followed by a track name, where fwd is short for forward and bwd is short for backward. Figure 6.6 shows the complete set of direction variables for Stenstrup station.

Whenever one of the variables is true then a train on the associated track section is heading in one direction, while the other is true when a train on the same track section is heading the other way. They are false if there is no train on the section or if no train on the section is heading in the given direction.

Moving from Odense to Svendborg denotes a travelling direction of forward and moving from Svendborg to Odense is backward.

```
fwd_A12 : Bool := false,
bwd_A12 : Bool := false
```

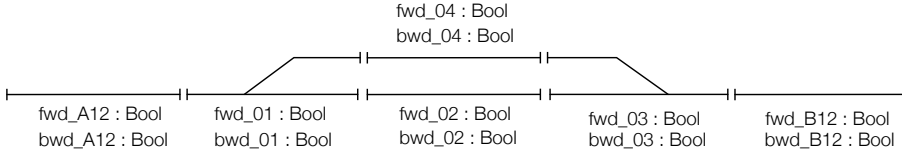


Figure 6.6: Relation between track sections and direction variables.

Example Let us consider a subset of Stenstrup station using track sections A12, 01, 02 and 04 shown on figure 6.7. Train A is moving backward towards Odense, while train B is moving forward towards Svendborg. Nothing is moving on track section 04 leaving both direction variables false.

Notice that train B is moving forward on track section 01 and 02, since it occupies both tracks.

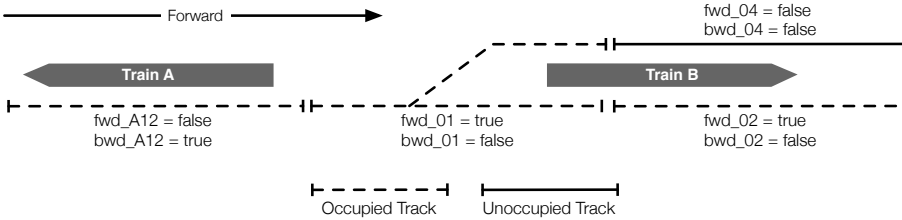


Figure 6.7: An example of train A and B travelling on track sections A12, 01 and 02 showing the direction variables.

6.3 Model of Train Movements

Regardless of how the trains are modelled, they must obey a set of general movement rules, as discussed in the following sections.

The following properties are true about train movements:

- Respects signal aspects, i.e. only passes facing signals showing a proceed aspect.
- Follows the tracks in its initial direction.
- Not allowed to change direction while traversing a train route.

6.3.1 Enter Station

A train is allowed to enter a station at track section ts , if ts is at the station limit and the entrance signal covering ts shows a proceed aspect.

The general rule for entering a station has the following form.

$$\begin{aligned} &[\text{enter_ts_from_line}] \text{ idle} \wedge \text{sGreen} \wedge \sim \text{sRed} \longrightarrow \\ &\text{idle}' = \mathbf{false}, \text{ts}' = \mathbf{false}, \\ &\text{cnt_ts}' = \text{cnt_ts} + 1, \text{dir_ts}' = \mathbf{true} \end{aligned}$$

where \mathbf{ts} is a track sections and \mathbf{dir} is either \mathbf{fwd} or \mathbf{bwd} . \mathbf{sGreen} and \mathbf{sRed} are signal relays for signal \mathbf{s} .

In the example below a rule for entering a station will be constructed and explained step by step showing the requirements before a train is allowed to enter and which parameters that the train affects when doing so.

Example Let us consider the following case on Stenstrup station when entering the station on track section A12 from the line. External events can happen only when the system is idle, which is the first requirement for any external rule. While any action that requires the internal system to respond on external events the system is no longer idle.

Hence, this can be applied as a template for constructing a rule for entering Stenstrup station.

$$\begin{aligned} &[\text{enter_A12_from_line}] \text{ idle} \wedge \dots \longrightarrow \\ &\text{idle}' = \mathbf{false}, \dots \end{aligned}$$

The only requirement for a train to enter a station is that the entry signal shows a proceed aspect. Again, using Stenstrup this requires that the entry signal A must show proceed aspect and not stop aspect.

$$\begin{aligned} &[\text{enter_A12_from_line}] \text{ idle} \wedge \text{aGreen} \wedge \sim \text{aRed} \longrightarrow \\ &\text{idle}' = \mathbf{false}, \dots \end{aligned}$$

When the train enters track section A12 the associated track relay is dropped. Therefore the corresponding relay variable is set to false. The train movement also needs to be updated, thus the counter for A12 is incremented, since there has entered a train on the track section. Finally the train is given a direction denoting the direction on track A12.

$$\begin{aligned} &[\text{enter_A12_from_line}] \text{ idle} \wedge \text{aGreen} \wedge \sim \text{aRed} \longrightarrow \\ &\text{idle}' = \mathbf{false}, \text{a12}' = \mathbf{false}, \\ &\text{cnt_A12}' = \text{cnt_A12} + 1, \text{fwd_A12}' = \mathbf{true} \end{aligned}$$

6.3.2 Enter Track Sections

When the train is within the station limits, it must follow the tracks until it is either out of the station again or sees a stop aspect. It is important that the train is not forced to follow the intended route, but rather follows the physical layout of the tracks.

The general rule for entering track sections has the following form.

$$\begin{aligned} &[\text{enter_ts1_from_ts2}] \text{ idle} \wedge \text{cnt_ts2} = 1 \wedge \text{dir_ts2} \wedge \sim \text{con_ts1_ts2} \wedge \dots \longrightarrow \\ &\text{idle}' = \mathbf{false}, \text{ts1}' = \mathbf{false}, \\ &\text{cnt_ts1}' = \text{cnt_ts1} + 1, \text{dir_ts1}' = \mathbf{true}, \text{con_ts1_ts2}' = \mathbf{true} \end{aligned}$$

where **ts1** and **ts2** are neighbouring track sections and **dir** is the direction needed to go from **ts2** to **ts1**, which is either **fwd** or **bwd**. Finally, $\wedge \dots$ denotes additional requirements to points and signals, which includes point positioning and signal aspects.

The reason why $\text{cnt_ts2} = 1$ is required and not $\text{cnt_ts2} \geq 1$ before entering is that, if there exists more than one train on a track section *ts2*, then the system is already in an unsafe state.

Train behaviour after an unsafe state is non-deterministic, and once the system enters an unsafe state it is not of interest what happens afterwards. If the model checker encounters an unsafe state it concludes that the system is faulty.

Consider figure 6.8 that shows an example of the state space of the model of train movement. The train travels, and remains in a safe state until it reaches an

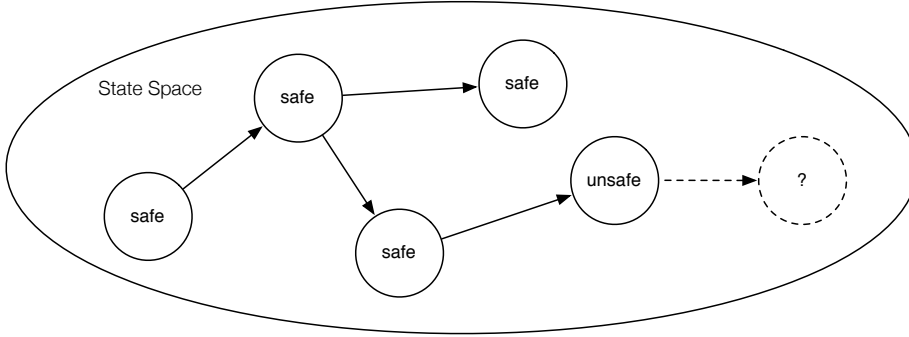


Figure 6.8: State space of trains showing safe and unsafe states.

unsafe state. At this point the model checker evaluates a fault. Hence, modelling behaviour after an unsafe state does not make sense.

In the running example below a rule for entering a track section within a station will be constructed using the general form. It will be explained step by step showing the requirements before a train is allowed to enter and which parameters that the train affects when doing so.

Example Let us consider the following case on Stenstrup station. Like entering a station the idle condition is required when entering track sections and as a result of the transition, the interlocking system is set to busy. The associated track relay is also dropped.

Only the front end of the train is allowed to enter a new track section. Therefore it is checked, as part of the guard, that a train does not already extend onto the next track section. This also ensures that the train cannot go through another train. Consider a situation where train 1 is entirely on section A and train 2 is on section A and B, which are neighbours. Train 1 now attempts to enter section B, but since there is a connection (due to train 2), it is not allowed to enter. If there is no connection then the train may enter, which results in a connection between the track that it is coming from and entering indicating that the train is occupying both tracks. Considering Stenstrup when entering section 02 from 01 we have the following.

```
[enter_02_from_01]
idle ∧ ¬(con_01_02) ∧ ... →
idle' = false, t02' = false,
con_01_02' = true, ...
```


A train may only enter a track section if it is already occupying the neighbouring track it is coming from. Hence, the track counter of the neighbouring track must be equal to one. This condition is to avoid the scenario where a train just suddenly appear out of nowhere. Additionally the counter for the track that is entered to is incremented by one to denote that a train is occupying the track.

```
[enter_02_from_01]
idle ∧ cnt_01 = 1 ∧ ¬(con_01_02) ∧ ...  →
idle' = false, t02' = false,
cnt_02' = cnt_02 + 1, con_01_02' = true, ...
```

When entering track section 02 from 01 the direction on track section 01 must be forward. Resulting to the direction of 02 being forward as well.

```
[enter_02_from_01]
idle ∧ cnt_01 = 1 ∧ fwd_01 ∧ ¬(con_01_02) ∧ ...  →
idle' = false, t02' = false,
cnt_02' = cnt_02 + 1, fwd_02' = true, con_01_02' = true
```

It is only physical possible for a train to enter track section 02 if the positioning of point 01 is in plus. Thus, the last requirement for entering track section 02 is when the point positioning is set to plus.

```
[enter_02_from_01]
idle ∧ cnt_01 = 1 ∧ fwd_01 ∧ ¬(con_01_02) ∧ plus01  →
idle' = false, t02' = false,
cnt_02' = cnt_02 + 1, fwd_02' = true, con_01_02' = true
```

Summarizing the above. A train may enter a track section ts if the front of the train is at a neighbouring track section ts' and the train is heading towards ts . Additionally, if a signal s is located on ts' and covers ts , the train may enter ts only if the signal shows a proceed aspect.

$s\text{Green} \wedge \neg s\text{Red}$

This requirement is added as a guard if there is a signal s covering a track section. Additionally, rules for entering points sections depends on the trains entry point. All these rules are explained in the following sections.

6.3.2.1 Travelling on Points Sections

The entry point on a points section matters greatly, both in relation to where the train ends up and in relation to safety. When entering from the stem side, the train will travel by the plus branch if the points section is locked in the plus position and likewise with the minus position. However, if the points section is not locked into position, i.e. is in the intermediate position, then the train runs the risk of derailling.

When entering from the branch side, the train risk derailling not only when the points section is not locked, but also when it is locked in the other branch position.

Recalling from figure 2.4 that showed the notion of a facing and a trailing move. A facing move is when a train enters the stem side or leaves the branch, while a trailing move is when a train enters the branch side or leaves the stem.

Facing Move - Stem Side A facing move onto a points section or a facing move to the neighbour of a points section is treated the same way as moving on linear sections.

Consider the following cases on Stenstrup. The following transition rule describes the situation, where a train is entering the points section 01 from the linear section A12.

$$\begin{aligned}
 &[\text{enter_01_from_A12}] \text{ idle} \wedge \text{cnt_A12} = 1 \wedge \text{fwd_A12} \wedge \sim \text{con_A12_01} \longrightarrow \\
 &\text{idle}' = \mathbf{false}, \text{t01}' = \mathbf{false}, \\
 &\text{cnt_01}' = \text{cnt_01} + 1, \text{con_A12_01}' = \mathbf{true}, \text{fwd_01}' = \mathbf{true}
 \end{aligned}$$

Notice that it follows the same pattern as movement on linear track sections. However, the same cannot be said about movements that involve the branch side of a points section.

Facing Move - Branch Side When entering the neighbouring track section from one of the branches of a points section, the position of the points has to be considered. It is a regular enter rule except that it requires the points section to be locked into position. Travelling from 01 to 02 requires the point to be locked in the plus position, which directs the train towards track section 02.

$$\begin{aligned}
& [\text{enter_02_from_01}] \text{ idle} \wedge \text{cnt_01} = 1 \wedge \text{fwd_01} \wedge \sim \text{con_01_02} \wedge \text{plus01} \longrightarrow \\
& \text{idle}' = \mathbf{false}, \text{t02}' = \mathbf{false}, \\
& \text{cnt_02}' = \text{cnt_02} + 1, \text{con_01_02}' = \mathbf{true}, \text{fwd_02}' = \mathbf{true}
\end{aligned}$$

When a points section is in an intermediate state, while a train is in the act of a facing move at the branch side the outcome is non-deterministic. The train may risk a derailment or it may continue an unknown path, since the points are just about to lock into a position, for instance. Referring back to figure 6.8 it is not of interest what happens after an unsafe state, hence the case of an intermediate points position will be left out.

Trailing Move - Stem Side A facing move at the stem side is similar to a trailing move at the stem side. The following transition rule describes the situation, where a train is entering the linear section A12 from the points section 01.

$$\begin{aligned}
& [\text{enter_A12_from_01}] \text{ idle} \wedge \text{cnt_01} = 1 \wedge \text{bwd_01} \wedge \sim \text{con_A12_01} \longrightarrow \\
& \text{idle}' = \mathbf{false}, \text{a12}' = \mathbf{false}, \\
& \text{cnt_A12}' = \text{cnt_A12} + 1, \text{con_A12_01}' = \mathbf{true}, \text{bwd_A12}' = \mathbf{true}
\end{aligned}$$

Notice once again, that it follows the same pattern as movement on linear track sections.

Trailing Move - Branch Side Entering a points section on the branch side requires two transition rules to cover the cases. If the correct branch is locked, then the train will not derail and proceed as normal. On the other hand, if the points are in the intermediate position or locked to the wrong branch, then the train will run the risk of derailling.

Consider the following rules for Stenstrup. A train enters to points section 01 from section 02. The rules follow the same pattern as before, but with an added requirement about the position of the points. This ensures that the case where the train travels without derailling is applied. Since there is located a signal covering the points section, the proceed aspect requirement is added as well.

$$\begin{aligned}
& [\text{enter_01_from_02_no_derail}] \text{ idle} \wedge \text{cnt_02} = 1 \wedge \text{bwd_02} \wedge \\
& \sim \text{con_01_02} \wedge \text{eGreen} \wedge \sim \text{eRed} \wedge \text{plus01} \longrightarrow \\
& \text{idle}' = \mathbf{false}, \text{t01}' = \mathbf{false},
\end{aligned}$$

$\text{cnt_01}' = \text{cnt_01} + 1, \text{con_01_02}' = \mathbf{true},$
 $\text{bwd_01}' = \mathbf{true},$

The second case is when a train enters the same points section, but the points positioning is not in plus i.e. either intermediate or minus.

$[\text{enter_01_from_02_derail}] \text{ idle} \wedge \text{cnt_02} = 1 \wedge \text{bwd_02} \wedge$
 $\sim \text{con_01_02} \wedge \text{eGreen} \wedge \sim \text{eRed} \wedge \sim \text{plus01} \longrightarrow$
 $\text{idle}' = \mathbf{false}, \text{t01}' = \mathbf{false},$
 $\text{cnt_01}' = \text{cnt_01} + 1, \text{con_01_02}' = \mathbf{true}$

Notice that the train is not assigned a direction when it derails³.

6.3.3 Leave Station

A train is allowed to leave a station if the back end of the train occupies a track section ts , which is located at the station limit, and it is heading towards the line.

The general rule for leaving a station has the following form.

$[\text{leave_ts_to_line}] \text{ idle} \wedge \text{cnt_ts} = 1 \wedge \text{dir_ts} \wedge \sim \text{con_tsNeighbour_ts} \longrightarrow$
 $\text{idle}' = \mathbf{false}, \text{ts}' = \mathbf{true},$
 $\text{cnt_ts}' = 0, \text{dir_ts}' = \mathbf{false}$

where ts is a track section, $tsNeighbour$ is the neighbouring track section to ts .

In the following example, a transition rule for leaving Stenstrup will be constructed. Again, explaining step by step the requirements before a train is allowed to leave a station and which parameters that the train affects when doing so.

Example Let us consider a train leaving Stenstrup where the rear end of the train is located at track section B12. First we shall begin by using idle which is the first requirement before leaving.

³Derailed trains will stop.

$$[\text{leave_B12_to_line}] \text{ idle} \wedge \dots \longrightarrow \\ \text{idle}' = \mathbf{false}, \dots$$

A train can leave section B12 towards the line only if there is a train at B12 and it is heading towards the line. Furthermore it must be the back end of the train.

$$[\text{leave_B12_to_line}] \text{ idle} \wedge \text{cnt_B12} = 1 \wedge \text{fwd_B12} \wedge \sim \text{con_03_B12} \longrightarrow \\ \text{idle}' = \mathbf{false}, \dots$$

Since the model only allows movements in a safe state, only one train can occupy each section when the transition is taken. Therefore the track relay variable associated with the track that the train is leaving, can be set to true, i.e. the track is vacant. Likewise the train movement variables (the counter and the direction) can be set to 0 and false respectively, as the track is now empty and nothing is no longer moving on track section B12.

$$[\text{leave_B12_to_line}] \text{ idle} \wedge \text{cnt_B12} = 1 \wedge \text{fwd_B12} \wedge \sim \text{con_03_B12} \longrightarrow \\ \text{idle}' = \mathbf{false}, \text{b12}' = \mathbf{true}, \\ \text{cnt_B12}' = 0, \text{fwd_B12}' = \mathbf{false}$$

6.3.4 Leave Track Section

A train can leave a track section ts , if the back end of the train occupies ts and the neighbouring track section ts' is occupied by the train as well. Furthermore the back end of the train must be heading towards ts' .

The ends of a train is located by looking at the connection variables. If a train is located at the linear section $ts1$, which is neighbour with $ts0$ in the backwards direction and $ts2$ in the forward direction, then the train has an end at $ts1$ if the connection variable between $ts0$ and $ts1$ is false. The back end of a train is the end which is opposite of the movement direction.

The general rule for leaving track sections has the following form.

$$[\text{leave_ts1_to_ts2}] \text{ idle} \wedge \text{cnt_ts1} = 1 \wedge \text{dir_ts1} \wedge \\ \text{con_ts1_ts2} \wedge \sim \text{con_ts0_ts1} \wedge \dots \longrightarrow \\ \text{idle}' = \mathbf{false}, \text{ts1}' = \mathbf{true}, \\ \text{cnt_ts1}' = 0, \text{dir_ts1}' = \mathbf{false}, \text{con_ts1_ts2}' = \mathbf{false}$$

where **ts0** and **ts2** are neighbouring track sections to track section **ts1** and **dir** is the direction needed to go from **ts1** to **ts2**, which is either **fwd** or **bwd**. Finally, $\wedge \dots$ denotes additional requirements to points positioning or train connections.

Note that $\sim \text{con_ts0_ts1}$ checks that it is in fact the end of the moving train. If the neighbour *ts0* does not exist this requirement is excluded.

Again we will introduce an example below showing how a leave rule is constructed using the general form.

Example Let us consider a case on Stenstrup station where the rear end of a train is leaving track section 02 moving away from track section 01 i.e. the train is travelling forward. The usual idle requirement is used to ensure the internal system is not busy, and the track relay is updated to denote a vacant track.

$$[\text{leave_02_to_03}] \text{ idle} \wedge \sim(\text{con_01_02}) \wedge \text{fwd_02} \wedge \dots \longrightarrow \\ \text{idle}' = \mathbf{false}, \text{t02}' = \mathbf{true}, \dots$$

The end of the train must occupy track section 02 and be connected to track section 03 to allow to leave. When leaving the counter is set to zero indicating there no longer is a train on track section 02. The direction is set to false and the connection between track section 02 and 03 is removed since the train has left 02.

$$[\text{leave_02_to_03}] \text{ con_02_03} \wedge \text{cnt_02} = 1 \wedge \text{idle} \wedge \sim(\text{con_01_02}) \wedge \text{fwd_02} \longrightarrow \\ \text{idle}' = \mathbf{false}, \text{t02}' = \mathbf{true}, \\ \text{cnt_02}' = 0, \text{fwd_02}' = \mathbf{false}, \text{con_02_03}' = \mathbf{false}$$

6.3.4.1 Travelling On Points Sections

While travelling on points sections there are certain cases where the general form needs modification. In the following sections it will be described when this is needed.

Moving Back End To a Points Section When moving the back end of a train onto a points section, the normal rules, as described above, apply.

Given here are two examples for Stenstrup. The first shows the rule concerning moving the back end of the train to the branch side of a points section.

$$\begin{aligned} &[\text{leave_02_to_01}] \text{ idle} \wedge \text{cnt_02} = 1 \wedge \text{bwd_02} \wedge \text{con_01_02} \wedge \sim \text{con_02_03} \longrightarrow \\ &\text{idle}' = \mathbf{false}, \text{t02}' = \mathbf{true}, \\ &\text{cnt_02}' = 0, \text{bwd_02}' = \mathbf{false}, \text{con_01_02}' = \mathbf{false} \end{aligned}$$

The second example shows the rule for moving the back end of a train to the stem side of a points section.

$$\begin{aligned} &[\text{leave_A12_to_01}] \text{ idle} \wedge \text{cnt_A12} = 1 \wedge \text{fwd_A12} \wedge \text{con_A12_01} \longrightarrow \\ &\text{idle}' = \mathbf{false}, \text{a12}' = \mathbf{true}, \\ &\text{cnt_A12}' = 0, \text{fwd_A12}' = \mathbf{false}, \text{con_A12_01}' = \mathbf{false}, \end{aligned}$$

Notice that this rules does not check whether the back end of the train is on A12 or not. This is because A12 neighbours the line on the other end and no line-to-section connection variable exist.

Leaving A Points Section At The Stem End When moving the back end of a train away from a points section at the stem end, a special consideration has to be made. Ensuring that the back end of the train is on the section is normally done by ensuring that the connection variable on the other end is false, i.e. the train does not extend further backwards.

In this situation, however, there are two possible paths the train could possibly extend in. One solution could be to create a rule for each position of the points, such that if the points are locked in plus, then we ensure that the connection variable to the neighbour connected on the plus branch is false. However, this does not accurately evaluate where the back end of the train really is. Imagine that the points are switched while the mid section of the train is on it. If the points then locks to the other branch, then the model would think that the back end of the train indeed is on the points section, whereas the back end actually is further down another path.

Instead we suggest a solution where only a single rule is needed. Regardless of the position of the points, it is ensured that the train does not extend to any of the neighbours connected to either branch.

$$[\text{leave_01_to_A12}] \text{ idle} \wedge \text{cnt_01} = 1 \wedge \text{bwd_01} \wedge$$

$$\begin{aligned} & \text{con_A12_01} \wedge \sim \text{con_01_02} \wedge \sim \text{con_01_04} \longrightarrow \\ & \text{idle}' = \mathbf{false}, \text{t01}' = \mathbf{true}, \\ & \text{cnt_01}' = 0, \text{bwd_01}' = \mathbf{false}, \text{con_A12_01}' = \mathbf{false} \end{aligned}$$

Leaving a Points Section At The Branch End This situation is similar to the entering points section at branch end case. The entering scenario required two rules to handle the situation properly, but in this case only one is needed.

If the branch is locked in the proper position in relation to the desired movement, then anything is fine and the standard transition rule is used, with the added requirement that the points is locked properly.

$$\begin{aligned} & [\text{leave_01_to_02}] \text{idle} \wedge \text{cnt_01} = 1 \wedge \text{fwd_01} \wedge \\ & \text{con_01_02} \wedge \sim \text{con_A12_01} \wedge \text{plus01} \longrightarrow \\ & \text{idle}' = \mathbf{false}, \text{t01}' = \mathbf{true}, \\ & \text{cnt_01}' = 0, \text{fwd_01}' = \mathbf{false}, \text{con_01_02}' = \mathbf{false} \end{aligned}$$

However, if the points is not locked in the proper position, a rule to handle this is not needed. To explain why let us consider a case where the points are in the intermediate position while leaving, then the back end should derail. This means that it is not able to move forward. The absence of a transition rule that covers this case, ensures that the back end is not moved. If the points are locked in the other position then the connection variable, between the points section and the neighbouring section which is connected to the branch will be false. It can be true only if another train is heading into the section from that branch, but in that case the guard would disallow the movement as well, since it requires that only a single train occupy the points section.

Therefore no additional rules are needed, since the back end of the train does not move. In the case of entrance, the additional rule was needed to move the train into the section, since it first derails at the switching area.

6.3.5 Change Direction

A train movement has a direction and the train is not allowed to change direction during that movement. However two consecutive train movements can be in opposite directions. Therefore, if a train is located between two train route start

points, the train would be able to travel in either direction depending on which route is locked.

This leads to the introduction of a rule that allows trains to “change direction” in that the train can begin a train movement in the opposite direction of what it is currently travelling (recall that the direction of a train in the model is only removed if the train has derailed).

This does present an issue. We assign a direction to each section occupied by a train. If the train is longer than a single section, then multiple direction variables would have to be updated. Since the transition rules cannot dynamically determine the length of the train and update each of the direction variables, one would have to create a rule for each possibility. Instead we decide to limit the rules to only allowing trains to change direction if the train occupies only a single track section.

Such a rule shall now be constructed, using Stenstrup as case. A train is allowed to change direction if it occupied just at single track section.

```
[change_direction_at_02_towards_03]
idle ∧ cnt_02 = 1 ∧ ... ∧
∼(con_01_02) ∧ ∼(con_02_03) →
...
```

Notice that since this transition does not affect the internal interlocking system, the idle variable is *not* set to false.

Furthermore, the signal located opposite of the trains direction must show proceed.

```
[change_direction_at_02_towards_03]
idle ∧ cnt_02 = 1 ∧ bwd_02
gGreen ∧ ∼(gRed) ∧
∼(con_01_02) ∧ ∼(con_02_03) →
bwd_02' = false, fwd_02' = true
```

The effect is that the direction of the train is changed to the opposite.

The train can then travel past the signal, if the enter 03 from 02 transition is taken.

6.4 Rubber Band Trains

Since the lengths of track sections and trains are not considered in this project, trains gain a peculiar property. They act like rubber band. The model of train movement allows an arbitrary length of trains, resulting in this rubber band effect to take place when trains move. Depending on how the model checker is evaluating the movement of a train, it may decide to move the front of the train twice and then the end of the train once, for instance.

Let us consider an example on figure 6.9 showing seven states of a moving train on four linear track sections. The train is initially located at section A, where the model checker changes its state to move the front to section B, then C and finally D at the fourth state.

The end of the train has remained at track section A making the train grow from being one track section long to four track sections long. In the fifth state the end of the train is removed from A, then B, then C and finally the train is once again one track section long at the seventh state. Note that the model checker does not move trains on four linear track sections in this strictly fashion, but this scenario will most likely occur.

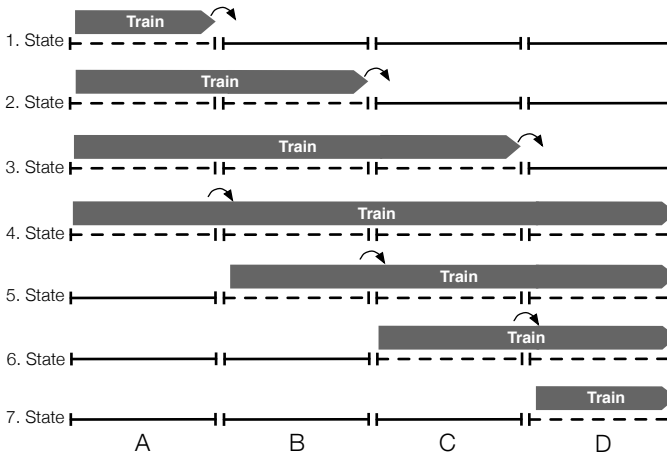


Figure 6.9: An example of a train moving on linear track sections.

A train with this rubber band property retains a flexibility that a real life train movement behaviour cannot feature. The train movement model uses these discrete track section units to avoid the notion of length. Hence, using this model of movement it captures a general behaviour of the property of a real life train having a fixed length, and as consequence, making the real life behaviour a

subset of the train movement model.

CHAPTER 7

Behavioural Model of Buttons and Points

Without any kind of simulated external events, the interlocking system will remain in its normal state. Therefore external events are introduced. External events are buttons being pressed and released, points being operated and tracks being occupied and released due to train movements¹.

Section 7.1 will describe the rules for the behaviour of buttons.

Section 7.2 will describe the rules for the behaviour of operating points.

7.1 Button Behaviour

The buttons are part of the circuitry and rules for their behaviour should therefore be generated from the circuit diagrams. We intend to generate the button behaviour, since they are not generated by Kjær and Le Bliguets [2] tool.

¹Assuming that no other external event can cause the relay to drop. Objects that fall on tracks may cause the track relays to drop, for instance. All objects making track relays drop, except trains, are disregarded.

The rules for pressing buttons at Stenstrup station, as suggested by Kjær and Le Bliguet [2, p. 120–124] have the following form.

```
[pushButton_id]
idle →
idle' = false, id' = true
```

where `id` is the button identifier.

Note that the route locking buttons can be pressed at any time when the system is idle. Once the internal system returns to the idle state the button is released.

The following rules are the behaviour for pressing buttons located on the operators panel for Stenstrup station.

```
[pushButton_b00406]
idle → idle' = false, b00406' = true
[]
[pushButton_b03106]
idle → idle' = false, b03106' = true
[]
[pushButton_b00606]
idle → idle' = false, b00606' = true
[]
[pushButton_b03306]
idle → idle' = false, b03306' = true
```

As stated earlier we intend to generate the rules for button behaviour. Since the buttons are part of the internal system the button identifiers cannot be extracted from the interlocking plan nor can they be derived. Thus, generating the above rules requires the button identifiers as input.

Releasing buttons are handled by the internal system, which was shown in section 5.3.2.

7.2 Points Behaviour

Points are operated by pressing the + or – button next to the points section of the operators panel, as described in section 2.5.1.3. The interlocking system will

register the button press and switch the point if it is safe to do so. This logic is implemented in the circuitry of the interlocking system and is documented by circuit diagrams.

This means that the correct way to implement the behaviour is to always allow the operator to press the buttons and then let the interlocking system sort out whether to move the points or not. Kjær and Le Bliguet [2, p. 119] chose to introduce a model abstraction from the real point control, since the hardware implementation of the point control was too complex to implement in the internal system. Instead they supplied some hand made rules to simulate the logic for Stenstrup station. The provided rules for points behaviour allows the model checker to switch the points without simulating button presses on the operators panel.

Recall the point states described in section 2.1.1.1, points can be in a minus position, a plus position or an intermediate position. The intermediate position is a state when the points are in the act of changing from plus to minus or vice versa.

Thus there are four rules for each points section. Consider the point machine P the general form is the following.

```
[plusToIntermediateP]
idle ∧ plusP ∧ Q →
idle' = false, plusP' = false
[]
[minusToIntermediateP]
idle ∧ minusP ∧ Q →
idle' = false, minusP' = false
[]
[intermediateToPlusP]
idle ∧ ∼ plusP ∧ ∼ minusP ∧ Q →
idle' = false, plusP' = true
[]
[intermediateToMinusP]
idle ∧ ∼ plusP ∧ ∼ minusP ∧ Q →
idle' = false, minusP' = true
```

where Q is a boolean expression that requires no train routes that involves P are locked and that the points section associated with P is free. The internal system ensures that if Q is false the point cannot change i.e. if any of the locking relays are dropped which involves P or the associated point section is occupied, then

the point is not allowed to change position. Figure 7.1 shows the transition rules for the point machine behaviour of P.

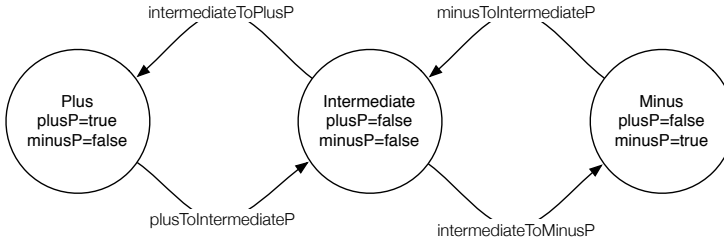


Figure 7.1: Transition system of point P.

Example Let us consider the point machine 01 on Stenstrup station. Using the pattern for rules of point behaviour shown above we will start by requiring that the system needs to be idle, since this is an external event. Recall that Q requires that the points section associated with P must be free, which is part of the no derailling safety rule, dictating that the points must not be operated while occupied. Therefore it is required that the section is vacant, i.e. the track relay variable $t01$ is true. Notice that the name is *operatePoint01*, since the rule by now is general enough to be any of the four rules.

```
[operatePoint01]
idle ∧ t01 ∧ ... →
idle' = false, ...
```

Additionally Q requires that no route involving P is locked, which means the switch must not be operated while a route that covers it is locked, i.e. route locking relays ia , ib and ua must not be locked. If a route is locked, then the corresponding route locking relay will be dropped.

```
[operatePoint01]
idle ∧ t01 ∧ ia ∧ ib ∧ ua ∧ ... →
idle' = false, ...
```

By now the rule is generic for any of the four cases of the point behaviour, hence the name is still *operatePoint01*. The key difference between these rules are which position the point currently are in and which position to switch to. Let us consider point 01 locked in plus position changing to intermediate. Switching from a locked position, the points relay must drop, hence $plus01$ is set to false.

[plusToIntermediateP]
 $\text{idle} \wedge \text{t01} \wedge \text{ia} \wedge \text{ib} \wedge \text{ua} \wedge \text{plus01} \longrightarrow$
 $\text{idle}' = \mathbf{false}, \text{plus01}' = \mathbf{false}$

The other case is when switching from the intermediate position to a locked position, both points relays must first be dropped before one of them are drawn, i.e. locked into a position. This case the point position is intermediate locking into plus.

[intermediateToPlusP]
 $\text{idle} \wedge \text{t01} \wedge \text{ia} \wedge \text{ib} \wedge \text{ua} \wedge \sim \text{plus01} \wedge \sim \text{minus01} \longrightarrow$
 $\text{idle}' = \mathbf{false}, \text{plus01}' = \mathbf{true}$

To account for the behaviour of point P, rules must exist for the minus position as well. Below are all the rules that constitutes for point 01 for Stenstrup station. The rules for point machine 02 will not be shown, since they are made in a similar fashion.

[plusToIntermediate01]
 $\text{idle} \wedge \text{plus01} \wedge \text{ia} \wedge \text{ib} \wedge \text{ua} \wedge \text{t01} \longrightarrow$
 $\text{idle}' = \mathbf{false}, \text{plus01}' = \mathbf{false}$
 \square
 [minusToIntermediate01]
 $\text{idle} \wedge \text{minus01} \wedge \text{ia} \wedge \text{ib} \wedge \text{ua} \wedge \text{t01} \longrightarrow$
 $\text{idle}' = \mathbf{false}, \text{minus01}' = \mathbf{false}$
 \square
 [intermediateToPlus01]
 $\text{idle} \wedge \sim \text{plus01} \wedge \sim \text{minus01} \wedge \text{ia} \wedge \text{ib} \wedge \text{ua} \wedge \text{t01} \longrightarrow$
 $\text{idle}' = \mathbf{false}, \text{plus01}' = \mathbf{true}$
 \square
 [intermediateToMinus01]
 $\text{idle} \wedge \sim \text{plus01} \wedge \sim \text{minus01} \wedge \text{ia} \wedge \text{ib} \wedge \text{ua} \wedge \text{t01} \longrightarrow$
 $\text{idle}' = \mathbf{false}, \text{minus01}' = \mathbf{true}$

Associations

This chapter will introduce two structures, both of which relates entities in the station documentation to variables in the model. A human may guess that the boolean variable *plus01* is the point detection relay for the plus position of a points section, but it is a difficult task for a computer program.

It would be desirable to have a direct mapping between the physical elements of the station and the relays that observe them in terms of naming conventions. Unfortunately, this is not the case. In many cases one could derive the name of the relays from the name of the element, but this is not generally true. For example, the point machine *02* is not located on track section *02* (happens to a linear), but is located on points section *03*.

Section [8.1](#) introduces the object relay association, which is a collection of associations between physical objects (and train routes) and the variables that models relays in the internal system.

Section [8.2](#) introduces train movement association, which is a collection of associations between the track sections and the variables introduced by the model of train movements.

8.1 Object Relay Associations

Object relation associations are associations between physical objects (and train routes) and relay variable that models them, e.g. for Stenstrup, the track relay of track section 01 is modelled by the boolean variable t01.

type

```
ObjectRelayAssociations ::
  routeRelayAssoc : TrainRouteId  $\overrightarrow{m}$  VarId
  trackRelayAssoc : TrackId  $\overrightarrow{m}$  VarId
  pointRelayAssoc : PointMachineId  $\times$  Branch  $\overrightarrow{m}$  VarId
  signalRelayAssoc : SignalId  $\times$  Lamp  $\overrightarrow{m}$  VarId
```

where we consider only the green and red lamp.

type

```
Lamp == gr | re
```

8.1.1 Well-formedness of Object Relay Associations

The object relay association is well-formed if each of its components are well-formed and no relay variable is shared between the associations, e.g. the same relay cannot be used both as a train route locking relay and as a point detection relay.

value

```
isWfObjectRelayAssociations : Diagram  $\times$  TrainRouteTable  $\times$ 
  ObjectRelayAssociations  $\rightarrow$  Bool
isWfObjectRelayAssociations(d, trt, ora)  $\equiv$ 
  noRelayVariableOverlap(ora)  $\wedge$ 
  isWfRouteAssoc(trt, ora)  $\wedge$ 
  isWfTrackAssoc(d, ora)  $\wedge$ 
  isWfPointAssoc(d, ora)  $\wedge$ 
  isWfSignalAssoc(d, ora)
```

No relay variable is pointed to in more than one of the association maps.

```

value
  noRelayVariableOverlap : ObjectRelayAssociations → Bool
  noRelayVariableOverlap(ora) ≡
    let routeVars = rng routeRelayAssoc(ora),
        trackVars = rng trackRelayAssoc(ora),
        pointVars = rng pointRelayAssoc(ora),
        signalVars = rng signalRelayAssoc(ora)
    in routeVars ∩ trackVars = {} ∧
        routeVars ∩ pointVars = {} ∧
        routeVars ∩ signalVars = {} ∧
        trackVars ∩ pointVars = {} ∧
        trackVars ∩ signalVars = {} ∧
        pointVars ∩ signalVars = {}
  end

```

8.1.1.1 Well-formedness of Route Associations

The train route association map is well-formed if the map has the same domain has the train route table, i.e. all the train routes.

```

value
  isWfRouteAssoc : TrainRouteTable ×
    ObjectRelayAssociations → Bool
  isWfRouteAssoc(trt, ora) ≡
    dom routeRelayAssoc(ora) = dom trt

```

8.1.1.2 Well-formedness of Track Associations

The track section association map is well-formed if the map has exactly all of the track sections of the station in its domain.

```

value
  isWfTrackAssoc : Diagram × ObjectRelayAssociations → Bool
  isWfTrackAssoc(d, ora) ≡
    let allTrackSections = allLinears(d) ∪ allPoints(d)
    in dom trackRelayAssoc(ora) = allTrackSections
  end

```

8.1.1.3 Well-formedness of Point Machine Associations

The point machine association map is well-formed if:

1. Every point machine on the station is part of the association and every point machine in the association exists on the station.
2. The same relay may not appear in the association twice.

value

```
isWfPointAssoc : Diagram × ObjectRelayAssociations → Bool
isWfPointAssoc(d, ora) ≡
  existingPointMachines(d, ora) ∧
  differentPointRelays(ora)
```

1. Existing Point Machines The point machines in the association must exist on the station and vice versa.

value

```
existingPointMachines : Diagram ×
  ObjectRelayAssociations → Bool
existingPointMachines(d, ora) ≡
  ( ∀ (pmId, b) : PointMachineId × Branch •
    (pmId, b) ∈ dom pointRelayAssoc(ora) ⇒
      pmId ∈ dom pointMachineTrack(d)
  ) ∧
  ( ∀ pmId : PointMachineId •
    pmId ∈ dom pointMachineTrack(d) ⇒
      ( ∃ (pmId', b) : PointMachineId × Branch •
        (pmId', b) ∈ dom pointRelayAssoc(ora) ∧
        pmId = pmId' )
  )
```

2. Each relay in the association, must only appear once The same relay may not be used to lock different points or the same point in different positions.

value

```

differentPointRelays : ObjectRelayAssociations → Bool
differentPointRelays(ora) ≡
  ( ∀ (pmId, b) : PointMachineId × Branch •
    (pmId, b) ∈ dom pointRelayAssoc(ora) ⇒
      ~ ( ∃ (pmId', b') : PointMachineId × Branch •
        (pmId', b') ∈ dom pointRelayAssoc(ora) ∧
        (pmId, b) ≠ (pmId', b') ∧
        pointRelayAssoc(ora)(pmId, b) =
        pointRelayAssoc(ora)(pmId', b')
      )
    )
  )

```

8.1.1.4 Well-formedness of Signal Associations

The signal association map is well-formed if:

1. Every signal on the station is part of the association and every signal in the association exists on the station.
2. The same relay may not appear in the association twice.

value

```

isWfSignalAssoc : Diagram × ObjectRelayAssociations → Bool
isWfSignalAssoc(d, ora) ≡
  existingSignals(d, ora) ∧
  differentLampRelays(ora)

```

1. Existing signals The signals in the association must exist on the station and vice versa.

value

```

existingSignals : Diagram ×
  ObjectRelayAssociations → Bool
existingSignals(d, ora) ≡
  ( ∀ (sId, l) : SignalId × Lamp •
    (sId, l) ∈ dom signalRelayAssoc(ora) ⇒

```

```

      sId ∈ allSignals(d)
    ) ∧
    ( ∀ sId : SignalId •
      sId ∈ allSignals(d) ⇒
        ( ∃ (sId', l) : SignalId × Lamp •
          (sId', l) ∈ dom signalRelayAssoc(ora) ∧
            sId = sId' )
        )
  )

```

2. Each relay in the association, must only appear once The same relay may not be used to control different lamps.

value

```

differentLampRelays : ObjectRelayAssociations → Bool
differentLampRelays(ora) ≡
  ( ∀ (sId, l) : SignalId × Lamp •
    (sId, l) ∈ dom signalRelayAssoc(ora) ⇒
      ~ ( ∃ (sId', l') : SignalId × Lamp •
        (sId', l') ∈ dom signalRelayAssoc(ora) ∧
          (sId, l) ≠ (sId', l') ∧
            signalRelayAssoc(ora)(sId, l) =
              signalRelayAssoc(ora)(sId', l')
        )
      )
  )

```

8.2 Train Movement Associations

Train movement associations is a collection of maps that associates track sections to the variables used in the model of train movements.

type

```

TrainMovementAssociations ::
  connectionVars : (TrackId × TrackId)  $\xrightarrow{m}$  VarId
  counterVars : TrackId  $\xrightarrow{m}$  VarId
  directionFwdVars : TrackId  $\xrightarrow{m}$  VarId
  directionBwdVars : TrackId  $\xrightarrow{m}$  VarId
  directions : (TrackId × TrackId)  $\xrightarrow{m}$  Direction

```


8.2.1 Well-formedness of Train Movement Associations

The train movement association is well-formed if each of its components are well-formed and no relay variable is shared between the associations, e.g. the same relay cannot be used both as a forward direction variable and as a backward direction variable.

value

```

isWfTrainMovementAssociations : Diagram ×
  TrainMovementAssociations → Bool
isWfTrainMovementAssociations(d, tma) ≡
  noRelayVariableOverlap(tma) ∧
  isWfConnectionVars(d, tma) ∧
  isWfCounterVars(d, tma) ∧
  isWfDirectionFwdVars(d, tma) ∧
  isWfDirectionBwdVars(d, tma) ∧
  isWfDirections(d, tma)

```

No relay variable is pointed to more than one of the association map.

```

noRelayVariableOverlap : TrainMovementAssociations → Bool
noRelayVariableOverlap(tma) ≡
  let connectionVars = rng connectionVars(tma),
      counterVars = rng counterVars(tma),
      directionFwdVars = rng directionFwdVars(tma),
      directionBwdVars = rng directionBwdVars(tma)
  in connectionVars ∩ counterVars = {} ∧
      connectionVars ∩ directionFwdVars = {} ∧
      connectionVars ∩ directionBwdVars = {} ∧
      counterVars ∩ directionFwdVars = {} ∧
      counterVars ∩ directionBwdVars = {} ∧
      directionFwdVars ∩ directionBwdVars = {}
  end

```

8.2.1.1 Well-formedness of connectionVars

The connection variables describe whether two neighbouring track sections are occupied by a single train, or by two separate trains.

The connection variables association is well-formed if

1. The pair of track sections in each key are neighbours.
2. Variables are not shared.

value

```
isWfConnectionVars : Diagram ×
  TrainMovementAssociations → Bool
isWfConnectionVars(d, tma) ≡
  conVarsValidDomain(d, tma) ∧
  conVarsValidRange(connectionVars(tma))
```

1. Only neighbours can have a connection variable Only track sections that are neighbours may have a connection variable.

value

```
conVarsValidDomain : Diagram ×
  TrainMovementAssociations → Bool
conVarsValidDomain(d, tma) ≡
  (∀ (tId1, tId2) : (TrackId × TrackId) •
    (tId1, tId2) ∈ dom connectionVars(tma) ⇒
      areNeighbours(tId1, tId2, d) )
```

2. No shared variables No two neighbour pair may share a connection variable.

value

```
conVarsValidRange : ((TrackId × TrackId)  $\xrightarrow{m}$  VarId) → Bool
conVarsValidRange(connectionVars) ≡
  (∀ key : (TrackId × TrackId) •
    key ∈ dom connectionVars ⇒
      ~( ∃ key' : (TrackId × TrackId) •
        key' ∈ dom connectionVars ∧
        key ≠ key' ∧
        connectionVars(key) =
        connectionVars(key') )
  )
```

8.2.1.2 Well-formedness of counterVars

The track section to counter variable association is well-formed if:

1. Each track section on the station is part of the domain and every element in the domain is part of the station.
2. Variables are not shared.

value

```

isWfCounterVars : Diagram ×
  TrainMovementAssociations → Bool
isWfCounterVars(d, tma) ≡
  cntVarsValidDomain(d, tma) ∧
  cntVarsValidRange(counterVars(tma))

```

1. All track sections Each track section on the station has a counter variable and each track section in the domain of the map is also part of the station.

value

```

cntVarsValidDomain : Diagram ×
  TrainMovementAssociations → Bool
cntVarsValidDomain(d, tma) ≡
  let allTrackSections = allLinears(d) ∪ allPoints(d)
  in dom counterVars(tma) = allTrackSections
end

```

2. No shared variables No two track sections may share a counter variable.

value

```

cntVarsValidRange : (TrackId  $\xrightarrow{m}$  VarId) → Bool
cntVarsValidRange(counterVars) ≡
  (∀ key : TrackId •
    key ∈ dom counterVars ⇒
    ¬( ∃ key' : TrackId •
      key' ∈ dom counterVars ∧

```

$$\begin{aligned}
 & \text{key} \neq \text{key}' \wedge \\
 & \text{counterVars}(\text{key}) = \\
 & \text{counterVars}(\text{key}')) \\
 &)
 \end{aligned}$$

8.2.1.3 Well-formedness of fwdVars

The track section to forward direction variable association is well-formed if:

1. Each track section on the station in part of the domain and every element in the domain is part of the station.
2. Variables are not shared.

value

```

isWfDirectionFwdVars : Diagram ×
  TrainMovementAssociations → Bool
isWfDirectionFwdVars(d, tma) ≡
  fwdVarsValidDomain(d, tma) ∧
  fwdVarsValidRange(directionFwdVars(tma))

```

1. All track sections Each track section on the station has a forward direction variable and each track section in the domain of the map is also part of the station.

value

```

fwdVarsValidDomain : Diagram ×
  TrainMovementAssociations → Bool
fwdVarsValidDomain(d, tma) ≡
  let allTrackSections = allLinears(d) ∪ allPoints(d)
  in dom directionFwdVars(tma) = allTrackSections
end

```

2. No shared variables No two track sections may share a forward direction variable.

value

```

fwdVarsValidRange : (TrackId  $\xrightarrow{m}$  VarId)  $\rightarrow$  Bool
fwdVarsValidRange(directionFwdVars)  $\equiv$ 
  ( $\forall$  key : TrackId •
    key  $\in$  dom directionFwdVars  $\Rightarrow$ 
       $\sim$ (  $\exists$  key' : TrackId •
        key'  $\in$  dom directionFwdVars  $\wedge$ 
        key  $\neq$  key'  $\wedge$ 
        directionFwdVars(key) =
        directionFwdVars(key') )
  )

```

8.2.1.4 Well-formedness of bwdVars

The track section to backward direction variable association is well-formed if:

1. Each track section on the station in part of the domain and every element in the domain is part of the station.
2. Variables are not shared.

value

```

isWfDirectionBwdVars : Diagram  $\times$ 
  TrainMovementAssociations  $\rightarrow$  Bool
isWfDirectionBwdVars(d, tma)  $\equiv$ 
  bwdVarsValidDomain(d, tma)  $\wedge$ 
  bwdVarsValidRange(directionBwdVars(tma))

```

1. All track sections Each track section on the station has a backward direction variable and each track section in the domain of the map is also part of the station.

value

```

bwdVarsValidDomain : Diagram ×
  TrainMovementAssociations → Bool
bwdVarsValidDomain(d, tma) ≡
  let allTrackSections = allLinears(d) ∪ allPoints(d)
  in dom directionBwdVars(tma) = allTrackSections
end

```

2. No shared variables No two track sections may share a backward direction variable.

```

value
bwdVarsValidRange : (TrackId  $\overrightarrow{\text{m}}$  VarId) → Bool
bwdVarsValidRange(directionBwdVars) ≡
  (∀ key : TrackId •
    key ∈ dom directionBwdVars ⇒
    ~ ( ∃ key' : TrackId •
      key' ∈ dom directionBwdVars ∧
      key ≠ key' ∧
      directionBwdVars(key) =
      directionBwdVars(key') )
  )

```

8.2.1.5 Well-formedness of directions

The directions association is well-formed if

1. The pair of track sections in each key are neighbours.
2. Each neighbour pair have both and forward and a backward direction.

```

value
isWfDirections : Diagram ×
  TrainMovementAssociations → Bool
isWfDirections(d, tma) ≡
  directionsValidDomain(d, tma) ∧
  bothDirections(directions(tma))

```

1. Track section pairs are neighbours Each track section pair in the domain are neighbours.

value

```

directionsValidDomain : Diagram ×
  TrainMovementAssociations → Bool
directionsValidDomain(d, tma) ≡
  (∀ (tId1, tId2) : (TrackId × TrackId) •
    (tId1, tId2) ∈ dom directions(tma) ⇒
      areNeighbours(tId1, tId2, d) )

```

2. Both directions present Each neighbour pair has both an associated forward and backward variable.

value

```

bothDirections :
  ((TrackId × TrackId)  $\overrightarrow{\text{map}}$  Direction) → Bool
bothDirections(directions) ≡
  ( ∀ (tId1, tId2) : (TrackId × TrackId) •
    (tId1, tId2) ∈ dom directions ⇒
      ( ∃ (tId1', tId2') : (TrackId × TrackId) •
        (tId1', tId2') ∈ dom directions ∧
          tId1 = tId2' ∧ tId2 = tId1' ∧
            directions((tId1, tId2)) ≠
              directions((tId1', tId2'))
        )
    )
  )

```


CHAPTER 9

Inter-Model Consistency

The following chapter introduces conditions for the inter-model consistency that are requirements to ensure that the interaction between the model of train movement and the relay system model behaves as expected.

The requirements are first described informally in *italics*. Then the requirements are written formally as LTL assertions. Finally each formulae is used to write a concrete example for Stenstrup station.

The model of train movement in chapter 6 introduced variables used to model the behaviour of the external environment. This chapter uses these variables to present formulae to ensure the inter-model consistency.

The presented patterns in this chapter are useful when we in chapter 11 introduce the Generator, that are using these formulae to generate the assertions.

The formal properties are written as formulas in linear temporal logic (LTL).

Section 9.1 will introduce variables used in the LTL expressions discussed in this chapter and chapter 10.

The rest of the sections within this chapter will introduce the inter-model consistency conditions.

9.1 Variables

The model of train movement simulates the movement on the track sections. We want to ensure that changes to the state space of the model of train movement is reflected in the model of the interlocking system, i.e. when the model of train movement claims a train occupy a given track section, the interlocking system should detect that track section as occupied and vice versa.

The variables introduced below should be interpreted as place holders for the actual relays in the internal system model. Their values are boolean, meaning it consists by either true or false. *TrackCounter* is, on the other hand, a variable that contains unsigned natural numbers.

Variables for the relay system model:

<i>TrackRelay_T</i>	False when track section <i>T</i> is occupied by a train.
<i>PlusRelay_P</i>	True when points section <i>P</i> is positioned in plus. False when <i>P</i> is not positioned in plus configuration.
<i>MinusRelay_P</i>	True when points section <i>P</i> is positioned in minus. False when <i>P</i> is not positioned in minus configuration.

Note when *PlusRelay_P* and *MinusRelay_P* for points section *P* is both false the points position configuration is in the intermediate state. Refer to section [7.2](#) on point behaviour.

Variables introduced in the model of train movement:

<i>TrackCounter_T</i>	Counts the number of trains on a track section <i>T</i> .
<i>Forward_T</i>	True if a train is moving forward on track section <i>T</i> . False when no train is moving forward on track section <i>T</i> .
<i>Backward_T</i>	True if a train is moving backward on track <i>T</i> . False when no train is moving backward on track section <i>T</i> .
<i>Connection_{T₁T₂}</i>	True if a train is intersecting track section <i>T₁</i> and track section <i>T₂</i> .

9.2 Track Occupation

A track section can either be occupied or vacant. The model of the internal interlocking system uses this information to set the relay as a boolean value, where the train movement model uses counters to enable more trains on a track section. The agreement among the two models is that the model of train movement knows its own location and this location must be detected by the internal system.

To ensure that the relays and the counters are interacting as expected two requirements are formed.

If a track relay $TrackRelay_T$, for track T is dropped, then the associated counter is greater than zero.

$$\bigwedge_{T \in AllTracks} \Box \neg TrackRelay_T \Rightarrow TrackCounter_T > 0$$

If a track counter $TrackCounter_T$ associated to track T is greater than zero, then track relay $TrackRelay_T$ is dropped

$$\bigwedge_{T \in AllTracks} \Box TrackCounter_T > 0 \Rightarrow \neg TrackRelay_T$$

These two requirements ensure that if and only if there is detected a train on track T the $TrackCounter_T$ is greater than zero and the track relay $TrackRelay_T$ is dropped.

Example The expression below requires that if track relay $a12$ is dropped, then track counter cnt_A12 must be greater than zero.

$$\Box \neg a12 \Rightarrow cnt_A12 > 0$$

And if the track counter cnt_A12 is greater than zero, then track relay $a12$ must be dropped.

$$\Box cnt_A12 > 0 \Rightarrow \neg a12$$

9.3 Track Free

Similar to occupying tracks, requirements of freeing tracks must be formulated. Thus, when the model of train movement dictates there is no train on a certain

track the relay system model detects no train and vice versa.

To ensure that the counters behaviour is as expected two requirements are formed when tracks are free.

If a track relay $TrackRelay_T$, for track T is drawn, then the associated counter is equal to zero.

$$\bigwedge_{T \in \text{AllTracks}} \Box TrackRelay_T \Rightarrow TrackCounter_T = 0$$

If a track counter $TrackCounter_T$ associated to track T is equal zero, then track relay $TrackRelay_T$ is drawn

$$\bigwedge_{T \in \text{AllTracks}} \Box TrackCounter_T = 0 \Rightarrow TrackRelay_T$$

Again, this ensures that if and only if there is not detected a train on track T the $TrackCounter_T$ is equal to zero and the track relay $TrackRelay_T$ is drawn.

Example The expression below requires that if track relay $t04$ is drawn, then track counter cnt_04 must be equal to zero.

$$\Box t04 \Rightarrow cnt_04 = 0$$

And if the track counter cnt_04 is equal to zero, then track relay $t04$ must be drawn.

$$\Box cnt_04 = 0 \Rightarrow t04$$

9.4 Train Direction

The model uses the notion of train direction, where the direction is either going forward or backward. Each track section have these associated directions described in section 6.2.3. To maintain the inter-model consistency it must be ensured that a moving train is in fact on a track section, hence the associated track section counter is greater than zero. It is never the case that a moving train on a track section is "invisible" i.e. the counter has to detect the train.

It is tempting to ensure that when the counter is greater than zero, then a train is moving. However, this is not the case since a train can be stopped if there is a

signal with stop aspect. For same reason, it can not be concluded that a train is moving if a track relay is dropped.

Another tempting requirement is that the direction cannot be forward and backward at the same time. However, physically the model is able to have two trains on a track section going with an opposite direction.

If a moving train is on track T , then the associated counter for track T is greater than zero.

$$\bigwedge_{T \in \text{AllTracks}} \Box \text{Backward}_T \vee \text{Forward}_T \Rightarrow \text{TrackCounter}_T > 0$$

Example The expression below requires that if a moving train is on track section 03, then track counter *cnt_03* must be greater than zero.

$$\Box \text{bwd_03} \vee \text{fwd_03} \Rightarrow \text{cnt_03} > 0$$

9.5 Points Configuration

The physical limitation of a points section is the position configuration can be in either plus, minus or intermediate. Where intermediate simulates a state that is neither plus nor minus. Thus somewhere in between plus and minus.

This physical requirement must be ensured in the model. The points relay controls the point machines that is the mechanical arm to switch between plus and minus. Assuming that the point machine always obeys the points relay following requirement is stated.

For all points sections P it is never the case they are plus and minus.

$$\bigwedge_{P \in \text{AllPoints}} \Box \text{Idle} \Rightarrow \neg(\text{PlusRelay}_P \wedge \text{MinusRelay}_P)$$

Note by using this expression the intermediate state is also accepted, which is when both *Plus_P* and *Minus_P* are false.

Example The expression below requires that points section 01, is in either plus, minus or intermediate state.

$$\Box \text{Idle} \Rightarrow \neg(\text{plus01} \wedge \text{minus01})$$

9.6 Train Connection

The model uses the notion of train connection, where the connection is a boolean value that for each neighbour pair of track sections is a connection variable. This connection variable indicates whether a train intersects (overlaps) any track sections.

If a train has a connection that intersects track T_1 and T_2 , then the associated track counter for track T_1 and T_2 are greater than zero.

$$\bigwedge_{(T_1, T_2) \in \text{allNeighbours}} \Box \text{Connection}_{T_1 T_2} \Rightarrow \text{TrackCounter}_{T_1} > 0 \wedge \text{TrackCounter}_{T_2} > 0$$

Note that it cannot be concluded that if $\text{TrackCounter}_{T_1}$ and $\text{TrackCounter}_{T_2}$ are greater than zero, then there is a connection. This is simply because two distinct trains can occupy each of these track sections.

Example The expression below requires that if a train is on track section 02 and 03, then track counter cnt_02 and cnt_03 must be greater than zero.

$$\Box \text{con_02_03} \Rightarrow \text{cnt_02} > 0 \wedge \text{cnt_03} > 0$$

Safety Properties

The following chapter describes the safety requirements that ensure safe train operation, which first and foremost includes the safety properties for no collision and no derailling introduced in section 2.4. Additional safety properties derived from the train route table for a given station will be described.

The requirements are first described informally in *italics*. Then the requirements are written formally, but abstract. Finally the abstract formulae are used to write a concrete example for Stenstrup station.

Stenstrup station is secured by a relay based interlocking system. For the remainder of this document, it is assumed that there exists a boolean variable for each relay in the interlocking system, which mirrors the relays state, i.e. is true when the relay is drawn.

The formal properties are written as formulas in linear temporal logic (LTL).

10.1 Variables

We introduce the following variables in addition to those from section 9.1:

$LockingRelay_x$	False when train route x is locked. Occasionally routes share the same locking relay and the specific routes are then differentiated by the position of the points on the route ($PointsLocked_x$).
$PointsLocked_x$	True when the points in route x are locked in the position dictated by the train route table for that route x .
$RouteLocked_x$	True when train route x is locked and the points of the route are locked in the position required for that route, i.e.
$RouteLocked_x = \neg LockingRelay_x \wedge PointsLocked_x$	
$TrackFree_x$	True if every track section in route x is unoccupied, as specified in the train route table for route x .
$SignalsSet_x$	True if every signal on route x is set to the aspect required by the train route table for route x .
$ProceedAspect_S$	True when signal S signals proceed.
$StopAspect_S$	True when signal S signals stop.
$SignalReleaseIsolation_x$	False when signal release isolation on route x is occupied.
$PositionRelay_{T,P}$	True when the points section P is locked to the branch, that the neighbouring track section T is connected to.
$Indl_x$	True when the first condition (Indl) of the train route release in the train route table for route x holds.
Opl_x	True when the second condition (Opl) of the train route release in the train route table for route x holds.

10.2 No Collision

The first safety requirement mentioned in section 2.4.1.1 that would ensure secure train operation is no collision.

In our model the definition of a train collision is when more than one train occupy the same track section. By using the track counter this enables the detection of a train collision.

For all track sections it is always the case that a track counter associated to track section T is less than two.

$$\bigwedge_{T \in \text{AllTracks}} \Box \text{TrackCounter}_T < 2$$

Example The expression below expresses that track counter associated with track section 01 is always less than two.

$$\Box \text{cnt_01} < 2$$

10.3 No Derailing

The second safety requirement mentioned in section 2.4.1.2 that would ensure secure train operation is no derailing.

In our model there are two kinds of a train derailment.

- First kind is that the points configuration is not locked in a position when a train is occupying the points section. Observe derailing scenarios B and C on figure 2.8.
- Second kind is when a train is connected to a branch of a points section, but the points configuration is not set correspondingly. Observe scenarios A and D on figure 2.8.

Considering the first kind of derailment it must be ensured, that as long as there is atleast one train occupying a points section, the points stay locked in

a position. This will avoid an intermediate state when trains are travelling on points sections.

For all points sections if the associated track counter for points section P is greater than or equal to one, then the relay must be in either plus or minus.

$$\bigwedge_{P \in \text{AllPoints}} \square \text{TrackCounter}_P \geq 1 \Rightarrow \text{PlusRelay}_P \vee \text{MinusRelay}_P$$

From the requirement given in section 9.5 it is ensured that it is never the case that a points section is configured to plus and minus.

Example The expression below shows that points section 01 is occupied then the points positioning is in either plus or minus.

$$\square \text{cnt_01} \geq 1 \Rightarrow \text{plus01} \vee \text{minus01}$$

Considering the second kind of derailment it must be ensured that the points section position configuration is in accordance to the connection. Thus this rule ensures that the configuration is set when there is a connection.

If a train occupies track section T and the neighbouring points section P at the branch side, then the points section must be positioned as required in the interlocking plan.

$$\bigwedge_{(T,P) \in \text{allNeighbours}} \square \text{Connection}_{T,P} \Rightarrow \text{PositionRelay}_{T,P}$$

One could be tempted to say that if the points section is locked then there is a connection, however that is not always the case.

Example The example below shows that if there is a train covering track section 02 and points section 03, then point machine 02 associated with points section 03 must be positioned in plus.

$$\square \text{con_02_03} \Rightarrow \text{plus02}$$

10.4 Points Position

If a train route has been locked, with locking relay lr , then for one of the routes x , which is controlled by lr , the points of that route, must be positioned as required in the train route table for route x .

$$\bigwedge_{lr \in \text{AllLockingRelays}} \Box \neg lr \Rightarrow \bigvee_{x \in \text{RoutesOf}(lr)} \text{PointsLocked}_x$$

where $\text{RoutesOf}(lr)$ is the set of routes controlled by the locking relay lr .

Example If train route 2 or 3 are locked, then both points are either locked in the plus or the minus position as required by the train route table for route 2 and 3 respectively.

$$\Box \neg ia \Rightarrow (\text{plus01} \wedge \text{plus02}) \vee (\text{minus01} \wedge \text{minus02})$$

10.5 Signal

A signal can either show a proceed aspect or a stop aspect. A proceed aspect is indicated by a green and not a red light, and a stop aspect is indicated by a red and not a green light. In a malfunctioning relay system undesired aspects could arise, such as a signal neither shows green nor red light or when a signal shows both green and red light simultaneously. Since they are not valid aspects, the train should stop.

It is always the case that when a signal shows the proceed aspect it only shows a green light and when showing stop aspect it only shows a red light.

For all signals S it is never the case they are red and green.

$$\bigwedge_{S \in \text{AllSignals}} \Box \text{Idle} \Rightarrow \neg(\text{Red}_S \wedge \text{Green}_S)$$

where Idle is true, when the relay system is ready to accept external events. Red_S is true, when signal S is showing stop aspect and Green_S is true, when the signal S is showing proceed aspect.

If a signal S shows proceed aspect, then a route x , starting from S , is locked and all the related track sections for that route are unoccupied, and all points of that route are positioned as required in the train route table.

$$\Box \text{Idle} \wedge \text{ProceedAspect}_S \Rightarrow \bigvee_{x \in \text{Routes}(S)} \text{RouteLocked}_x \wedge \text{TracksFree}_x \wedge \text{SignalsSet}_x$$

where ProceedAspect_S is true, when signal S is showing proceed aspect. $\text{Routes}(S)$ is the set of routes starting from signal S .

Example Applying this to the concrete example for entrance signal A the following abstract expression is yielded:

$$\begin{aligned} &\Box \text{Idle} \wedge \text{ProceedAspect}_A \Rightarrow \\ &(\text{RouteLocked}_2 \wedge \text{TracksFree}_2 \wedge \text{SignalsSet}_2) \vee \\ &(\text{RouteLocked}_3 \wedge \text{TracksFree}_3 \wedge \text{SignalsSet}_3) \end{aligned}$$

as train routes 2 and 3 starts from signal A.

Replacing the abstract variables with their concrete counterparts in terms of relay variables gives:

$$\begin{aligned} &\Box \text{idle} \wedge \text{aGreen} \Rightarrow \\ &((\neg \text{ia} \wedge \text{plus01} \wedge \text{plus02}) \wedge \\ &(\text{a12} \wedge \text{t01} \wedge \text{t02} \wedge \text{t03} \wedge \text{b12}) \wedge \\ &(\text{fRed} \wedge (\text{gRed} \vee (\text{gGreen} \wedge \neg \text{ub} \wedge \text{plus02})))) \vee \\ &((\neg \text{ia} \wedge \text{minus01} \wedge \text{minus02}) \wedge \\ &(\text{a12} \wedge \text{t01} \wedge \text{t04} \wedge \text{t03} \wedge \text{b12}) \wedge \\ &(\text{eRed} \wedge (\text{hRed} \vee (\text{hGreen} \wedge \neg \text{ub} \wedge \text{minus02})))) \end{aligned}$$

This can be reduced to:

$$\begin{aligned} &\Box \text{idle} \wedge \text{aGreen} \Rightarrow \\ &\neg \text{ia} \wedge \text{a12} \wedge \text{t01} \wedge \text{t03} \wedge \text{b12} \wedge \\ &(\text{plus01} \wedge \text{plus02} \wedge \text{t02} \wedge \text{fRed} \wedge (\text{gRed} \vee (\text{gGreen} \wedge \neg \text{ub} \wedge \text{plus02})))) \vee \\ &\text{minus01} \wedge \text{minus02} \wedge \text{t04} \wedge \text{eRed} \wedge (\text{hRed} \vee (\text{hGreen} \wedge \neg \text{ub} \wedge \text{minus02}))) \end{aligned}$$

The expression above shows that if entrance signal A is showing proceed aspect, then train route 2 or 3 is locked and their common track sections are unoccupied. Depending on whether route 2 or 3 is locked the points are set correspondingly, as described in section 10.4, and the track section, $t02$ or $t04$, for that route is unoccupied, required in the train route table. Signals E and F are showing stop aspect depending on the point positioning. Finally, signals G and H are showing stop aspect unless an exit route has already been locked¹. For route 2 this means that G shows stop unless route 9 is also locked in which case G may show a drive aspect. Similarly, for route 3 this means that H shows stop unless route 10 is also locked in which case H may show a proceed aspect².

10.6 Signal Release

If the track section $SignalReleaseIsolation_x$, specified in the signal release field for route x in the train route table, is occupied, then the signal S in the signal release field must show stop aspect.

$$\Box Idle \wedge \neg SignalReleaseIsolation_x \Rightarrow StopAspect_x$$

If signal release isolation on route x is occupied the signal release signal on route x must show stop aspect.

Example The expression below requires that if track section $a12$ is occupied by an object, then signal A must show a stop aspect and no longer show a drive aspect.

$$\Box idle \wedge \neg a12 \Rightarrow aRed$$

Recall from section 10.5 that a signal may not show a red and a green light simultaneously.

10.7 Conflicting Routes

If a train route x is locked, then a conflicting route y must not be locked.

$$\Box RouteLocked_x \Rightarrow \bigwedge_{y \in ConflictingRoutes(x)} \neg RouteLocked_y$$

¹Viser "kør" hvis der i forvejen er stillet udkørsel

²Note that these routes are not conflicting routes.

Example If route 2 is locked the following abstract expression is given:

$$\begin{aligned} \Box \text{RouteLocked}_2 \Rightarrow & \neg \text{RouteLocked}_3 \wedge \\ & \neg \text{RouteLocked}_5 \wedge \\ & \neg \text{RouteLocked}_6 \wedge \\ & \neg \text{RouteLocked}_7 \wedge \\ & \neg \text{RouteLocked}_8 \wedge \\ & \neg \text{RouteLocked}_{10} \end{aligned}$$

Applying this to the concrete example for route 2 the following is given:

$$\begin{aligned} \Box \neg ia \wedge plus01 \wedge plus02 \Rightarrow & \\ \neg(\neg ia \wedge minus01 \wedge minus02) \wedge & \\ \neg(\neg ib \wedge plus01 \wedge plus02) \wedge & \\ \neg(\neg ib \wedge minus01 \wedge minus02) \wedge & \\ \neg(\neg ua \wedge plus01) \wedge & \\ \neg(\neg ua \wedge minus01) \wedge & \\ \neg(\neg ub \wedge minus02) & \end{aligned}$$

The expression above requires that if train route 2 is locked, then train routes 3, 5, 6, 7, 8 and 10 are not locked.

10.8 Train Route Release

In order to release a locked train route there must occur a *release sequence*. Recall from section 2.6.2.7, that the release sequence involves a train to occupy and leave track sections in a certain order.

There are two properties for train route release. One is a safety property, the other a liveness property.

Safety *If a train route has been locked, then the route is not released until after the release sequence (Indl. \rightarrow Opl.)*

Liveness *After the release sequence, the route is eventually released³*

³Not considered

10.8.1 LTL General Form

The safety property is expressed by:

$$\begin{aligned} & \Box(\text{LockingRelay}_x \wedge \bigcirc(\text{RouteLocked}_x \wedge \Diamond \text{LockingRelay}_x) \Rightarrow \\ & \bigcirc(\neg \text{LockingRelay}_x \mathcal{U} \neg \text{LockingRelay}_x \wedge \text{Indl}_x \wedge \\ & \bigcirc(\neg \text{LockingRelay}_x \mathcal{U} \neg \text{LockingRelay}_x \wedge \text{Opl}_x))) \end{aligned}$$

where \mathcal{U} is the LTL until operator, x is the train route, Indl_x is the start condition for the release sequence for route x and Opl_x is the end condition.

Step-by-step Explanation We begin the expression by finding the exact moment when the route is locked. This is done by finding the critical time, where the route is not locked in the current state, but is locked in the next state. When the route is not locked the position of the points is irrelevant, however when the route is locked it is necessary to know the position of the points in addition to the state of the route relay to determine which route is locked.

$$\text{LockingRelay}_x \wedge \bigcirc(\text{RouteLocked}_x)$$

From this point on, there are two options. Either the route is never released or it eventually is. Since the model of external events allow infinite repeated actions (such as pressing a button repeatedly infinitely many times on the operator's panel), the case where the route is never released is allowed. If the route is never released, then it follows that the route is not released before the release sequence. On the other hand, if the route is eventually released, then it must occur after the release sequence.

$$\text{LockingRelay}_x \wedge \bigcirc(\text{RouteLocked}_x \wedge \Diamond \text{LockingRelay}_x) \Rightarrow \bigcirc(\dots)$$

If the route is never released, then the left side of the implication is false. That means that the entire expression is true in that state, regardless of the right side. However, if the left side is true, then the right side must evaluate to true as well for the entire expression to be true.

Therefore the dots must be replaced with an expression that describes, that the route is not released until after the release sequence.

The route must be locked until (and including) the time of the initiation (*Indl*) of the release sequence.

$$\neg \text{LockingRelay}_x \mathcal{U} \neg \text{LockingRelay}_x \wedge \text{Indl}_x$$

And once the initiation (*Indl*) is reached, we may still not release until the last condition *Opl*, which completes the sequence:

$$\begin{aligned} & \neg \text{LockingRelay}_x \mathcal{U} \neg \text{LockingRelay}_x \wedge \text{Indl}_x \wedge \\ & \bigcirc (\neg \text{LockingRelay}_x \mathcal{U} \neg \text{LockingRelay}_x \wedge \text{Opl}_x) \end{aligned}$$

The formula is then completed by combining the two fragments and ensuring that it holds for all states:

$$\begin{aligned} & \Box (\text{LockingRelay}_x \wedge \bigcirc (\text{RouteLocked}_x \wedge \Diamond \text{LockingRelay}_x) \Rightarrow \\ & \bigcirc (\neg \text{LockingRelay}_x \mathcal{U} \neg \text{LockingRelay}_x \wedge \text{Indl}_x \wedge \\ & \bigcirc (\neg \text{LockingRelay}_x \mathcal{U} \neg \text{LockingRelay}_x \wedge \text{Opl}_x))) \end{aligned}$$

Example The following is a concrete example for train route 2 of Stenstrup Station.

$$\begin{aligned} & \Box (\text{ia} \wedge \bigcirc (\neg \text{ia} \wedge \text{plus01} \wedge \text{plus02} \wedge \Diamond \text{ia}) \Rightarrow \\ & \bigcirc (\neg \text{ia} \mathcal{U} \neg \text{ia} \wedge \neg \text{t01} \wedge \text{t02} \wedge \\ & \bigcirc (\neg \text{ia} \mathcal{U} \neg \text{ia} \wedge \neg \text{t02} \wedge \text{t01}))) \end{aligned}$$

where *plus01*, *plus02* are true when points 01 and 02 are locked in the plus position and *ia* is false when route 2 or 3 is locked.

Notice that even though some routes share the same locking relay, it is only needed to check the points position, when the route is established, as the points are not allowed to be changed when a route is locked.

Generator

Our generator takes the data model of the interlocking plan discussed in section 4.1 and transforms it into a transition system containing a state space, transition rules of the external events, and assertions (section 4.4). The generated transition system can then be combined with the transition system of the internal events to obtain the complete system for verification.

This chapter will describe the flow of the generator, the required input of the generator, how the generator is creating the output and finally how the output is then unparsed to RSL-SAL.

Section 11.2 shows the sub-generator that facilitates the generation of the state space.

Section 11.3 shows the sub-generator that facilitates the generation of the transition rules of the external behaviour.

Section 11.4 shows the sub-generator that facilitates the generation of the safety properties and the inter-model consistency assertions.

Section 11.5 shows how the output is generated by using an unparser.

11.1 Overview

The generator requires as input the interlocking plan that contains a station layout diagram and a train route table, which chapter 4 showed the type definitions of. Additionally, the associations are needed for the generator to map the names from the station elements with the relay system. The generator outputs a transition system, which contains a state space, transition rules and assertions. Figure 11.1 illustrates this simplified partitioning.

The specific input and output will be shown in the study case of Stenstrup in chapter 13.

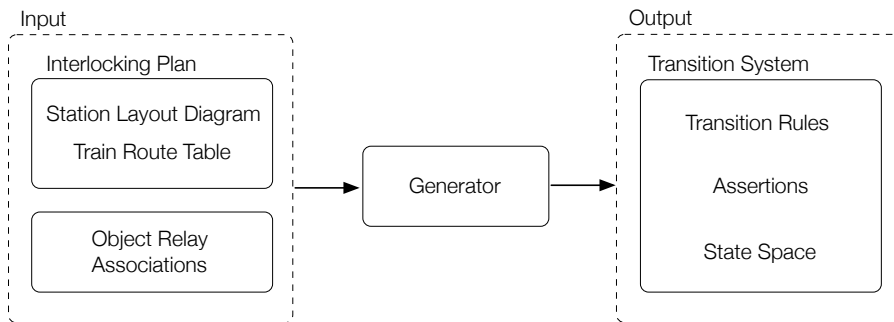


Figure 11.1: Overview of the generator with the required input and generated output.

An overview of the application flow is illustrated on figure 11.2. The file `Stenstrup`, specifically made for the Stenstrup station, instantiates `StationLayout` and `ObjectRelayAssociations` and invokes the `Generator`. Within the generator subclasses facilitates the generation of state space, transition rules and assertions.

The State Generator creates and initialises the state space of the transition system (contained in the type `TrainMovementAssociations`). Transition Generator facilitates the creation of the transition rules of the external events and the Assertion Generator facilitates the creation of assertions containing safety properties and inter-model consistency conditions. The output from each generator are then stored in a transition system, where the unparser prints it in a form that RSL-SAL accepts.

The *generate* function, located in the `Generator.rsl` file, constructs a transition system by requiring an interlocking plan, object relay associations used to map the station objects with the relays, the set of button identifiers¹, and finally a

¹Recall from section 7.1 that button identifiers cannot be extracted from the interlocking

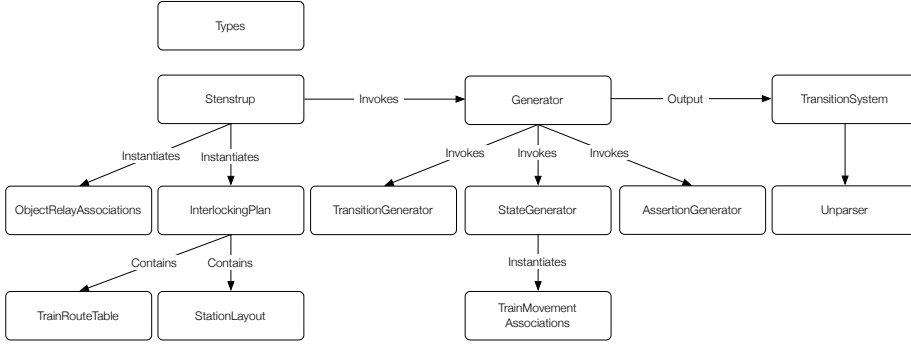


Figure 11.2: Structure of the implementation.

text string to name the transition system. Each sub-generator is then invoked to create the transition system.

value

```

generate : Text × ObjectRelayAssociations ×
  InterlockingPlan × Text-set → TransitionSystem
generate(name, ora, ip, buttonIds) ≡
  let sld = sld(ip),
    tma = genTrainMovementAssociations(sld)
  in mk_TransitionSystem(
    name,
    generateStateSpace(ip, tma),
    generateTransitionSystem(ora, ip, tma, buttonIds),
    generateAssertions(ora, ip, tma))
end
pre isWfInterlockingPlan(ip) ∧
  isWfObjectRelayAssociations(ora, ip)

```

11.2 State Generator

According to our model of train movement, the *counter*, *connection*, and *direction* variables are to be generated.

All of this is stored in a *TrainMovementAssociation* value and is created by plan nor can they be derived.

functions defined in `StateGenerator.rsl`, which can be found in appendix section E.3.3 and E.2.4, respectively.

type

```
TrainMovementAssociations ::
  connectionVars : (TrackId × TrackId)  $\overrightarrow{m}$  VarId
  counterVars : TrackId  $\overrightarrow{m}$  VarId
  directionFwdVars : TrackId  $\overrightarrow{m}$  VarId
  directionBwdVars : TrackId  $\overrightarrow{m}$  VarId
  directions : (TrackId × TrackId)  $\overrightarrow{m}$  Direction
```

where direction is

type

```
Direction == fwd | bwd
```

The generated TrainMovementAssociations are then used as input to the state generator to generate all the variables by calling the function generateStateSpace. The preconditions requires that the interlocking plan and the train movement associations are well-formed.

value

```
generateStateSpace : InterlockingPlan ×
  TrainMovementAssociations → Var-set
generateStateSpace(ip, tma) ≡
  genBoolVars(rng connectionVars(tma)) ∪
  genNatVars(rng counterVars(tma)) ∪
  genBoolVars(rng directionFwdVars(tma)) ∪
  genBoolVars(rng directionBwdVars(tma))
pre isWfInterlockingPlan(ip) ∧
  isWfTrainMovementAssociations(sld(ip), tma)
```

In the following sections the three types of variables will be introduced.

11.2.1 Counter Variables

Every track section has a counter which keeps track of how many trains currently occupy the section. Therefore the state generator recursively goes through every

track section (except for *line*) and creates a name of the counter variable for it. The name is created by prefixing “cnt_” to the name of the track section.

```

value
genCntVars : TrackId-set  $\times$  Diagram  $\rightarrow$  TrackId  $\xrightarrow{m}$  VarId
genCntVars(ids, d)  $\equiv$ 
  if ids = {} then [] else
    let id = hd ids, ids' = ids \ {id}
    in if isLine(id, d) then []
      else genCntVar(id)
    end  $\cup$  genCntVars(ids', d)
  end
end,

genCntVar : TrackId  $\rightarrow$  TrackId  $\xrightarrow{m}$  VarId
genCntVar(id)  $\equiv$  [id  $\mapsto$  "cnt_" ^ id]

```

It is then stored in a map, where the id of the track section is used as key.

```

value
counterVars : TrackId  $\xrightarrow{m}$  VarId

```

This creates an *association* between a track section and the variable.

11.2.2 Connection Variables

Every track section has a connection variable for each of its neighbours. Therefore, the generator uses the neighbour relation, introduced in section 4.2.4, as it only contains one entry for each pair of neighbours. As with the counter variables the *line* section is disregard. The name is generated by prefixing “con_” to the identifiers of the neighbouring track section, which have been concatenated with an underscore in between.

```

value
genConVars : (TrackId  $\times$  TrackId)-set  $\times$  Diagram  $\rightarrow$ 
  (TrackId  $\times$  TrackId)  $\xrightarrow{m}$  VarId
genConVars(nbs, d)  $\equiv$ 
  if nbs = {} then [] else

```

```

    let (nb1,nb2) = hd nbs, nbs' = nbs \ {(nb1,nb2)}
    in if isLine(nb1, d)  $\vee$  isLine(nb2, d)
        then genConVars(nbs', d)
        else genConVar(nb1, nb2)  $\cup$  genConVars(nbs', d)
    end
end
end,

```

```

genConVar : TrackId  $\times$  TrackId  $\rightarrow$ 
            (TrackId  $\times$  TrackId)  $\xrightarrow{m}$  VarId
genConVar(id1, id2)  $\equiv$  [(id1,id2)  $\mapsto$  "con_"  $\wedge$  id1  $\wedge$  "_"  $\wedge$  id2],

```

The ids of the connection variables are stored in a map that use the neighbouring pair of track sections as key.

```

value
connectionVars : (TrackId  $\times$  TrackId)  $\xrightarrow{m}$  VarId

```

11.2.3 Direction Variables

Recall section 6.2.3, that each track section has one forward and one backward variable. They are created in a similar fashion to the counters and therefore the code will not be shown here.

The state generator creates these variables by using the principle of source and sink. Travelling away from the source (towards the sinks) are considered forward, whereas the opposite is considered backward. The source and sinks are the station borders, where trains can enter and exit the station, i.e. the *line*. One of these are chosen as the source while the rest are considered sinks (fig. 11.3).

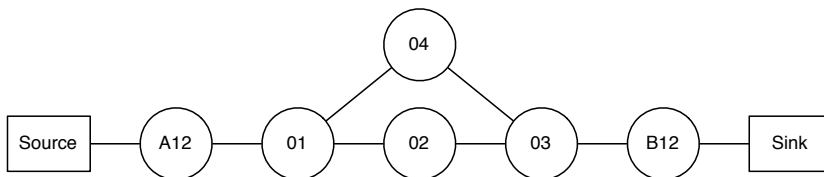


Figure 11.3: Stenstrup in graph form.

It is noteworthy that the model of the station only has a single *line* section (fig. 11.4). Therefore it is necessary to consider it a special case. Otherwise it would act only as source and never as sink.

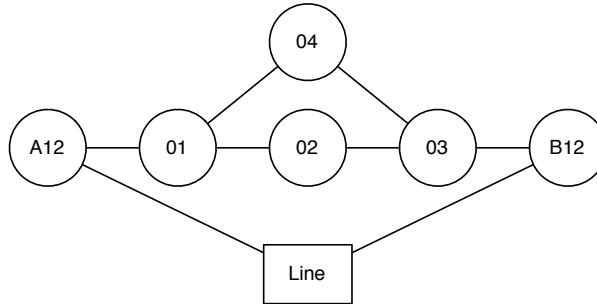


Figure 11.4: Stenstrup in graph form.

The algorithm assigns the directions in the following manner. First all the neighbours of *line* are collected. Lets call them nb_1, \dots, nb_n where n is the number of neighbours to *line* and $n \geq 1$. Then the edge from *line* to nb_1 is given the direction forward and the edge from nb_1 to *line* is marked as backward.

For the remainder, the backward direction will always be the opposite direction of the forward and will therefore not be mentioned again. For the remaining neighbours of *line*, the direction from the neighbour to *line* is forward. In this way a single source with possible multiple sinks have been created (fig. 11.5).

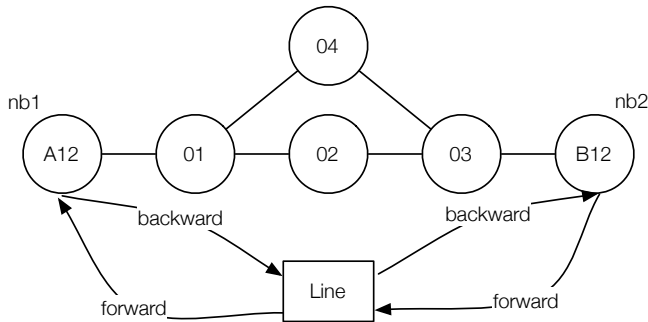


Figure 11.5: Stenstrup in graph form.

The rest of the network is then traversed from the neighbours of the source. Visited nodes are marked so that they are not visited again. In this manner directions are given to the edges of the entire network, except for sources and sinks, which have already been visited in the first step. See figure 11.6 for a step-by-step run-through of the algorithm.

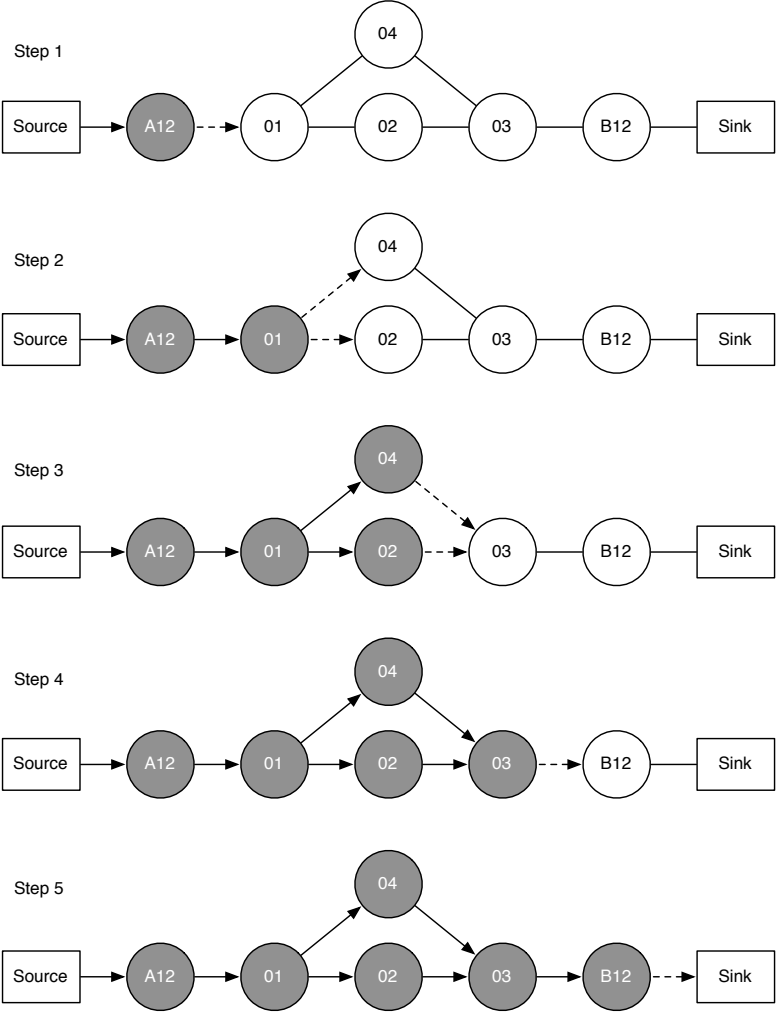


Figure 11.6: Step by step demonstration of the direction generating algorithm.

The result is a mapping from a neighbour pair to a direction, which can be used in creating the rules of train movement.

value

directions : (TrackId \times TrackId) $\overrightarrow{\mapsto}$ Direction

This algorithm has a weakness in that it recognises the station borders as a single line node. Consider a station that has the following station layout on figure 11.7

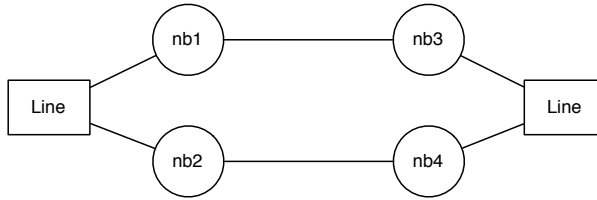


Figure 11.7: Station layout

The algorithm, like before, finds all neighbours to the line. In this case the line has four neighbours, nb1, nb2, nb3 and nb4.

The edge from line to nb1 gets the forward direction, and from nb1 to line gets backward. As stated earlier the algorithm then assigns all the remaining neighbours such that going from the neighbour to line is forward. Figure 11.8 shows the algorithms representation of the station.

Notice that the dotted arrows from line to nb2 and from nb2 to line. This creates inconsistency for the notion of direction, since going from line to nb2 should be forward. Figure 11.9 shows the forward directions and how the dotted forward direction from nb2 to line creates this inconsistency. The backward direction is ignored since it always is the opposite of forward.

As the example shows, the algorithm currently will not work when the line has more than two neighbours. However, one way of solving this is by naming the station borders uniquely. In case of Stenstrup, this could be *Odense* and *Svendborg*.

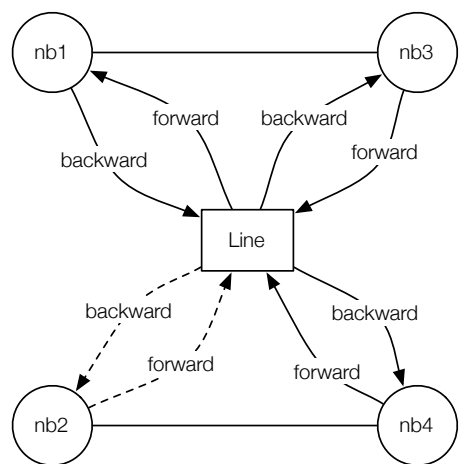


Figure 11.8: The algorithms representation of the station layout

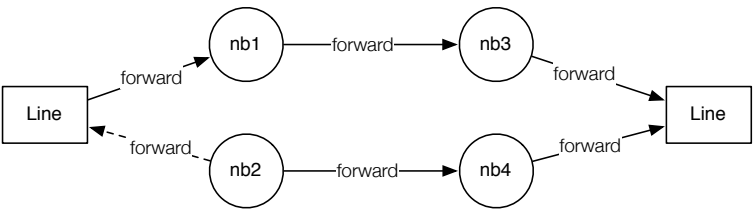


Figure 11.9: Station layout with directions

11.3 Transition Generator

The transition generator facilitates the creation of all the transition rules for the external events, which involves rules for train movement i.e. entering and leaving a station and track sections, rules for points and rules for buttons.

The list below shows each rule that it generates with reference back to the patterns mentioned earlier and reference to implementation details in this section:

- Entering track section rules (Section 6.3.2 and 11.3.2.2).
- Leaving track section rules (Section 6.3.4 and 11.3.2.3).
- Leaving station rule (Section 6.3.3 and 11.3.2.3).
- Button rules (Section 7.1 and 11.3.3.1).
- Points rules (Section 7.2 and 11.3.3.2).

The specification can be found in appendix E.2.5.

11.3.1 Generating All Transition Rules

All transition rules are generated by calling the function `generateTransitionSystem`. The precondition of the transition generator is, that the interlocking plan and the two associations are well-formed.

```

value
  generateTransitionSystem : ObjectRelayAssociations ×
    InterlockingPlan × TrainMovementAssociations ×
    Text-set → TransitionRule-set
  generateTransitionSystem(ora, ip, tma, buttons) ≡
    let d = sld(ip),
      allSections = allSections(d)
    in genTrainMovementTransitions(ora, d, tma) ∪
      genOtherExternalTransitions(buttons, ora, ip)
    end
  pre isWfInterlockingPlan(ip) ∧
    isWfTrainMovementAssociations(sld(ip), tma) ∧
    isWfObjectRelayAssociations(ora, ip)

```

11.3.2 Train Movement Generator

Train movement rules are generated by iterating each track section on a station. Entering and leaving rules are then generated depending on the type of track sections.

This section will describe how the transition rules of train movements are generated from the patterns given in section 6.3.

value

```

genTrainMovementTransitions :
  ObjectRelayAssociations × Diagram ×
  TrainMovementAssociations → TransitionRule-set
genTrainMovementTransitions(ora, d, tma) ≡
  -- 6.3.1 Enter Station
  genEnterStationRules(ora, d, tma) ∪
  -- 6.3.2 Enter Track Section
  genEnterSectionRules(ora, d, tma) ∪
  -- 6.3.3 Leave Station
  genLeaveStationRules(ora, d, tma) ∪
  -- 6.3.4 Leave Track Section
  genLeaveSectionRules(ora, d, tma) ∪
  -- 6.3.5 Change Direction
  genChangeDirectionRules(ora, d, tma)

```

11.3.2.1 Enter Station

Special enter rules are needed where a train enters a track section from the line. Therefore the overloaded functions `genEnterStationRules` recursively iterates through the set of all neighbour pair, where the line is part of the pair.

value -- 6.3.1 Enter Station

```

genEnterStationRules :
  ObjectRelayAssociations × Diagram ×
  TrainMovementAssociations → TransitionRule-set
genEnterStationRules(ora, d, tma) ≡
  let line = line(d),
      nbs = getNeighboursOf(line, d)
  in genEnterStationRules(nbs, ora, d, tma)

```

```

end,

genEnterStationRules : TrackId-set ×
  ObjectRelayAssociations × Diagram ×
  TrainMovementAssociations → TransitionRule-set
genEnterStationRules(nbs, ora, d, tma) ≡
  if nbs = {} then {} else
    let nb = hd nbs,
        nbs' = nbs \ {nb}
    in {genEnterStationRule(nb, ora, d, tma)} ∪
        genEnterStationRules(nbs', ora, d, tma) -- Recursive Call
  end
end

```

For transition from the line to a track section on the station, a rule is created following the pattern shown in section 6.3.1.

```

value
  genEnterStationRule :
    TrackId × ObjectRelayAssociations ×
    Diagram × TrainMovementAssociations → TransitionRule
  genEnterStationRule(nb, ora, d, tma) ≡
    let line = line(d),
        dir = getDirection(line, nb, tma),
        name = genEnterRuleName(line, nb),
        guard = and( { idle } ∪
            proceedAspect(line, nb, ora, d) ),
        update = { systemBusy,
            addTrain(nb, tma),
            dropTrackRelay(nb, ora),
            setDirection(nb, dir, tma) }
    in mk_TransitionRule(name, guard, update)
end

```

where auxiliary functions are defined as follows:

```

value
  genEnterRuleName : TrackId × TrackId → Text
  genEnterRuleName(from, to) ≡ "enter_" ^ to ^ "_from_" ^ from,

```

```

idle : BooleanExp = literal("idle"),

proceedAspect : TrackId × TrackId ×
  ObjectRelayAssociations × Diagram → BooleanExp-set
proceedAspect(from, to, ora, d) ≡
  let sId = signalLocations(d)(from, to),
      sId_gr = signalRelayAssoc(ora)(sId, gr),
      sId_re = signalRelayAssoc(ora)(sId, re)
  in { literal(sId_gr), neg(literal(sId_re)) }
end,

systemBusy : TS.Assignment =
  mk_Assignment("idle", T.BoolAssign(false)),

addTrain : TrackId ×
  TrainMovementAssociations → Assignment
addTrain(tId, tma) ≡
  let cVar = getCounterVar(tId, tma)
  in mk_Assignment(cVar, AddAssign(1))
end,

dropTrackRelay : TrackId ×
  ObjectRelayAssociations → Assignment
dropTrackRelay(tId, ora) ≡
  let rId = getTrackRelay(tId, ora)
  in mk_Assignment(rId, BoolAssign(false))
end,

setDirection : TrackId × Direction ×
  TrainMovementAssociations → Assignment
setDirection(tId, dir, tma) ≡
  if dir = fwd
  then let dVar = getFwdDirVar(tId, tma)
       in mk_Assignment(dVar, BoolAssign(true))
       end
  else let dVar = getBwdDirVar(tId, tma)
       in mk_Assignment(dVar, BoolAssign(true))
       end
  end
end

```

11.3.2.2 Rules for Entering Track Sections

Each track section can be entered from each of its neighbours. The overloaded functions `genEnterSectionRules` recursively iterates through the set of neighbours, but disregards the neighbours to the line, as these have already been considered in section 11.3.2.1. For each pair of neighbours, $(ts1, ts2)$, `genEnterSectionRule` is then called twice, once with $ts1$ as the first argument and once with $ts2$ as the first argument. This creates an enter rule in both directions, i.e. entering $ts1$ from $ts2$ and entering $ts2$ from $ts1$.

```

value -- 6.3.2 Enter Track Section
  genEnterSectionRules : ObjectRelayAssociations × Diagram ×
    TrainMovementAssociations → TransitionRule-set
  genEnterSectionRules(ora, d, tma) ≡
    let nbs = { (tId1, tId2) | (tId1, tId2) : (TrackId × TrackId) •
      (tId1, tId2) ∈ neighbours(d) ∧
      ~isLine(tId1, d) ∧ ~isLine(tId2, d) }
    in genEnterSectionRules(nbs, ora, d, tma)
    end,

  genEnterSectionRules : (TrackId × TrackId)-set ×
    ObjectRelayAssociations × Diagram ×
    TrainMovementAssociations → TransitionRule-set
  genEnterSectionRules(nbs, ora, d, tma) ≡
    if nbs = {} then {} else
      let (tId1, tId2) = hd nbs,
        nbs' = nbs \ {(tId1, tId2)}
      in -- From tId1 to tId2
        genEnterSectionRule(tId1, tId2, ora, d, tma) ∪
        -- From tId2 to tId1
        genEnterSectionRule(tId2, tId1, ora, d, tma) ∪
        -- Recursive Call
        genEnterSectionRules(nbs', ora, d, tma)
    end
end

```

As described in section 6.3.4.1, it matters whether a train enters a points section from the stem side or the branch and likewise if it enters another section from a points section. Each of these cases are handled in `genEnterSectionRule`. Recall, from section 4.2.4, that two points sections are not allowed to be neighbours.

value

```

genEnterSectionRule : TrackId × TrackId ×
  ObjectRelayAssociations × Diagram ×
  TrainMovementAssociations → TransitionRule-set
genEnterSectionRule(from, to, ora, d, tma) ≡
  -- 6.3.2.1 Facing Move - Stem Side
  if isPoints(to, d) ∧ isStemNb(to, from, d)
  then {genEnterLinearSectionRule(from, to, ora, d, tma)}

  -- 6.3.2.1 Facing Move - Branch Side
  elsif isPoints(from, d) ∧ isBranchNb(from, to, d)
  then {genEnterFacingBranchSideRule(from, to, ora, d, tma)}

  -- 6.3.2.1 Trailing Move - Stem Side
  elsif isPoints(from, d) ∧ isStemNb(from, to, d)
  then {genEnterLinearSectionRule(from, to, ora, d, tma)}

  -- 6.3.2.1 Trailing Move - Branch Side
  elsif isPoints(to, d) ∧ isBranchNb(to, from, d)
  then {genEnterFacingBranchSideRule(from, to, ora, d, tma), -- Correct pos
    genEnterTrailingBranchSideRule(from, to, ora, d, tma)} -- Incorrect pos

  else -- 6.3.2 Linear
    {genEnterLinearSectionRule(from, to, ora, d, tma)}
  end

```

The structure of the functions for each case is the same, but the rule that is generated varies slightly. Therefore only `genEnterLinearSectionRule` will be shown and explained here. In fact it follows the same structure as for generating enter station rules, which were shown in section [11.3.2.1](#).

value

```

genEnterLinearSectionRule : TrackId × TrackId ×
  ObjectRelayAssociations × Diagram ×
  TrainMovementAssociations → TransitionRule
genEnterLinearSectionRule(from, to, ora, d, tma) ≡
  let dir = getDirection(from, to, tma),
    name = "enter_" ^ to ^ "_from_" ^ from,
    guard = and(
      { idle,
        trackOccupied(from, tma),
        direction(from, to, tma),
        notConnected(from, to, tma) } ∪

```



```

-- If move passes signal
if hasSignal(from, to, d)
  then proceedAspect(from, to, ora, d) else {} end),
update = {
  systemBusy,
  addTrain(to, tma),
  dropTrackRelay(to, ora),
  connect(to, from, tma),
  setDirection(to, dir, tma) }
in mk_TransitionRule(name, guard, update)
end

```

where

value

```

trackOccupied : TrackId ×
  TrainMovementAssociations → BooleanExp
trackOccupied(tId, tma) ≡
  equals(literal(getCounterVar(tId, tma)), literal("1")),

direction : TrackId × TrackId ×
  TrainMovementAssociations → BooleanExp
direction(from, to, tma) ≡
  literal(getDirVar(from, to, tma)),

notConnected : TrackId × TrackId ×
  TrainMovementAssociations → BooleanExp
notConnected(tId1, tId2, tma) ≡
  neg(literal(getConnectionVar(tId1, tId2, tma))),

connect : TrackId × TrackId ×
  TrainMovementAssociations → Assignment
connect(tId1, tId2, tma) ≡
  let cVar = getConnectionVar(tId1, tId2, tma)
  in mk_Assignment(cVar, T.BoolAssign(true))
end

```

11.3.2.3 Rules for Leaving Track Sections

The rules for leaving track sections are similar to the rules for entering and will not be shown. The patterns for leaving track sections are located at section 6.3.3 and 6.3.4 for behaviour of train movements.

Additionally, generating rules for leaving a station is similar to leaving track sections, except that it is not required to have a connection to the line. Hence, the function `genLineLeaveRules` called by `genRules` will not be shown.

11.3.2.4 Change Direction

Trains are allowed to change direction under certain conditions only, as described in section 6.3.5. Trains may only change direction when it occupies only a single section and by turning around and entering the next section, it would pass a signal showing proceed. The overloaded function `genChangeDirectionRules` recursively iterates through the locations of the signals and creates a rule allowing the train to change direction at those locations. Locations on the line is disregarded as no train is modelled there, so it does not make sense to have a rule allowing it to change direction.

```

value -- 6.3.5 Change Direction
  genChangeDirectionRules :
    ObjectRelayAssociations × Diagram ×
    TrainMovementAssociations → TransitionRule-set
  genChangeDirectionRules(ora, d, tma) ≡
    let allSignalLocations = dom signalLocations(d)
    in genChangeDirectionRules(allSignalLocations, ora, d, tma)
    end,

  genChangeDirectionRules : (TrackId × TrackId)-set ×
    ObjectRelayAssociations × Diagram ×
    TrainMovementAssociations → TransitionRule-set
  genChangeDirectionRules(signalLocations, ora, d, tma) ≡
    if signalLocations = {} then {} else
      let (tId1, tId2) = hd signalLocations,
      signalLocations' = signalLocations \ {(tId1, tId2)}
      in -- Recursive Call
        genChangeDirectionRules(signalLocations', ora, d, tma) ∪
        if tId1 = line(d) then {}
        else {genChangeDirectionRule(tId1, tId2, ora, d, tma)}

```

```

    end
  end
end

```

The transition rule itself is created by the function `genChangeDirectionRule`, which is in the same way as the other transition generating functions.

```

value
  genChangeDirectionRule : TrackId × TrackId ×
    ObjectRelayAssociations × Diagram ×
    TrainMovementAssociations → TransitionRule
  genChangeDirectionRule(from, to, a, d, tma) ≡
    let curDir = directions(tma)(from, to),
        oppDir = oppositeDirection(curDir),
        name = "change_direction_at_" ^ from ^ "_towards_" ^ to,
        guard = { idle,
                  trackOccupied(from, tma),
                  direction(from, to, tma) } ∪
                  proceedAspect(from, to, a, d) ∪
                  isTailOfTrainGuards(from, to, d, tma),
        update = { setDirection(from, oppDir, tma),
                  removeDirection(from, curDir, tma) }
    in mk_TransitionRule(name, T.and(guard), update)
end

```

11.3.3 Other External Events Generator

In addition to the train movements, there are two kinds of external events that have to be modelled. These are the behaviour of buttons and the operation of points, as described in section 7.

The function `genOtherExternalTransitions` is supplied by the generator of other external events and is called by the transition generator.

```

value
  genOtherExternalTransitions :
    Text-set × ObjectRelayAssociations ×
    InterlockingPlan → TransitionRule-set

```

```

genOtherExternalTransitions(buttons, ora, ip)  $\equiv$ 
  -- 7.1 Button Behaviour
  genAllButtonRules(buttons)  $\cup$ 
  -- 7.2 Points Behaviour
  genAllPointsOpRules(ora, ip)

```

The following will explain the generation of the transitions that describe the button and points behaviour in detail.

11.3.3.1 Rules for Buttons

The button behaviour patterns are described in section 7.1. The rules for button presses are simple and so is the generator that generates them. To generate the transition rules for pressing buttons, the set of all button identifiers is required. Refer to the case study of Stenstrup where button identifiers are provided shown in section 13.1.5.

```

value -- 7.1 Button Behaviour
  genAllButtonRules : Text-set  $\rightarrow$  TransitionRule-set
  genAllButtonRules(bs)  $\equiv$ 
    if bs = {} then {} else
      let b = hd bs, bs' = bs \ {b}
      in {genButtonRule(b)}  $\cup$  genAllButtonRules(bs')
    end
  end,

  genButtonRule : Text  $\rightarrow$  TransitionRule
  genButtonRule(b)  $\equiv$ 
    mk_TransitionRule(
      "pushButton_" ^ b,
      T.literal("idle"),
      { busySystem, mk_Assignment(b, BoolAssign(true)) } )

```

A transition rule is generated for each button, by recursively iterating through the set of all buttons. The rule is very simple. If the system is standing idle, the button can be pressed. This causes the system to become busy and the variable that represent the state of the button is set to true, which represents that the button is pressed.

11.3.3.2 Rules for Operation of Points

The rule patterns for operation of points are described in section 7.2.

Even though these rules should be generated from the circuit diagrams, they can also be generated from the interlocking plan.

The rules are generated as part of the transition generator and are mostly trivial. For each points section in the station layout diagram, the four rules are generated. The difficult part is figuring out which routes covers the points. This is done by searching for each points section in the track section part of each route in the train route table. If the track relay for the points section should be drawn before the route can be locked, then the points section is indeed part of the route.

A total of four transition rules are needed for each points section. The overloaded functions `genAllPointsOpRules` collectively calls `genPointsOpRules` for each points section on the station.

value -- 7.2 Points Behaviour

```

genAllPointsOpRules : ObjectRelayAssociations ×
  InterlockingPlan → TransitionRule-set
genAllPointsOpRules(ora, ip) ≡
  let d = sld(ip),
    allPoints = allPoints(d)
  in genAllPointsOpRules(allPoints, ora, ip)
end,

genAllPointsOpRules :
  TrackId-set × ObjectRelayAssociations ×
  InterlockingPlan → TransitionRule-set
genAllPointsOpRules(allPoints, ora, ip) ≡
  if allPoints = {} then {} else
    let p = hd allPoints,
      allPoints' = allPoints \ {p}
    in genPointsOpRules(p, ora, ip) ∪
      genAllPointsOpRules(allPoints', ora, ip) -- Recursive Call
    end
  end
end

```

Each of the four transition rules are then generated by `genPointsOpRules`.

value

```

genPointsOpRules :
  TrackId × ObjectRelayAssociations ×
  InterlockingPlan → TransitionRule-set
genPointsOpRules(pId, ora, ip) ≡
  let d = sld(ip),
      trt = trt(ip),
      rlrs = getRouteLockingRelaysCoveringP(pId, ora, trt) in {

    -- From plus to intermediate
    mk_TransitionRule(
      "plusToIntermediate" ^ pId,
      genPointsOpGuard(pId, plus, rlrs, ora, d),
      genPointsOpUpd(pId, plus, false, ora, d) ),

    -- From intermediate to plus
    mk_TransitionRule(
      "intermediateToPlus" ^ pId,
      genPointsOpGuard(pId, rlrs, ora, d),
      genPointsOpUpd(pId, plus, true, ora, d) ),

    -- From minus to intermediate
    mk_TransitionRule(
      "minusToIntermediate" ^ pId,
      genPointsOpGuard(pId, minus, rlrs, ora, d),
      genPointsOpUpd(pId, minus, false, ora, d) ),

    -- From intermediate to minus
    mk_TransitionRule(
      "intermediateToMinus" ^ pId,
      genPointsOpGuard(pId, rlrs, ora, d),
      genPointsOpUpd(pId, minus, true, ora, d) )
  }
end

```

The following explains the remaining functions used to generate the transitions for the points operation.

genPointsOpGuard Are two overloaded functions, which handle the cases of moving from one of the two locked positions to intermediate and the other way, respectively. They call `getRouteLockingRelaysCoveringP` to get the route locking relays that covers the points section in question.

getRouteLockingRelaysCoveringP A wrapper function that finds all the routes from the train route table and calls `getRLRsCoveringP`. Then `getRLRsCoveringP` recursively runs through the routes and calls `getRLRCoveringP`. Finally, if the given points section is covered by the given route, then the related route locking relay is returned by `getRLRCoveringP`.

genPointsOpUpdate Returns a set of assignments by updating the idle and updating the points relay to either being locked into a position or intermediate depending on which case.

convRelayIdsToLiterals Converts a set of relay ids into literals to be used in a boolean expression.

11.4 Assertion Generator

This section introduces the assertion generator and describes important parts in the specification. The patterns introduced in chapter 9 about inter-model consistency, and chapter 10 about safety properties described the assertions in general form. This section will use these patterns and apply them as guidelines to implement the assertion generator.

The assertion generator handles the generation of the safety properties and inter-model consistency conditions. The safety properties consists of no collision and no derailling assertions and other requirements derived from the train route table.

The inter-model consistency consists of conditions that ensure the interaction between the train movement model and the internal relay system model behaves as expected.

Thus, the assertion generator is split into three parts, one for inter-model consistency, one for safety conditions assertions and one for the assertions derived from the train route table.

Inter-model consistency consist of the following conditions each with reference back to the patterns mentioned earlier.

- Track Occupation (Section 9.2).

- Track Free (Section 9.3).
- Train Direction (Section 9.4).
- Points Configuration (Section 9.5).
- Train Connection (Section 9.6).

The basic safety properties consists of the following conditions each with a reference back to the patterns mentioned earlier.

- No collision (Section 10.2).
- No derailling (Section 10.3).

And lastly, the safety properties derived from the train route table consists of the following conditions each with a reference back to the patterns mentioned earlier.

- Points Position (Section 10.4).
- Signal (Section 10.5).
- Signal Release (Section 10.6).
- Conflicting Routes (Section 10.7).
- Train Route Release (Section 10.8).

The following sections will present a description of how a single assertion is generated by introducing some general rules and then an overview of how the assertions are generated. Finally an example from our specification showing how the signal release assertion is generated.

11.4.1 Generating a Single Assertion

RSL-SAL assertions have the following general form:

[assertionName] TransitionSystem \vdash TemporalLogic

where `assertionName` is a unique identifier for the assertion. The `TransitionSystem` is the name of the transition system and `TemporalLogic` is the LTL expression checking the validity of the model/transition system.

The abstract syntax of this is given by the RSL Assertion type. This type was described in section 4.4.4.

In the example below the *generateSomeAssertion* function accepts two parameters. This is simply to show the structure, where the number of parameters typically range from two to five.

Generating an assertion has the following general form:

```
generateSomeAssertion : ... × ... → T.Assertion
generateSomeAssertion(..., ...) ≡
    ("some_assertion_name" ^ toString(...),
      TransitionSystem,
      genSomeFormula(..., ...))
    ),
```

The name is made unique by concatenating an indicative identifier to the name, where we usually use a track section id, route id or signal id. The *TransitionSystem* is simply a variable, that contains the name of the current transition system used in all the generated assertions, namely *InterlockingSystem*.

Finally, the *genSomeFormula* function is invoked to generate the LTL expressions. In the simplest form all the assertions consist of a boolean expression implying another boolean expression. The inter-model consistency conditions and the safety properties described in chapter 9 and chapter 10 all requires that the expressions must hold true in all states. Thus, applying *globally* on the general form.

By invoking the function *genSomeFormula* it returns a LTLformula defined in section 4.4.4.

Hence, the generated LTLformula has the following general form:

```
genSomeFormula : ... × ... → LTLformula
genSomeFormula(..., ...) ≡
    g(impl(...,...))
```

This would generate the following LTL formula:

$$G(\dots \Rightarrow \dots)$$

11.4.2 Specifying the Assertion Generator

AssertionGenerator.rsl supply the function generateAssertions, which makes calls to each of the subgenerators.

value

```

generateAssertions :
  ObjectRelayAssociations × InterlockingPlan ×
  TrainMovementAssociations → Assertion-set
generateAssertions(ora, ip, tma) ≡
  genConsistencyAssertions(ora, ip, tma) ∪
  genSafetyAssertions(ora, ip, tma) ∪
  genTrtAssertions(ora, ip, tma)
pre isWfInterlockingPlan(ip) ∧
  isWfTrainMovementAssociations(sld(ip), tma) ∧
  isWfObjectRelayAssociations(ora, ip)

```

11.4.2.1 Consistency Assertion Generator

The inter-model consistency assertions are generated by the consistency assertion generator. The more general assertion generator call the genConsistencyAssertions method of the consistency assertion generator.

value

```

genConsistencyAssertions :
  ObjectRelayAssociations × InterlockingPlan ×
  TrainMovementAssociations → Assertion-set
genConsistencyAssertions(ora, ip, tma) ≡
  -- 9.2 Track Occupation
  generateOccupationAssertions(ora, ip, tma) ∪
  -- 9.3 Track Free
  generateFreeAssertions(ora, ip, tma) ∪
  -- 9.4 Train Direction
  generateDirectionAssertions(ip, tma) ∪
  -- 9.5 Points Configuration

```

```

generatePointsConfigurationAssertions(ora, ip)  $\cup$ 
-- 9.6 Train Connection
generateConnectionAssertions(ip, tma)
pre isWfInterlockingPlan(ip)  $\wedge$ 
    isWfTrainMovementAssociations(sld(ip), tma)  $\wedge$ 
    isWfObjectRelayAssociations(ora, ip)

```

Consider the track occupation consistency requirement presented in section 9.2. In the following, the specification of the generator for the related assertions will be explained.

```

value -- 9.2 Track Occupation
generateOccupationAssertions : ObjectRelayAssociations  $\times$ 
    InterlockingPlan  $\times$  TrainMovementAssociations  $\rightarrow$  Assertion-set
generateOccupationAssertions(ora, ip, tma)  $\equiv$ 
    let d = sld(ip),
        sections = allLinears(d)  $\cup$  allPoints(d) -- NB: No line pieces!
    in generateOccupationAssertions(sections, ora, tma)
end

```

Track Occupation requires that there must be consistency, for each track section, between the track relay variables and the track counters. Therefore the set of all track sections is given to generateOccupationAssertions.

```

value
generateOccupationAssertions :
    TrackId-set  $\times$  ObjectRelayAssociations  $\times$ 
        TrainMovementAssociations  $\rightarrow$  Assertion-set
generateOccupationAssertions(sections, ora, tma)  $\equiv$ 
    if sections = {} then {} else
        let s = hd sections,
            sections' = sections  $\setminus$  {s}
        in {generateOccupationAssertion1(s, ora, tma),
            generateOccupationAssertion2(s, ora, tma)}  $\cup$ 
            generateOccupationAssertions(sections', ora, tma) -- Recursive
        end
    end

```

The function generateOccupationAssertions runs through all the track sections recursively and calls the two functions which are responsible for generating the actual assertion for a given track section.

value

```

generateOccupationAssertion1 :
  TrackId × ObjectRelayAssociations ×
  TrainMovementAssociations → Assertion
generateOccupationAssertion1(s, ora, tma) ≡
  ("occ_" ^ s,
  genOccuFormula1(s, ora, tma)),

genOccuFormula1 :
  TrackId × ObjectRelayAssociations ×
  TrainMovementAssociations → LTLformula
genOccuFormula1(s, ora, tma) ≡
  g(impl(b(greaterthan(
    literal(getCounterVar(s, tma)),
    literal("0"))),
    neg(ltrl(trackRelayAssoc(ora)(s))))),

```

generateOccupationAssertion2 is very similar to generateOccupationAssertion1. However, it generates the other direction of the bi-implication.

value

```

generateOccupationAssertion2 :
  TrackId × ObjectRelayAssociations ×
  TrainMovementAssociations → Assertion
generateOccupationAssertion2(s, ora, tma) ≡
  ( "occ_" ^ s ^ "_",
  genOccuFormula2(s, ora, tma) ),

genOccuFormula2 :
  TrackId × ObjectRelayAssociations ×
  TrainMovementAssociations → LTLformula
genOccuFormula2(s, ora, tma) ≡
  g(
    impl(neg(ltrl(trackRelayAssoc(ora)(s))),
      b(greaterthan(literal(getCounterVar(s, tma)),
        literal("0")))))

```

The functions for the other consistency requirements are similar.

11.4.2.2 Safety Generator

This subgenerator is responsible for generating the assertions for the basic safety requirements, i.e. no collision and no derailling.

```

value
  genSafetyAssertions :
    ObjectRelayAssociations × InterlockingPlan ×
      TrainMovementAssociations → Assertion-set
  genSafetyAssertions(ora, ip, tma) ≡
    -- 10.2 No Collision
    genNoCollisionAssertions(ip, tma) ∪
    -- 10.3 No Derailling
    genNoDeraillingAssertions(ora, ip, tma)
  pre isWfInterlockingPlan(ip) ∧
    isWfTrainMovementAssociations(sld(ip), tma) ∧
    isWfObjectRelayAssociations(ora, ip)

```

The no collision assertions are generated by the function `genNoCollisionAssertions`, by recursively running through each track section on the station.

```

value -- 10.2 No Collision
  genNoCollisionAssertions : InterlockingPlan ×
    TrainMovementAssociations → Assertion-set
  genNoCollisionAssertions(ip, tma) ≡
    let d = sld(ip),
      sections = allLinears(d) ∪ allPoints(d) -- NB: No line pieces!
    in generateNoCollisionAssertions(sections, tma)
  end,

  generateNoCollisionAssertions : TrackId-set ×
    TrainMovementAssociations → Assertion-set
  generateNoCollisionAssertions(sections, tma) ≡
    if sections = {} then {} else
      let s = hd sections,
        sections' = sections \ {s}
      in {generateNoCollisionAssertion(s, tma)} ∪
        generateNoCollisionAssertions(sections', tma) -- Recursive call
    end
  end

```

The pattern described in sections 10.2 and 10.3 is applied to each track section on the station.

value

```

generateNoCollisionAssertion : TrackId ×
  TrainMovementAssociations → Assertion
generateNoCollisionAssertion(s, tma) ≡
  ("no_collision_" ^ s,
   genNoColFormula(s, tma) ),

genNoColFormula : TrackId ×
  TrainMovementAssociations → LTLformula
genNoColFormula(s, tma) ≡
  g(b(lessthan(literal(getCounterVar(s, tma)), literal("2"))))

```

11.4.2.3 TRT Generator

The trt generator creates the assertions of the requirements derived from the train route table. Each requirement, as described in section 10.4 through 10.8.

value

```

genTrtAssertions :
  ObjectRelayAssociations × InterlockingPlan ×
  TrainMovementAssociations → Assertion-set
genTrtAssertions(ora, ip, tma) ≡
  -- 10.4 Points Position
  genPointsPositionAssertions(ora, trt(ip)) ∪
  -- 10.5 Signal
  genSignalAssertions(ora, ip) ∪
  -- 10.6 Signal Release
  genSignalReleaseAssertions(ora, ip) ∪
  -- 10.7 Conflicting Routes
  genConflictingRoutesAssertions(ora, ip) ∪
  -- 10.8 Train Route Release
  genTrainRouteReleaseAssertions(ora, ip)
pre isWfInterlockingPlan(ip) ∧
  isWfTrainMovementAssociations(sld(ip), tma) ∧
  isWfObjectRelayAssociations(ora, ip)

```

The implementation of each requirement is very similar to what has already been shown in the preceeding section and will therefore not be discussed further here. Refer to the source code for the entire specification.

11.5 Output

The Stenstrup file instantiates `InterlockingPlan` and `ObjectRelayAssociations` for Stenstrup station. By executing the test case within the Stenstrup file it will invoke the generator, asking it to generate the transition system for Stenstrup and the unparser to print it in a form that RSL-SAL accepts.

11.5.1 Unparser

The unparser takes a transition system and prints a textual representation of it, using the syntax of RSL-SAL.

A transition system in RSL-SAL looks as follows:

```
transition_system [name]
local
[variable_decl]
in
[transition_decl]
end

ltl_assertions
[assertions_decl]
```

The unparser creates this structure and calls the overloaded method `print` on each component of the transition system, such that they are inserted at the right place.

```
value
  print : TransitionSystem → Text
  print(ts) ≡
    let name = name(ts),
        state = state(ts),
```

```

transitionRules = transitionRules(ts),
assertions = assertions(ts)
in "\ntransition_system\n\n[" ^ name ^ "]\n\nlocal" ^
  print(state) ^ "\n\nin\n" ^
  print(transitionRules) ^ "\n\nend\n\nltl_assertion\n" ^
  print(name, assertions) ^ "\n"
end

```

11.5.1.1 State

The state is unparsed by recursively running through all the variables in the state space. Each variable is written on its own line and they are separated by commas.

value

```

print : Var-set → Text
print(vars) ≡
  if vars = {} then "" else
    let var = hd vars, vars' = vars \ {var}
    in "\n" ^ print(var) ^
      if vars' = {} then ""
      else ", " ^ print(vars')
    end
  end
end
end

```

The identifier of the variable is written and then the type definition and initial assignment is unparsed.

value

```

print : Var → Text
print(var) ≡
  let id = id(var),
    val = val(var)
  in id ^ " : " ^ print(val)
end

```

The unparsed support the types boolean and natural numbers.


```

value
  print : Val → Text
  print(val) ≡
    case val of
      mk_BoolVal(b) → "Bool := " ^ toString(b),
      mk_NatVal(n)  → "Nat  := " ^ toString(n)
    end

```

11.5.1.2 Transitions

The transition rules are written out using the same method as the variables. However, they are separated by the choice operator, [=].

```

value
  print : TransitionRule-set → Text
  print(trs) ≡
    if trs = {} then ""
    else let tr = hd trs, trs' = trs \ {tr}
      in "\n" ^ print(tr) ^
        if trs' = {} then ""
        else "\n[=" ^ print(trs')
      end
    end
  end

```

The name can be completely unparsed at this point, however both the guard and update needs further treatment.

```

value
  print : TransitionRule → Text
  print(tr) ≡
    "[" ^ name(tr) ^ "]" ^ "\n" ^
    print(guard(tr)) ^ " ==>\n" ^
    print(assignments(tr))

```

Recall that the guard is a boolean expression and is unparsed in a similar fashion as the update and will therefore not be shown. The update consists of assignments separated by commas.

```

value
  print : MultipleAssignment → Text
  print(assignments) ≡
    if assignments = {} then "" else
      let assignment = hd assignments,
        assignments' = assignments \ {assignment}
      in print(assignment) ^
        if assignments' = {} then ""
        else ", " ^ print(assignments')
      end
    end
end

```

Each different type of assignment is treated independently.

```

value
  print : Assignment → Text
  print(assignment) ≡
    let id = id(assignment)
    in case assign(assignment) of
      BoolAssign(b) → id ^ ", " = " ^ toString(b),
      NatAssign(n)  → id ^ ", " = " ^ toString(n),
      AddAssign(a)  → id ^ ", " = " ^ id ^ " + " ^ toString(a),
      SubAssign(s)  → id ^ ", " = " ^ id ^ " - " ^ toString(s)
    end
end

```

11.5.1.3 Assertions

Assertions are written under the *ltl_assertions* tag, where each assertion gets its own line separated by comma.

```

value
  print : Text × Assertion-set → Text
  print(tsName, asserts) ≡
    if asserts = {} then "" else
      let assert = hd asserts, asserts' = asserts \ {assert}
      in "\n" ^ print(tsName, assert) ^
        if asserts' = {} then ""
    end

```

```

        else "\n" ^ print(tsName, asserts')
      end
    end
  end
end

```

The LTL formula is unparsed in a similar manner as the boolean expression (not shown here).

```

value
  print : Text × (Text × LTLformula) → Text
  print(tsName, (nm, ltl)) ≡
    "[" ^ nm ^ "]" ^ tsName ^ " |-\n" ^ print(ltl)

```


The test strategy of the implementation is to create scenarios of invalid and valid data models and associations. The expected invalid scenarios should then be caught by the well-formedness expressions defined within each of the data models and associations, as well as the expected valid scenarios should satisfy the requirements.

In the following chapter a thorough test of the station layout will be conducted in section 12.1. The tests of train route table, transition system, train movement associations and object relay association uses the same test strategy as the station layout. These tests can be found in appendix E.5.2, E.5.3, E.5.5, E.5.4, respectively.

All of the tests can be found in appendix E.5, and all the output from each test can be found in appendix F.3.

12.1 Station Layout

The test strategy for the station layout diagram is to create a simple station consisting of two linear track sections 02 and 03, and a points sections 01, where

Test case name	Description
isWfIdentifiers_fail	id 01 used both as a linear section and a points section
isWfIdentifiers_succeed	distinct ids used in linear, signal and point

Table 12.1: Test of well-formedness of identifiers

each linear track section is connected to the points section at the branch side, and a *line* connected to the stem, as illustrated on figure 12.1. Additionally, the points section is covered by two signals *A* and *B* at the branch side. Using the simple station allows us to easily modify it creating invalid station layouts.

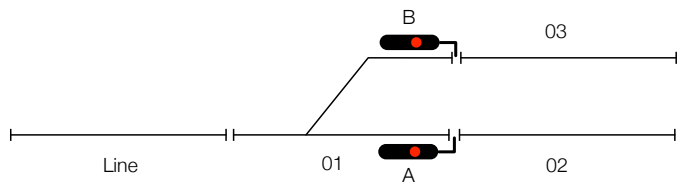


Figure 12.1: A simple station for testing the station layout diagram well-formedness.

Each well-formedness requirement defined in the station layout is to be tested. The tables from 12.1–12.7 are divided in groups each listed with names of well-formedness expressions followed by *fail* or *success*. *Fail* indicates that it is expected to fail, while *success* indicates a scenario that satisfies the requirements by the given well-formedness expression. Each test is attached with a description of the tested scenario.

12.2 Results

Each test case shown for the station layout diagram using the simple station passed all tests. All the invalid input failed as expected, and the valid input succeeded as expected.

All the tested output can be found in appendix F.3, where each test case is prefixed with the name of the well-formedness expression followed by either fail or succeed. The result is printed to the right of the name, where true denotes the test was as expected, and false would imply a failed test.

Test case name	Description
neighbours_OnlySections_fail	id which is not a section id
neighbours_OnlySections_succeed	id which is a section id
neighbours_EverySection_fail	a section without any neighbour
neighbours_EverySection_succeed	all sections has a neighbour, 01 is connected to line
neighbours_Irreflexive_linear_fail	a linear section neighbour with itself
neighbours_Irreflexive_points_fail	a points section neighbour with itself
neighbours_Antisymmetrical_fail	neighbouring sections are inserted as double entry
neighbours_Antisymmetrical_succeed	neighbouring sections are inserted not as double entry

Table 12.2: Test well-formedness of neighbours

Test case name	Description
linears_1or2Neighbours_fail_1	linear section 03 has no neighbours
linears_1or2Neighbours_fail_2	linear section 02 has 3 neighbours
linears_1or2Neighbours_succeed_1	linear sections has 1 neighbour each
linears_1or2Neighbours_succeed_2	linear sections has 2 neighbour each

Table 12.3: Test well-formedness of linear track sections

Test case name	Description
points_3Neighbours_fail_1	point section 01 has 2 neighbours
points_3Neighbours_fail_2	point section 01 has 4 neighbours
points_3Neighbours_succeed	point section 01 has 3 neighbours

Table 12.4: Test well-formedness of points sections

Test case name	Description
branchNeighbours_areNeighbours_fail	the neighbours in the branch neighbours relation does not match the neighbours relation
branchNeighbours_areNeighbours_succeed	the neighbours in the branch neighbours relation match the neighbours relation
branchNeighbours_FirstIsPoints_fail	The first parameter in the tuple is a linear section
branchNeighbours_FirstIsPoints_succeed	The first parameter in the tuple is a points section
branchNeighbours_everyPoints_fail	Not every points defined in allPoints are used in branchNeighbours
branchNeighbours_everyPoints_succeed	every points defined in allPoints and are used in branchNeighbours

Table 12.5: Test well-formedness of branch neighbours

Test case name	Description
signalLocation_UsesExistingSignals_fail	signal location does not use existing signals from allsignals
signalLocation_UsesExistingSignals_succeed	signal location use existing signals from allsignals
signalsArePlacedAtValtIdNeighbours_fail	signals are placed at invalid locations (between sections that does not exist)
signalsArePlacedAtValtIdNeighbours_succeed	signals are placed at valid locations (between existing sections)

Table 12.6: Test well-formedness of signals

Test case name	Description
pointMachines_areSymmetrical_fail1	inconsistency between pointMachineTrack and trackPointMachine
pointMachines_areSymmetrical_fail2	inconsistency between pointMachineTrack and trackPointMachine
pointMachines_areSymmetrical_fail3	inconsistency between pointMachineTrack and trackPointMachine
pointMachines_areSymmetrical_succeed	consistency between pointMachineTrack and trackPointMachine
pointMachines_ExistingTrackSections_fail	uses a non-existing points section
pointMachines_ExistingTrackSections_succeed	uses an existing points section

Table 12.7: Test well-formedness of point machines

CHAPTER 13

Case Study - Stenstrup

This chapter shows the entire process of verifying the interlocking system of Stenstrup station, from filling in the data representation to model checking the generated model of the interlocking system and environment.

This case study follows the user guide given in [appendix C](#).

Refer to [appendix B](#) for the documentation of Stenstrup and [appendix E.4.5](#) for the specification.

Section [13.1](#) describes, using Stenstrup station, how it is constructed step by step.

Section [13.2](#) then shows how to generate the external events by using Stenstrup. Some of the generated output is then compared with variables and transition rules for the external behaviour.

13.1 Defining Stenstrup

Beginning from the provided template (see [appendix E.4.4](#)), every entry is filled out as described in the following sections.

13.1.1 Interlocking Plan

An interlocking plan consists of a *station layout diagram* and a *train route table*.

value

```
stenstrupIP : InterlockingPlan =
  mk_InterlockingPlan(
    stenstrupSLD,
    stenstrupTRT
  )
```

13.1.2 Station Layout Diagram

The station layout diagram of Stenstrup is defined as a well-formed station layout diagram. The content of each of the constituents is shown in the following sections.

value

```
stenstrupSLD : Diagram =
  mk_Diagram(
    "line",
    allLinears,
    allPoints,
    allSignals,
    neighbours,
    branchNeighbours,
    pointMachineTrack,
    trackPointMachine,
    signalLocations
  )
```

13.1.2.1 Physical Objects

The linear track sections A12, 02, 04 and B12 are stored in the set *allLinears*, the points sections 01 and 03 are stored in *allPoints* and the signals A, B, D, F, G and H are stored in *allSignals*.

value

```
allLinears : TrackId-set = {"A12", "02", "04", "B12"},
allPoints  : TrackId-set = {"01", "03"},
allSignals : SignalId-set = {"A", "B", "E", "F", "G", "H"}
```

13.1.2.2 Neighbour Relation

The pairs of physically connected track sections are stored in *neighbours*. Recall from section 4.2.4, that the line is part of this relation and that the relation is asymmetric.

value

```
neighbours : (TrackId × TrackId)-set =
  { ("line", "A12"), ("A12", "01"),
    ("01", "02"), ("01", "04"),
    ("02", "03"), ("03", "04"),
    ("03", "B12"), ("B12", "line") }
```

13.1.2.3 Points

As described in section 4.2.5, the neighbours relation provide insufficient information about the points sections, namely that it does not provide any information about which branch a points section and its branch side neighbours are connected to.

value

```
branchNeighbours : (TrackId × TrackId)  $\xrightarrow{m}$  Branch =
  [ ("01", "02")  $\mapsto$  plus,
    ("01", "04")  $\mapsto$  minus,
    ("03", "02")  $\mapsto$  plus,
    ("03", "04")  $\mapsto$  minus ]
```

Furthermore the point sections needs to be associated with the point machine that control them.

value

```
pointMachineTrack : PointMachineId  $\xrightarrow{m}$  TrackId =
  [ "01"  $\mapsto$  "01",
    "02"  $\mapsto$  "03" ],

trackPointMachine : TrackId  $\xrightarrow{m}$  PointMachineId =
  [ "01"  $\mapsto$  "01",
    "03"  $\mapsto$  "02" ]
```

13.1.2.4 Signals

Finishing out the station layout diagram of Stenstrup, is defining the signals locations.

value

```
signalLocations : (TrackId  $\times$  TrackId)  $\xrightarrow{m}$  SignalId =
  [ ("line", "A12")  $\mapsto$  "A", ("line", "B12")  $\mapsto$  "B",
    ("02", "01")  $\mapsto$  "E", ("04", "01")  $\mapsto$  "F",
    ("02", "03")  $\mapsto$  "G", ("04", "03")  $\mapsto$  "H" ]
```

13.1.1.3 Train Route Table

The train route table of Stenstrup (fig. 2.15), shows that Stenstrup has eight train routes. The data for each route is stored in a DataRow structure. Only the content of two of the routes are shown here. Please refer to the full listing in appendix E.4.5 for the remaining routes.

value

```
stenstrupTRT : TrainRouteTable =
  [ "2"  $\mapsto$  dataRow2, "3"  $\mapsto$  dataRow3,
    "5"  $\mapsto$  dataRow5, "6"  $\mapsto$  dataRow6,
    "7"  $\mapsto$  dataRow7, "8"  $\mapsto$  dataRow8,
    "9"  $\mapsto$  dataRow9, "10"  $\mapsto$  dataRow10 ]
```

Recall the specification of a DataRow from section 4.3.2. The following shows a DataRow object instantiated with the data from the row of train route 2.

value

```

dataRow2 : DataRow =
  mk_DataRow(
    {"03", "B12"},
    [ "A" ↦ gr, "B" ↦ arbitrary, "E" ↦ arbitrary,
      "F" ↦ re, "G" ↦ re, "H" ↦ arbitrary ],
    [ "01" ↦ plus, "02" ↦ plus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ( "A", "A12" ),
    ( ("01", dropped), ("02", drawn)),
    ( ("02", dropped), ("01", drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
      "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ true ]
  )

```

The data for train route 10 is shown below.

value

```

dataRow10 : DataRow =
  mk_DataRow(
    {},
    [ "A" ↦ arbitrary, "B" ↦ arbitrary, "E" ↦ arbitrary,
      "F" ↦ arbitrary, "G" ↦ re, "H" ↦ gr ],
    [ "01" ↦ arbitrary, "02" ↦ minus ],
    [ "A12" ↦ false, "01" ↦ false, "02" ↦ false,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ( "H", "03" ),
    ( ("03", dropped), ("B12", drawn)),
    ( ("B12", dropped), ("03", drawn)) ),
    [ "2" ↦ true, "3" ↦ false, "5" ↦ true, "6" ↦ true,
      "7" ↦ false, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
  )

```

13.1.4 Object Relay Associations

The object relay associations from section 8.1 is a collection of data that associates relays of the interlocking system, with the physical objects of the station and the route ids.

value

```
stenstrupAssocs : ObjectRelayAssociations =
  mk_ObjectRelayAssociations(
    routeRelayAssoc,
    trackRelayAssoc,
    pointRelayAssoc,
    signalRelayAssoc
  )
```

Table 5.5 shows the relation between train routes and train route locking relays, i.e. which relay must be dropped for a given train route to be locked.

value

```
routeRelayAssoc : TrainRouteId  $\overrightarrow{m}$  RelayId =
  [ "2"  $\mapsto$  "ia", "3"  $\mapsto$  "ia",
    "5"  $\mapsto$  "ib", "6"  $\mapsto$  "ib",
    "7"  $\mapsto$  "ua", "8"  $\mapsto$  "ua",
    "9"  $\mapsto$  "ub", "10"  $\mapsto$  "ub" ]
```

Table 5.2 shows which track sections are associated with which track relay variables, i.e. the names of the variables that are false when the associated track section is occupied.

value

```
trackRelayAssoc : TrackId  $\overrightarrow{m}$  RelayId =
  [ "A12"  $\mapsto$  "a12", "01"  $\mapsto$  "t01", "02"  $\mapsto$  "t02",
    "B12"  $\mapsto$  "b12", "03"  $\mapsto$  "t03", "04"  $\mapsto$  "t04" ]
```

Table 5.3 shows the relation between point machines and the point relay variables for the points sections.

value

```
pointRelayAssoc : PointMachineId  $\times$  Branch  $\overrightarrow{m}$  RelayId =
  [ ("01", plus)  $\mapsto$  "plus01", ("01", minus)  $\mapsto$  "minus01",
    ("02", plus)  $\mapsto$  "plus02", ("02", minus)  $\mapsto$  "minus02" ]
```

Table 5.4 shows the relations between the signals and the signal relay variables to show stop and proceed.

value

```
signalRelayAssoc : SignalId × Aspect  $\xrightarrow{\overline{m}}$  RelayId =
  [ ("A", gr)  $\mapsto$  "aGreen", ("A", re)  $\mapsto$  "aRed",
    ("B", gr)  $\mapsto$  "bGreen", ("B", re)  $\mapsto$  "bRed",
    ("E", gr)  $\mapsto$  "eGreen", ("E", re)  $\mapsto$  "eRed",
    ("F", gr)  $\mapsto$  "fGreen", ("F", re)  $\mapsto$  "fRed",
    ("G", gr)  $\mapsto$  "gGreen", ("G", re)  $\mapsto$  "gRed",
    ("H", gr)  $\mapsto$  "hGreen", ("H", re)  $\mapsto$  "hRed" ]
```

13.1.5 Buttons

The buttons are stored in the set *allButtons*. Table 5.5 shows the button identifiers.

value

```
allButtons : Text-set = {"b00406", "b03106", "b00606", "b03306"}
```

13.2 Generating Model of External Events

The model of the external behaviour is generated by first translating the RSL specification of the tool to SML and then calling the generators generate function.

test_case

```
[generate.transitionSystem]
  print(generate(
    "InterlockingSystem",
    stenstrupAssocs,
    stenstrupIP,
    allButtons))
```

13.2.1 Result

The generator creates a transition system containing 24 variables, 47 transitions and 102 assertions. The entire unparsed output is available in appendix F.1.

Lets have a look at some of the output in order to convince ourselves that it is correct.

13.2.1.1 Variables

The generated variables are shown in tables 13.1, 13.2 and 13.3. Comparing the generated output for Stenstrup station on the right column with the left column for the model of train movement presented in section 6.2, we see that it is exactly as expected.

Model	Generated
cnt_04 : Nat := 0,	cnt_04 : Nat := 0,
cnt_02 : Nat := 0,	cnt_02 : Nat := 0,
cnt_B12 : Nat := 0,	cnt_B12 : Nat := 0,
cnt_A12 : Nat := 0,	cnt_A12 : Nat := 0,
cnt_03 : Nat := 0,	cnt_03 : Nat := 0,
cnt_01 : Nat := 0	cnt_01 : Nat := 0

Table 13.1: Generated output of counter variables compared to the model of train movement

Model	Generated
con_01_04 : Bool := false,	con_01_04 : Bool := false,
con_02_03 : Bool := false,	con_02_03 : Bool := false,
con_01_02 : Bool := false,	con_01_02 : Bool := false,
con_03_04 : Bool := false,	con_03_04 : Bool := false,
con_A12_01 : Bool := false,	con_A12_01 : Bool := false,
con_03_B12 : Bool := false	con_03_B12 : Bool := false

Table 13.2: Generated output of connection variables compared to the model of train movement

13.2.1.2 Transitions

Comparing the transition rules from section 7.1 with the generated transition rules for button behaviour we get the expected output.

```
[push_b03106] idle ==>
```

Model	Generated
fwd_04 : Bool := false, bwd_04 : Bool := false, fwd_02 : Bool := false, bwd_02 : Bool := false, fwd_B12 : Bool := false, bwd_B12 : Bool := false, fwd_A12 : Bool := false, bwd_A12 : Bool := false, fwd_03 : Bool := false, bwd_03 : Bool := false, fwd_01 : Bool := false, bwd_01 : Bool := false	fwd_04 : Bool := false, bwd_04 : Bool := false, fwd_02 : Bool := false, bwd_02 : Bool := false, fwd_B12 : Bool := false, bwd_B12 : Bool := false, fwd_A12 : Bool := false, bwd_A12 : Bool := false, fwd_03 : Bool := false, bwd_03 : Bool := false, fwd_01 : Bool := false, bwd_01 : Bool := false

Table 13.3: Generated output of direction variables compared to the model of train movement

```

b03106' = true, idle' = false
[=]
[push_b00606] idle ==>
b00606' = true, idle' = false
[=]
[push_b00406] idle ==>
b00406' = true, idle' = false
[=]
[push_b03306] idle ==>
b03306' = true, idle' = false

```

Again, comparing the transition rules presented in section 7.2 with the generated output we get the expected output.

```

[point01FromPlus] idle /\ ib /\ plus01 /\ ua /\ t01 /\ ia ==>
plus01' = false, idle' = false
[=]
[point01ToPlus] idle /\ ib /\ ~(plus01) /\ ua /\ ~(minus01) /\
ia /\ t01 ==>
plus01' = true, idle' = false
[=]
[point01FromMinus] idle /\ ib /\ minus01 /\ ua /\ t01 /\ ia ==>
minus01' = false, idle' = false
[=]
[point01ToMinus] idle /\ ib /\ ~(plus01) /\ ua /\ ~(minus01) /\

```

```

ia /\ t01 ==>
minus01' = true, idle' = false

```

Comparing an entering rule and a leaving rule from section 6.3.1 and 6.3.3 with the generated output we get the expected result.

```

[enter_A12_from_line] idle /\ aGreen /\ ~(aRed) ==>
fwd_A12' = true, idle' = false, a12' = false,
cnt_A12' = cnt_A12 + 1
[=]
[leave_A12_to_line] bwd_A12 /\ idle /\ ~(con_A12_01) /\
cnt_A12 = 1 ==>
idle' = false, cnt_A12' = 0, a12' = true, bwd_A12' = false

```

13.2.1.3 Assertions

Finally, comparing one of the assertions presented in section 10.8 with the generated output below we get the expected result.

```

[train_route_release_2] InterlockingSystem |- G(ia /\ X(plus02 /\
plus01 /\ ~(ia) /\ F(ia)) => X(U(~(ia), ~(t01) /\ t02 /\ ~(ia)
/\ X(U(~(ia), ~(ia) /\ ~(t02) /\ t01))))))

```

13.3 Combining the Models

We now have the RSL-SAL model of the external system. This needs to be combined with the internal system into a single model, that can then be model checked.

The model of the internal system is acquired, as described in the user guide found in appendix C, by using the tool developed by Kjær and Le Bliguet [2].

The combined RSL-SAL model of the internal and external system can then be translated to SAL, and finally model checked. The results of model checking Stenstrup station will be presented in the next section.

13.4 Results Stenstrup

The result shows that the consistency conditions are satisfied. This means that the internal system responds according to the external events provided by the model of train movement. Since the safety properties (chapter 10) are satisfied as well, we can conclude that the model of the interlocking system ensures the safety of the modelled train movements (chapter 6).

The results of the model checker can be found in appendix F.2. All the inter-model consistency and safety property assertions were satisfied including the assertions provided by Kjær and Le Bliguet [2]

The model checking was conducted on the following hardware which took approximately 1.5 hour.

CPU Intel Core i5-2400 CPU @ 3.10 GHz, 6MB cache

RAM 8 GB 1333 MHz DDR3 non-ECC SDRAM

Operating System Xubuntu 11.04, Linux kernel 2.6.38-13-generic-pae

RSLTC Version 2.5-1

SAL Version 3.0

Conclusion

Banedanmark wanted to automate the process of validating relay interlocking systems. Some work had already been carried out, but part of the generation process was still to be done manually.

In this project we have specified data models of station layout diagrams, train route tables and the RSL-SAL transition system, and created well-formedness functions that essentially acts as data validators. Furthermore, we created associations between the physical station elements and the model of the internal system, and associations to allow the generation of a representation of trains.

We have, in RSL, specified a tool which is able to generate a behavioural model of train movements and the behaviour of operating points and buttons for Stenstrup station. We then formalised basic safety properties, identified safety properties from a train route table and specified a generator that can generate these properties automatically. Our model of train movement is compatible with Kjær and Le Bliguet's [2] model of the internal system and is indeed intended to be a substitute for the train movement model they suggested.

To verify the compatibility between the model of train movement and internal system we identified, formalised and generated inter-model consistency conditions to ensure that the external behaviours were observed by the internal system.

Thus, the tool is capable of generating a RSL-SAL transition system of the external events from station documentation using our data models, in addition to generating the safety and consistency assertions.

With our contribution Banedanmark now has a functioning prototype for automated validation of relay interlocking systems. With such a tool Banedanmark can validate their station documentation faster and eliminate the risk of human error. Furthermore, one can speculate that the proposed tool would verify more properties, compared to now.

Our model of train movements allows for trains of arbitrary length and uses implicitly defined trains, which allows for an arbitrary number of trains on the station. This model was chosen over a model using explicitly defined trains, as that model was not only limited to a certain number of trains, but also model checked slower, which could cause problems for larger systems.

Finally, we model checked a combined RSL-SAL transition system of the external events and the internal events. The result of our case study of Stenstrup station showed that all safety and consistency conditions were satisfied using our model of train movement. From this fact we concluded that the model of the interlocking system ensures the safety of the modelled train movements.

We are aware that the algorithm for assigning directions may not function properly if more than one track is connected to the line at each end. This could be solved by differentiating between the line sections at each end.

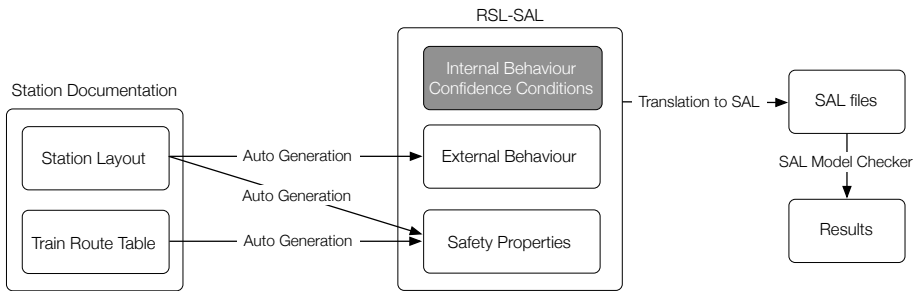


Figure 14.1: Auto generation of external behaviour and safety properties.

14.1 Future Work

In this section, suggestions will be made on how to proceed from here and how the developed tool and the tools used can be improved.

14.1.1 Directly Related

While the tool has been tested on Stenstrup, it is not known how well it work with another station using the DSB type 1954 relay interlocking system. The problem with testing this, is the amount of work needed to transform the physical documentation into the digital data structure. This is especially true considering the assumptions made in both this work and that of Kjær and Le Bliguet [2].

A graphical editor would help speed up this process. Unfortunately the output of the editor developed by Eriksen and Pedersen [7] is not compatible with tool set.

If the tool set should see use in industry, it would most likely have to be expanded with some of the parts that where not considered due to scope, e.g. support for level crossings.

Finally, some work would be required to fully automate the verification process. As of now, the model of the internal and the external system has to be combined by hand before the combined system can be given to the model checker. One could choose to expand the Java implementation of the generator of the internal system, using our specification of the external system. However, it would seem more natural to merge the specifications, make the specification of the internal system executable and combine it with ours at this level. By making the complete specification executable, the output of the complete system can be tested before implementation.

14.1.2 Indirectly Related

During this work, we found that it would be useful for RSL-SAL to support multiple models in a single file.

```
scheme SchemeName =  
  class
```

```
    transition_system [Internal]
    local
    ...
    in
    ...
    end

    transition_system [External]
    local
    ...
    in
    ...
    end
end
```

This would greatly improve the readability of the file, as the parts would be logically divided.

Even if SAL does not support interaction between two models, the RSL-SAL translator could combine them into a single SAL model. Any conflicting names could be solved by adding the system name to the conflicting variable or transition name.

Bibliography

- [1] RAISE Tools. Home page: <http://www.iist.unu.edu/newrh/III/3/1/page.html>.
- [2] Andreas Andersen Kjær and Marie Le Bliguet. Modelling Interlocking System for Railway Stations. Master's thesis, DTU, 2008.
- [3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [4] Banedanmark. Functional Safety Requirements Version 1.1 Issue 1.3. Internal Banedanmark Publication, September 2003.
- [5] Banedanmark. *Sikkerhedsreglement af 1975 (SR)*. Banedanmark, 2008. Available as PDF from <http://www.bane.dk/hentmedie.asp?filID=11350> (Danish).
- [6] Leonardo de Moura, Sam Owre, and Natarajan Shankar. The SAL Language Manual. Technical Report SRI-CSL-01-02, SRI International.
- [7] Louise Elmoose Eriksen and Boe Pedersen. Simulation of Relay Interlocking Systems. Bachelor thesis, DTU, 2007.
- [8] RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall, 1 edition, 1992.
- [9] Anne Elisabeth Haxthausen. Towards a Framework for Modelling and Verification of Relay Interlocking Systems. In *Modelling Development and Verification of Adaptive Systems*, pages 101–111, 2010.

- [10] Henrik W. Karlsson and Carsten S. Lundsten. *Introduktion til sikrings- og fjernstyringsanlæg*. Rambøll, 2006. (Danish).
- [11] Juan Ignacio Perna and Chris George. Model Checking RAISE Specifications. Technical Report 331, UNU-IIST, 2006.
- [12] Gregor Theeg and Sergej Vlasenko. *Railway Signalling & Interlocking*. DVV Media Group — Eurailpress, 1 edition, 2009.

APPENDIX A

Word List

The table below has a translation from english to danish of the words and terminology used throughout this thesis.

English	Danish
(Track) Section	Sporafsnit
(Set of) Points	Sporskifte
Branch	Forgrening
Relay	Relæ
Signal lamp relay	Lampekontrolrelæ
Point relay	Sporskiftekontrolrelæ
Interlocking System	Sikringsanlæg
Train route	Togvej
Train route table	Togvejstabel
Interlocking Plan	Sikringsplan
Shunting movement	Rangering
Open Line	Fri bane
Line block	Linieblok
Distant signal	Fremskudt signal
(Station) Entry signal	Indkørelsessignal
(Station) Pre-exit signal	Perronudkørelsessignal
(Station) Exit signal	Udkørelsessignal
Shunt signal	Signal for rangering
Station limit	Stationsgrænse
Signal Release	Stop Fald

APPENDIX B

Documentation of Stenstrup Station

Togveje		Signaler				Sporskifter			Sporisolationer				Ovk	Stop	Togvejsopl.		Gensidige spæringer	
nr		Spørforløb	a	b	A	B	E	F	G	H	I	J	K	L	Indl	Opt		
2	fra Odense	1 strækkn	$\frac{B}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	2	
3		2 strækkn	$\frac{B}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	3	
5	fra Svendborg	1 strækkn	$\frac{B}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	5	
6		2 strækkn	$\frac{B}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	$\frac{A}{2}$	6	
7	til Odense	1													$\frac{A}{2}$	$\frac{A}{2}$	7	
8		2													$\frac{A}{2}$	$\frac{A}{2}$	8	
9	til Svendborg	1													$\frac{A}{2}$	$\frac{A}{2}$	9	
10		2													$\frac{A}{2}$	$\frac{A}{2}$	10	

Figure B.2: Train Route Table for Stenstrup Station.

APPENDIX C

User Guide

The following section will describe how to use the tool.

C.1 Requirements

Ensure that the following software is installed and running correctly:

- rsltc version 2.5
- Standard ML of New Jersey v110.72
- SAL 3.0
- Java Runtime Environment 6 (1.6)
- Emacs 23

C.2 Stenstrup

Assuming everything is in the same folder, call the generator of the internal model.

```
java -jar InternalBehaviourGeneration.jar
```

It will ask for parameters, so fill in as below:

```
Please enter the path of the input XML file:
> stenstrup.xml
Please enter the path of the output RSL-SAL file:
> stenstrup.rsl
Please enter the name of the generated RSL-SAL scheme:
> stenstrup
```

This will generate the file `stenstrup.rsl` in the folder.

Open `stenstrup.rsl`, locate the transition *setIdle* and replace:

```
==> idle' = true
```

with

```
==> idle' = ~ (b00406 \/ b00606 \/ b03106 \/ b03306),
          b00406' = false,
          b00606' = false,
          b03106' = false,
          b03306' = false
```

Open `Stenstrup.rsl` (another file) in emacs and click RSL → Type check. This should return 0 errors and 0 warnings.

Now click RSL → SML → Translate to SML and run. Without closing the window that appears, hit RSL → SML → End SML run and save. Click yes.

This output shall now be combined with `stenstrup.rsl`. Copy anything between “local” and “in” from the output and place it after “local” in `stenstrup.rsl` (insert comma if needed).

Next copy everything between “in” and “end” from the output and place it after “in” in `stenstrup.rsl` (insert `[=]` if needed).

Finally copy everything after “ltl_assertion” from the output and paste it in after “ltl_assertion” in `stenstrup.rsl` and type check `stenstrup.rsl`. This should return no errors and no warnings.

Close the output buffer and `Stenstrup.rsl`, leaving just `stenstrup.rsl` open. Click RSL → SAL → Translate to SAL. This will generate a great number of files. Next click RSL → SAL → Run SAL well-formed checker. All three checks should be successful.

Finally click RSL → SAL → Run model checker → base. Hit enter. Expect the model to run for at least 1.5 hour.

APPENDIX D

Behavioural Model of Train Movements Using Explicitly Defined Trains

This solution explores explicit declaration of trains. A train occupies a number of connected track sections and has a direction. Here the tracks occupied by the train is collected in a set¹ and the direction is a pointer to the next section. Alternatively a up/down type mechanism could be employed.

```
transition_system [InterlockingSystem]
  local
    ...
    trainldir : TrackId := none,
    trainlloc : TrackId-set := {},
  in
    ...
  end
```

¹Using RSL-SAL, a set is translated into a boolean variable for each possible element in the set, which is true if the element is in the set. Not all RSL set operators are translatable, unfortunately, most noticeable **card**, which would actually be easy to implement, but that is an aside.

where `TrackId` is defined as follows

```
type TrackId ==  
  t_a12 | t_01 | t_02 | t_03 | t_04 | t_b12 | none
```

The initial state of the system is the normal state. Here no track sections are occupied and therefore, no trains are at the station. Hence the direction is initialised to `none` (could be set to either `a12` or `b12`, but is not, since we don't want to dictate its entrance.) and the location is the empty set to represent that the train is not at the station.

It is assumed that the trains respect the aspects of the signals. Therefore no train are supposed to enter the station before a green light at a station entry signal is given. Assuming a working and well-formed interlocking system, this only happens once a route has been locked. This require an external event, which we assumed has happened in the following. Every rule also require the system to be idle, so that the relays are in a correct position. As a consequence every rule introduced here will also set the system in the busy state, to allow the relays to response to the external events.

D.1 Enter

Lets look at the rules for entering the station at section `a12`.

```
[t1_enter_a12_from_line]  
idle ∧ aGreen ∧ train1loc = {} →  
  idle' = false,  
  a12' = false,  
  train1dir' = t_01,  
  train1loc' = train1loc ∪ {t_a12}
```

The guard require the station entry signal to show green, while the train cannot already be at the station. If these conditions are met, the train is allowed to enter `a12`. Therefore the relay `a12` is dropped, the track identifier is added to the location set and the direction is set to section `01`, since the train came from the line. This information is derived from the station layout.

Entering the station is a special case and therefore has rules that are not in the general case. Lets look at another special case, which happens to be the next in our scenario.

```
[t1_enter_01_in_plus_from_a12]
idle ∧ train1dir = t_01 ∧ t_a12 ∈ train1loc ∧ plus01 →
  idle' = false,
  t01' = false,
  train1dir' = t_02,
  train1loc' = train1loc ∪ {t_01}
```

In this situation the train enters 01 from a12. For this to occur, we need to have a train on a12, which is headed towards 01. Since the direction points to a specific section, a rule is needed for each possible state of the point. Here it is in plus. Therefore the next section will be 02. Had it been in minus, the next would be 04 and if the point is not locked, the train will derail, so the direction is set to none.

```
[t1_enter_01_in_neither]
idle ∧ train1dir = t_01 ∧ ∼ plus01 ∧ ∼ minus01 →
  idle' = false,
  t01' = false,
  train1dir' = none,
  train1loc' = train1loc ∪ {t_01}
```

The interlocking system should ensure that both the plus and minus relay cannot be down at the same time, eliminating the case where the point is locked in both plus and minus position, which also happens to be physically impossible.

Lets continue the example. If 01 was in plus, the next section would be 02. 02 is a regular track segment, but since the trains comes from a point, we again need to ensure that the point is in the correct position.

```
[t1_enter_02_from_01]
idle ∧ train1dir = t_02 ∧
t_01 ∈ train1loc ∧ plus01 →
  idle' = false,
  t02' = false,
  train1dir' = t_03,
  train1loc' = train1loc ∪ {t_02}
```

The next section is yet another point, but this time it is not entered from the stem. In addition there is a signal on 02, which we have to respect. As such the details have already been discussed.

D.2 Leave

Since the train is allowed to occupy multiple sections and we would like to be able to move the back end of the train as well, rules for leaving a section is introduced.

Imagine that the train from before has reached b12 and is leaving the station. We want to correctly move the back end of the train off the station. First we check that the train is leaving the station by checking that the direction is set to none. We should not be allowed to remove the train from this section unless it is the tail end. Therefore we require that the train is only on b12. If it was at other sections as well, we would not be looking at the tail end.

```
[t1.leave_b12_to_line]
idle ∧ train1dir = none ∧ train1loc = {t.b12} →
  idle' = false,
  b12' = true,
  train1loc' = train1loc \ {t.b12}
```

Lets say that the train is on 03 as well. Everything else being equal, the tail end of the train would now be on 03. Therefore we can remove it from 03 and then afterwards use `t1.leave_b12_to_line` to remove it entirely.

```
[t1.leave_03_to_b12]
idle ∧ train1dir = t.b12 ∧ train1loc = {t.03, t.b12} →
  idle' = false,
  t03' = true,
  train1loc' = train1loc \ {t.03}
```

One could follow the same method and keep adding requirements about the location of the train the further back along the track we look, but instead we'll introduce a more general rule.

Lets consider the train leaving 02 in the direction of 03 as an example to demonstrate the rule. We require that the section prior to 02, i.e. 01 is not in

the location set. This ensures that we do not split the train. In addition, it is required that the train is on both 02 and 03. The reason for this is that we cannot remove a train from a section where it is not located and we should not remove the entire train, just the tail end. If the train was positioned on just 02 and we did not require it to be on 03 as well, both the head and tail end would be at 02 and therefore we would remove the entire train. Obviously a train is not allowed to disappear at random locations.

```
[t1_leave_02_to_03]
idle ∧ ∼ (t_01 ∈ train1loc) ∧
{t_02, t_03} ⊆ train1loc ∧ train1dir ≠ t_01 →
  idle' = false,
  t02' = true,
  train1loc' = train1loc \ {t_02}
```

220 Behavioural Model of Train Movements Using Explicitly Defined Trains

APPENDIX E

Specifications

E.1 Data Models

E.1.1 Objects

context: InterlockingPlan
object IP : InterlockingPlan

context: StationLayout
object SL : StationLayout

context: TrainRouteTable
object TRT : TrainRouteTable

context: TransitionSystem
object TS : TransitionSystem

E.1.2 Interlocking Plan

```

context: T, TRT, SL
scheme InterlockingPlan =
  class
    type
      InterlockingPlan ::
        sld : SL.Diagram
        trt : TRT.TrainRouteTable

    value
      isWfInterlockingPlan : InterlockingPlan → Bool
      isWfInterlockingPlan(ip) ≡
        let d = sld(ip),
            trt = trt(ip)
        in SL.isWfDiagram(d) ∧ TRT.isWfTrainRouteTable(trt, d)
        end

  end

```

E.1.3 Station Layout Diagram

```

context: T
scheme StationLayout =
  class
    type
      TrackId = Text,
      PointMachineId = Text,
      SignalId = Text

    type /* Station Layout */
      Branch = {| ps : T.PointsState • ps ≠ T.arbitrary |},

      Diagram :: /* Station Layout Diagram */
        line : TrackId
        allLinears : TrackId-set
        allPoints : TrackId-set
        allSignals : SignalId-set
        neighbours : (TrackId × TrackId)-set
        branchNeighbours : (TrackId × TrackId)  $\nrightarrow$  Branch
        pointMachineTrack : PointMachineId  $\nrightarrow$  TrackId
        trackPointMachine : TrackId  $\nrightarrow$  PointMachineId
        signalLocations : (TrackId × TrackId)  $\nrightarrow$  SignalId

    value /* Observer functions */
      areNeighbours : TrackId × TrackId × Diagram → Bool
      areNeighbours(tId1, tId2, d) ≡
        (tId1, tId2) ∈ neighbours(d) ∨
        (tId2, tId1) ∈ neighbours(d),

```

```

isLine : TrackId × Diagram → Bool
isLine(tId, d) ≡ tId = line(d),

isPoints : TrackId × Diagram → Bool
isPoints(tId, d) ≡ tId ∈ allPoints(d),

isLinear : TrackId × Diagram → Bool
isLinear(tId, d) ≡ tId ∈ allLinears(d),

isSignal : SignalId × Diagram → Bool
isSignal(sId, d) ≡ sId ∈ allSignals(d),

isSection : TrackId × Diagram → Bool
isSection(tId, d) ≡ tId ∈ allSections(d),

hasSignal : TrackId × TrackId × Diagram → Bool
hasSignal(tId1, tId2, d) ≡
  (tId1, tId2) ∈ dom signalLocations(d),

isBranchNb : TrackId × TrackId × Diagram → Bool
isBranchNb(points, nb, d) ≡
  (points, nb) ∈ dom branchNeighbours(d)
pre isPoints(points, d),

isStemNb : TrackId × TrackId × Diagram → Bool
isStemNb(points, nb, d) ≡
  ~isBranchNb(points, nb, d) ∧ areNeighbours(points, nb, d)
pre isPoints(points, d)

value /* Aux functions */
getNeighboursOf : TrackId × Diagram → TrackId-set
getNeighboursOf(tId, d) ≡
  getNbs(tId, neighbours(d)),

getNbs : TrackId × (TrackId × TrackId)-set → TrackId-set
getNbs(tId, nbs) ≡
  if nbs = {} then {} else
    let nb = hd nbs, nbs' = nbs \ {nb}
    in getNbs(tId, nbs') ∪ getNb(tId, nb)
  end
end,

getNb : TrackId × (TrackId × TrackId) → TrackId-set
getNb(tId, (nb1, nb2)) ≡
  if tId = nb1 then {nb2} else
    if tId = nb2 then {nb1} else {} end
  end,

allSections : Diagram → TrackId-set
allSections(d) ≡
  allLinears(d) ∪ allPoints(d) ∪ {line(d)},

xor : Bool-set → Bool
xor(bools) ≡
  if bools = {} then false else

```

```

    let bool = hd bools, bools' = bools \ {bool}
    in xor(bool, xor(bools'))
  end
end,

xor : Bool × Bool → Bool
xor(p, q) ≡ (p ∧ ~q) ∨ (~p ∧ q)

value /* Well-formedness */
isWfDiagram : Diagram → Bool
isWfDiagram(d) ≡
  isWfIdentifiers(d) ∧
  isWfNeighbours(d) ∧
  isWfBranchNeighbours(d) ∧
  isWfPointMachines(d) ∧
  isWfSignalLocations(d),

isWfIdentifiers : Diagram → Bool
isWfIdentifiers(d) ≡
  allLinears(d) ∩ allPoints(d) = {}

value /* Well-formedness neighbours */
isWfNeighbours : Diagram → Bool
isWfNeighbours(d) ≡
  neighbours.OnlySections(d) ∧
  neighbours.EverySection(d) ∧
  neighbours.Irreflexive(d) ∧
  neighbours.Antisymmetrical(d) ∧
  linears_1or2Neighbours(d) ∧
  points_3Neighbours(d) ∧
  pointsNotNeighbours(d),

neighbours.OnlySections : Diagram → Bool
neighbours.OnlySections(d) ≡
  ( ∀ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ neighbours(d) ⇒
      isSection(tId1, d) ∧ isSection(tId2, d) ),

neighbours.EverySection : Diagram → Bool
neighbours.EverySection(d) ≡
  ( ∀ tId : TrackId • tId ∈ allSections(d) ⇒
    ( ∃ nb : TrackId • nb ∈ allSections(d) ∧
      areNeighbours(tId, nb, d)) ),

neighbours.Irreflexive : Diagram → Bool
neighbours.Irreflexive(d) ≡
  ( ∀ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ neighbours(d) ⇒
      tId1 ≠ tId2 ),

neighbours.Antisymmetrical : Diagram → Bool
neighbours.Antisymmetrical(d) ≡
  ( ∀ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ neighbours(d) ⇒
      (tId2, tId1) ∉ neighbours(d) ),

```



```

linears_1or2Neighbours : Diagram → Bool
linears_1or2Neighbours(d) ≡
  ( ∀ tId : TrackId •
    tId ∈ allLinears(d) ⇒
      let n = card getNeighboursOf(tId, d) in
        n ≥ 1 ∧ n ≤ 2
    end),

points_3Neighbours : Diagram → Bool
points_3Neighbours(d) ≡
  ( ∀ tId : TrackId •
    tId ∈ allPoints(d) ⇒
      card getNeighboursOf(tId, d) = 3),

pointsNotNeighbours : Diagram → Bool
pointsNotNeighbours(d) ≡
  ~( ∃ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ neighbours(d) ∧
    isPoints(tId1, d) ∧ isPoints(tId2, d) )

value /* Well-formedness branchNeighbours */
isWfBranchNeighbours : Diagram → Bool
isWfBranchNeighbours(d) ≡
  branchNeighbours_areNeighbours(d) ∧
  branchNeighbours_FirstIsPoints(d) ∧
  branchNeighbours_everyPoints_2branchNbs(d) ∧
  branchNeighbours_diffNbsOnBranch(d),

branchNeighbours_areNeighbours : Diagram → Bool
branchNeighbours_areNeighbours(d) ≡
  ( ∀ (tId1, tId2) : (TrackId × TrackId) •
    (tId1, tId2) ∈ dom branchNeighbours(d) ⇒
      areNeighbours(tId1, tId2, d) ),

branchNeighbours_FirstIsPoints : Diagram → Bool
branchNeighbours_FirstIsPoints(d) ≡
  ( ∀ (tId1, tId2) : TrackId × TrackId •
    (tId1, tId2) ∈ dom branchNeighbours(d) ⇒
      isPoints(tId1, d)
  ),

branchNeighbours_everyPoints_2branchNbs : Diagram → Bool
branchNeighbours_everyPoints_2branchNbs(d) ≡
  (∀ p : TrackId • p ∈ allPoints(d) ⇒
    (∃ (tId1, tId2) : TrackId × TrackId •
      (tId1, tId2) ∈ dom branchNeighbours(d) ∧
      p = tId1 ∧
      (∃ (tId1', tId2') : TrackId × TrackId •
        (tId1', tId2') ∈ dom branchNeighbours(d) ∧
        p = tId1' ∧ tId2' ≠ tId2')
    )
  ),

branchNeighbours_diffNbsOnBranch : Diagram → Bool

```

```

branchNeighbours_diffNbsOnBranch(d) ≡
  (∀ (p, tId1) : TrackId × TrackId •
    (p, tId1) ∈ branchNeighbours(d) ⇒
      (∀ (p', tId2) : TrackId × TrackId •
        (p', tId2) ∈ branchNeighbours(d) ⇒
          (p = p' ∧ tId1 ≠ tId2 ⇒
            (branchNeighbours(d)(p, tId1) = T.plus ∧
              branchNeighbours(d)(p, tId2) = T.minus) ∨
            (branchNeighbours(d)(p, tId1) = T.minus ∧
              branchNeighbours(d)(p, tId2) = T.plus))))))

value /* Well-formedness point machines */
isWfPointMachines : Diagram → Bool
isWfPointMachines(d) ≡
  areSymmetrical(pointMachineTrack(d), trackPointMachine(d)) ∧
  pointsSectionsExists(d),

areSymmetrical : (PointMachineId  $\overrightarrow{m}$  TrackId) ×
  (TrackId  $\overleftarrow{m}$  PointMachineId) → Bool
areSymmetrical(pmt, tpm) ≡
  dom pmt = rng tpm ∧
  dom tpm = rng pmt ∧
  ( ∀ pmlId : PointMachineId • pmlId ∈ dom pmt ⇒
    let tId = pmt(pmlId)
    in pmlId = tpm(tId)
    end ) ∧
  ( ∀ tId : TrackId • tId ∈ dom tpm ⇒
    let pmlId = tpm(tId)
    in tId = pmt(pmlId)
    end ),

pointsSectionsExists : Diagram → Bool
pointsSectionsExists(d) ≡
  dom trackPointMachine(d) = allPoints(d)

value /* Well-formedness signalLocations */
isWfSignalLocations : Diagram → Bool
isWfSignalLocations(d) ≡
  signalLocation_UsesExistingSignals(d) ∧
  signalsArePlacedAtValtIdNeighbours(d),

signalLocation_UsesExistingSignals : Diagram → Bool
signalLocation_UsesExistingSignals(d) ≡
  rng signalLocations(d) = allSignals(d),

signalsArePlacedAtValtIdNeighbours : Diagram → Bool
signalsArePlacedAtValtIdNeighbours(d) ≡
  ( ∀ (tId1, tId2) : (TrackId × TrackId) •
    (tId1, tId2) ∈ dom signalLocations(d) ⇒
      areNeighbours(tId1, tId2, d) )

end

```

E.1.4 Train Route Table

```

context: T, SL
scheme TrainRouteTable =
  class
    type
      RelayState == drawn | dropped,
      Aspect == gr | re | arbitrary,

      ReleaseCond = (SL.TrackId × RelayState) × (SL.TrackId × RelayState),
      ReleaseSequence = ReleaseCond × ReleaseCond, -- Indl i; Opl

    DataRow ::
      overlap      : SL.TrackId-set
      signals      : SL.SignalId  $\overline{m}$  Aspect
      points       : SL.PointMachineId  $\overline{m}$  T.PointsState
      trainDetection : SL.TrackId  $\overline{m}$  Bool
      signalRelease  : SL.SignalId × SL.TrackId
      trainRouteRelease : ReleaseSequence
      mutuallyBlocking : T.TrainRouteId  $\overline{m}$  Bool,

    TrainRouteTable = T.TrainRouteId  $\overline{m}$  DataRow

  value -- AUX
    getEntrySignal : DataRow → SL.SignalId
    getEntrySignal(dr) ≡
      let (sId, tId) = signalRelease(dr) in sId end,

    coveredSections : (SL.TrackId  $\overline{m}$  Bool) → SL.TrackId-set
    coveredSections(trainDetection) ≡
      { tId | tId : SL.TrackId • tId ∈ dom trainDetection ∧
        trainDetection(tId)},

    areConnected : SL.TrackId-set × SL.Diagram → Bool
    areConnected(sections, d) ≡
      if sections = {} then true else
        let s = hd sections, sections' = sections \ {s}
        in findConnected({s}, {s}, sections, d) = sections
      end
    end,

    findConnected : SL.TrackId-set × SL.TrackId-set ×
      SL.TrackId-set × SL.Diagram → SL.TrackId-set
    findConnected(queue, visited, sectionsInRoute, d) ≡
      if queue = {} then visited else
        let q = hd queue,
          nbs = SL.getNeighboursOf(q, d) ∩ sectionsInRoute,
          queue' = (queue \ {q}) ∪ (nbs \ visited),
          visited' = visited ∪ {q}
        in findConnected(queue', visited', sectionsInRoute, d)
      end
    end

  value -- Well-formedness

```

```

isWfTrainRouteTable : TrainRouteTable × SL.Diagram → Bool
isWfTrainRouteTable(trt, d) ≡
  (∀ trId : T.TrainRouteId • trId ∈ dom trt ⇒
    isWfDataRow(trId, d, trt)),

isWfDataRow : T.TrainRouteId × SL.Diagram × TrainRouteTable → Bool
isWfDataRow(trId, d, trt) ≡
  let dr = trt(trId)
  in isWfOverlap(dr, d) ∧
    isWfSignals(dr, d) ∧
    isWfPoints(dr, d) ∧
    isWfTrainDetection(trainDetection(dr), d) ∧
    isWfSignalRelease(signalRelease(dr), d) ∧
    isWfTrainRouteRelease(dr, d) ∧
    isWfMutuallyBlocking(trId, trt)
  end

value -- Well-formedness of overlap
isWfOverlap : DataRow × SL.Diagram → Bool
isWfOverlap(dr, d) ≡
  let overlap = overlap(dr)
  in card overlap > 0 ⇒
    overlapCovered(overlap, trainDetection(dr)) ∧
    overlapIsConnected(overlap, d) ∧
    endOfRoute(dr, d)
  end,

-- 4.3.3.1.1
overlapCovered : SL.TrackId-set × (SL.TrackId  $\overline{m}$  Bool) → Bool
overlapCovered(overlap, trainDetection) ≡
  overlap ⊆ coveredSections(trainDetection),

-- 4.3.3.1.2
overlapIsConnected : SL.TrackId-set × SL.Diagram → Bool
overlapIsConnected(overlap, d) ≡
  areConnected(overlap, d),

-- 4.3.3.1.3
endOfRoute : DataRow × SL.Diagram → Bool
endOfRoute(dr, d) ≡
  let overlap = overlap(dr),
    coveredSections = coveredSections(trainDetection(dr)),
    route = coveredSections \ overlap
  in ( ∃! tId : SL.TrackId • tId ∈ overlap ∧
    let nbs = SL.getNeighboursOf(tId, d),
    nbsInOverlap = nbs ∩ route
    in card nbsInOverlap ≥ 1
    end
  )
  end

value -- Well-formedness of signals
isWfSignals : DataRow × SL.Diagram → Bool
isWfSignals(dr, d) ≡
  let signals = signals(dr)

```

```

    in existingSignals(signals, d) ∧
      proceedEntry(getEntrySignal(dr), signals)
  end,

-- 4.3.4.1.1
existingSignals : (SL.SignalId  $\multimap$  Aspect) × SL.Diagram → Bool
existingSignals(signals, d) ≡
  dom signals = SL.allSignals(d),

-- 4.3.4.1.2
proceedEntry : SL.SignalId × (SL.SignalId  $\multimap$  Aspect) → Bool
proceedEntry(entrySignal, signals) ≡
  signals(entrySignal) = gr

value -- Well-formedness of points
isWfPoints : DataRow × SL.Diagram → Bool
isWfPoints(dr, d) ≡
  let points = points(dr)
  in -- 4.3.5.1.1
    pointMachinesExist(dom points, d) ∧
    -- 4.3.5.1.2
    pointMachinesInTRT(dr, d) ∧
    -- 4.3.5.1.3
    pointsInRouteLocked(dr, d) ∧
    -- 4.3.5.1.4
    correctPos(dr, d)
  end,

-- 4.3.5.1.1
pointMachinesExist : SL.PointMachineId-set × SL.Diagram → Bool
pointMachinesExist(points, d) ≡
  ( ∀ pId : SL.PointMachineId • pId ∈ points ⇒
    pId ∈ dom SL.pointMachineTrack(d) ∧
    let tId = SL.pointMachineTrack(d)(pId)
    in tId ∈ SL.allPoints(d)
  end ),

-- 4.3.5.1.2
pointMachinesInTRT : DataRow × SL.Diagram → Bool
pointMachinesInTRT(dr, d) ≡
  ( ∀ tId : SL.TrackId •
    tId ∈ dom trainDetection(dr) ⇒
    SL.isPoints(tId, d) ⇒
    let pmId = SL.trackPointMachine(d)(tId)
    in pmId ∈ dom points(dr)
  end ),

-- 4.3.5.1.3
pointsInRouteLocked : DataRow × SL.Diagram → Bool
pointsInRouteLocked(dr, d) ≡
  ( ∀ pId : SL.PointMachineId • pId ∈ points(dr) ⇒
    let pos = points(dr)(pId),
        tId = SL.pointMachineTrack(d)(pId)
    in tId ∈ coveredSections(trainDetection(dr)) ⇒
      (pos = T.plus ∨ pos = T.minus)
  )

```

```

    end ),

-- 4.3.5.1.4
correctPos : DataRow × SL.Diagram → Bool
correctPos(dr, d) ≡
  ( ∀ pId : SL.TrackId •
    pId ∈ coveredSections(trainDetection(dr)) ∩
    SL.allPoints(d) ⇒
    ( ∀ nb : SL.TrackId •
      nb ∈ SL.getNeighboursOf(pId, d) ∩
      coveredSections(trainDetection(dr)) ⇒
      SL.isBranchNb(pId, nb, d) ⇒
      points(dr)(SL.trackPointMachine(d)(pId)) =
      SL.branchNeighbours(d)(pId, nb) ))

value -- Well-formedness of train detection
isWfTrainDetection : (SL.TrackId  $\overline{m}$  Bool) × SL.Diagram → Bool
isWfTrainDetection(trainDetection, d) ≡
  existingTrackSections(dom trainDetection, d) ∧
  atLeastOneSectionInRoute(trainDetection) ∧
  connectedRoute(trainDetection, d) ∧
  justOneBranchNeighbour(trainDetection, d),

-- 4.3.6.1.1
existingTrackSections : SL.TrackId-set × SL.Diagram → Bool
existingTrackSections(sections, d) ≡
  sections = SL.allLinears(d) ∪ SL.allPoints(d),

-- 4.3.6.1.2
atLeastOneSectionInRoute : (SL.TrackId  $\overline{m}$  Bool) → Bool
atLeastOneSectionInRoute(trainDetection) ≡
  (∃ b : Bool • b ∈ rng trainDetection ∧ b),

-- 4.3.6.1.3
connectedRoute : (SL.TrackId  $\overline{m}$  Bool) × SL.Diagram → Bool
connectedRoute(trainDetection, d) ≡
  areConnected(coveredSections(trainDetection), d),

-- 4.3.6.1.4
justOneBranchNeighbour :
  (SL.TrackId  $\overline{m}$  Bool) × SL.Diagram → Bool
justOneBranchNeighbour(trainDetection, d) ≡
  ( ∀ tId : SL.TrackId • tId ∈ coveredSections(trainDetection) ⇒
    -- Only interested in points sections
    SL.isPoints(tId, d) ⇒
    -- Does not have two branch side neighbour
    ~ ( ∃ bNb1 : SL.TrackId •
      bNb1 ∈ coveredSections(trainDetection) ∧
      SL.isBranchNb(tId, bNb1, d) ∧
      ( ∃ bNb2 : SL.TrackId •
        bNb2 ∈ coveredSections(trainDetection) ∧
        SL.isBranchNb(tId, bNb2, d) ∧
        bNb1 ≠ bNb2 )
    )
  )
)

```

```

value -- Well-formedness of signal release
isWfSignalRelease : (SL.SignalId × SL.TrackId) × SL.Diagram → Bool
isWfSignalRelease((sId, tId), d) ≡
  sId ∈ SL.allSignals(d) ∧
  tId ∈ SL.allLinears(d) ∪ SL.allPoints(d) ∧
  signalFollowedByTrackSection(sId, tId, d),

-- 4.3.7.1
signalFollowedByTrackSection : SL.SignalId × SL.TrackId ×
  SL.Diagram → Bool
signalFollowedByTrackSection(sId, tId, d) ≡
  (∃ tId2 : SL.TrackId • tId2 ∈ SL.allSections(d) ∧
    (tId2, tId) ∈ dom SL.signalLocations(d) ∧
    SL.signalLocations(d)(tId2, tId) = sId
  )

value -- Well-formedness of train route release
isWfTrainRouteRelease : DataRow × SL.Diagram → Bool
isWfTrainRouteRelease(dr, d) ≡
  let (indl, opl) = trainRouteRelease(dr),
    ((tId1, rs1), (tId2, rs2)) = indl,
    ((tId3, rs3), (tId4, rs4)) = opl
  in reverseTrackSections(tId1, tId2, tId3, tId4) ∧
    reverseRelayState(rs1, rs2, rs3, rs4) ∧
    trackSectionPartOfRoute(tId1, dr) ∧
    trackSectionPartOfRoute(tId2, dr) ∧
    properRelayState(rs1, rs2) ∧
    SL.areNeighbours(tId1, tId2, d)
  end,

reverseTrackSections : SL.TrackId × SL.TrackId ×
  SL.TrackId × SL.TrackId → Bool
reverseTrackSections(tId1, tId2, tId3, tId4) ≡
  tId1 = tId4 ∧ tId2 = tId3,

reverseRelayState : RelayState × RelayState ×
  RelayState × RelayState → Bool
reverseRelayState(rs1, rs2, rs3, rs4) ≡
  rs1 = rs3 ∧ rs2 = rs4,

trackSectionPartOfRoute : SL.TrackId × DataRow → Bool
trackSectionPartOfRoute(tId, dr) ≡
  tId ∈ coveredSections(trainDetection(dr)),

properRelayState : RelayState × RelayState → Bool
properRelayState(rs1, rs2) ≡
  rs1 = dropped ∧ rs2 = drawn

value -- Well-formedness of mutually blocking
isWfMutuallyBlocking : T.TrainRouteId × TrainRouteTable → Bool
isWfMutuallyBlocking(trId, trt) ≡
  let dr = trt(trId)
  in entryForEveryRoute(mutuallyBlocking(dr), trt) ∧
    conflictingWhenShareSection(trId, trt) ∧

```

```

    conflictingWhenDifferentPos(trId, trt)
end,

-- 4.3.9.1 Every Train Route Has An Entry
entryForEveryRoute : (T.TrainRouteId  $\overline{\mapsto}$  Bool)  $\times$ 
    TrainRouteTable  $\rightarrow$  Bool
entryForEveryRoute(mutuallyBlocking, trt)  $\equiv$ 
    dom mutuallyBlocking  $\subseteq$  dom trt  $\wedge$ 
    dom mutuallyBlocking  $\supseteq$  dom trt,

-- 4.3.9.2 Conflicting routes when sharing sections
conflictingWhenShareSection : T.TrainRouteId  $\times$ 
    TrainRouteTable  $\rightarrow$  Bool
conflictingWhenShareSection(route1, trt)  $\equiv$ 
    let routes = dom trt
    in conflictingWhenShareSection(route1, routes, trt)
end,

conflictingWhenShareSection : T.TrainRouteId  $\times$ 
    T.TrainRouteId-set  $\times$  TrainRouteTable  $\rightarrow$  Bool
conflictingWhenShareSection(route1, routes, trt)  $\equiv$ 
    if routes = {} then true else
        let route2 = hd routes, routes' = routes \ {route2}
        in conflictingWhenShareSection(route1, route2, trt)  $\wedge$ 
            conflictingWhenShareSection(route1, routes', trt) -- Recursive Call
        end
    end,

conflictingWhenShareSection : T.TrainRouteId  $\times$  T.TrainRouteId  $\times$ 
    TrainRouteTable  $\rightarrow$  Bool
conflictingWhenShareSection(route1, route2, trt)  $\equiv$ 
    let -- Route 1
        dr1 = trt(route1),
        sections1 = coveredSections(trainDetection(dr1)),
        overlap1 = overlap(dr1),
        sectionsNoOverlap1 = sections1 \ overlap1,
        -- Route 2
        dr2 = trt(route2),
        sections2 = coveredSections(trainDetection(dr2)),
        overlap2 = overlap(dr2),
        sectionsNoOverlap2 = sections2 \ overlap2
    in card (sectionsNoOverlap1  $\cap$  sectionsNoOverlap2) > 0  $\Rightarrow$ 
        mutuallyBlocking(dr1)(route2)
    end,

-- 4.3.9.3 Conflicting routes if different point positions
conflictingWhenDifferentPos : T.TrainRouteId  $\times$  TrainRouteTable  $\rightarrow$  Bool
conflictingWhenDifferentPos(route1, trt)  $\equiv$ 
    let routes = dom trt
    in conflictingWhenDifferentPos(route1, routes, trt)
end,

conflictingWhenDifferentPos : T.TrainRouteId  $\times$ 
    T.TrainRouteId-set  $\times$  TrainRouteTable  $\rightarrow$  Bool
conflictingWhenDifferentPos(route1, routes, trt)  $\equiv$ 

```



```

    if routes = {} then true else
      let route2 = hd routes,
          routes' = routes \ {route2}
      in conflictingWhenDifferentPos(route1, route2, trt) ∧
          conflictingWhenDifferentPos(route1, routes', trt) -- Recursive Call
      end
    end,

conflictingWhenDifferentPos : T.TrainRouteId × T.TrainRouteId ×
  TrainRouteTable → Bool
conflictingWhenDifferentPos(route1, route2, trt) ≡
  let -- Route 1
      dr1 = trt(route1),
      points1 = points(dr1),
      -- Route 2
      dr2 = trt(route2),
      points2 = points(dr2)
  in ( ∃ pmId : SL.PointMachineId •
      pmId ∈ dom points1 ∧
      pmId ∈ dom points2 ∧
      points1(pmId) ≠ T.arbitrary ∧
      points2(pmId) ≠ T.arbitrary ∧
      points1(pmId) ≠ points2(pmId)
    ) ⇒ mutuallyBlocking(dr1)(route2)
  end

end

```

E.1.5 Transition System

```

context: T
scheme TransitionSystem =
  class
    type
      /* Variable in state of transition system */
      Var ::
        id : T.VarId
        val : Val,

      /* Value as boolean or natural number */
      Val == mk_BoolVal(b : Bool) | mk_NatVal(n : Nat),

      Assignment ::
        id : T.VarId
        assign : T.AssignExp,

      MultipleAssignment = Assignment-set,

      TransitionRule ::
        name : Text
        guard : T.BooleanExp

```

```

    assignments : MultipleAssignment,

    /* Assertion */
    Assertion = Text × T.LTLformula,

    TransitionSystem ::
      name : Text
      state : Var-set
      transitionRules : TransitionRule-set
      assertions : Assertion-set

value /* Auxiliary functions */
  isIdInState : T.VarId × Var-set → Bool
  isIdInState(id, state) ≡
    ( ∃ var : Var • var ∈ state ∧ id = id(var) ),

  idsInBoolExp : T.BooleanExp → T.VarId-set
  idsInBoolExp(boolExp) ≡
    case boolExp of
      T.bool(b) → {},
      T.literal(l) → {l},
      T.and(a) → idsInBoolExp(a),
      T.or(o) → idsInBoolExp(o),
      T.neg(n) → idsInBoolExp(n),
      T.lessThan(l,r) →
        idsInArithmeticExp(l) ∪ idsInArithmeticExp(r),
      T.greaterThan(l,r) →
        idsInArithmeticExp(l) ∪ idsInArithmeticExp(r),
      T.geq(l,r) →
        idsInArithmeticExp(l) ∪ idsInArithmeticExp(r),
      T.equals(l,r) →
        idsInArithmeticExp(l) ∪ idsInArithmeticExp(r)
    end,

  idsInBoolExp : T.BooleanExp-set → T.VarId-set
  idsInBoolExp(boolExps) ≡
    let boolExp = hd boolExps, boolExps' = boolExps \ {boolExp}
    in if boolExps = {} then {}
      else idsInBoolExp(boolExp) ∪ idsInBoolExp(boolExps')
    end
    end,

  idsInArithmeticExp : T.ArithmeticExp → T.VarId-set
  idsInArithmeticExp(aExp) ≡
    case aExp of
      T.add(a) → idsInArithmeticExp(a),
      T.sub(s) → idsInArithmeticExp(s),
      T.literal(l) → {l}
    end,

  idsInArithmeticExp : T.ArithmeticExp-set → T.VarId-set
  idsInArithmeticExp(aExps) ≡
    let aExp = hd aExps, aExps' = aExps \ {aExp}
    in if aExps = {} then {}
      else idsInArithmeticExp(aExp) ∪

```

```

        idsInArithmeticExp(aExps')
    end
end,

idsInLTL : T.LTLformula → T.VarId-set
idsInLTL(ltl) ≡
    case ltl of
        T.b(boolExp) → idsInBoolExp(boolExp),
        T.ltrl(l) → {l},
        T.conj(ltls) → idsInLTL(ltls),
        T.disj(ltls) → idsInLTL(ltls),
        T.paren(ltl) → idsInLTL(ltl),
        T.neg(ltl) → idsInLTL(ltl),
        T.impl(ltl1, ltl2) → idsInLTL(ltl1) ∪ idsInLTL(ltl2),
        T.x(ltl) → idsInLTL(ltl),
        T.g(ltl) → idsInLTL(ltl),
        T.f(ltl) → idsInLTL(ltl),
        T.u(ltl1, ltl2) → idsInLTL(ltl1) ∪ idsInLTL(ltl2)
    end,

idsInLTL : T.LTLformula-set → T.VarId-set
idsInLTL(ltls) ≡
    let ltl = hd ltls, ltls' = ltls \ {ltl}
    in if ltls = {} then {}
        else idsInLTL(ltl) ∪ idsInLTL(ltls')
    end
end

value /* Well-formness */
isWfTransitionSystem : TransitionSystem → Bool
isWfTransitionSystem(ts) ≡
    isWfName(name(ts)) ∧
    isWfState(state(ts)) ∧
    isWfTransitionRules(ts) ∧
    isWfAssertions(ts)

value /* Well-formness of name */
isWfName : Text → Bool
isWfName(name) ≡ name ≠ ""

value /* Well-formness of state */
isWfState : Var-set → Bool
isWfState(vars) ≡ uniqueIds(vars) ∧ validIds(vars),

uniqueIds : Var-set → Bool
uniqueIds(vars) ≡
    ( ∀ var1 : Var • var1 ∈ vars ⇒
        ~ ( ∃ var2 : Var • var2 ∈ vars ∧
            id(var1) = id(var2) ∧ var1 ≠ var2 )
    ),

validIds : Var-set → Bool
validIds(vars) ≡
    ( ∀ var : Var • var ∈ vars ⇒
        id(var) ≠ "" ∧

```

```

    let char = hd id(var)
    in char ≠ '0' ∧ char ≠ '1' ∧
      char ≠ '2' ∧ char ≠ '3' ∧
      char ≠ '4' ∧ char ≠ '5' ∧
      char ≠ '6' ∧ char ≠ '7' ∧
      char ≠ '8' ∧ char ≠ '9'
  end )

value /* Well-formness of transition rules */
isWfTransitionRules : TransitionSystem → Bool
isWfTransitionRules(ts) ≡
  let state = state(ts),
      trs = transitionRules(ts)
  in validNames(trs) ∧
    uniqueNames(trs) ∧
    assignmentToExistingIds(ts) ∧
    idsInGuardMustBeInState(state, trs) ∧
    nonemptyAssignment(trs) ∧
    idUpdOnlyOnce(trs)
  end,

validNames : TransitionRule-set → Bool
validNames(trs) ≡
  ( ∀ tr : TransitionRule • tr ∈ trs ⇒
    name(tr) ≠ ""
  ),

uniqueNames : TransitionRule-set → Bool
uniqueNames(trs) ≡
  ( ∀ tr1 : TransitionRule • tr1 ∈ trs ⇒
    ~( ∃ tr2 : TransitionRule • tr2 ∈ trs ∧
      tr1 ≠ tr2 ∧ name(tr1) = name(tr2) )
  ),

assignmentToExistingIds : TransitionSystem → Bool
assignmentToExistingIds(ts) ≡
  ( ∀ tr : TransitionRule • tr ∈ transitionRules(ts) ⇒
    ( ∀ assign : Assignment • assign ∈ assignments(tr) ⇒
      ( ∃ var : Var • var ∈ state(ts) ∧
        id(assign) = id(var) )
    )
  ),

idsInGuardMustBeInState : Var-set × TransitionRule-set → Bool
idsInGuardMustBeInState(state, trs) ≡
  ( ∀ tr : TransitionRule • tr ∈ trs ⇒
    ( ∀ id : T.VarId • id ∈ idsInBoolExp(guard(tr)) ⇒
      ( ∃ var : Var • var ∈ state ∧
        id = id(var) )
    )
  ),

nonemptyAssignment : TransitionRule-set → Bool
nonemptyAssignment(trs) ≡

```

```

    (  $\forall$  tr : TransitionRule • tr  $\in$  trs  $\Rightarrow$ 
      assignments(tr)  $\neq$  {}
    ),

idUpdOnlyOnce : TransitionRule-set  $\rightarrow$  Bool
idUpdOnlyOnce(trs)  $\equiv$ 
  (  $\forall$  tr : TransitionRule • tr  $\in$  trs  $\Rightarrow$ 
    let ma = assignments(tr)
    in (  $\forall$  assign : Assignment • assign  $\in$  ma  $\Rightarrow$ 
       $\sim$ (  $\exists$  assign2 : Assignment • assign2  $\in$  ma  $\wedge$ 
        assign  $\neq$  assign2  $\wedge$  id(assign) = id(assign2) )
    )
  end
)

value /* Well-formness of assertions */
isWfAssertions : TransitionSystem  $\rightarrow$  Bool
isWfAssertions(ts)  $\equiv$ 
  let state = state(ts),
      assertions = assertions(ts)
  in nonemptyNames(assertions)  $\wedge$ 
    uniqueNames(assertions)  $\wedge$ 
    existingVars(state, assertions)
  end,

nonemptyNames : Assertion-set  $\rightarrow$  Bool
nonemptyNames(assertions)  $\equiv$ 
  (  $\forall$  (name, ltl) : Assertion • (name, ltl)  $\in$  assertions  $\Rightarrow$ 
    name  $\neq$  "" ),

uniqueNames : Assertion-set  $\rightarrow$  Bool
uniqueNames(assertions)  $\equiv$ 
  (  $\forall$  (name, ltl) : Assertion • (name, ltl)  $\in$  assertions  $\Rightarrow$ 
     $\sim$ (  $\exists$  (name2, ltl2) : Assertion •
      (name2, ltl2)  $\in$  assertions  $\wedge$ 
      (name, ltl)  $\neq$  (name2, ltl2)  $\wedge$ 
      name = name2 )
    ),

existingVars : Var-set  $\times$  Assertion-set  $\rightarrow$  Bool
existingVars(state, assertions)  $\equiv$ 
  (  $\forall$  (name, ltl) : Assertion • (name, ltl)  $\in$  assertions  $\Rightarrow$ 
    (  $\forall$  id : T.VarId • id  $\in$  idsInLTL(ltl)  $\Rightarrow$ 
      isIdInState(id, state) )
    )
end

```

E.2 Generators

E.2.1 Objects

E.2.1.1 Generator

context: Generator
object G : Generator

E.2.1.2 AssertionGenerator

context: AssertionGenerator
object AG : AssertionGenerator

context: ConsistencyAssertionGenerator
object CAG : ConsistencyAssertionGenerator

context: SafetyAssertionGenerator
object SAG : SafetyAssertionGenerator

context: TrtAssertionGenerator
object TAG : TrtAssertionGenerator

E.2.1.3 StateGenerator

context: StateGenerator
object SG : StateGenerator

E.2.1.4 TransitionGenerator

context: TransitionGenerator
object TG : TransitionGenerator

context: TrainMovementTransitionGenerator
object TMTG : TrainMovementTransitionGenerator

context: OtherExternalTransitionsGenerator
object OXTG : OtherExternalTransitionsGenerator

E.2.2 Generator

context: AG, SG, TG, ORA, TMA
scheme Generator =
 class
 value
 generate : **Text** \times ORA.ObjectRelayAssociations \times
 IP.InterlockingPlan \times **Text-set** \rightarrow TS.TransitionSystem
 generate(name, ora, ip, buttonIds) \equiv
 let sld = IP.sld(ip),
 tma = genTrainMovementAssociations(sld)
 in TS.mk_TransitionSystem(
 name,
 SG.generateStateSpace(ip, tma),
 TG.generateTransitionSystem(ora, ip, tma, buttonIds),
 AG.generateAssertions(ora, ip, tma))
 end
 pre IP.isWfInterlockingPlan(ip) \wedge
 ORA.isWfObjectRelayAssociations(ora, ip),

 genTrainMovementAssociations :
 SL.Diagram \rightarrow TMA.TrainMovementAssociations
 genTrainMovementAssociations(d) \equiv
 TMA.mk_TrainMovementAssociations(
 SG.genConVars(SL.neighbours(d), d),
 SG.genCntVars(SL.allSections(d), d),
 SG.genDirFwdVars(SL.allSections(d), d),
 SG.genDirBwdVars(SL.allSections(d), d),
 SG.genDirections(d))
 end

E.2.3 Assertion Generator

context: CAG, SAG, TAG
scheme AssertionGenerator =
 class
 value
 generateAssertions :

```

ORA.ObjectRelayAssociations × IP.InterlockingPlan ×
TMA.TrainMovementAssociations → TS.Assertion-set
generateAssertions(ora, ip, tma) ≡
  CAG.genConsistencyAssertions(ora, ip, tma) ∪
  SAG.genSafetyAssertions(ora, ip, tma) ∪
  TAG.genTrtAssertions(ora, ip, tma)
pre IP.isWfInterlockingPlan(ip) ∧
      TMA.isWfTrainMovementAssociations(IP.sld(ip), tma) ∧
      ORA.isWfObjectRelayAssociations(ora, ip)

end

```

E.2.3.1 Consistency Assertion Generator

context: ORA, IP, TMA

scheme ConsistencyAssertionGenerator =

```

class
  value
    genConsistencyAssertions :
      ORA.ObjectRelayAssociations × IP.InterlockingPlan ×
      TMA.TrainMovementAssociations → TS.Assertion-set
    genConsistencyAssertions(ora, ip, tma) ≡
      -- 9.2 Track Occupation
      generateOccupationAssertions(ora, ip, tma) ∪
      -- 9.3 Track Free
      generateFreeAssertions(ora, ip, tma) ∪
      -- 9.4 Train Direction
      generateDirectionAssertions(ip, tma) ∪
      -- 9.5 Points Configuration
      generatePointsConfigurationAssertions(ora, ip) ∪
      -- 9.6 Train Connection
      generateConnectionAssertions(ip, tma)
    pre IP.isWfInterlockingPlan(ip) ∧
        TMA.isWfTrainMovementAssociations(IP.sld(ip), tma) ∧
        ORA.isWfObjectRelayAssociations(ora, ip)

  value -- 9.2 Track Occupation
    generateOccupationAssertions :
      ORA.ObjectRelayAssociations × IP.InterlockingPlan ×
      TMA.TrainMovementAssociations → TS.Assertion-set
    generateOccupationAssertions(ora, ip, tma) ≡
      let d = IP.sld(ip),
          sections = SL.allLinears(d) ∪ SL.allPoints(d) -- NB: No line pieces!
      in generateOccupationAssertions(sections, ora, tma)
    end,

    generateOccupationAssertions :
      SL.TrackId-set × ORA.ObjectRelayAssociations ×
      TMA.TrainMovementAssociations → TS.Assertion-set
    generateOccupationAssertions(sections, ora, tma) ≡
      if sections = {} then {} else

```



```

    let s = hd sections,
        sections' = sections \ {s}
    in {generateOccupationAssertion1(s, ora, tma),
        generateOccupationAssertion2(s, ora, tma)} ∪
        generateOccupationAssertions(sections', ora, tma) -- Recursive
    end
end,

-- cnt_01 ∷ 0 = ∷ ∼01
generateOccupationAssertion1 :
    SL.TrackId × ORA.ObjectRelayAssociations ×
    TMA.TrainMovementAssociations → TS.Assertion
generateOccupationAssertion1(s, ora, tma) ≡
    ("occ_" ^ s,
    genOccuFormula1(s, ora, tma)),

genOccuFormula1 :
    SL.TrackId × ORA.ObjectRelayAssociations ×
    TMA.TrainMovementAssociations → T.LTLformula
genOccuFormula1(s, ora, tma) ≡
    T.g(T.impl(T.b(T.greaterthan(
        T.literal(TMA.getCounterVar(s, tma)),
        T.literal("0")),
        T.neg(T.ltrl(ORA.trackRelayAssoc(ora)(s)))))),

-- ∼01 = ∷ cnt_01 ∷ 0
generateOccupationAssertion2 :
    SL.TrackId × ORA.ObjectRelayAssociations ×
    TMA.TrainMovementAssociations → TS.Assertion
generateOccupationAssertion2(s, ora, tma) ≡
    ("occ_" ^ s ^ "_",
    genOccuFormula2(s, ora, tma) ),

genOccuFormula2 :
    SL.TrackId × ORA.ObjectRelayAssociations ×
    TMA.TrainMovementAssociations → T.LTLformula
genOccuFormula2(s, ora, tma) ≡
    T.g(
        T.impl(T.neg(T.ltrl(ORA.trackRelayAssoc(ora)(s))),
            T.b(T.greaterthan(T.literal(TMA.getCounterVar(s, tma)),
                T.literal("0"))))),

value -- 9.3 Track Free
generateFreeAssertions :
    ORA.ObjectRelayAssociations × IP.InterlockingPlan ×
    TMA.TrainMovementAssociations → TS.Assertion-set
generateFreeAssertions(ora, ip, tma) ≡
    let d = IP.sld(ip),
        sections = SL.allLinears(d) ∪ SL.allPoints(d) -- NB: No line pieces!
    in generateFreeAssertions(sections, ora, tma)
end,

generateFreeAssertions :
    SL.TrackId-set × ORA.ObjectRelayAssociations ×
    TMA.TrainMovementAssociations → TS.Assertion-set

```

```

generateFreeAssertions(sections, ora, tma)  $\equiv$ 
  if sections = {} then {} else
    let s = hd sections,
        sections' = sections \ {s}
    in {generateFreeAssertion1(s, ora, tma),
        generateFreeAssertion2(s, ora, tma)}  $\cup$ 
        generateFreeAssertions(sections', ora, tma) -- Recursive
    end
end,

-- cnt_01 = 0 =i t01
generateFreeAssertion1 :
  SL.TrackId  $\times$  ORA.ObjectRelayAssociations  $\times$ 
  TMA.TrainMovementAssociations  $\rightarrow$  TS.Assertion
generateFreeAssertion1(s, ora, tma)  $\equiv$ 
  ("free_" ^ s,
   genFreeFormula1(s, ora, tma)),

genFreeFormula1 :
  SL.TrackId  $\times$  ORA.ObjectRelayAssociations  $\times$ 
  TMA.TrainMovementAssociations  $\rightarrow$  T.LTLformula
genFreeFormula1(s, ora, tma)  $\equiv$ 
  T.g(T.impl(T.b(T.equals(T.literal(TMA.getCounterVar(s, tma)),
                           T.literal("0"))),
              T.ltrl(ORA.trackRelayAssoc(ora)(s)))),

-- t01 =i cnt_01 = 0
generateFreeAssertion2 :
  SL.TrackId  $\times$  ORA.ObjectRelayAssociations  $\times$ 
  TMA.TrainMovementAssociations  $\rightarrow$  TS.Assertion
generateFreeAssertion2(s, ora, tma)  $\equiv$ 
  ("free_" ^ s ^ "_",
   genFreeFormula2(s, ora, tma)),

genFreeFormula2 :
  SL.TrackId  $\times$  ORA.ObjectRelayAssociations  $\times$ 
  TMA.TrainMovementAssociations  $\rightarrow$  T.LTLformula
genFreeFormula2(s, ora, tma)  $\equiv$ 
  T.g(T.impl(T.ltrl(ORA.trackRelayAssoc(ora)(s)),
              T.b(T.equals(T.literal(TMA.getCounterVar(s, tma)),
                           T.literal("0"))))),

value -- 9.4 Train Direction
generateDirectionAssertions : IP.InterlockingPlan  $\times$ 
  TMA.TrainMovementAssociations  $\rightarrow$  TS.Assertion-set
generateDirectionAssertions(ip, tma)  $\equiv$ 
  let d = IP.sld(ip),
      sections = SL.allLinears(d)  $\cup$  SL.allPoints(d) -- NB: No line pieces!
  in generateDirectionAssertions(sections, tma)
end,

generateDirectionAssertions : SL.TrackId-set  $\times$ 
  TMA.TrainMovementAssociations  $\rightarrow$  TS.Assertion-set
generateDirectionAssertions(sections, tma)  $\equiv$ 
  if sections = {} then {} else

```

```

    let s = hd sections,
        sections' = sections \ {s}
    in {generateDirectionAssertion(s, tma)} ∪
        generateDirectionAssertions(sections', tma) -- Recursive
    end
end,

generateDirectionAssertion : SL.TrackId ×
    TMA.TrainMovementAssociations → TS.Assertion
generateDirectionAssertion(tId, tma) ≡
    ("direction_" ^ tId,
    genDirFormula(tId, tma)
    ),

genDirFormula : SL.TrackId ×
    TMA.TrainMovementAssociations → T.LTLformula
genDirFormula(tId, tma) ≡
    let trackFwd = TMA.getDirVar(tId, T.fwd, tma),
        trackBwd = TMA.getDirVar(tId, T.bwd, tma),
        trackCnt = TMA.getCounterVar(tId, tma)
    in T.g(T.impl(T.disj({T.ltrl(trackFwd), T.ltrl(trackBwd)}),
        T.b(T.greaterthan(T.literal(trackCnt), T.literal("0")))))
    end

value -- 9.5 Points Configuration
generatePointsConfigurationAssertions :
    ORA.ObjectRelayAssociations ×
    IP.InterlockingPlan → TS.Assertion-set
generatePointsConfigurationAssertions(ora, ip) ≡
    let d = IP.sld(ip),
        allPoints = SL.allPoints(d)
    in generatePointsConfigurationAssertions(allPoints, d, ora)
    end,

generatePointsConfigurationAssertions :
    SL.TrackId-set × SL.Diagram ×
    ORA.ObjectRelayAssociations → TS.Assertion-set
generatePointsConfigurationAssertions(sections, d, ora) ≡
    if sections = {} then {} else
        let s = hd sections,
            sections' = sections \ {s}
        in {generatePointsConfigurationAssertion(s, d, ora)} ∪
            generatePointsConfigurationAssertions(sections', d, ora) -- Recursive
        end
    end,

generatePointsConfigurationAssertion :
    SL.TrackId × SL.Diagram ×
    ORA.ObjectRelayAssociations → TS.Assertion
generatePointsConfigurationAssertion(s, d, ora) ≡
    ("points_configuration_" ^ s,
    genPointsConfigFormula(s, d, ora)
    ),

genPointsConfigFormula : SL.TrackId × SL.Diagram ×

```

```

ORA.ObjectRelayAssociations → T.LTLformula
genPointsConfigFormula(s, d, ora) ≡
  T.g(T.impl(T.ltl("idle"), T.neg(T.conj(
    {T.ltl(ORA.getPointRelay(s, T.plus, ora, d)),
     T.ltl(ORA.getPointRelay(s, T.minus, ora, d))
    }))))

value -- 9.6 Train Connection
generateConnectionAssertions : IP.InterlockingPlan ×
  TMA.TrainMovementAssociations → TS.Assertion-set
generateConnectionAssertions(ip, tma) ≡
  let d = IP.sld(ip),
      neighbours = SL.neighbours(d) -- NB: Includes line pieces
  in generateConnectionAssertions(neighbours, d, tma)
end,

generateConnectionAssertions :
  (SL.TrackId × SL.TrackId)-set × SL.Diagram ×
  TMA.TrainMovementAssociations → TS.Assertion-set
generateConnectionAssertions(neighbours, d, tma) ≡
  if neighbours = {} then {} else
    let nb = hd neighbours,
        neighbours' = neighbours \ {nb}
    in generateConnectionAssertion(nb, d, tma) ∪
        generateConnectionAssertions(neighbours', d, tma) -- Recursive
  end
end,

generateConnectionAssertion :
  (SL.TrackId × SL.TrackId) × SL.Diagram ×
  TMA.TrainMovementAssociations → TS.Assertion-set
generateConnectionAssertion((nb1, nb2), d, tma) ≡
  if SL.isLine(nb1, d) ∨ SL.isLine(nb2, d) then {} -- NB: Discard line pieces
  else {( TMA.getConnectionVar(nb1, nb2, tma),
          genConFormula(nb1, nb2, tma) )}
  end,

genConFormula : SL.TrackId × SL.TrackId ×
  TMA.TrainMovementAssociations → T.LTLformula
genConFormula(from, to, tma) ≡
  T.g(T.impl(T.ltl(TMA.getConnectionVar(from, to, tma)),
    T.conj({T.b(
      T.greaterthan(T.literal(TMA.getCounterVar(from, tma)),
        T.literal("0")),
      T.b(T.greaterthan(T.literal(TMA.getCounterVar(to, tma)),
        T.literal("0")))}))))))

end

```

E.2.3.2 SafetyAssertionGenerator

context: ORA, IP, TMA

```

scheme SafetyAssertionGenerator =
  class
    value
      genSafetyAssertions :
        ORA.ObjectRelayAssociations × IP.InterlockingPlan ×
          TMA.TrainMovementAssociations → TS.Assertion-set
      genSafetyAssertions(ora, ip, tma) ≡
        -- 10.2 No Collision
        genNoCollisionAssertions(ip, tma) ∪
        -- 10.3 No Derailing
        genNoDerailingAssertions(ora, ip, tma)
    pre IP.isWfInterlockingPlan(ip) ∧
        TMA.isWfTrainMovementAssociations(IP.sld(ip), tma) ∧
        ORA.isWfObjectRelayAssociations(ora, ip)

  value -- 10.2 No Collision
    genNoCollisionAssertions : IP.InterlockingPlan ×
      TMA.TrainMovementAssociations → TS.Assertion-set
    genNoCollisionAssertions(ip, tma) ≡
      let d = IP.sld(ip),
        sections = SL.allLinears(d) ∪ SL.allPoints(d) -- NB: No line pieces!
      in generateNoCollisionAssertions(sections, tma)
    end,

    generateNoCollisionAssertions : SL.TrackId-set ×
      TMA.TrainMovementAssociations → TS.Assertion-set
    generateNoCollisionAssertions(sections, tma) ≡
      if sections = {} then {} else
        let s = hd sections,
          sections' = sections \ {s}
        in {generateNoCollisionAssertion(s, tma)} ∪
          generateNoCollisionAssertions(sections', tma) -- Recursive call
      end
    end,

    generateNoCollisionAssertion : SL.TrackId ×
      TMA.TrainMovementAssociations → TS.Assertion
    generateNoCollisionAssertion(s, tma) ≡
      ("no_collision_" ^ s,
        genNoColFormula(s, tma) ),

    genNoColFormula : SL.TrackId ×
      TMA.TrainMovementAssociations → T.LTLformula
    genNoColFormula(s, tma) ≡
      T.g(T.b(T.lessThan(T.literal(TMA.getCounterVar(s, tma))),
        T.literal("2"))))

  value -- 10.3 No Derailing
    genNoDerailingAssertions :
      ORA.ObjectRelayAssociations × IP.InterlockingPlan ×
        TMA.TrainMovementAssociations → TS.Assertion-set
    genNoDerailingAssertions(ora, ip, tma) ≡
      let d = IP.sld(ip),
        sections = SL.allPoints(d),
        branchNbs = dom SL.branchNeighbours(d)

```

```

in -- G(cnt_01  $i = 1 = i$  plus01  $\setminus$  minus01)
    generateNoDerailingAssertions(sections, d, ora, tma)  $\cup$ 
    -- G(con_02_03  $= i$  plus02)
    generateNoDerailingAssertions(branchNbs, d, ora, tma)
end,

-- G(cnt_01  $i = 1 = i$  plus01  $\setminus$  minus01)
generateNoDerailingAssertions : SL.TrackId-set  $\times$ 
    SL.Diagram  $\times$  ORA.ObjectRelayAssociations  $\times$ 
    TMA.TrainMovementAssociations  $\rightarrow$  TS.Assertion-set
generateNoDerailingAssertions(sections, d, ora, tma)  $\equiv$ 
    if sections = {} then {} else
        let s = hd sections,
            sections' = sections  $\setminus$  {s}
        in {generateNoDerailingAssertion(s, d, ora, tma)}  $\cup$ 
            generateNoDerailingAssertions(sections', d, ora, tma) -- Recursive call
    end
end,

generateNoDerailingAssertion : SL.TrackId  $\times$ 
    SL.Diagram  $\times$  ORA.ObjectRelayAssociations  $\times$ 
    TMA.TrainMovementAssociations  $\rightarrow$  TS.Assertion
generateNoDerailingAssertion(s, d, ora, tma)  $\equiv$ 
    ("points_locked_when_occupied_"  $\wedge$  s,
     genNoDerailFormula(s, d, ora, tma) ),

genNoDerailFormula : SL.TrackId  $\times$ 
    SL.Diagram  $\times$  ORA.ObjectRelayAssociations  $\times$ 
    TMA.TrainMovementAssociations  $\rightarrow$  T.LTLformula
genNoDerailFormula(s, d, ora, tma)  $\equiv$ 
    T.g(T.impl(T.b(T.geq(T.literal(
        TMA.getCounterVar(s, tma)), T.literal("1"))),
        T.disj({T.ltrl(ORA.getPointRelay(s, T.plus, ora, d)),
            T.ltrl(ORA.getPointRelay(s, T.minus, ora, d))
        }))),

-- G(con_02_03  $= i$  plus02)
generateNoDerailingAssertions : (SL.TrackId  $\times$  SL.TrackId)-set  $\times$ 
    SL.Diagram  $\times$  ORA.ObjectRelayAssociations  $\times$ 
    TMA.TrainMovementAssociations  $\rightarrow$  TS.Assertion-set
generateNoDerailingAssertions(bNbs, d, ora, tma)  $\equiv$ 
    if bNbs = {} then {} else
        let bNb = hd bNbs,
            bNbs' = bNbs  $\setminus$  {bNb}
        in {generateNoDerailingAssertion(bNb, d, ora, tma)}  $\cup$ 
            generateNoDerailingAssertions(bNbs', d, ora, tma) -- Recursive call
    end
end,

generateNoDerailingAssertion : (SL.TrackId  $\times$  SL.TrackId)  $\times$ 
    SL.Diagram  $\times$  ORA.ObjectRelayAssociations  $\times$ 
    TMA.TrainMovementAssociations  $\rightarrow$  TS.Assertion
generateNoDerailingAssertion(bNb, d, ora, tma)  $\equiv$ 
    let (pointsSection, nb) = bNb,
        branch = SL.branchNeighbours(d)(pointsSection, nb),

```

```

        conVar = TMA.getConnectionVar(pointsSection, nb, tma),
        branchVar = ORA.getPointRelay(pointsSection, branch, ora, d)
    in ("no_derailing_" ^ pointsSection ^ "_" ^ nb,
        T.g(T.impl(T.ltrl(conVar), T.ltrl(branchVar))))
    end

end

```

E.2.3.3 TrtAssertionGenerator

context: ORA, IP, TMA

scheme TrtAssertionGenerator =

```

    class
        value
            genTrtAssertions :
                ORA.ObjectRelayAssociations × IP.InterlockingPlan ×
                TMA.TrainMovementAssociations → TS.Assertion-set
            genTrtAssertions(ora, ip, tma) ≡
                -- 10.4 Points Position
                genPointsPositionAssertions(ora, IP.trt(ip)) ∪
                -- 10.5 Signal
                genSignalAssertions(ora, ip) ∪
                -- 10.6 Signal Release
                genSignalReleaseAssertions(ora, ip) ∪
                -- 10.7 Conflicting Routes
                genConflictingRoutesAssertions(ora, ip) ∪
                -- 10.8 Train Route Release
                genTrainRouteReleaseAssertions(ora, ip)
            pre IP.isWfInterlockingPlan(ip) ∧
                TMA.isWfTrainMovementAssociations(IP.sld(ip), tma) ∧
                ORA.isWfObjectRelayAssociations(ora, ip)

        value -- 10.4 Points Position
            genPointsPositionAssertions : ORA.ObjectRelayAssociations ×
            TRT.TrainRouteTable → TS.Assertion-set
            genPointsPositionAssertions(ora, trt) ≡
                let lockingRelays = rng ORA.routeRelayAssoc(ora)
                in generatePointPositionAssertions(lockingRelays, ora, trt)
            end,

            generatePointPositionAssertions :
                T.VarId-set × ORA.ObjectRelayAssociations ×
                TRT.TrainRouteTable → TS.Assertion-set
            generatePointPositionAssertions(lockingRelays, ora, trt) ≡
                if lockingRelays = {} then {} else
                    let relayId = hd lockingRelays,
                        lockingRelays' = lockingRelays \ {relayId}
                    in {genPointPosAssert(relayId, ora, trt)} ∪
                        generatePointPositionAssertions(lockingRelays', ora, trt)
                    end
                end,
        end,
    end,

```

```

genPointPosAssert : T.VarId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → TS.Assertion
genPointPosAssert(relayId, ora, trt) ≡
  ("point_position_" ^ relayId,
   genPointPosFormula(relayId, ora, trt) ),

genPointPosFormula : T.VarId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.LTLformula
genPointPosFormula(relayId, ora, trt) ≡
let routeIds = getRouteIdsOfLockingRelay(
  ORA.routeRelayAssoc(ora), relayId)
in T.g(T.impl(T.neg(T.ltrl(relayId)),
  T.disj(relayPointPosConfig(routeIds, ora, trt))))

end,

relayPointPosConfig : T.TrainRouteId-set ×
  ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.LTLformula-set
relayPointPosConfig(routeIds, ora, trt) ≡
  if routeIds = {} then {} else
    let routeId = hd routeIds,
      routeIds' = routeIds \ {routeId}
    in {T.conj(routePointPosConfig(routeId, ora, trt))} ∪
      relayPointPosConfig(routeIds', ora, trt)
  end
end,

routePointPosConfig : T.TrainRouteId ×
  ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.LTLformula-set
routePointPosConfig(routeId, ora, trt) ≡
  let pointsMap = TRT.points(trt(routeId)),
    pointsSet = mapToSet(pointsMap)
  in routePointPosConfig_(pointsSet, ora, trt)
end,

routePointPosConfig_ : (SL.PointMachineId × T.PointsState)-set ×
  ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.LTLformula-set
routePointPosConfig_(pointsSet, ora, trt) ≡
  if pointsSet = {} then {} else
    let (pId, branch) = hd pointsSet,
      pointsSet' = pointsSet \ {(pId, branch)}
    in if branch = T.arbitrary then {}
      else {T.ltrl(ORA.pointRelayAssoc(ora)(pId, branch))}
      end ∪ routePointPosConfig_(pointsSet', ora, trt)
    end
end,

getRouteIdsOfLockingRelay : (T.TrainRouteId  $\overrightarrow{m}$  T.VarId) ×
  T.VarId → T.TrainRouteId-set
getRouteIdsOfLockingRelay(rrAssocs, relayId) ≡
  { routeId | routeId : T.VarId •
    routeId ∈ dom rrAssocs ∧

```



```

rrAssocs(routeId) = relayId}

value -- 10.5 Signal
  genSignalAssertions : ORA.ObjectRelayAssociations ×
    IP.InterlockingPlan → TS.Assertion-set
  genSignalAssertions(ora, ip) ≡
    let d = IP.sld(ip),
        trt = IP.trt(ip)
    in generateSignalAspectAssertions(d, ora, trt) ∪
        generateSignalLockingAssertions(d, ora, trt)
    end,

  generateSignalAspectAssertions :
    SL.Diagram × ORA.ObjectRelayAssociations ×
    TRT.TrainRouteTable → TS.Assertion-set
  generateSignalAspectAssertions(d, ora, trt) ≡
    generateSignalAspectAssertions(SL.allSignals(d), ora, trt),

  generateSignalAspectAssertions :
    SL.SignalId-set × ORA.ObjectRelayAssociations ×
    TRT.TrainRouteTable → TS.Assertion-set
  generateSignalAspectAssertions(allSignals, ora, trt) ≡
    if allSignals = {} then {} else
      let signalId = hd allSignals,
          allSignals' = allSignals \ {signalId}
      in {generateSignalAspectAssertion(signalId, ora)} ∪
          -- Recursive call
          generateSignalAspectAssertions(allSignals', ora, trt)
    end
    end,

  generateSignalAspectAssertion : SL.SignalId ×
    ORA.ObjectRelayAssociations → TS.Assertion
  generateSignalAspectAssertion(signalId, a) ≡
    ("signal_" ^ signalId,
     genSignalAspectFormula(signalId, a)),

  genSignalAspectFormula : SL.SignalId ×
    ORA.ObjectRelayAssociations → T.LTLformula
  genSignalAspectFormula(signalId, a) ≡
    let gr = ORA.signalRelayAssoc(a)(signalId, ORA.gr),
        re = ORA.signalRelayAssoc(a)(signalId, ORA.re)
    in T.g(T.impl(T.ltrl("idle"),
                  T.neg(T.conj({T.ltrl(gr), T.ltrl(re)}))))
    end,

  -- Signal locking
  generateSignalLockingAssertions :
    SL.Diagram × ORA.ObjectRelayAssociations ×
    TRT.TrainRouteTable → TS.Assertion-set
  generateSignalLockingAssertions(d, ora, trt) ≡
    let allSignals = SL.allSignals(d)
    in generateSignalLockingAssertions(allSignals, ora, trt)
    end,

```

```

generateSignalLockingAssertions :
  SL.SignalId-set × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → TS.Assertion-set
generateSignalLockingAssertions(signals, ora, trt) ≡
  if signals = {} then {} else
    let s = hd signals,
        signals' = signals \ {s}
    in {generateSignalLockingAssertion(s, ora, trt)} ∪
        -- Recursive Call
        generateSignalLockingAssertions(signals', ora, trt)
    end
  end,

generateSignalLockingAssertion :
  SL.SignalId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → TS.Assertion
generateSignalLockingAssertion(signalId, ora, trt) ≡
  ("signal_exp_" ^ signalId,
   genSigFormula(signalId, ora, trt)),

genSigFormula :
  SL.SignalId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.LTLformula
genSigFormula(signalId, ora, trt) ≡
  let coveredRoutes = getRoutesCoveredBySignal(signalId, trt),
      grVarId = ORA.signalRelayAssoc(ora)(signalId, ORA.gr)
  in T.g(T.impl(T.conj({T.ltl("idle"), T.ltl(grVarId)}),
               T.disj(getReservations(
                 coveredRoutes, signalId, ora, trt))))
  end,

getReservations : T.TrainRouteId-set ×
  SL.SignalId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.LTLformula-set
getReservations(trainRoutes, signalId, a, trt) ≡
  if trainRoutes = {} then {} else
    let routeId = hd trainRoutes,
        trainRoutes' = trainRoutes \ {routeId},
        tracksFree = getReqFreeTracks(trt, routeId),
        redSignals = getSignalAspectRoutes1(routeId, TRT.re, trt),
        nonConflictRoutes = getNonConflictingRoutes(
          TRT.mutuallyBlocking(trt(routeId)), routeId)
    in if getIfGreenSignalIsMoreThanOnce1(
        signalId, dom trt, trt, 0) > 1
    then
      {T.paren(T.conj({T.neg(T.ltl(
        ORA.routeRelayAssoc(a)(routeId))))}
        ∪
        routePointPosConfig(routeId, a, trt) ∪
        getTrackRelays(tracksFree, a) ∪
        getSignalRelays(redSignals, ORA.re, a) ∪
        {T.disj(getRouteRelayPointPosConfigSignal(
          nonConflictRoutes, routeId, a, trt))})}
      else
      {T.paren(T.conj({T.neg(T.ltl(

```

```

ORA.routeRelayAssoc(a)(routeId)))} ∪
routePointPosConfig(routeId, a, trt) ∪
getTrackRelays(tracksFree, a) ∪
getSignalRelays(redSignals, ORA.re, a)))}
end
∪
getReservations(trainRoutes', signalId, a, trt)
end
end,

getRouteRelayPointPosConfigSignal : T.TrainRouteId-set ×
T.TrainRouteId × ORA.ObjectRelayAssociations ×
TRT.TrainRouteTable → T.LTLformula-set
getRouteRelayPointPosConfigSignal(routeIds, currentRouteId, a, trt) ≡
if routeIds = {} then {} else
  let routeId = hd routeIds, routeIds' = routeIds \ {routeId},
  signalsGr =
    getSignalAspectRoutes(routeId, TRT.gr, trt),
  signalsRe =
    getSignalAspectRoutes(routeId, TRT.re, trt),
  signalRED =
    getSignalAspectRoutes(currentRouteId, TRT.re, trt)
  in {T.paren(T.disj({
    T.conj(getSignalRelays(signalRED, ORA.re, a)),
    T.paren(T.conj({T.neg(T.ltrl(
      ORA.routeRelayAssoc(a)(routeId)))} ∪
      routePointPosConfig(routeId, a, trt) ∪
      getSignalRelays(signalsGr, ORA.gr, a) ∪
      getSignalRelays(signalsRe, ORA.re, a)))
    })))}
  ∪ getRouteRelayPointPosConfigSignal(
    routeIds', currentRouteId, a, trt)
end
end,

getTrackRelays : SL.TrackId-set ×
ORA.ObjectRelayAssociations → T.LTLformula-set
getTrackRelays(tIds, a) ≡
if tIds = {} then {} else
  let tId = hd tIds, tIds' = tIds \ {tId}
  in {T.ltrl(ORA.getTrackRelay(tId, a))} ∪
    getTrackRelays(tIds', a)
end
end,

getSignalRelays : SL.SignalId-set × ORA.Lamp ×
ORA.ObjectRelayAssociations → T.LTLformula-set
getSignalRelays(signals, asp, a) ≡
if signals = {} then {} else
  let signalId = hd signals,
  signals' = signals \ {signalId}
  in {T.ltrl(ORA.signalRelayAssoc(a)(signalId, asp))} ∪
    getSignalRelays(signals', asp, a)
end
end,

```

```

getSignalAspectRoutes :
  T.TrainRouteId × TRT.Aspect ×
  TRT.TrainRouteTable → SL.SignalId-set
getSignalAspectRoutes(routeId, aspect, trt) ≡
  {signalId | signalId : SL.SignalId •
    signalId ∈ dom TRT.signals(trt(routeId)) ∧
    TRT.signals(trt(routeId))(signalId) = aspect},

getSignalAspectRoutes1 :
  T.TrainRouteId × TRT.Aspect ×
  TRT.TrainRouteTable → SL.SignalId-set
getSignalAspectRoutes1(routeId, aspect, trt) ≡
  {signalId | signalId : SL.SignalId •
    signalId ∈ dom TRT.signals(trt(routeId)) ∧
    TRT.signals(trt(routeId))(signalId) = aspect ∧
    ~isSignalGreenInNonConflictingRoute(
      routeId, dom trt, signalId, trt)},

isSignalGreenInNonConflictingRoute :
  T.TrainRouteId × T.TrainRouteId-set ×
  SL.SignalId × TRT.TrainRouteTable → Bool
isSignalGreenInNonConflictingRoute(routeId, allRoutes, signalId, trt) ≡
  if allRoutes = {} then false else
    let curRouteId = hd allRoutes,
        allRoutes' = allRoutes \ {curRouteId},
        mutualBlock = TRT.mutuallyBlocking(trt(routeId)),
        nonConflict = getNonConflictingRoutes(mutualBlock, routeId)
    in containsTrueInSet(nonConflict, signalId, trt)
  end
end,

containsTrueInSet :
  T.TrainRouteId-set × SL.SignalId ×
  TRT.TrainRouteTable → Bool
containsTrueInSet(allNonConflictRoutes, signalId, trt) ≡
  let set = isSignalGreenInNonConflictingRoutes(
    allNonConflictRoutes, signalId, trt),
    set' = set \ {hd set}
  in hd set
end,

isSignalGreenInNonConflictingRoutes : T.TrainRouteId-set ×
  SL.SignalId × TRT.TrainRouteTable → Bool-set
isSignalGreenInNonConflictingRoutes(allNonConflictRoutes, signalId, trt) ≡
  if allNonConflictRoutes = {} then {false} else
    let routeId = hd allNonConflictRoutes,
        allNonConflictRoutes' = allNonConflictRoutes \ {routeId}
    in if TRT.signals(trt(routeId))(signalId) = TRT.gr then {true}
       else {false}
    end ∪ isSignalGreenInNonConflictingRoutes(
      allNonConflictRoutes', signalId, trt)
  end
end,

```

```

getRoutesCoveredBySignal : SL.SignalId ×
  TRT.TrainRouteTable → T.TrainRouteId-set
getRoutesCoveredBySignal(signalId, trt) ≡
  {routeId | routeId : T.TrainRouteId •
    routeId ∈ dom trt ∧
    signalId ∈ dom TRT.signals(trt(routeId)) ∧
    TRT.signals(trt(routeId))(signalId) = TRT.gr},

getReqFreeTracks : TRT.TrainRouteTable ×
  T.TrainRouteId → T.VarId-set
getReqFreeTracks(trt, routeId) ≡
  {trackId | trackId : SL.TrackId •
    trackId ∈ dom TRT.trainDetection(trt(routeId)) ∧
    TRT.trainDetection(trt(routeId))(trackId) },

getIfGreenSignalIsMoreThanOnce1 : SL.SignalId ×
  T.TrainRouteId-set × TRT.TrainRouteTable × Nat → Nat
getIfGreenSignalIsMoreThanOnce1(signalId, allRoutes, trt, counter) ≡
  if allRoutes = {} then counter else
    let routeId = hd allRoutes,
      allRoutes' = allRoutes \ {routeId}
    in if TRT.signals(trt(routeId))(signalId) = TRT.gr
      then getIfGreenSignalIsMoreThanOnce1(
        signalId, allRoutes', trt, counter + 1)
      else getIfGreenSignalIsMoreThanOnce1(
        signalId, allRoutes', trt, counter)
    end
  end
end

value -- 10.6 Signal Release
genSignalReleaseAssertions : ORA.ObjectRelayAssociations ×
  IP.InterlockingPlan → TS.Assertion-set
genSignalReleaseAssertions(ora, ip) ≡
  let trt = IP.trt(ip),
    routeIds = dom trt
  in genSignalReleaseAssertions(dom IP.trt(ip), ora, trt)
end,

genSignalReleaseAssertions :
  T.TrainRouteId-set × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → TS.Assertion-set
genSignalReleaseAssertions(routeIds, ora, trt) ≡
  if routeIds = {} then {} else
    let routeId = hd routeIds,
      routeIds' = routeIds \ {routeId}
    in {generateSignalReleaseAssertion(routeId, ora, trt)} ∪
      -- Recursive Call
      genSignalReleaseAssertions(routeIds', ora, trt)
    end
  end,

generateSignalReleaseAssertion :
  T.TrainRouteId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → TS.Assertion

```

```

generateSignalReleaseAssertion(routeId, ora, trt) ≡
  ("signal_release_" ^ routeId,
   genSignalReleaseFormula(routeId, ora, trt)),

genSignalReleaseFormula :
  T.TrainRouteId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.LTLformula
genSignalReleaseFormula(routeId, ora, trt) ≡
  let (sId, tId) = TRT.signalRelease(trt(routeId)),
      trackRelayId = ORA.trackRelayAssoc(ora)(tId),
      signalAspect = ORA.signalRelayAssoc(ora)(sId, ORA.re)
  in
    T.g(T.impl(T.conj({T.ltrl("idle"),
                       T.neg(T.ltrl(trackRelayId))}),
            T.ltrl(signalAspect)))
  end

value -- 10.7 Conflicting Routes
genConflictingRoutesAssertions : ORA.ObjectRelayAssociations ×
  IP.InterlockingPlan → TS.Assertion-set
genConflictingRoutesAssertions(ora, ip) ≡
  let trt = IP.trt(ip),
      routeIds = dom trt
  in genConflictingRoutesAssertions(routeIds, ora, trt)
  end,

genConflictingRoutesAssertions :
  T.TrainRouteId-set × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → TS.Assertion-set
genConflictingRoutesAssertions(routeIds, ora, trt) ≡
  if routeIds = {} then {} else
    let routeId = hd routeIds,
        routeIds' = routeIds \ {routeId}
    in {generateConflictingRouteAssertion(routeId, ora, trt)} ∪
        -- Recursive Call
        genConflictingRoutesAssertions(routeIds', ora, trt)
    end
  end,

generateConflictingRouteAssertion :
  T.TrainRouteId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → TS.Assertion
generateConflictingRouteAssertion(routeId, ora, trt) ≡
  ("conflicting_route_" ^ routeId,
   genConflictRouteFormula(routeId, ora, trt)
  ),

genConflictRouteFormula :
  T.TrainRouteId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.LTLformula
genConflictRouteFormula(routeId, ora, trt) ≡
  T.g(T.impl(T.conj(
    {T.neg(T.ltrl(ORA.routeRelayAssoc(ora)(routeId)))} ∪
    routePointPosConfig(routeId, ora, trt)),
    T.conj(getRouteRelayAndPointPosConfig(

```

```

getConflictingRoutes(TRT.mutuallyBlocking(trt(routeId)),
  routeId), routeId, ora, trt))),

getConflictingRoutes : (T.TrainRouteId  $\overline{m}$  Bool)  $\times$ 
  T.TrainRouteId  $\rightarrow$  T.TrainRouteId-set
getConflictingRoutes(mutuallyBlocking, routeId)  $\equiv$ 
  {routeId | routeId : T.TrainRouteId •
    routeId  $\in$  dom mutuallyBlocking  $\wedge$ 
    mutuallyBlocking(routeId)},

getRouteRelayAndPointPosConfig : T.TrainRouteId-set  $\times$ 
  T.TrainRouteId  $\times$  ORA.ObjectRelayAssociations  $\times$ 
  TRT.TrainRouteTable  $\rightarrow$  T.LTLformula-set
getRouteRelayAndPointPosConfig(routeIds, routeId, ora, trt)  $\equiv$ 
  if routeIds = {} then {} else
    let currentRouteId = hd routeIds,
        routeIds' = routeIds \ {currentRouteId}
    in if currentRouteId = routeId then {} else
      {T.neg(T.conj({T.neg(T.ltl(
        ORA.routeRelayAssoc(ora)(currentRouteId))))}
         $\cup$  routePointPosConfig(currentRouteId, ora, trt)
      )}}
    end  $\cup$  getRouteRelayAndPointPosConfig(
      routeIds', routeId, ora, trt)
  end
end

value -- 10.8 Train Route Release
genTrainRouteReleaseAssertions : ORA.ObjectRelayAssociations  $\times$ 
  IP.InterlockingPlan  $\rightarrow$  TS.Assertion-set
genTrainRouteReleaseAssertions(ora, ip)  $\equiv$ 
  let trt = IP.trt(ip),
      routeIds = dom trt
  in genTrainRouteReleaseAssertions(routeIds, ora, trt)
end,

genTrainRouteReleaseAssertions :
  T.TrainRouteId-set  $\times$  ORA.ObjectRelayAssociations  $\times$ 
  TRT.TrainRouteTable  $\rightarrow$  TS.Assertion-set
genTrainRouteReleaseAssertions(routeIds, ora, trt)  $\equiv$ 
  if routeIds = {} then {} else
    let routeId = hd routeIds,
        routeIds' = routeIds \ {routeId}
    in {generateRouteReleaseAssertion(routeId, ora, trt)}  $\cup$ 
      -- Recursive Call
      genTrainRouteReleaseAssertions(routeIds', ora, trt)
    end
  end,

generateRouteReleaseAssertion :
  T.TrainRouteId  $\times$  ORA.ObjectRelayAssociations  $\times$ 
  TRT.TrainRouteTable  $\rightarrow$  TS.Assertion
generateRouteReleaseAssertion(routeId, ora, trt)  $\equiv$ 
  ("train_route_release_" ^ routeId,
    genRouteReleaseFormula(routeId, ora, trt)),

```

```

trackRelayState : SL.TrackId × TRT.RelayState ×
  ORA.ObjectRelayAssociations → T.LTLformula
trackRelayState(tId, rs, ora) ≡
  let rId = ORA.trackRelayAssoc(ora)(tId)
  in if rs = TRT.dropped then T.neg(T.ltrl(rId))
    else T.ltrl(rId) end
end,

genPointReqPos : (SL.PointMachineId × T.PointsState)-set ×
  ORA.ObjectRelayAssociations → T.LTLformula-set
genPointReqPos(points, ora) ≡
  if points = {} then {}
  else let (pId, ps) = hd points,
    points' = points \ {(pId, ps)}
    in if ps = T.arbitrary then {}
      else {T.ltrl(ORA.pointRelayAssoc(ora)(pId, ps))}
      end ∪ genPointReqPos(points', ora)
  end
end,

genRouteReleaseFormula :
  T.TrainRouteId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.LTLformula
genRouteReleaseFormula(routeId, ora, trt) ≡
  let routeRelay = ORA.routeRelayAssoc(ora)(routeId),
    route_not_locked = T.ltrl(routeRelay),
    route_locked = T.neg(route_not_locked),

    requiredPointPositions = T.conj(
      genPointReqPos(mapToSet(
        TRT.points(trt(routeId))), ora ),

    (initiation, dissolve) =
      TRT.trainRouteRelease(trt(routeId)),
    ((init_tId1, init_rs1), (init_tId2, init_rs2)) = initiation,
    ((diss_tId1, diss_rs1), (diss_tId2, diss_rs2)) = dissolve,

    -- Initiation trigger
    init_tr1 = trackRelayState(init_tId1, init_rs1, ora),
    init_tr2 = trackRelayState(init_tId2, init_rs2, ora),
    init = T.conj( {init_tr1, init_tr2} ),

    -- Dissolve trigger
    diss_tr1 = trackRelayState(diss_tId1, diss_rs1, ora),
    diss_tr2 = trackRelayState(diss_tId2, diss_rs2, ora),
    diss = T.conj( {diss_tr1, diss_tr2} )
  in T.g( T.impl( T.conj({
    route_not_locked,
    T.x(T.conj({
      route_locked,
      requiredPointPositions,
      T.f(route_not_locked) }) ) ),
    T.x(T.u(route_locked, T.conj({
      route_locked,

```



```

        init,
        T.x(T.u(route_locked, T.conj({
            route_locked, diss }))) ) })
    ))
end

value -- AUX --
-- helper function for point position and train route release
mapToSet : (SL.PointMachineId  $\rightsquigarrow$  T.PointsState)  $\rightarrow$ 
            (SL.PointMachineId  $\times$  T.PointsState)-set
mapToSet(map)  $\equiv$  mapToSet(dom map, map),

mapToSet : SL.PointMachineId-set  $\times$ 
            (SL.PointMachineId  $\rightsquigarrow$  T.PointsState)  $\rightarrow$ 
            (SL.PointMachineId  $\times$  T.PointsState)-set
mapToSet(keys, map)  $\equiv$ 
    if keys = {} then {} else
        let key = hd keys, keys' = keys \ {key}
        in {(key, map(key))}  $\cup$  mapToSet(keys, map)
    end
end,

getNonConflictingRoutes : (T.TrainRouteId  $\rightsquigarrow$  Bool)  $\times$ 
    T.TrainRouteId  $\rightarrow$  T.TrainRouteId-set
getNonConflictingRoutes(mutuallyBlocking, routeId)  $\equiv$ 
    {routeId | routeId : T.TrainRouteId •
        routeId  $\in$  dom mutuallyBlocking  $\wedge$ 
         $\sim$ mutuallyBlocking(routeId)}
end

```

E.2.4 State Generator

context: TS, SL, TMA, IP

scheme StateGenerator =

```

class
    value /* Generator functions for model catalog */
    genConVars :
        (SL.TrackId  $\times$  SL.TrackId)-set  $\times$  SL.Diagram  $\rightarrow$ 
        (SL.TrackId  $\times$  SL.TrackId)  $\rightsquigarrow$  T.VarId
    genConVars(nbs, d)  $\equiv$ 
        if nbs = {} then [] else
            let (nb1, nb2) = hd nbs, nbs' = nbs \ {(nb1, nb2)}
            in if SL.isLine(nb1, d)  $\vee$  SL.isLine(nb2, d)
                then genConVars(nbs', d)
                else genConVar(nb1, nb2)  $\cup$  genConVars(nbs', d)
            end
        end
    end,
end

genConVar : SL.TrackId  $\times$  SL.TrackId  $\rightarrow$ 

```

```

(SL.TrackId × SL.TrackId)  $\overrightarrow{m}$  T.VarId
genConVar(id1, id2)  $\equiv$  [(id1, id2)  $\mapsto$  "con_"  $\wedge$  id1  $\wedge$  " "  $\wedge$  id2],

genCntVars : SL.TrackId-set × SL.Diagram  $\rightarrow$  SL.TrackId  $\overrightarrow{m}$  T.VarId
genCntVars(ids, d)  $\equiv$ 
  if ids = {} then [] else
    let id = hd ids, ids' = ids \ {id}
    in if SL.isLine(id, d) then []
       else genCntVar(id)
       end  $\cup$  genCntVars(ids', d)
    end
  end,

genCntVar : SL.TrackId  $\rightarrow$  SL.TrackId  $\overrightarrow{m}$  T.VarId
genCntVar(id)  $\equiv$  [id  $\mapsto$  "cnt_"  $\wedge$  id],

genDirFwdVars : SL.TrackId-set ×
  SL.Diagram  $\rightarrow$  SL.TrackId  $\overrightarrow{m}$  T.VarId
genDirFwdVars(ids, d)  $\equiv$ 
  if ids = {} then [] else
    let id = hd ids, ids' = ids \ {id}
    in if SL.isLine(id, d) then []
       else genFwdVar(id)
       end  $\cup$  genDirFwdVars(ids', d)
    end
  end,

genFwdVar : SL.TrackId  $\rightarrow$  SL.TrackId  $\overrightarrow{m}$  T.VarId
genFwdVar(id)  $\equiv$  [id  $\mapsto$  "fwd_"  $\wedge$  id],

genDirBwdVars : SL.TrackId-set ×
  SL.Diagram  $\rightarrow$  SL.TrackId  $\overrightarrow{m}$  T.VarId
genDirBwdVars(ids, d)  $\equiv$ 
  if ids = {} then [] else
    let id = hd ids, ids' = ids \ {id}
    in if SL.isLine(id, d) then []
       else genBwdVar(id)
       end  $\cup$  genDirBwdVars(ids', d)
    end
  end,

genBwdVar : SL.TrackId  $\rightarrow$  SL.TrackId  $\overrightarrow{m}$  T.VarId
genBwdVar(id)  $\equiv$  [id  $\mapsto$  "bwd_"  $\wedge$  id],

genDirections : SL.Diagram  $\rightarrow$ 
  (SL.TrackId × SL.TrackId)  $\overrightarrow{m}$  T.Direction
genDirections(d)  $\equiv$ 
  let nbs = SL.getNeighboursOf(SL.line(d), d), nb = hd nbs
  in dirTraverse({nb}, {SL.line(d)}, d)  $\uparrow$ 
     dirSourceAndSinks(nb, nbs, d)
  end

value /* Helper functions for generating the directions */
dirSourceAndSinks : SL.TrackId ×
  SL.TrackId-set × SL.Diagram  $\rightarrow$ 

```

```

    (SL.TrackId × SL.TrackId)  $\rightsquigarrow$  T.Direction
    dirSourceAndSinks(nb, nbs, d)  $\equiv$ 
        dirSinks(nbs, d)  $\dagger$  genDirection(SL.line(d), nb),

    dirSinks : SL.TrackId-set × SL.Diagram  $\rightarrow$ 
        (SL.TrackId × SL.TrackId)  $\rightsquigarrow$  T.Direction
    dirSinks(nbs, d)  $\equiv$ 
        if nbs = {} then []
        else let nb = hd nbs, nbs' = nbs \ {nb}
            in dirSinks(nbs', d)  $\dagger$  genDirection(nb, SL.line(d))
        end
    end,

    dirTraverse : SL.TrackId-set ×
        SL.TrackId-set × SL.Diagram  $\rightarrow$ 
        (SL.TrackId × SL.TrackId)  $\rightsquigarrow$  T.Direction
    dirTraverse(ids, visited, d)  $\equiv$ 
        if ids = {} then []
        else let id = hd ids,
            nbs = SL.getNeighboursOf(id, d),
            visited' = visited  $\cup$  {id},
            ids' = (nbs  $\cup$  ids) \ visited'
            in dirNbs(id, nbs \ visited')  $\dagger$ 
                dirTraverse(ids', visited', d)
            end
        end,

    dirNbs : SL.TrackId × SL.TrackId-set  $\rightarrow$ 
        (SL.TrackId × SL.TrackId)  $\rightsquigarrow$  T.Direction
    dirNbs(id, nbs)  $\equiv$ 
        if nbs = {} then []
        else let nb = hd nbs, nbs' = nbs \ {nb}
            in genDirection(id, nb)  $\cup$  dirNbs(id, nbs')
            end
        end,

    genDirection : SL.TrackId × SL.TrackId  $\rightarrow$ 
        (SL.TrackId × SL.TrackId)  $\rightsquigarrow$  T.Direction
    genDirection(id, nb)  $\equiv$  [(id, nb)  $\mapsto$  T.fwd, (nb, id)  $\mapsto$  T.bwd]

value /* Generator function for state */
    generateStateSpace : IP.InterlockingPlan ×
        TMA.TrainMovementAssociations  $\rightarrow$  TS.Var-set
    generateStateSpace(ip, tma)  $\equiv$ 
        genBoolVars(rng TMA.connectionVars(tma))  $\cup$ 
        genNatVars(rng TMA.counterVars(tma))  $\cup$ 
        genBoolVars(rng TMA.directionFwdVars(tma))  $\cup$ 
        genBoolVars(rng TMA.directionBwdVars(tma))
    pre IP.isWfInterlockingPlan(ip)  $\wedge$ 
        TMA.isWfTrainMovementAssociations(IP.sld(ip), tma),

    genBoolVars : SL.TrackId-set  $\rightarrow$  TS.Var-set
    genBoolVars(ids)  $\equiv$ 
        if ids = {} then {} else
            let id = hd ids, ids' = ids \ {id}

```

```

    in genBoolVars(ids')  $\cup$  {TS.mk_Var(id, TS.mk_BoolVal(false))}
  end
end,

genNatVars : SL.TrackId-set  $\rightarrow$  TS.Var-set
genNatVars(ids)  $\equiv$ 
  if ids = {} then {} else
    let id = hd ids, ids' = ids \ {id}
    in genNatVars(ids')  $\cup$  {TS.mk_Var(id, TS.mk_NatVal(0))}
  end
end

end

```

E.2.5 Transition Generator

context: OXTG, TMTG

scheme TransitionGenerator =

```

  class
    value
      generateTransitionSystem : ORA.ObjectRelayAssociations  $\times$ 
        IP.InterlockingPlan  $\times$  TMA.TrainMovementAssociations  $\times$ 
          Text-set  $\rightarrow$  TS.TransitionRule-set
      generateTransitionSystem(ora, ip, tma, buttons)  $\equiv$ 
        let d = IP.sld(ip),
            allSections = SL.allSections(d)
        in TMTG.genTrainMovementTransitions(ora, d, tma)  $\cup$ 
          OXTG.genOtherExternalTransitions(buttons, ora, ip)
        end
    pre IP.isWfInterlockingPlan(ip)  $\wedge$ 
        TMA.isWfTrainMovementAssociations(IP.sld(ip), tma)  $\wedge$ 
        ORA.isWfObjectRelayAssociations(ora, ip)

  end
end

```

E.2.5.1 Train Movement Transition Generator

context: ORA, TMA

scheme TrainMovementTransitionGenerator =

```

  class
    value
      genTrainMovementTransitions :
        ORA.ObjectRelayAssociations  $\times$  SL.Diagram  $\times$ 
          TMA.TrainMovementAssociations  $\rightarrow$  TS.TransitionRule-set
      genTrainMovementTransitions(ora, d, tma)  $\equiv$ 
        -- 6.3.1 Enter Station
        genEnterStationRules(ora, d, tma)  $\cup$ 

```

```

-- 6.3.2 Enter Track Section
genEnterSectionRules(ora, d, tma)  $\cup$ 
-- 6.3.3 Leave Station
genLeaveStationRules(ora, d, tma)  $\cup$ 
-- 6.3.4 Leave Track Section
genLeaveSectionRules(ora, d, tma)  $\cup$ 
-- 6.3.5 Change Direction
genChangeDirectionRules(ora, d, tma)

value -- 6.3.1 Enter Station
genEnterStationRules :
  ORA.ObjectRelayAssociations  $\times$  SL.Diagram  $\times$ 
  TMA.TrainMovementAssociations  $\rightarrow$  TS.TransitionRule-set
genEnterStationRules(ora, d, tma)  $\equiv$ 
  let line = SL.line(d),
      nbs = SL.getNeighboursOf(line, d)
  in genEnterStationRules(nbs, ora, d, tma)
end,

genEnterStationRules : SL.TrackId-set  $\times$ 
  ORA.ObjectRelayAssociations  $\times$  SL.Diagram  $\times$ 
  TMA.TrainMovementAssociations  $\rightarrow$  TS.TransitionRule-set
genEnterStationRules(nbs, ora, d, tma)  $\equiv$ 
  if nbs = {} then {} else
    let nb = hd nbs,
        nbs' = nbs  $\setminus$  {nb}
    in {genEnterStationRule(nb, ora, d, tma)}  $\cup$ 
        genEnterStationRules(nbs', ora, d, tma) -- Recursive Call
  end
end,

genEnterStationRule :
  SL.TrackId  $\times$  ORA.ObjectRelayAssociations  $\times$ 
  SL.Diagram  $\times$  TMA.TrainMovementAssociations  $\rightarrow$  TS.TransitionRule
genEnterStationRule(nb, ora, d, tma)  $\equiv$ 
  let line = SL.line(d),
      dir = TMA.getDirection(line, nb, tma),
      name = genEnterRuleName(line, nb),
      guard = T.and( { idle }  $\cup$ 
        proceedAspect(line, nb, ora, d) ),
      update = { systemBusy,
        addTrain(nb, tma),
        dropTrackRelay(nb, ora),
        setDirection(nb, dir, tma) }
  in TS.mk_TransitionRule(name, guard, update)
end

value -- 6.3.2 Enter Track Section
genEnterSectionRules : ORA.ObjectRelayAssociations  $\times$  SL.Diagram  $\times$ 
  TMA.TrainMovementAssociations  $\rightarrow$  TS.TransitionRule-set
genEnterSectionRules(ora, d, tma)  $\equiv$ 
  let nbs = { (tId1, tId2) | (tId1, tId2) : (SL.TrackId  $\times$  SL.TrackId)  $\bullet$ 
    (tId1, tId2)  $\in$  SL.neighbours(d)  $\wedge$ 
     $\sim$ SL.isLine(tId1, d)  $\wedge$   $\sim$ SL.isLine(tId2, d) }
  in genEnterSectionRules(nbs, ora, d, tma)

```

```

end,

genEnterSectionRules : (SL.TrackId × SL.TrackId)-set ×
  ORA.ObjectRelayAssociations × SL.Diagram ×
  TMA.TrainMovementAssociations → TS.TransitionRule-set
genEnterSectionRules(nbs, ora, d, tma) ≡
  if nbs = {} then {} else
    let (tId1, tId2) = hd nbs,
        nbs' = nbs \ {(tId1, tId2)}
    in -- From tId1 to tId2
        genEnterSectionRule(tId1, tId2, ora, d, tma) ∪
        -- From tId2 to tId1
        genEnterSectionRule(tId2, tId1, ora, d, tma) ∪
        -- Recursive Call
        genEnterSectionRules(nbs', ora, d, tma)
    end
  end,

genEnterSectionRule : SL.TrackId × SL.TrackId ×
  ORA.ObjectRelayAssociations × SL.Diagram ×
  TMA.TrainMovementAssociations → TS.TransitionRule-set
genEnterSectionRule(from, to, ora, d, tma) ≡
  -- 6.3.2.1 Facing Move - Stem Side
  if SL.isPoints(to, d) ∧ SL.isStemNb(to, from, d)
  then {genEnterLinearSectionRule(from, to, ora, d, tma)}

  -- 6.3.2.1 Facing Move - Branch Side
  elseif SL.isPoints(from, d) ∧ SL.isBranchNb(from, to, d)
  then {genEnterFacingBranchSideRule(from, to, ora, d, tma)}

  -- 6.3.2.1 Trailing Move - Stem Side
  elseif SL.isPoints(from, d) ∧ SL.isStemNb(from, to, d)
  then {genEnterLinearSectionRule(from, to, ora, d, tma)}

  -- 6.3.2.1 Trailing Move - Branch Side
  elseif SL.isPoints(to, d) ∧ SL.isBranchNb(to, from, d)
  then { -- Correct pos
        genEnterFacingBranchSideRule(from, to, ora, d, tma),
        -- Incorrect pos
        genEnterTrailingBranchSideRule(from, to, ora, d, tma)}

  else -- 6.3.2 Linear
    {genEnterLinearSectionRule(from, to, ora, d, tma)}
  end,

genEnterLinearSectionRule : SL.TrackId × SL.TrackId ×
  ORA.ObjectRelayAssociations × SL.Diagram ×
  TMA.TrainMovementAssociations → TS.TransitionRule
genEnterLinearSectionRule(from, to, ora, d, tma) ≡
  let dir = TMA.getDirection(from, to, tma),
      name = "enter_" ^ to ^ "_from_" ^ from,
      guard = T.and(
        { idle,
          trackOccupied(from, tma),
          direction(from, to, tma),

```

```

        notConnected(from, to, tma) } ∪
    -- If move passes signal
    if SL.hasSignal(from, to, d)
    then proceedAspect(from, to, ora, d) else {} end),
update = {
    systemBusy,
    addTrain(to, tma),
    dropTrackRelay(to, ora),
    connect(to, from, tma),
    setDirection(to, dir, tma) }
in TS.mk_TransitionRule(name, guard, update)
end,

genEnterFacingBranchSideRule : SL.TrackId × SL.TrackId ×
    ORA.ObjectRelayAssociations × SL.Diagram ×
    TMA.TrainMovementAssociations → TS.TransitionRule
genEnterFacingBranchSideRule(from, to, ora, d, tma) ≡
    let dir = TMA.getDirection(from, to, tma),
        name = "enter_" ^ to ^ "_from_" ^ from,
        guard = T.and(
            { idle,
              trackOccupied(from, tma),
              direction(from, to, tma),
              notConnected(from, to, tma),
              branchGuard(to, from, ora, d) } ∪
            -- If move passes signal
            if SL.hasSignal(from, to, d)
            then proceedAspect(from, to, ora, d) else {} end),
        update = {
            systemBusy,
            addTrain(to, tma),
            dropTrackRelay(to, ora),
            connect(to, from, tma),
            setDirection(to, dir, tma) }
    in TS.mk_TransitionRule(name, guard, update)
end,

genEnterTrailingBranchSideRule : SL.TrackId × SL.TrackId ×
    ORA.ObjectRelayAssociations × SL.Diagram ×
    TMA.TrainMovementAssociations → TS.TransitionRule
genEnterTrailingBranchSideRule(from, to, ora, d, tma) ≡
    let name = "enter_" ^ to ^ "_from_" ^ from ^ "_derail",
        guard = T.and(
            { idle,
              trackOccupied(from, tma),
              direction(from, to, tma),
              notConnected(from, to, tma),
              T.neg(branchGuard(from, to, ora, d)) } ∪
            -- If move passes signal
            if SL.hasSignal(from, to, d)
            then proceedAspect(from, to, ora, d) else {} end),
        update = {
            systemBusy,
            addTrain(to, tma),
            dropTrackRelay(to, ora),

```

```

        connect(to, from, tma) }
    in TS.mk_TransitionRule(name, guard, update)
end

value -- 6.3.3 Leave Station
genLeaveStationRules : ORA.ObjectRelayAssociations × SL.Diagram ×
    TMA.TrainMovementAssociations → TS.TransitionRule-set
genLeaveStationRules(ora, d, tma) ≡
    let line = SL.line(d),
        nbs = SL.getNeighboursOf(line, d)
    in genLeaveStationRules(nbs, ora, d, tma)
end,

genLeaveStationRules : SL.TrackId-set ×
    ORA.ObjectRelayAssociations × SL.Diagram ×
    TMA.TrainMovementAssociations → TS.TransitionRule-set
genLeaveStationRules(nbs, ora, d, tma) ≡
    if nbs = {} then {} else
        let nb = hd nbs,
            nbs' = nbs \ {nb}
        in {genLeaveStationRule(nb, ora, d, tma)} ∪
            genLeaveStationRules(nbs', ora, d, tma) -- Recursive Call
        end
    end,

genLeaveStationRule : SL.TrackId ×
    ORA.ObjectRelayAssociations × SL.Diagram ×
    TMA.TrainMovementAssociations → TS.TransitionRule
genLeaveStationRule(nb, ora, d, tma) ≡
    let line = SL.line(d),
        name = genLeaveRuleName(nb, SL.line(d)),
        guard =
            { idle,
              trackOccupied(nb, tma),
              direction(nb, line, tma) } ∪
            isTailOfTrainGuards(nb, line, d, tma),
        update =
            { systemBusy,
              drawTrackRelay(nb, ora),
              removeTrain(nb, tma),
              removeDirection(nb, line, tma) }
    in TS.mk_TransitionRule(name, T.and(guard), update)
end

value -- 6.3.4 Leave Track Section
genLeaveSectionRules :
    ORA.ObjectRelayAssociations × SL.Diagram ×
    TMA.TrainMovementAssociations → TS.TransitionRule-set
genLeaveSectionRules(ora, d, tma) ≡
    let nbs = { (tId1, tId2) | (tId1, tId2) : (SL.TrackId × SL.TrackId) •
        (tId1, tId2) ∈ SL.neighbours(d) ∧
        ~SL.isLine(tId1, d) ∧ ~SL.isLine(tId2, d) }
    in genLeaveSectionRules(nbs, d, ora, tma)
end,

```



```

genLeaveSectionRules : (SL.TrackId × SL.TrackId)-set ×
  SL.Diagram × ORA.ObjectRelayAssociations ×
  TMA.TrainMovementAssociations → TS.TransitionRule-set
genLeaveSectionRules(nbs, d, ora, tma) ≡
  if nbs = {} then {} else
    let (tId1, tId2) = hd nbs,
        nbs' = nbs \ {(tId1, tId2)}
    in {genLeaveSectionRule(tId1, tId2, d, ora, tma),
        genLeaveSectionRule(tId2, tId1, d, ora, tma)} ∪
        genLeaveSectionRules(nbs', d, ora, tma) -- Recursive Call
    end
  end,

genLeaveSectionRule : SL.TrackId × SL.TrackId ×
  SL.Diagram × ORA.ObjectRelayAssociations ×
  TMA.TrainMovementAssociations → TS.TransitionRule
genLeaveSectionRule(from, to, d, ora, tma) ≡
  -- 6.3.4.1 Leave towards points section
  if SL.isPoints(to, d)
  then genLeaveLinearSectionRule(from, to, d, ora, tma)

  -- 6.3.4.1 Leave points section at stem end
  elseif SL.isPoints(from, d) ∧ SL.isStemNb(from, to, d)
  then genLeaveLinearSectionRule(from, to, d, ora, tma)

  -- 6.3.4.1 Leave points section at branch end
  elseif SL.isPoints(from, d) ∧ SL.isBranchNb(from, to, d)
  then genLeavePointsBranchEnd(from, to, d, ora, tma)

  -- 6.3.4 Leave Linear Section
  else genLeaveLinearSectionRule(from, to, d, ora, tma)
  end,

genLeaveLinearSectionRule : SL.TrackId × SL.TrackId ×
  SL.Diagram × ORA.ObjectRelayAssociations ×
  TMA.TrainMovementAssociations → TS.TransitionRule
genLeaveLinearSectionRule(from, to, d, ora, tma) ≡
  let dir = TMA.getDirection(from, to, tma),
      name = genLeaveRuleName(from, to),
      guard = T.and({
        idle,
        direction(from, to, tma),
        connected(from, to, tma),
        trackOccupied(from, tma)
      }) ∪
        isTailOfTrainGuards(from, to, d, tma) ),
      update = {
        systemBusy,
        drawTrackRelay(from, ora),
        removeTrain(from, tma),
        removeConnection(from, to, tma),
        removeDirection(from, dir, tma)
      }
  in TS.mk_TransitionRule(name, guard, update)
  end,

```

```

genLeavePointsBranchEnd : SL.TrackId × SL.TrackId ×
  SL.Diagram × ORA.ObjectRelayAssociations ×
  TMA.TrainMovementAssociations → TS.TransitionRule
genLeavePointsBranchEnd(from, to, d, ora, tma) ≡
  let dir = TMA.getDirection(from, to, tma),
      name = genLeaveRuleName(from, to),
      guard = T.and({
        idle,
        direction(from, to, tma),
        connected(from, to, tma),
        trackOccupied(from, tma),
        branchGuard(from, to, ora, d)
      }) ∪
        isTailOfTrainGuards(from, to, d, tma) ),
      update = {
        systemBusy,
        drawTrackRelay(from, ora),
        removeTrain(from, tma),
        removeConnection(from, to, tma),
        removeDirection(from, dir, tma)
      }
  in TS.mk_TransitionRule(name, guard, update)
end

value -- 6.3.5 Change Direction
genChangeDirectionRules :
  ORA.ObjectRelayAssociations × SL.Diagram ×
  TMA.TrainMovementAssociations → TS.TransitionRule-set
genChangeDirectionRules(ora, d, tma) ≡
  let allSignalLocations = dom SL.signalLocations(d)
  in genChangeDirectionRules(allSignalLocations, ora, d, tma)
end,

genChangeDirectionRules : (SL.TrackId × SL.TrackId)-set ×
  ORA.ObjectRelayAssociations × SL.Diagram ×
  TMA.TrainMovementAssociations → TS.TransitionRule-set
genChangeDirectionRules(signalLocations, ora, d, tma) ≡
  if signalLocations = {} then {} else
    let (tId1, tId2) = hd signalLocations,
        signalLocations' = signalLocations \ {(tId1, tId2)}
    in -- Recursive Call
      genChangeDirectionRules(signalLocations', ora, d, tma) ∪
      if tId1 = SL.line(d) then {}
      else {genChangeDirectionRule(tId1, tId2, ora, d, tma)}
    end
  end
end,

genChangeDirectionRule : SL.TrackId × SL.TrackId ×
  ORA.ObjectRelayAssociations × SL.Diagram ×
  TMA.TrainMovementAssociations → TS.TransitionRule
genChangeDirectionRule(from, to, a, d, tma) ≡
  let curDir = TMA.directions(tma)(from, to),
      oppDir = T.oppositeDirection(curDir),

```

```

    name = "change_direction_at_" ^ from ^ "_towards_" ^ to,
    guard = { idle,
              trackOccupied(from, tma),
              direction(from, to, tma) } ∪
              proceedAspect(from, to, a, d) ∪
              isTailOfTrainGuards(from, to, d, tma),
    update = { setDirection(from, oppDir, tma),
               removeDirection(from, curDir, tma) }
  in TS.mk_TransitionRule(name, T.and(guard), update)
end

value -- AUX
isHeadOfTrainGuard : SL.TrackId × SL.TrackId ×
  TMA.TrainMovementAssociations → T.BooleanExp
isHeadOfTrainGuard(s, inFront, tma) ≡
  T.neg(T.literal(TMA.getConnectionVar(s, inFront, tma))),

isTailOfTrainGuards : SL.TrackId × SL.TrackId ×
  SL.Diagram × TMA.TrainMovementAssociations → T.BooleanExp-set
isTailOfTrainGuards(from, to, d, tma) ≡
  let nbs = SL.getNeighboursOf(from, d),
      otherNbs = {tId | tId : SL.TrackId •
                  tId ∈ nbs ∧ tId ≠ to ∧ ~SL.isLine(tId, d) }
  in isTailOfTrainGuards(otherNbs, from, tma)
end,

isTailOfTrainGuards : SL.TrackId-set × SL.TrackId ×
  TMA.TrainMovementAssociations → T.BooleanExp-set
isTailOfTrainGuards(nbs, from, tma) ≡
  if nbs = {} then {} else
    let nb = hd nbs,
        nbs' = nbs \ {nb}
    in {notConnected(from, nb, tma)} ∪
        isTailOfTrainGuards(nbs', from, tma) -- Recursive Call
    end
  end

value -- Name
genEnterRuleName : SL.TrackId × SL.TrackId → Text
genEnterRuleName(from, to) ≡ "enter_" ^ to ^ "_from_" ^ from,

genLeaveRuleName : SL.TrackId × SL.TrackId → Text
genLeaveRuleName(from, to) ≡ "leave_" ^ from ^ "_to_" ^ to

value -- Guards
idle : T.BooleanExp = T.literal("idle"),

trackOccupied : SL.TrackId ×
  TMA.TrainMovementAssociations → T.BooleanExp
trackOccupied(tId, tma) ≡
  T.equals(T.literal(TMA.getCounterVar(tId, tma)), T.literal("1")),

trackUnoccupied : SL.TrackId ×
  TMA.TrainMovementAssociations → T.BooleanExp
trackUnoccupied(tId, tma) ≡

```

```

T.equals(T.literal(TMA.getCounterVar(tId, tma)), T.literal("0")),

connected : SL.TrackId × SL.TrackId ×
  TMA.TrainMovementAssociations → T.BooleanExp
connected(tId1, tId2, tma) ≡
  T.literal(TMA.getConnectionVar(tId1, tId2, tma)),

notConnected : SL.TrackId × SL.TrackId ×
  TMA.TrainMovementAssociations → T.BooleanExp
notConnected(tId1, tId2, tma) ≡
  T.neg(T.literal(TMA.getConnectionVar(tId1, tId2, tma))),

direction : SL.TrackId × SL.TrackId ×
  TMA.TrainMovementAssociations → T.BooleanExp
direction(from, to, tma) ≡
  T.literal(TMA.getDirVar(from, to, tma)),

proceedAspect : SL.TrackId × SL.TrackId ×
  ORA.ObjectRelayAssociations × SL.Diagram → T.BooleanExp-set
proceedAspect(from, to, ora, d) ≡
  let sId = SL.signalLocations(d)(from, to),
      sId_gr = ORA.signalRelayAssoc(ora)(sId, ORA.gr),
      sId_re = ORA.signalRelayAssoc(ora)(sId, ORA.re)
  in { T.literal(sId_gr), T.neg(T.literal(sId_re)) }
  end,

branchGuard : SL.TrackId × SL.TrackId ×
  ORA.ObjectRelayAssociations ×
  SL.Diagram → T.BooleanExp
branchGuard(tId1, tId2, ora, d) ≡
  let points = if SL.isPoints(tId1, d) then tId1 else tId2 end,
      section = if SL.isPoints(tId1, d) then tId2 else tId1 end,
      branch = SL.branchNeighbours(d)(points, section)
  in T.literal(ORA.getPointRelay(points, branch, ora, d))
  end

value -- update
systemBusy : TS.Assignment =
  TS.mk_Assignment("idle", T.BoolAssign(false)),

drawTrackRelay : SL.TrackId ×
  ORA.ObjectRelayAssociations → TS.Assignment
drawTrackRelay(tId, ora) ≡
  let rId = ORA.getTrackRelay(tId, ora)
  in TS.mk_Assignment(rId, T.BoolAssign(true))
  end,

dropTrackRelay : SL.TrackId ×
  ORA.ObjectRelayAssociations → TS.Assignment
dropTrackRelay(tId, ora) ≡
  let rId = ORA.getTrackRelay(tId, ora)
  in TS.mk_Assignment(rId, T.BoolAssign(false))
  end,

addTrain : SL.TrackId ×

```

```

    TMA.TrainMovementAssociations → TS.Assignment
addTrain(tId, tma) ≡
    let cVar = TMA.getCounterVar(tId, tma)
    in TS.mk_Assignment(cVar, T.AddAssign(1))
end,

removeTrain : SL.TrackId ×
    TMA.TrainMovementAssociations → TS.Assignment
removeTrain(tId, tma) ≡
    let cVar = TMA.getCounterVar(tId, tma)
    in TS.mk_Assignment(cVar, T.SubAssign(1))
end,

setDirection : SL.TrackId × SL.TrackId ×
    TMA.TrainMovementAssociations → TS.Assignment
setDirection(from, to, tma) ≡
    let dVar = TMA.getDirVar(from, to, tma)
    in TS.mk_Assignment(dVar, T.BoolAssign(true))
end,

setDirection : SL.TrackId × T.Direction ×
    TMA.TrainMovementAssociations → TS.Assignment
setDirection(tId, dir, tma) ≡
    if dir = T.fwd
    then let dVar = TMA.getFwdDirVar(tId, tma)
         in TS.mk_Assignment(dVar, T.BoolAssign(true))
         end
    else let dVar = TMA.getBwdDirVar(tId, tma)
         in TS.mk_Assignment(dVar, T.BoolAssign(true))
         end
    end,

removeDirection : SL.TrackId × SL.TrackId ×
    TMA.TrainMovementAssociations → TS.Assignment
removeDirection(from, to, tma) ≡
    let dVar = TMA.getDirVar(from, to, tma)
    in TS.mk_Assignment(dVar, T.BoolAssign(false))
end,

removeDirection : SL.TrackId × T.Direction ×
    TMA.TrainMovementAssociations → TS.Assignment
removeDirection(tId, dir, tma) ≡
    if dir = T.fwd
    then let dVar = TMA.getFwdDirVar(tId, tma)
         in TS.mk_Assignment(dVar, T.BoolAssign(false))
         end
    else let dVar = TMA.getBwdDirVar(tId, tma)
         in TS.mk_Assignment(dVar, T.BoolAssign(false))
         end
    end,

connect : SL.TrackId × SL.TrackId ×
    TMA.TrainMovementAssociations → TS.Assignment
connect(tId1, tId2, tma) ≡
    let cVar = TMA.getConnectionVar(tId1, tId2, tma)

```

```

    in TS.mk_Assignment(cVar, T.BoolAssign(true))
  end,

  removeConnection : SL.TrackId × SL.TrackId ×
    TMA.TrainMovementAssociations → TS.Assignment
  removeConnection(tId1, tId2, tma) ≡
    let cVar = TMA.getConnectionVar(tId1, tId2, tma)
    in TS.mk_Assignment(cVar, T.BoolAssign(false))
  end

end

```

E.2.5.2 Other External Transitions Generator

context: ORA, TS

scheme OtherExternalTransitionsGenerator =

```

  class
    value
      genOtherExternalTransitions :
        Text-set × ORA.ObjectRelayAssociations ×
          IP.InterlockingPlan → TS.TransitionRule-set
      genOtherExternalTransitions(buttons, ora, ip) ≡
        -- 7.1 Button Behaviour
        genAllButtonRules(buttons) ∪
        -- 7.2 Points Behaviour
        genAllPointsOpRules(ora, ip)

    value -- 7.1 Button Behaviour
      genAllButtonRules : Text-set → TS.TransitionRule-set
      genAllButtonRules(bs) ≡
        if bs = {} then {} else
          let b = hd bs, bs' = bs \ {b}
          in {genButtonRule(b)} ∪ genAllButtonRules(bs')
        end
      end,

      genButtonRule : Text → TS.TransitionRule
      genButtonRule(b) ≡
        TS.mk_TransitionRule(
          "pushButton_" ^ b,
          T.literal("idle"),
          { busySystem, TS.mk_Assignment(b, T.BoolAssign(true)) } )

    value -- 7.2 Points Behaviour
      genAllPointsOpRules : ORA.ObjectRelayAssociations ×
        IP.InterlockingPlan → TS.TransitionRule-set
      genAllPointsOpRules(ora, ip) ≡
        let d = IP.sld(ip),
            allPoints = SL.allPoints(d)
        in genAllPointsOpRules(allPoints, ora, ip)
      end,

```

```

genAllPointsOpRules :
  SL.TrackId-set × ORA.ObjectRelayAssociations ×
  IP.InterlockingPlan → TS.TransitionRule-set
genAllPointsOpRules(allPoints, ora, ip) ≡
  if allPoints = {} then {} else
    let p = hd allPoints,
        allPoints' = allPoints \ {p}
    in genPointsOpRules(p, ora, ip) ∪
        genAllPointsOpRules(allPoints', ora, ip) -- Recursive Call
    end
  end,

genPointsOpRules :
  SL.TrackId × ORA.ObjectRelayAssociations ×
  IP.InterlockingPlan → TS.TransitionRule-set
genPointsOpRules(pId, ora, ip) ≡
  let d = IP.sld(ip),
      trt = IP.trt(ip),
      rlrs = getRouteLockingRelaysCoveringP(pId, ora, trt) in {

    -- From plus to intermediate
    TS.mk.TransitionRule(
      "plusToIntermediate" ^ pId,
      genPointsOpGuard(pId, T.plus, rlrs, ora, d),
      genPointsOpUpd(pId, T.plus, false, ora, d) ),

    -- From intermediate to plus
    TS.mk.TransitionRule(
      "intermediateToPlus" ^ pId,
      genPointsOpGuard(pId, rlrs, ora, d),
      genPointsOpUpd(pId, T.plus, true, ora, d) ),

    -- From minus to intermediate
    TS.mk.TransitionRule(
      "minusToIntermediate" ^ pId,
      genPointsOpGuard(pId, T.minus, rlrs, ora, d),
      genPointsOpUpd(pId, T.minus, false, ora, d) ),

    -- From intermediate to minus
    TS.mk.TransitionRule(
      "intermediateToMinus" ^ pId,
      genPointsOpGuard(pId, rlrs, ora, d),
      genPointsOpUpd(pId, T.minus, true, ora, d) )
  }
  end,

genPointsOpUpd : SL.TrackId × SL.Branch × Bool ×
  ORA.ObjectRelayAssociations × SL.Diagram → TS.MultipleAssignment
genPointsOpUpd(pId, branch, locked, ora, d) ≡
  { busySystem,
    TS.mk.Assignment( ORA.getPointRelay(pId, branch, ora, d),
                      T.BoolAssign(locked)) },

  -- From locked to intermediate

```

```

genPointsOpGuard : SL.TrackId × SL.Branch × T.VarId-set ×
  ORA.ObjectRelayAssociations × SL.Diagram → T.BooleanExp
genPointsOpGuard(pId, branch, rlrs, ora, d) ≡
  T.and({ T.literal("idle"),
          T.literal(ORA.getPointRelay(pId, branch, ora, d)),
          T.literal(ORA.trackRelayAssoc(ora)(pId))
        } ∪
        convRelayIdsToLiterals(rlrs)
  ),
  -- From intermediate to locked
  genPointsOpGuard : SL.TrackId × T.VarId-set ×
    ORA.ObjectRelayAssociations × SL.Diagram → T.BooleanExp
genPointsOpGuard(pId, rlrs, ora, d) ≡
  T.and({ T.literal("idle"),
          T.neg(T.literal(ORA.getPointRelay(pId, T.plus, ora, d))),
          T.neg(T.literal(ORA.getPointRelay(pId, T.minus, ora, d))),
          T.literal(ORA.trackRelayAssoc(ora)(pId))
        } ∪
        convRelayIdsToLiterals(rlrs)
  ),

convRelayIdsToLiterals : T.VarId-set → T.BooleanExp-set
convRelayIdsToLiterals(rIds) ≡
  if rIds = {} then {} else
    let rId = hd rIds, rIds' = rIds \ {rId}
    in {T.literal(rId)} ∪ convRelayIdsToLiterals(rIds')
  end
end,

getRouteLockingRelaysCoveringP :
  SL.TrackId × ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.VarId-set
getRouteLockingRelaysCoveringP(tId, ora, trt) ≡
  getRLRsCoveringP(dom ORA.routeRelayAssoc(ora), tId, ora, trt),

getRLRsCoveringP : T.TrainRouteId-set × SL.TrackId ×
  ORA.ObjectRelayAssociations × TRT.TrainRouteTable → T.VarId-set
getRLRsCoveringP(allRoutes, tId, ora, trt) ≡
  if allRoutes = {} then {} else
    let route = hd allRoutes, allRoutes' = allRoutes \ {route}
    in getRLRCoveringP(route, tId, ora, trt) ∪
       getRLRsCoveringP(allRoutes', tId, ora, trt)
  end
end,

getRLRCoveringP : T.TrainRouteId × SL.TrackId ×
  ORA.ObjectRelayAssociations ×
  TRT.TrainRouteTable → T.VarId-set
getRLRCoveringP(route, tId, ora, trt) ≡
  if TRT.trainDetection(trt(route))(tId)
  then {ORA.routeRelayAssoc(ora)(route)}
  else {}
end

```

value -- Aux functions


```

    busySystem : TS.Assignment = TS.mk_Assignment("idle", T.BoolAssign(false))

end

```

E.3 Associations

E.3.1 Objects

```

context: ObjectRelayAssociations
object ORA : ObjectRelayAssociations

```

```

context: TrainMovementAssociations
object TMA : TrainMovementAssociations

```

E.3.2 Object Relay Assocations

```

context: IP
scheme ObjectRelayAssociations =
  class
    type
      Lamp == gr | re,

      ObjectRelayAssociations ::
        routeRelayAssoc : T.TrainRouteId  $\rightsquigarrow$  T.VarId
        trackRelayAssoc : SL.TrackId  $\rightsquigarrow$  T.VarId
        pointRelayAssoc : SL.PointMachineId  $\times$  SL.Branch  $\rightsquigarrow$  T.VarId
        signalRelayAssoc : SL.SignalId  $\times$  Lamp  $\rightsquigarrow$  T.VarId

    value /* getters */
      getPointRelay : SL.TrackId  $\times$  SL.Branch  $\times$ 
        ObjectRelayAssociations  $\times$  SL.Diagram  $\rightarrow$  T.VarId
      getPointRelay(tId, branch, a, d)  $\equiv$ 
        let pId = SL.trackPointMachine(d)(tId)
        in pointRelayAssoc(a)(pId, branch)
      end,

      getTrackRelay : SL.TrackId  $\times$  ObjectRelayAssociations  $\rightarrow$  T.VarId
      getTrackRelay(tId, a)  $\equiv$  trackRelayAssoc(a)(tId)

    value /* Well-formedness */
      isWfObjectRelayAssociations : ObjectRelayAssociations  $\times$ 
        IP.InterlockingPlan  $\rightarrow$  Bool

```

```

isWfObjectRelayAssociations(ora, ip)  $\equiv$ 
  let d = IP.sld(ip),
      trt = IP.trt(ip)
  in noRelayVariableOverlap(ora)  $\wedge$ 
      isWfRouteAssoc(trt, ora)  $\wedge$ 
      isWfTrackAssoc(d, ora)  $\wedge$ 
      isWfPointAssoc(d, ora)  $\wedge$ 
      isWfSignalAssoc(d, ora)
  end
pre IP.isWfInterlockingPlan(ip),

noRelayVariableOverlap : ObjectRelayAssociations  $\rightarrow$  Bool
noRelayVariableOverlap(ora)  $\equiv$ 
  let routeVars = rng routeRelayAssoc(ora),
      trackVars = rng trackRelayAssoc(ora),
      pointVars = rng pointRelayAssoc(ora),
      signalVars = rng signalRelayAssoc(ora)
  in routeVars  $\cap$  trackVars = {}  $\wedge$ 
      routeVars  $\cap$  pointVars = {}  $\wedge$ 
      routeVars  $\cap$  signalVars = {}  $\wedge$ 
      trackVars  $\cap$  pointVars = {}  $\wedge$ 
      trackVars  $\cap$  signalVars = {}  $\wedge$ 
      pointVars  $\cap$  signalVars = {}
  end

value /* Well-formedness of routeRelayAssoc */
isWfRouteAssoc : TRT.TrainRouteTable  $\times$ 
  ObjectRelayAssociations  $\rightarrow$  Bool
isWfRouteAssoc(trt, ora)  $\equiv$ 
  dom routeRelayAssoc(ora) = dom trt

value /* Well-formedness of trackRelayAssoc */
isWfTrackAssoc : SL.Diagram  $\times$ 
  ObjectRelayAssociations  $\rightarrow$  Bool
isWfTrackAssoc(d, ora)  $\equiv$ 
  let allTrackSections = SL.allLinears(d)  $\cup$  SL.allPoints(d)
  in dom trackRelayAssoc(ora) = allTrackSections
  end

value /* Well-formedness of pointRelayAssoc */
isWfPointAssoc : SL.Diagram  $\times$ 
  ObjectRelayAssociations  $\rightarrow$  Bool
isWfPointAssoc(d, ora)  $\equiv$ 
  existingPointMachines(d, ora)  $\wedge$ 
  differentPointRelays(ora),

existingPointMachines : SL.Diagram  $\times$ 
  ObjectRelayAssociations  $\rightarrow$  Bool
existingPointMachines(d, ora)  $\equiv$ 
  (  $\forall$  (pmId, b) : SL.PointMachineId  $\times$  SL.Branch •
    (pmId, b)  $\in$  dom pointRelayAssoc(ora)  $\Rightarrow$ 
    pmId  $\in$  dom SL.pointMachineTrack(d)
  )  $\wedge$ 
  (  $\forall$  pmId : SL.PointMachineId •

```

```

    pmId ∈ dom SL.pointMachineTrack(d) ⇒
      ( ∃ (pmId', b) : SL.PointMachineId × SL.Branch •
        (pmId', b) ∈ dom pointRelayAssoc(ora) ∧
        pmId = pmId' )
  ),

differentPointRelays : ObjectRelayAssociations → Bool
differentPointRelays(ora) ≡
  ( ∀ (pmId, b) : SL.PointMachineId × SL.Branch •
    (pmId, b) ∈ dom pointRelayAssoc(ora) ⇒
      ~ ( ∃ (pmId', b') : SL.PointMachineId × SL.Branch •
        (pmId', b') ∈ dom pointRelayAssoc(ora) ∧
        (pmId, b) ≠ (pmId', b') ∧
        pointRelayAssoc(ora)(pmId, b) =
        pointRelayAssoc(ora)(pmId', b')
      )
    )
  )

value /* Well-formedness of signalRelayAssoc */
isWfSignalAssoc : SL.Diagram ×
  ObjectRelayAssociations → Bool
isWfSignalAssoc(d, ora) ≡
  existingSignals(d, ora) ∧
  differentLampRelays(ora),

existingSignals : SL.Diagram ×
  ObjectRelayAssociations → Bool
existingSignals(d, ora) ≡
  ( ∀ (sId, l) : SL.SignalId × Lamp •
    (sId, l) ∈ dom signalRelayAssoc(ora) ⇒
      sId ∈ SL.allSignals(d)
    )
  ) ∧
  ( ∀ sId : SL.SignalId •
    sId ∈ SL.allSignals(d) ⇒
      ( ∃ (sId', l) : SL.SignalId × Lamp •
        (sId', l) ∈ dom signalRelayAssoc(ora) ∧
        sId = sId' )
    )
  ),

differentLampRelays : ObjectRelayAssociations → Bool
differentLampRelays(ora) ≡
  ( ∀ (sId, l) : SL.SignalId × Lamp •
    (sId, l) ∈ dom signalRelayAssoc(ora) ⇒
      ~ ( ∃ (sId', l') : SL.SignalId × Lamp •
        (sId', l') ∈ dom signalRelayAssoc(ora) ∧
        (sId, l) ≠ (sId', l') ∧
        signalRelayAssoc(ora)(sId, l) =
        signalRelayAssoc(ora)(sId', l')
      )
    )
  )

end

```

E.3.3 Train Movement Associations

context: T, SL, TS

scheme TrainMovementAssociations =
class

type

```
TrainMovementAssociations ::
  connectionVars : (SL.TrackId × SL.TrackId)  $\multimap$  T.VarId
  counterVars : SL.TrackId  $\multimap$  T.VarId
  directionFwdVars : SL.TrackId  $\multimap$  T.VarId
  directionBwdVars : SL.TrackId  $\multimap$  T.VarId
  directions : (SL.TrackId × SL.TrackId)  $\multimap$  T.Direction
```

value /* Auxiliary functions */

```
getConnectionVar : SL.TrackId × SL.TrackId ×
  TrainMovementAssociations → Text
getConnectionVar(id, nb, tma)  $\equiv$ 
  let conVars = connectionVars(tma)
  in if (id,nb)  $\in$  dom conVars
    then conVars(id, nb) else conVars(nb, id) end
end,
```

```
getCounterVar : SL.TrackId ×
  TrainMovementAssociations → T.VarId
getCounterVar(tId, tma)  $\equiv$ 
  counterVars(tma)(tId),
```

```
getFwdDirVar : SL.TrackId ×
  TrainMovementAssociations → T.VarId
getFwdDirVar(tId, tma)  $\equiv$  directionFwdVars(tma)(tId),
```

```
getBwdDirVar : SL.TrackId ×
  TrainMovementAssociations → T.VarId
getBwdDirVar(tId, tma)  $\equiv$  directionBwdVars(tma)(tId),
```

```
getDirVar : SL.TrackId × SL.TrackId ×
  TrainMovementAssociations → T.VarId
getDirVar(id, nb, tma)  $\equiv$ 
  getDirVar(id, directions(tma)(id, nb), tma),
```

```
getDirVar_ : SL.TrackId × SL.TrackId ×
  TrainMovementAssociations → T.VarId
getDirVar_(id, nb, tma)  $\equiv$ 
  getDirVar(id, directions(tma)(nb, id), tma),
```

```
getDirVar : SL.TrackId × T.Direction ×
  TrainMovementAssociations → T.VarId
getDirVar(id, dir, tma)  $\equiv$ 
  if dir = T.fwd then getFwdDirVar(id, tma)
  else getBwdDirVar(id, tma) end,
```

```
getDirection : SL.TrackId × SL.TrackId ×
  TrainMovementAssociations → T.Direction
```

```

getDirection(tId1, tId2, tma)  $\equiv$ 
  let directions = directions(tma)
  in if (tId1, tId2)  $\in$  dom directions
    then directions(tId1, tId2)
    else directions(tId2, tId1)
  end
end

value /* Well-formedness of train movement associations */
isWfTrainMovementAssociations : SL.Diagram  $\times$ 
  TrainMovementAssociations  $\rightarrow$  Bool
isWfTrainMovementAssociations(d, tma)  $\equiv$ 
  noRelayVariableOverlap(tma)  $\wedge$ 
  isWfConnectionVars(d, tma)  $\wedge$ 
  isWfCounterVars(d, tma)  $\wedge$ 
  isWfDirectionFwdVars(d, tma)  $\wedge$ 
  isWfDirectionBwdVars(d, tma)  $\wedge$ 
  isWfDirections(d, tma),

noRelayVariableOverlap : TrainMovementAssociations  $\rightarrow$  Bool
noRelayVariableOverlap(tma)  $\equiv$ 
  let connectionVars = rng connectionVars(tma),
    counterVars = rng counterVars(tma),
    directionFwdVars = rng directionFwdVars(tma),
    directionBwdVars = rng directionBwdVars(tma)
  in connectionVars  $\cap$  counterVars = {}  $\wedge$ 
    connectionVars  $\cap$  directionFwdVars = {}  $\wedge$ 
    connectionVars  $\cap$  directionBwdVars = {}  $\wedge$ 
    counterVars  $\cap$  directionFwdVars = {}  $\wedge$ 
    counterVars  $\cap$  directionBwdVars = {}  $\wedge$ 
    directionFwdVars  $\cap$  directionBwdVars = {}
  end

value
isWfConnectionVars : SL.Diagram  $\times$ 
  TrainMovementAssociations  $\rightarrow$  Bool
isWfConnectionVars(d, tma)  $\equiv$ 
  conVarsValidDomain(d, tma)  $\wedge$ 
  conVarsValidRange(connectionVars(tma)),

conVarsValidDomain : SL.Diagram  $\times$ 
  TrainMovementAssociations  $\rightarrow$  Bool
conVarsValidDomain(d, tma)  $\equiv$ 
  ( $\forall$  (tId1, tId2) : (SL.TrackId  $\times$  SL.TrackId)  $\bullet$ 
    (tId1, tId2)  $\in$  dom connectionVars(tma)  $\Rightarrow$ 
    SL.areNeighbours(tId1, tId2, d) ),

conVarsValidRange : ((SL.TrackId  $\times$  SL.TrackId)  $\multimap$  T.VarId)  $\rightarrow$  Bool
conVarsValidRange(connectionVars)  $\equiv$ 
  ( $\forall$  key : (SL.TrackId  $\times$  SL.TrackId)  $\bullet$ 
    key  $\in$  dom connectionVars  $\Rightarrow$ 
     $\sim$ ( $\exists$  key' : (SL.TrackId  $\times$  SL.TrackId)  $\bullet$ 
      key'  $\in$  dom connectionVars  $\wedge$ 
      key  $\neq$  key'  $\wedge$ 
      connectionVars(key) =

```

```

        connectionVars(key') )
    )

value
  isWfCounterVars : SL.Diagram ×
    TrainMovementAssociations → Bool
  isWfCounterVars(d, tma) ≡
    cntVarsValidDomain(d, tma) ∧
    cntVarsValidRange(counterVars(tma)),

  cntVarsValidDomain : SL.Diagram ×
    TrainMovementAssociations → Bool
  cntVarsValidDomain(d, tma) ≡
    let allTrackSections = SL.allLinears(d) ∪ SL.allPoints(d)
    in dom counterVars(tma) = allTrackSections
    end,

  cntVarsValidRange : (SL.TrackId  $\xrightarrow{m}$  T.VarId) → Bool
  cntVarsValidRange(counterVars) ≡
    (∀ key : SL.TrackId •
      key ∈ dom counterVars ⇒
        ∼( ∃ key' : SL.TrackId •
          key' ∈ dom counterVars ∧
          key ≠ key' ∧
          counterVars(key) =
            counterVars(key') )
    )

value
  isWfDirectionFwdVars : SL.Diagram ×
    TrainMovementAssociations → Bool
  isWfDirectionFwdVars(d, tma) ≡
    fwdVarsValidDomain(d, tma) ∧
    fwdVarsValidRange(directionFwdVars(tma)),

  fwdVarsValidDomain : SL.Diagram ×
    TrainMovementAssociations → Bool
  fwdVarsValidDomain(d, tma) ≡
    let allTrackSections = SL.allLinears(d) ∪ SL.allPoints(d)
    in dom directionFwdVars(tma) = allTrackSections
    end,

  fwdVarsValidRange : (SL.TrackId  $\xrightarrow{m}$  T.VarId) → Bool
  fwdVarsValidRange(directionFwdVars) ≡
    (∀ key : SL.TrackId •
      key ∈ dom directionFwdVars ⇒
        ∼( ∃ key' : SL.TrackId •
          key' ∈ dom directionFwdVars ∧
          key ≠ key' ∧
          directionFwdVars(key) =
            directionFwdVars(key') )
    )

value
  isWfDirectionBwdVars : SL.Diagram ×

```

```

    TrainMovementAssociations → Bool
isWfDirectionBwdVars(d, tma) ≡
    bwdVarsValidDomain(d, tma) ∧
    bwdVarsValidRange(directionBwdVars(tma)),

bwdVarsValidDomain : SL.Diagram ×
    TrainMovementAssociations → Bool
bwdVarsValidDomain(d, tma) ≡
    let allTrackSections = SL.allLinears(d) ∪ SL.allPoints(d)
    in dom directionBwdVars(tma) = allTrackSections
    end,

bwdVarsValidRange : (SL.TrackId  $\overrightarrow{m}$  T.VarId) → Bool
bwdVarsValidRange(directionBwdVars) ≡
    (∀ key : SL.TrackId •
        key ∈ dom directionBwdVars ⇒
            ~ ( ∃ key' : SL.TrackId •
                key' ∈ dom directionBwdVars ∧
                key ≠ key' ∧
                directionBwdVars(key) =
                directionBwdVars(key') )
    )

value
isWfDirections : SL.Diagram ×
    TrainMovementAssociations → Bool
isWfDirections(d, tma) ≡
    directionsValidDomain(d, tma) ∧
    bothDirections(directions(tma)),

directionsValidDomain : SL.Diagram ×
    TrainMovementAssociations → Bool
directionsValidDomain(d, tma) ≡
    (∀ (tId1, tId2) : (SL.TrackId × SL.TrackId) •
        (tId1, tId2) ∈ dom directions(tma) ⇒
            SL.areNeighbours(tId1, tId2, d) ),

bothDirections :
    ((SL.TrackId × SL.TrackId)  $\overrightarrow{m}$  T.Direction) → Bool
bothDirections(directions) ≡
    ( ∀ (tId1, tId2) : (SL.TrackId × SL.TrackId) •
        (tId1, tId2) ∈ dom directions ⇒
            ( ∃ (tId1', tId2') : (SL.TrackId × SL.TrackId) •
                (tId1', tId2') ∈ dom directions ∧
                tId1 = tId2' ∧ tId2 = tId1' ∧
                directions((tId1, tId2)) ≠
                directions((tId1', tId2'))
            )
        )
    )

end

```

E.4 Misc.

E.4.1 Objects

```
context: Types
object T : Types
```

```
context: Unparser
object UP : Unparser
```

E.4.2 Types

```
scheme Types =
  class
    type
      VarId = Text,
      TrainRouteId = Text,

      PointsState == plus | minus | arbitrary,

      /* a boolean expression */
      BooleanExp ==
        and(a : BooleanExp-set) |
        or(o : BooleanExp-set) |
        neg(n : BooleanExp) |
        lessthan(ArithmeticExp, ArithmeticExp) |
        greaterthan(ArithmeticExp, ArithmeticExp) |
        geq(ArithmeticExp, ArithmeticExp) | /* greater than equal to*/
        equals(ArithmeticExp, ArithmeticExp) |
        literal(id : VarId) |
        bool(b : Bool),
      /* an arithmetic expression */
      ArithmeticExp ==
        add(a : ArithmeticExp-set) |
        sub(s : ArithmeticExp-set) |
        literal(l : VarId),

      AssignExp ==
        BoolAssign(ba : Bool) |
        NatAssign(na : Nat) |
        AddAssign(aa : Nat) |
        SubAssign(sa : Nat) ,

      Direction == fwd | bwd

  value
```



```

oppositeDirection : Direction → Direction
oppositeDirection(dir) ≡
  case dir of
    fwd → bwd,
    bwd → fwd
  end

type
  LTLformula ==
    b(bExp : BooleanExp) |
    ltrl(l : Text) | /* Literal */
    conj(LTLformula-set) | /* Conjunction */
    disj(LTLformula-set) | /* Disjunction */
    paren(LTLformula) | /* Parenthesis */
    neg(LTLformula) | /* Negation */
    impl(LTLformula, LTLformula) | /* Implication */
    x(LTLformula) | /* Next */
    g(LTLformula) | /* Globally */
    f(LTLformula) | /* Finally */
    u(LTLformula, LTLformula) /* Until */
end

```

E.4.3 Unparser

```

context: T, TS
scheme Unparser =
  class
    value /* Functions to be called externally */

    /* Unparses given transition system */
    print : TS.TransitionSystem → Text
    print(ts) ≡
      let name = TS.name(ts),
          state = TS.state(ts),
          transitionRules = TS.transitionRules(ts),
          assertions = TS.assertions(ts)
      in "\ntransition_system\n\n[" ^ name ^ "]\n\nlocal" ^
        print(state) ^ "\n\nin\n\n" ^
        print(transitionRules) ^ "\n\nend\n\nltl_assertion\n" ^
        print(name, assertions) ^ "\n"
      end

    value /* Auxiliary functions */

    /* Converts a boolean value to text */
    toString : Bool → Text
    toString(b) ≡ if b then "true" else "false" end,

    /* Converts a natural number to text */
    toString : Nat → Text
    toString(n) ≡

```

```

    case n of
      0 → "0",
      1 → "1",
      2 → "2",
      3 → "3",
      4 → "4"
    end

value /* Unparse state */

/* State */
print : TS.Var-set → Text
print(vars) ≡
  if vars = {} then "" else
    let var = hd vars, vars' = vars \ {var}
    in "\n" ^ print(var) ^
      if vars' = {} then ""
      else ", " ^ print(vars')
    end
  end,

/* State variable */
print : TS.Var → Text
print(var) ≡
  let id = TS.id(var),
      val = TS.val(var)
  in id ^ " : " ^ print(val)
end,

/* Val */
print : TS.Val → Text
print(val) ≡
  case val of
    TS.mk.BoolVal(b) → "Bool := " ^ toString(b),
    TS.mk.NatVal(n)  → "Nat  := " ^ toString(n)
  end

value /* Unparse transition rules */

/* Transition rules */
print : TS.TransitionRule-set → Text
print(trs) ≡
  if trs = {} then ""
  else let tr = hd trs, trs' = trs \ {tr}
        in "\n" ^ print(tr) ^
          if trs' = {} then ""
          else "\n[=]" ^ print(trs')
        end
    end,

/* Transition rule */
print : TS.TransitionRule → Text
print(tr) ≡

```

```

    "[" ^ TS.name(tr) ^ "]" ^ "\n" ^
    print(TS.guard(tr)) ^ " ==>\n" ^
    print(TS.assignments(tr)),

/* Multiple assignment */
print : TS.MultipleAssignment → Text
print(assignments) ≡
  if assignments = {} then "" else
    let assignment = hd assignments,
        assignments' = assignments \ {assignment}
    in print(assignment) ^
      if assignments' = {} then ""
      else " , " ^ print(assignments')
    end
  end
end,

/* Assignment */
print : TS.Assignment → Text
print(assignment) ≡
  let id = TS.id(assignment)
  in case TS.assign(assignment) of
    T.BoolAssign(b) → id ^ " , " = " ^ toString(b),
    T.NatAssign(n)  → id ^ " , " = " ^ toString(n),
    T.AddAssign(a)  → id ^ " , " = " ^ id ^ " + " ^ toString(a),
    T.SubAssign(s)  → id ^ " , " = " ^ id ^ " - " ^ toString(s)
  end
end

value /* Unparse assertions */
print : Text × TS.Assertion-set → Text
print(tsName, asserts) ≡
  if asserts = {} then "" else
    let assert = hd asserts, asserts' = asserts \ {assert}
    in "\n" ^ print(tsName, assert) ^
      if asserts' = {} then ""
      else " , \n" ^ print(tsName, asserts')
    end
  end
end,

/* Assertion */
print : Text × (Text × T.LTLformula) → Text
print(tsName, (nm, ltl)) ≡
  "[" ^ nm ^ "]" ^ tsName ^ " |- \n" ^ print(ltl)

value /* Unparse expressions */
print : T.BooleanExp → Text
print(boolExp) ≡
  case boolExp of
    T.and(a) → let x = hd a, a' = a \ {x}
               in print(x) ^
                 if a' = {} then ""
                 else " /\ " ^ print(T.and(a')) end
  end,

```

```

T.or(o) → let x = hd o, o' = o \ {x}
        in print(x) ^ "
          if o' = {} then ""
          else " \\/ " ^ print(T.or(o')) end
        end,
T.neg(n) → "~(" ^ print(n) ^ ")",
T.lessthan(l,r) → print(l) ^ " < " ^ print(r),
T.greaterthan(l,r) → print(l) ^ " > " ^ print(r),
T.geq(l,r) → print(l) ^ " >= " ^ print(r),
T.equals(l,r) → print(l) ^ " = " ^ print(r),
T.literal(l) → l,
T.bool(b) → toString(b)
end,

print : T.ArithmeticExp → Text
print(arithExp) ≡
  case arithExp of
    T.add(a) → let x = hd a, a' = a \ {x}
                in print(x) ^ if a' = {} then ""
                             else " + " ^ print(T.add(a'))
                end
    end,
    T.sub(s) → let x = hd s, s' = s \ {x}
                in print(x) ^ if s' = {} then ""
                             else " - " ^ print(T.sub(s'))
                end
    end,
    T.literal(n) → n
  end

value /* Unparse LTL formula */
print : T.LTLformula → Text
print(ltl) ≡
  case ltl of
    T.paren(ltl) → "(" ^ print(ltl) ^ ")",
    T.b(b) → print(b),
    T.ltrl(l) → l,
    T.conj(set) → let s = hd set, set' = set \ {s}
                  in if set' = {} then ""
                     else print(T.conj(set')) ^ " /\ "
                  end ^ print(s)
                  end,
    T.disj(set) → let s = hd set, set' = set \ {s}
                  in if set' = {} then ""
                     else print(T.disj(set')) ^ " \/ "
                  end ^ print(s)
                  end,
    T.neg(n) → "~(" ^ print(n) ^ ")",
    T.impl(l,r) → print(l) ^ " => " ^ print(r),
    T.x(x) → "X(" ^ print(x) ^ ")",
    T.g(g) → "G(" ^ print(g) ^ ")",
    T.f(f) → "F(" ^ print(f) ^ ")",
    T.u(l,r) → "U(" ^ print(l) ^ ", " ^ print(r) ^ ")"
  end
end

```

end

E.4.4 Station Template

```

context: G, UP, ORA
scheme StationTemplate =
  class
    value -- Station Layout Diagram
      stationLayoutDiagram : SL.Diagram =
        SL.mk_Diagram(
          line, allLinears, allPoints, allSignals,
          neighbours, branchNeighbours, pointMachineTrack,
          trackPointMachine, signalLocations),
      line : Text = "line",
      allLinears : SL.TrackId-set = {},
      allPoints : SL.TrackId-set = {},
      allSignals : SL.SignalId-set = {},
      neighbours : (SL.TrackId  $\times$  SL.TrackId)-set = {},
      branchNeighbours :
        (SL.TrackId  $\times$  SL.TrackId)  $\mapsto$  SL.Branch = [],
      pointMachineTrack : SL.PointMachineId  $\mapsto$  SL.TrackId =
        [],
      trackPointMachine : SL.TrackId  $\mapsto$  SL.PointMachineId =
        [],
      signalLocations :
        (SL.TrackId  $\times$  SL.TrackId)  $\mapsto$  SL.SignalId = []

    value -- Train Route Table
      trainRouteTable : TRT.TrainRouteTable = [],
      dataRow : TRT.DataRow =
        TRT.mk_DataRow({}, [], [], [], (), (), [])

    value -- Interlocking Plan
      interlockingPlan : IP.InterlockingPlan =
        IP.mk_InterlockingPlan(
          stationLayoutDiagram, trainRouteTable)

    value -- Buttons
      allButtons : Text-set = {}

    value -- Object Relay Associations
      objectRelayAssociations : ORA.ObjectRelayAssociations =
        ORA.mk_ObjectRelayAssociations(
          routeRelayAssoc, trackRelayAssoc,
          pointRelayAssoc, signalRelayAssoc),
      routeRelayAssoc : T.TrainRouteId  $\mapsto$  T.VarId = [],
      trackRelayAssoc : SL.TrackId  $\mapsto$  T.VarId = [],
      pointRelayAssoc :
        SL.PointMachineId  $\times$  SL.Branch  $\mapsto$  T.VarId = [],
      signalRelayAssoc :
        SL.SignalId  $\times$  ORA.Lamp  $\mapsto$  T.VarId = []

```

```

test_case -- Well-formedness checks
  [isWfStationLayoutDiagram]
    SL.isWfDiagram(stationLayoutDiagram),
  [isWfTrainRouteTable]
    TRT.isWfTrainRouteTable(trainRouteTable,
      stationLayoutDiagram),
  [isWfInterlockingPlan]
    IP.isWfInterlockingPlan(interlockingPlan),
  [isWfObjectRelayAssociations]
    ORA.isWfObjectRelayAssociations(objectRelayAssociations,
      interlockingPlan)

test_case
  [generate_transitionSystem]
    UP.print(G.generate("InterlockingSystem",
      objectRelayAssociations,
      interlockingPlan,
      allButtons))

end

```

E.4.5 Stenstrup Station

context: G, UP

scheme Stenstrup =

```

class
  value
    stenstrupSLD : SL.Diagram =
      SL.mk_Diagram(
        "line",
        allLinears,
        allPoints,
        allSignals,
        neighbours,
        branchNeighbours,
        pointMachineTrack,
        trackPointMachine,
        signalLocations
      ),

    allLinears : SL.TrackId-set = {"A12", "02", "04", "B12"},
    allPoints  : SL.TrackId-set = {"01", "03"},
    allSignals : SL.SignalId-set = {"A", "B", "E", "F", "G", "H"},

    neighbours : (SL.TrackId × SL.TrackId)-set =
      { ("line", "A12"), ("A12", "01"),
        ("01", "02"), ("01", "04"),
        ("02", "03"), ("03", "04"),
        ("03", "B12"), ("B12", "line") },

    branchNeighbours : (SL.TrackId × SL.TrackId)  $\nrightarrow$  SL.Branch =

```

```
[ ("01", "02") ↦ T.plus,
  ("01", "04") ↦ T.minus,
  ("03", "02") ↦ T.plus,
  ("03", "04") ↦ T.minus ],
```

```
pointMachineTrack : SL.PointMachineId ↗ SL.TrackId =
  [ "01" ↦ "01",
    "02" ↦ "03"],
```

```
trackPointMachine : SL.TrackId ↗ SL.PointMachineId =
  [ "01" ↦ "01",
    "03" ↦ "02"],
```

```
signalLocations : (SL.TrackId × SL.TrackId) ↗ SL.SignalId =
  [ ("line", "A12") ↦ "A", ("line", "B12") ↦ "B",
    ("02", "01") ↦ "E", ("04", "01") ↦ "F",
    ("02", "03") ↦ "G", ("04", "03") ↦ "H"]
```

value

```
allButtons : Text-set = {"b00406", "b03106", "b00606", "b03306"}
```

value

```
stenstrupTRT : TRT.TrainRouteTable =
  [ "2" ↦ dataRow2, "3" ↦ dataRow3,
    "5" ↦ dataRow5, "6" ↦ dataRow6,
    "7" ↦ dataRow7, "8" ↦ dataRow8,
    "9" ↦ dataRow9, "10" ↦ dataRow10 ],
```

```
dataRow2 : TRT.DataRow =
```

```
TRT.mk_DataRow(
  {"03", "B12"},
  [ "A" ↦ TRT.gr, "E" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
    "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
  [ "01" ↦ T.plus, "02" ↦ T.plus ],
  [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
    "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
  ("A", "A12"),
  ((("01", TRT.dropped), ("02", TRT.drawn)),
   ((("02", TRT.dropped), ("01", TRT.drawn))) ),
  [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
    "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ true ]
),
```

```
dataRow3 : TRT.DataRow =
```

```
TRT.mk_DataRow(
  {"03", "B12"},
  [ "A" ↦ TRT.gr, "E" ↦ TRT.arbitrary, "E" ↦ TRT.re,
    "F" ↦ TRT.arbitrary, "G" ↦ TRT.arbitrary, "H" ↦ TRT.re ],
  [ "01" ↦ T.minus, "02" ↦ T.minus ],
  [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
    "04" ↦ true, "03" ↦ true, "B12" ↦ true ],
  ("A", "A12"),
  ((("01", TRT.dropped), ("04", TRT.drawn)),
   ((("04", TRT.dropped), ("01", TRT.drawn))) ),
  [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
```

```

    "7" ↦ true, "8" ↦ true, "9" ↦ true, "10" ↦ false]
  ),

dataRow5 : TRT.DataRow =
  TRT.mk.DataRow(
    { "A12" ↦ "01" },
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.gr, "E" ↦ TRT.re,
      "F" ↦ TRT.arbitrary, "G" ↦ TRT.arbitrary, "H" ↦ TRT.re ],
    [ "01" ↦ T.plus, "02" ↦ T.plus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ( "B", "B12" ),
    ( ("03", TRT.dropped), ("02", TRT.drawn)),
    ( ("02", TRT.dropped), ("03", TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
      "7" ↦ false, "8" ↦ true, "9" ↦ true, "10" ↦ true ]
  ),

dataRow6 : TRT.DataRow =
  TRT.mk.DataRow(
    { "A12" ↦ "01" },
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.gr, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.minus, "02" ↦ T.minus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
      "04" ↦ true, "03" ↦ true, "B12" ↦ true ],
    ( "B", "B12" ),
    ( ("03", TRT.dropped), ("04", TRT.drawn)),
    ( ("04", TRT.dropped), ("03", TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
      "7" ↦ true, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
  ),

dataRow7 : TRT.DataRow =
  TRT.mk.DataRow(
    {},
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.gr,
      "F" ↦ TRT.re, "G" ↦ TRT.arbitrary, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.plus, "02" ↦ T.arbitrary ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
      "04" ↦ false, "03" ↦ false, "B12" ↦ false ],
    ( "E", "01" ),
    ( ("01", TRT.dropped), ("A12", TRT.drawn)),
    ( ("A12", TRT.dropped), ("01", TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ false, "6" ↦ true,
      "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ false ]
  ),

dataRow8 : TRT.DataRow =
  TRT.mk.DataRow(
    {},
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.re,
      "F" ↦ TRT.gr, "G" ↦ TRT.arbitrary, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.minus, "02" ↦ T.arbitrary ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,

```



```

    "04" ↦ false, "03" ↦ false, "B12" ↦ false],
  ("F", "01"),
  ((("01", TRT.dropped), ("A12", TRT.drawn)),
   (("A12", TRT.dropped), ("01", TRT.drawn))),
  [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ false,
    "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ false ]
),

dataRow9 : TRT.DataRow =
  TRT.mk.DataRow(
    {},
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.arbitrary, "G" ↦ TRT.gr, "H" ↦ TRT.re ],
    [ "01" ↦ T.arbitrary, "02" ↦ T.plus ],
    [ "A12" ↦ false, "01" ↦ false, "02" ↦ false,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ("G", "03"),
    ((("03", TRT.dropped), ("B12", TRT.drawn)),
     (("B12", TRT.dropped), ("03", TRT.drawn))),
    [ "2" ↦ false, "3" ↦ true, "5" ↦ true, "6" ↦ true,
      "7" ↦ false, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
  ),

dataRow10 : TRT.DataRow =
  TRT.mk.DataRow(
    {},
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.arbitrary, "G" ↦ TRT.re, "H" ↦ TRT.gr ],
    [ "01" ↦ T.arbitrary, "02" ↦ T.minus ],
    [ "A12" ↦ false, "01" ↦ false, "02" ↦ false,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ("H", "03"),
    ((("03", TRT.dropped), ("B12", TRT.drawn)),
     (("B12", TRT.dropped), ("03", TRT.drawn))),
    [ "2" ↦ true, "3" ↦ false, "5" ↦ true, "6" ↦ true,
      "7" ↦ false, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
  )

value
  stenstrupIP : IP.InterlockingPlan =
    IP.mk.InterlockingPlan(
      stenstrupSLD,
      stenstrupTRT
    )

value
  stenstrupAssocs : ORA.ObjectRelayAssociations =
    ORA.mk.ObjectRelayAssociations(
      routeRelayAssoc,
      trackRelayAssoc,
      pointRelayAssoc,
      signalRelayAssoc
    ),

routeRelayAssoc : T.TrainRouteId → T.VarId =

```

```

[ "2" ↦ "ia", "3" ↦ "ia",
  "5" ↦ "ib", "6" ↦ "ib",
  "7" ↦ "ua", "8" ↦ "ua",
  "9" ↦ "ub", "10" ↦ "ub" ],

trackRelayAssoc : SL.TrackId ↗ T.VarId =
[ "A12" ↦ "a12", "01" ↦ "t01", "02" ↦ "t02",
  "B12" ↦ "b12", "03" ↦ "t03", "04" ↦ "t04" ],

pointRelayAssoc : SL.PointMachineId ×
  SL.Branch ↗ T.VarId =
[ ("01", T.plus) ↦ "plus01", ("01", T.minus) ↦ "minus01",
  ("02", T.plus) ↦ "plus02", ("02", T.minus) ↦ "minus02" ],

signalRelayAssoc : SL.SignalId × ORA.Lamp ↗ T.VarId =
[ ("A", ORA.gr) ↦ "aGreen", ("A", ORA.re) ↦ "aRed",
  ("B", ORA.gr) ↦ "bGreen", ("B", ORA.re) ↦ "bRed",
  ("E", ORA.gr) ↦ "eGreen", ("E", ORA.re) ↦ "eRed",
  ("F", ORA.gr) ↦ "fGreen", ("F", ORA.re) ↦ "fRed",
  ("G", ORA.gr) ↦ "gGreen", ("G", ORA.re) ↦ "gRed",
  ("H", ORA.gr) ↦ "hGreen", ("H", ORA.re) ↦ "hRed" ]

test_case -- Well-formedness checks
[ isWfStationLayoutDiagram
  SL.isWfDiagram(stenstrupSLD),
  isWfTrainRouteTable
  TRT.isWfTrainRouteTable(stenstrupTRT, stenstrupSLD),
  isWfInterlockingPlan
  IP.isWfInterlockingPlan(stenstrupIP),
  isWfObjectRelayAssociations
  ORA.isWfObjectRelayAssociations(stenstrupAssocs, stenstrupIP)

test_case
[ generate_transitionSystem
  UP.print(G.generate("InterlockingSystem",
    stenstrupAssocs,
    stenstrupIP,
    allButtons))

end

```

E.5 Test

E.5.1 Test of Station Layout

```

context: SL
scheme Test_StationLayout =
  class
    value

```

```

/* Fail test cases */
/* Well-formedness NEIGHBOURS start */
/* id which is not a section id */
neighbours_OnlySections_d_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_OnlySections_fail, branchNeighbours,
    pmt, tpm, signalLocations),

neighbours_OnlySections_fail :
  (SL.TrackId × SL.TrackId)-set =
    {"01", "02"}, ("01", "A")},

/* id which is a section id */
neighbours_OnlySections_d_succeed : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_OnlySections_succeed,
    branchNeighbours, pmt, tpm, signalLocations),
neighbours_OnlySections_succeed :
  (SL.TrackId × SL.TrackId)-set =
    {"01", "02"}, ("01", "03")},
/* a section without a neighbour */
neighbours_EverySection_d_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_EverySection_fail, branchNeighbours,
    pmt, tpm, signalLocations),

neighbours_EverySection_fail :
  (SL.TrackId × SL.TrackId)-set = {"01", "02"}},

/* all sections has a neighbour, 01 is connected
to line otherwise
the station is jsut planted in the middle of
no-where.*/
neighbours_EverySection_d_succeed : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_EverySection_succeed,
    branchNeighbours, pmt, tpm, signalLocations),

neighbours_EverySection_succeed :
  (SL.TrackId × SL.TrackId)-set =
    {"line", "01"}, ("01", "02"), ("01", "03")},

/* a linear section neighbour with itself */
neighbours_Irreflexive_d_linear_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_Irreflexive_linear_fail,
    branchNeighbours, pmt, tpm, signalLocations),

neighbours_Irreflexive_linear_fail :
  (SL.TrackId × SL.TrackId)-set = {"03", "03"}},

```

```

/* a points section neighbour with itself */
neighbours_Irreflexive_d_points_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_Irreflexive_points_fail,
    branchNeighbours, pmt, tpm, signalLocations),

neighbours_Irreflexive_points_fail :
  (SL.TrackId × SL.TrackId)-set = {"01", "01"},

/* neighbouring sections are inserted as double
entry */
neighbours_Antisymmetrical_d_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_Antisymmetrical_fail,
    branchNeighbours, pmt, tpm, signalLocations),

neighbours_Antisymmetrical_fail :
  (SL.TrackId × SL.TrackId)-set =
  {"01", "02"}, {"01", "03"}, {"02", "01"},
  {"03", "01"},

/* neighbouring sections are inserted not as double
entry */
neighbours_Antisymmetrical_d_succeed : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_Antisymmetrical_succeed,
    branchNeighbours, pmt, tpm, signalLocations),

neighbours_Antisymmetrical_succeed :
  (SL.TrackId × SL.TrackId)-set =
  {"01", "02"}, {"01", "03"},

/* linear section 03 has no neighbours */
linears_1or2Neighbours_d_fail_1 : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears_linears_1or2Neighbours_fail_2,
    allPoints, allSignals,
    neighbours_linears_1or2Neighbours_fail_1,
    branchNeighbours, pmt, tpm, signalLocations),

neighbours_linears_1or2Neighbours_fail_1 :
  (SL.TrackId × SL.TrackId)-set = {"01", "02"},

/* linear section 02 has 3 neighbours */
linears_1or2Neighbours_d_fail_2 : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears_linears_1or2Neighbours_fail_2,
    allPoints, allSignals,
    neighbours_linears_1or2Neighbours_fail_2,
    branchNeighbours, pmt, tpm, signalLocations),

```

```

allLinears_linears_1or2Neighbours_fail_2 :
  SL.TrackId-set = {"02", "03", "04"},

neighbours_linears_1or2Neighbours_fail_2 :
  (SL.TrackId × SL.TrackId)-set =
    {"01", "02"}, {"03", "02"}, {"04", "02"},

/* linear sections has 1 neighbour each */
linears_1or2Neighbours_d.succeed_1 : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_linears_1or2Neighbours_succeed_1,
    branchNeighbours, pmt, tpm, signalLocations),

neighbours_linears_1or2Neighbours_succeed_1 :
  (SL.TrackId × SL.TrackId)-set =
    {"01", "02"}, {"01", "03"},

/* linear sections has 2 neighbour each */
linears_1or2Neighbours_d.succeed_2 : SL.Diagram =
  SL.mk_Diagram(
    "line",
    allLinears_linears_1or2Neighbours_succeed_2,
    allPoints, allSignals,
    neighbours_linears_1or2Neighbours_succeed_2,
    branchNeighbours, pmt, tpm, signalLocations),

allLinears_linears_1or2Neighbours_succeed_2 :
  SL.TrackId-set = {"02", "03", "04", "05"},

neighbours_linears_1or2Neighbours_succeed_2 :
  (SL.TrackId × SL.TrackId)-set =
    {"01", "02"}, {"01", "03"}, {"02", "04"},
    {"03", "05"},

/* point section 01 has 2 neighbours */
points_3Neighbours_d.fail_1 : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours_points_3Neighbours_fail_1,
    branchNeighbours, pmt, tpm, signalLocations),

neighbours_points_3Neighbours_fail_1 :
  (SL.TrackId × SL.TrackId)-set =
    {"01", "02"}, {"01", "03"},

/* point section 01 has 4 neighbours */
points_3Neighbours_d.fail_2 : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears_points_3Neighbours_fail_2,
    allPoints, allSignals,
    neighbours_points_3Neighbours_fail_2,
    branchNeighbours, pmt, tpm, signalLocations),

allLinears_points_3Neighbours_fail_2 : SL.TrackId-set =

```

```

    {"02", "03", "04", "05"},

neighbours_points_3Neighbours_fail_2 :
  (SL.TrackId × SL.TrackId)-set =
    {"01", "02"}, {"01", "03"}, {"01", "04"},
    {"01", "05"}},

/* point section 01 has 3 neighbours */
points_3Neighbours_d_succeed : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears_points_3Neighbours_succeed,
    allPoints, allSignals,
    neighbours_points_3Neighbours_succeed,
    branchNeighbours, pmt, tpm, signalLocations),

allLinears_points_3Neighbours_succeed : SL.TrackId-set =
  {"02", "03", "04"},

neighbours_points_3Neighbours_succeed :
  (SL.TrackId × SL.TrackId)-set =
    {"01", "02"}, {"01", "03"}, {"01", "04"}},

neighbours_pointsNotNeighbours_d_fail :
  (SL.TrackId × SL.TrackId)-set =
    {"01", "03"}},

pointsNotNeighbours_d_fail : SL.Diagram =
  SL.mk_Diagram(
    "line",
    stenstrup_allLinears,
    stenstrup_allPoints,
    stenstrup_allSignals,
    neighbours_pointsNotNeighbours_d_fail,
    stenstrup_branchNeighbours,
    stenstrup_pointMachineTrack,
    stenstrup_trackPointMachine,
    stenstrup_signalLocations
  ),

/* Well-formedness NEIGHBOURS end */

/* Well-formedness BRANCHNEIGHBOURS start*/
/* the neighbours in the branch neighbours relation
does
not match the neighbours relation */
branchNeighbours_areNeighbours_d_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours, branchNeighbours_areNeighbours_fail,
    pmt, tpm, signalLocations),

branchNeighbours_areNeighbours_fail :
  (SL.TrackId × SL.TrackId)  $\mapsto$  SL.Branch =
  [{"02", "03"}  $\mapsto$  T.plus],

```

```

/* the neighbours in the branch neighbours relation
match the neighbours relation */
branchNeighbours_areNeighbours_d_succeed : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours,
    branchNeighbours_areNeighbours_succeed, pmt, tpm,
    signalLocations),

branchNeighbours_areNeighbours_succeed :
  (SL.TrackId × SL.TrackId)  $\rightsquigarrow$  SL.Branch =
  [ ("01", "02")  $\mapsto$  T.plus, ("01", "03")  $\mapsto$  T.minus],

/* The first parameter in the tuple is a linear
*/
branchNeighbours_FirstIsPoints_d_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours, branchNeighbours_FirstIsPoints_fail,
    pmt, tpm, signalLocations),

branchNeighbours_FirstIsPoints_fail :
  (SL.TrackId × SL.TrackId)  $\rightsquigarrow$  SL.Branch =
  [ ("02", "01")  $\mapsto$  T.plus, ("03", "01")  $\mapsto$  T.minus],

/* The first parameter in the tuple is a points
*/
branchNeighbours_FirstIsPoints_d_succeed : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours,
    branchNeighbours_FirstIsPoints_succeed, pmt, tpm,
    signalLocations),

branchNeighbours_FirstIsPoints_succeed :
  (SL.TrackId × SL.TrackId)  $\rightsquigarrow$  SL.Branch =
  [ ("01", "02")  $\mapsto$  T.plus, ("01", "03")  $\mapsto$  T.minus],

/* Not every points defined in allPoints are used
in branchNeighbours */
branchNeighbours_everyPoints_d_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours, branchNeighbours_everyPoints_fail,
    pmt, tpm, signalLocations),

branchNeighbours_everyPoints_fail :
  (SL.TrackId × SL.TrackId)  $\rightsquigarrow$  SL.Branch = [],

/* every points defined in allPoints and are used
in branchNeighbours */
branchNeighbours_everyPoints_d_succeed : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours, branchNeighbours_everyPoints_succeed,

```

```

    pmt, tpm, signalLocations),

branchNeighbours_everyPoints_succeed :
  (SL.TrackId × SL.TrackId)  $\overrightarrow{m}$  SL.Branch =
    [ ("01", "02")  $\mapsto$  T.plus, ("01", "03")  $\mapsto$  T.minus ],
/* Well-formedness BRANCHNEIGHBOURS end*/
/* Well-formedness SIGNALLOCATION start*/
/* signal location does not use existing signals
from allsignals */

signalLocation_UsesExistingSignals_d_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours, branchNeighbours, pmt, tpm,
    signalLocations_UsesExistingSignals_fail),

signalLocations_UsesExistingSignals_fail :
  (SL.TrackId × SL.TrackId)  $\overrightarrow{m}$  SL.SignalId =
    [ ("02", "01")  $\mapsto$  "A", ("03", "01")  $\mapsto$  "F" ],

/* signal location does not use existing signals
from allsignals */
signalLocation_UsesExistingSignals_d_succeed :
  SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours, branchNeighbours, pmt, tpm,
    signalLocations_UsesExistingSignals_succeed),

signalLocations_UsesExistingSignals_succeed :
  (SL.TrackId × SL.TrackId)  $\overrightarrow{m}$  SL.SignalId =
    [ ("02", "01")  $\mapsto$  "A", ("03", "01")  $\mapsto$  "B" ],

/* signals are placed at invalid locations (between
sections that does not exist) */
signalsArePlacedAtValtIdNeighbours_d_fail : SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours, branchNeighbours, pmt, tpm,
    signalsArePlacedAtValtIdNeighbours_fail),

signalsArePlacedAtValtIdNeighbours_fail :
  (SL.TrackId × SL.TrackId)  $\overrightarrow{m}$  SL.SignalId =
    [ ("02", "04")  $\mapsto$  "A", ("03", "06")  $\mapsto$  "B" ],

/* signals are placed at valid locations (between
sections that does exist) */
signalsArePlacedAtValtIdNeighbours_d_succeed :
  SL.Diagram =
  SL.mk_Diagram(
    "line", allLinears, allPoints, allSignals,
    neighbours, branchNeighbours, pmt, tpm,
    signalsArePlacedAtValtIdNeighbours_succeed),

signalsArePlacedAtValtIdNeighbours_succeed :

```



```

(SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  SL.SignalId =
  [ ("02", "01")  $\mapsto$  "A", ("03", "01")  $\mapsto$  "B"],

/* Well-formedness SIGNALLOCATION end*/
/* Success station layout */
allLinears : SL.TrackId-set = {"02", "03"},

allPoints : SL.TrackId-set = {"01"},

allSignals : SL.SignalId-set = {"A", "B"},

neighbours : (SL.TrackId × SL.TrackId)-set =
  {"line", "01"}, ("01", "02"), ("01", "03")},

branchNeighbours :
  (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  SL.Branch =
  [ ("01", "02")  $\mapsto$  T.plus, ("01", "03")  $\mapsto$  T.minus],

signalLocations :
  (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  SL.SignalId =
  [ ("02", "01")  $\mapsto$  "A", ("03", "01")  $\mapsto$  "B"]

/* Succeed test cases */
value
stenstrupSLD : SL.Diagram =
  SL.mk_Diagram(
    "line",
    stenstrup_allLinears,
    stenstrup_allPoints,
    stenstrup_allSignals,
    stenstrup_neighbours,
    stenstrup_branchNeighbours,
    stenstrup_pointMachineTrack,
    stenstrup_trackPointMachine,
    stenstrup_signalLocations
  ),

stenstrup_allLinears : SL.TrackId-set =
  {"A12", "02", "04", "B12"},
stenstrup_allPoints : SL.TrackId-set =
  {"01", "03"},
stenstrup_allSignals : SL.SignalId-set =
  {"A", "B", "E", "F", "G", "H"},

stenstrup_neighbours : (SL.TrackId × SL.TrackId)-set =
  { ("line", "A12"), ("A12", "01"),
    ("01", "02"), ("01", "04"),
    ("02", "03"), ("03", "04"),
    ("03", "B12"), ("B12", "line") },

stenstrup_branchNeighbours :
  (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  SL.Branch =
  [ ("01", "02")  $\mapsto$  T.plus,
    ("01", "04")  $\mapsto$  T.minus,
    ("03", "02")  $\mapsto$  T.plus,

```

```

    ("03", "04") ↦ T.minus ],

stenstrup_pointMachineTrack : SL.PointMachineId ↗ SL.TrackId =
  [ "01" ↦ "01",
    "02" ↦ "03"],

stenstrup_trackPointMachine : SL.TrackId ↗ SL.PointMachineId =
  [ "01" ↦ "01",
    "03" ↦ "02"],

stenstrup_signalLocations :
  (SL.TrackId × SL.TrackId) ↗ SL.SignalId =
  [ ("line", "A12") ↦ "A", ("line", "B12") ↦ "B",
    ("02", "01") ↦ "E", ("04", "01") ↦ "F",
    ("02", "03") ↦ "G", ("04", "03") ↦ "H"],

sld_fail1 : SL.Diagram =
  SL.mk_Diagram(
    "line",
    allLines_fail,
    allPoints_fail,
    stenstrup_allSignals,
    stenstrup_neighbours,
    stenstrup_branchNeighbours,
    stenstrup_pointMachineTrack,
    stenstrup_trackPointMachine,
    stenstrup_signalLocations
  ),

allLines_fail : SL.TrackId-set = {"A12", "01", "02", "04", "B12"},
allPoints_fail : SL.TrackId-set = {"01", "03"}

test_case
[ isWfIdentifiers_fail ]
  ~ SL.isWfIdentifiers(sld_fail1),
[ isWfIdentifiers_succeed ]
  SL.isWfIdentifiers(stenstrupSLD)

test_case
[ neighbours_OnlySections_fail ]
  ~ SL.neighbours_OnlySections(
    neighbours_OnlySections_d.fail),
[ neighbours_OnlySections_succeed ]
  SL.neighbours_OnlySections(
    neighbours_OnlySections_d.succeed),
[ neighbours_EverySection_fail ]
  ~ SL.neighbours_EverySection(
    neighbours_EverySection_d.fail),
[ neighbours_EverySection_succeed ]
  SL.neighbours_EverySection(
    neighbours_EverySection_d.succeed),
[ neighbours_Irreflexive_linear_fail ]
  ~ SL.neighbours_Irreflexive(
    neighbours_Irreflexive_d.linear.fail),
[ neighbours_Irreflexive_points_fail ]

```

```

    ~ SL.neighbours.Irreflexive(
        neighbours.Irreflexive.d_points_fail),
[neighbours_Antisymmetrical_fail]
    ~ SL.neighbours.Antisymmetrical(
        neighbours.Antisymmetrical.d_fail),
[neighbours_Antisymmetrical_succeed]
    SL.neighbours.Antisymmetrical(
        neighbours.Antisymmetrical.d_succeed),
[linear_1or2Neighbours_fail_1]
    ~ SL.linear_1or2Neighbours(
        linear_1or2Neighbours.d_fail_1),
[linear_1or2Neighbours_fail_2]
    ~ SL.linear_1or2Neighbours(
        linear_1or2Neighbours.d_fail_2),
[linear_1or2Neighbours_succeed_1]
    SL.linear_1or2Neighbours(
        linear_1or2Neighbours.d_succeed_1),
[linear_1or2Neighbours_succeed_2]
    SL.linear_1or2Neighbours(
        linear_1or2Neighbours.d_succeed_2),
[points_3Neighbours_fail_1]
    ~ SL.points_3Neighbours(points_3Neighbours.d_fail_1),
[points_3Neighbours_fail_2]
    ~ SL.points_3Neighbours(points_3Neighbours.d_fail_2),
[points_3Neighbours_succeed]
    SL.points_3Neighbours(points_3Neighbours.d_succeed),
[pointsNotNeighbours_fail]
    ~ SL.pointsNotNeighbours(pointsNotNeighbours.d_fail),
[pointsNotNeighbours_succeed]
    SL.pointsNotNeighbours(stenstrupsLD)

```

test_case

```

[branchNeighbours_areNeighbours_fail]
    ~ SL.branchNeighbours_areNeighbours(
        branchNeighbours_areNeighbours.d_fail),
[branchNeighbours_areNeighbours_succeed]
    SL.branchNeighbours_areNeighbours(
        branchNeighbours_areNeighbours.d_succeed),
[branchNeighbours_FirstIsPoints_fail]
    ~ SL.branchNeighbours_FirstIsPoints(
        branchNeighbours_FirstIsPoints.d_fail),
[branchNeighbours_FirstIsPoints_succeed]
    SL.branchNeighbours_FirstIsPoints(
        branchNeighbours_FirstIsPoints.d_succeed),
[branchNeighbours_everyPoints_fail]
    ~ SL.branchNeighbours_everyPoints_2branchNbs(
        branchNeighbours_everyPoints.d_fail),
[branchNeighbours_everyPoints_succeed]
    SL.branchNeighbours_everyPoints_2branchNbs(
        branchNeighbours_everyPoints.d_succeed)

```

test_case

```

[signalLocation_UsesExistingSignals_fail]
    ~ SL.signalLocation_UsesExistingSignals(
        signalLocation_UsesExistingSignals.d_fail),

```

```

[signalLocation_UsesExistingSignals_succeed]
  SL.signalLocation_UsesExistingSignals(
    signalLocation_UsesExistingSignals_d_succeed),
[signalsArePlacedAtValtIdNeighbours_fail]
  ~ SL.signalsArePlacedAtValtIdNeighbours(
    signalsArePlacedAtValtIdNeighbours_d_fail),
[signalsArePlacedAtValtIdNeighbours_succeed]
  SL.signalsArePlacedAtValtIdNeighbours(
    signalsArePlacedAtValtIdNeighbours_d_succeed)
/* Point Machines */

```

value

```

pmt_fail1 : SL.PointMachineId  $\overrightarrow{\mapsto}$  SL.TrackId =
  ["01"  $\mapsto$  "01", "02"  $\mapsto$  "03"],
tpm_fail1 : SL.TrackId  $\overrightarrow{\mapsto}$  SL.PointMachineId =
  ["01"  $\mapsto$  "01", "03"  $\mapsto$  "03"],
pmt_fail2 : SL.PointMachineId  $\overrightarrow{\mapsto}$  SL.TrackId =
  ["01"  $\mapsto$  "01", "02"  $\mapsto$  "03"],
tpm_fail2 : SL.TrackId  $\overrightarrow{\mapsto}$  SL.PointMachineId =
  ["01"  $\mapsto$  "01", "02"  $\mapsto$  "02"],
pmt_fail3 : SL.PointMachineId  $\overrightarrow{\mapsto}$  SL.TrackId =
  ["01"  $\mapsto$  "01", "02"  $\mapsto$  "03"],
tpm_fail3 : SL.TrackId  $\overrightarrow{\mapsto}$  SL.PointMachineId =
  ["01"  $\mapsto$  "02", "03"  $\mapsto$  "01"],
pmt : SL.PointMachineId  $\overrightarrow{\mapsto}$  SL.TrackId =
  ["01"  $\mapsto$  "01", "02"  $\mapsto$  "03"],
tpm : SL.TrackId  $\overrightarrow{\mapsto}$  SL.PointMachineId =
  ["01"  $\mapsto$  "01", "03"  $\mapsto$  "02"]

```

test_case

```

[pointMachines_areSymmetrical_fail1]
  ~ SL.areSymmetrical(pmt_fail1, tpm_fail1),
[pointMachines_areSymmetrical_fail2]
  ~ SL.areSymmetrical(pmt_fail2, tpm_fail2),
[pointMachines_areSymmetrical_fail3]
  ~ SL.areSymmetrical(pmt_fail3, tpm_fail3),
[pointMachines_areSymmetrical_succeed]
  SL.areSymmetrical(pmt, tpm)

```

value

```

sld_fail : SL.Diagram =
  SL.mk_Diagram(
    "line",
    allLinears_ok,
    allPoints_ok,
    allSignals_ok,
    neighbours_ok,
    branchNeighbours_ok,
    pointMachineTrack_ok,
    trackPointMachine_fail,
    signalLocations_ok
  ),

trackPointMachine_fail : SL.TrackId  $\overrightarrow{\mapsto}$  SL.PointMachineId =
  [ "01"  $\mapsto$  "01",

```

```

    "07" ↦ "02"],

sld_ok : SL.Diagram =
  SL.mk_Diagram(
    "line",
    allLinears_ok,
    allPoints_ok,
    allSignals_ok,
    neighbours_ok,
    branchNeighbours_ok,
    pointMachineTrack_ok,
    trackPointMachine_ok,
    signalLocations_ok
  ),

allLinears_ok : SL.TrackId-set = {"A12", "02", "04", "B12"},
allPoints_ok : SL.TrackId-set = {"01", "03"},
allSignals_ok : SL.SignalId-set = {"A", "B", "E", "F", "G", "H"},

neighbours_ok : (SL.TrackId × SL.TrackId)-set =
  { ("line", "A12"), ("A12", "01"),
    ("01", "02"), ("01", "04"),
    ("02", "03"), ("03", "04"),
    ("03", "B12"), ("B12", "line") },

branchNeighbours_ok : (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  SL.Branch =
  [ ("01", "02") ↦ T.plus,
    ("01", "04") ↦ T.minus,
    ("03", "02") ↦ T.plus,
    ("03", "04") ↦ T.minus ],

pointMachineTrack_ok : SL.PointMachineId  $\overrightarrow{\mapsto}$  SL.TrackId =
  [ "01" ↦ "01",
    "02" ↦ "03"],

trackPointMachine_ok : SL.TrackId  $\overrightarrow{\mapsto}$  SL.PointMachineId =
  [ "01" ↦ "01",
    "03" ↦ "02"],

signalLocations_ok : (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  SL.SignalId =
  [ ("line", "A12") ↦ "A", ("line", "B12") ↦ "B",
    ("02", "01") ↦ "E", ("04", "01") ↦ "F",
    ("02", "03") ↦ "G", ("04", "03") ↦ "H"]

test_case
  [pointMachines_ExistingTrackSections_fail]
  ~SL.pointsSectionsExists(sld_fail),
  [pointMachines_ExistingTrackSections_succeed]
  SL.pointsSectionsExists(sld_ok)

```

end

E.5.2 Test of Train Route Table

context: SL, TRT

scheme Test_TrainRouteTable =

class

value

```

/* Success station layout */
allLinears : SL.TrackId-set = {"02", "03"},
allPoints  : SL.TrackId-set = {"01"},
allSignals : SL.SignalId-set = {"A", "B"},

neighbours : (SL.TrackId × SL.TrackId)-set =
  { ("line", "01"), ("01", "02"), ("01", "03") },

branchNeighbours : (SL.TrackId × SL.TrackId)  $\multimap$  SL.Branch =
  [ ("01", "02")  $\mapsto$  T.plus, ("01", "03")  $\mapsto$  T.minus ],

trackPointMachine : SL.TrackId  $\multimap$  SL.PointMachineId =
  [ "01"  $\mapsto$  "01" ],

pointMachineTrack : SL.PointMachineId  $\multimap$  SL.TrackId =
  [ "01"  $\mapsto$  "01" ],

signalLocations : (SL.TrackId × SL.TrackId)  $\multimap$  SL.SignalId =
  [ ("02", "01")  $\mapsto$  "A", ("03", "01")  $\mapsto$  "B" ],

d : SL.Diagram =
  SL.mk_Diagram(
    "line",
    allLinears,
    allPoints,
    allSignals,
    neighbours,
    branchNeighbours,
    trackPointMachine,
    pointMachineTrack,
    signalLocations
  )

/* Well-formed overlap */

/* No overlap is WF */
value
dataRow : TRT.DataRow =
  TRT.mk_DataRow(
    emptyOverlap,
    [ "A"  $\mapsto$  TRT.gr, "B"  $\mapsto$  TRT.arbitrary, "E"  $\mapsto$  TRT.arbitrary,
      "F"  $\mapsto$  TRT.re, "G"  $\mapsto$  TRT.re, "H"  $\mapsto$  TRT.arbitrary ],
    [ "01"  $\mapsto$  T.plus, "02"  $\mapsto$  T.plus ],
    [ "A12"  $\mapsto$  true, "01"  $\mapsto$  true, "02"  $\mapsto$  true,
      "04"  $\mapsto$  false, "03"  $\mapsto$  true, "B12"  $\mapsto$  true ],
    ( "A", "A12" ),
    ( ("01", TRT.dropped), ("02", TRT.drawn)),
    ( ("02", TRT.dropped), ("01", TRT.drawn)) ),

```

```

        [ "2" ↦ true, "3" ↦ true ]
    ),
    emptyOverlap : SL.TrackId-set = {}
test_case
  [noOverlap] TRT.isWfOverlap(dataRow, d)

/* Overlap part of route */
value
  trainDetection : SL.TrackId  $\overline{\mapsto}$  Bool =
    [ "01" ↦ true, "02" ↦ true, "03" ↦ false ]

test_case
  [overlapNotPartofRoute]
    ~TRT.overlapCovered({"03"}, trainDetection),
  [overlapPartofRoute]
    TRT.overlapCovered({"02"}, trainDetection)

/* Overlap forms connected path */
test_case
  [overlapNotConnected]
    ~TRT.overlapIsConnected({"02", "03"}, d),
  [overlapConnectedA]
    TRT.overlapIsConnected({"01"}, d),
  [overlapConnectedB]
    TRT.overlapIsConnected({"01", "02"}, d)

/* Overlap end of route */
value
  stenstrupSLD : { | d : SL.Diagram • SL.isWfDiagram(d) | } =
    SL.mk_Diagram(
      "line",
      stenstrup_allLinears,
      stenstrup_allPoints,
      stenstrup_allSignals,
      stenstrup_neighbours,
      stenstrup_branchNeighbours,
      stenstrup_pointMachineTrack,
      stenstrup_trackPointMachine,
      stenstrup_signalLocations
    ),

  stenstrup_allLinears : SL.TrackId-set =
    {"A12", "02", "04", "B12"},
  stenstrup_allPoints : SL.TrackId-set =
    {"01", "03"},
  stenstrup_allSignals : SL.SignalId-set =
    {"A", "B", "E", "F", "G", "H"},

  stenstrup_neighbours : (SL.TrackId × SL.TrackId)-set =
    { ("line", "A12"), ("A12", "01"),
      ("01", "02"), ("01", "04"),
      ("02", "03"), ("03", "04"),
      ("03", "B12"), ("B12", "line") },

  stenstrup_branchNeighbours : (SL.TrackId × SL.TrackId)  $\overline{\mapsto}$  SL.Branch =

```

```

[ ("01'", "02'") ↦ T.plus,
  ("01'", "04'") ↦ T.minus,
  ("03'", "02'") ↦ T.plus,
  ("03'", "04'") ↦ T.minus ],

stenstrup_pointMachineTrack : SL.PointMachineId  $\overline{\mapsto}$  SL.TrackId =
[ "01'" ↦ "01'",
  "02'" ↦ "03'"],

stenstrup_trackPointMachine : SL.TrackId  $\overline{\mapsto}$  SL.PointMachineId =
[ "01'" ↦ "01'",
  "03'" ↦ "02'"],

stenstrup_signalLocations : (SL.TrackId  $\times$  SL.TrackId)  $\overline{\mapsto}$  SL.SignalId =
[ ("line", "A12'") ↦ "A", ("line", "B12'") ↦ "B",
  ("02'", "01'") ↦ "E", ("04'", "01'") ↦ "F",
  ("02'", "03'") ↦ "G", ("04'", "03'") ↦ "H"],

dataRow2 : TRT.DataRow =
  TRT.mk_DataRow(
    {"03'", "B12'"},
    [ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
    [ "01'" ↦ T.plus, "02'" ↦ T.plus ],
    [ "A12'" ↦ true, "01'" ↦ true, "02'" ↦ true,
      "04'" ↦ false, "03'" ↦ true, "B12'" ↦ true ],
    ("A", "A12'"),
    ((("01'", TRT.dropped), ("02'", TRT.drawn)),
     ((("02'", TRT.dropped), ("01'", TRT.drawn)) )),
    [ "2'" ↦ true, "3'" ↦ true ]
  ),

dataRow2D : TRT.DataRow =
  TRT.mk_DataRow(
    {"02'", "03'"},
    [ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
    [ "01'" ↦ T.plus, "02'" ↦ T.plus ],
    [ "A12'" ↦ true, "01'" ↦ true, "02'" ↦ true,
      "04'" ↦ false, "03'" ↦ true, "B12'" ↦ true ],
    ("A", "A12'"),
    ((("01'", TRT.dropped), ("02'", TRT.drawn)),
     ((("02'", TRT.dropped), ("01'", TRT.drawn)) )),
    [ "2'" ↦ true, "3'" ↦ true ]
  )

test_case
[ overlapNotEOR ]
  ~TRT.endOfRoute(dataRow2D, stenstrupSLD),
[ overlapEOR ]
  TRT.endOfRoute(dataRow2, stenstrupSLD)

/* Well-formed signals */
value
  signals1 : SL.SignalId  $\overline{\mapsto}$  TRT.Aspect =

```



```

    ["A" ↦ TRT.re, "C" ↦ TRT.gr],
    signals2 : SL.SignalId → TRT.Aspect =
    ["A" ↦ TRT.gr, "B" ↦ TRT.re]
test_case
  [nonExistingSignals]
    ~TRT.existingSignals(signals1, d),
  [existingSignals]
    TRT.existingSignals(signals2, d)

value
  entrySignal : SL.SignalId = "A"
test_case
  [stopEntry]
    ~TRT.proceedEntry(entrySignal, signals1),
  [proceedEntry]
    TRT.proceedEntry(entrySignal, signals2)

/* Well-formed points */
test_case
  [nonexistingPointMachine]
    ~TRT.pointMachinesExist({"00"}, d),
  [existingPointMachines]
    TRT.pointMachinesExist({"01"}, d)

value
  dataRow2B : TRT.DataRow =
    TRT.mk_DataRow(
      {"03", "B12"},
      [ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
        "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
      [ ],
      [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
        "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
      ( "A", "A12" ),
      ((("01", TRT.dropped), ("02", TRT.drawn)),
        (("02", TRT.dropped), ("01", TRT.drawn)) ),
      [ "2" ↦ true, "3" ↦ true ]
    ),

  dataRow2C : TRT.DataRow =
    TRT.mk_DataRow(
      {"03", "B12"},
      [ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
        "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
      [ "02" ↦ T.plus ],
      [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
        "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
      ( "A", "A12" ),
      ((("01", TRT.dropped), ("02", TRT.drawn)),
        (("02", TRT.dropped), ("01", TRT.drawn)) ),
      [ "2" ↦ true, "3" ↦ true ]
    )

test_case
  [pointMachinesNotInTRT]

```

```

    ~TRT.pointMachinesInTRT(dataRow2B, stenstrupSLD),
  [pointMachineNotInTRT]
    ~TRT.pointMachinesInTRT(dataRow2C, stenstrupSLD),
  [pointMachinesInTRT]
    TRT.pointMachinesInTRT(dataRow2, stenstrupSLD)

value
  dataRow10A : TRT.DataRow =
    TRT.mk_DataRow(
      {},
      [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
        "F" ↦ TRT.arbitrary, "G" ↦ TRT.re, "H" ↦ TRT.gr ],
      [ "01" ↦ T.arbitrary, "02" ↦ T.arbitrary ],
      [ "A12" ↦ false, "01" ↦ false, "02" ↦ false,
        "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
      ( "H", "03" ),
      ((("03", TRT.dropped), ("B12", TRT.drawn)),
        ((("B12", TRT.dropped), ("03", TRT.drawn)) ),
      [ "2" ↦ true, "3" ↦ false, "5" ↦ true, "6" ↦ true,
        "7" ↦ false, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
    )

test_case
  [pointsInRouteLocked_fail]
    ~TRT.pointsInRouteLocked(dataRow10A, stenstrupSLD),
  [pointsInRouteLocked_success]
    TRT.pointsInRouteLocked(dataRow2, stenstrupSLD)

value
  dataRow2A : TRT.DataRow =
    TRT.mk_DataRow(
      { "03", "B12" },
      [ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
        "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
      [ "01" ↦ T.plus, "02" ↦ T.minus ],
      [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
        "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
      ( "A", "A12" ),
      ((("01", TRT.dropped), ("02", TRT.drawn)),
        ((("02", TRT.dropped), ("01", TRT.drawn)) ),
      [ "2" ↦ true, "3" ↦ true ]
    )

test_case
  [pointsNotInCorrectPos]
    ~TRT.correctPos(dataRow2A, stenstrupSLD),
  [pointsInCorrectPos]
    TRT.correctPos(dataRow2, stenstrupSLD)

/* Well-formed train detection */
value
  sections1 : SL.TrackId-set = { "00", "01", "02" },
  sections2 : SL.TrackId-set = { "01", "02" },
  sections3 : SL.TrackId-set = { "01", "02", "03", "04" },
  sections4 : SL.TrackId-set = { "01", "02", "03" }

```

```

test_case
  [nonexistingTrackSections]
    ~TRT.existingTrackSections(sections1, d),
  [tooFewTrackSections]
    ~TRT.existingTrackSections(sections2, d),
  [tooManyTrackSections]
    ~TRT.existingTrackSections(sections3, d),
  [existingTrackSections]
    TRT.existingTrackSections(sections4, d)

value
  trainDetection1 : SL.TrackId  $\mapsto$  Bool = ["01"  $\mapsto$  false],
  trainDetection2 : SL.TrackId  $\mapsto$  Bool = ["01"  $\mapsto$  true]

test_case
  [emptyTrainDetection]
    ~TRT.atLeastOneSectionInRoute([],
  [noSectionInRoute]
    ~TRT.atLeastOneSectionInRoute(trainDetection1),
  [atLeastOneSectionInRoute]
    TRT.atLeastOneSectionInRoute(trainDetection2)

value
  trainDetection_fail : SL.TrackId  $\mapsto$  Bool =
    [ "A12"  $\mapsto$  true, "01"  $\mapsto$  true, "02"  $\mapsto$  false,
      "04"  $\mapsto$  false, "03"  $\mapsto$  true, "B12"  $\mapsto$  true ],
  trainDetection_succeed : SL.TrackId  $\mapsto$  Bool =
    [ "A12"  $\mapsto$  true, "01"  $\mapsto$  true, "02"  $\mapsto$  true,
      "04"  $\mapsto$  false, "03"  $\mapsto$  true, "B12"  $\mapsto$  true ]

test_case
  [notConnectedRoute]
    ~TRT.connectedRoute(trainDetection_fail, stenstrupSLD),
  [connectedRoute]
    TRT.connectedRoute(trainDetection_succeed, stenstrupSLD)

value
  trainDetection_fail2 : SL.TrackId  $\mapsto$  Bool =
    [ "A12"  $\mapsto$  true, "01"  $\mapsto$  true, "02"  $\mapsto$  true,
      "04"  $\mapsto$  true, "03"  $\mapsto$  true, "B12"  $\mapsto$  true ]

test_case
  [justOneBranchNeighbour_fail]
    ~TRT.justOneBranchNeighbour(trainDetection_fail2,
    stenstrupSLD),
  [justOneBranchNeighbour]
    TRT.justOneBranchNeighbour(trainDetection_succeed,
    stenstrupSLD)

/* Well-formed signal release */
test_case
  [nonexistingSignal]
    ~TRT.isWfSignalRelease(("C", "01"), d),
  [existingSignal]

```

```

    TRT.isWfSignalRelease(("A", "01"), d)

test_case
  [nonexistingTrackSection]
    ~TRT.isWfSignalRelease(("A", "00"), d),
  [existingTrackSection]
    TRT.isWfSignalRelease(("A", "01"), d),

  [signalNotFollowedByTrackSection]
    ~TRT.signalFollowedByTrackSection("A", "02", d),
  [signalFollowedByTrackSection]
    TRT.signalFollowedByTrackSection("A", "01", d)

/* Well-formed train route release */
test_case
  [notReverseTrackSections]
    ~TRT.reverseTrackSections("01", "02", "01", "02"),
  [reverseTrackSections]
    TRT.reverseTrackSections("01", "02", "02", "01")

test_case
  [notReverseRelayState]
    ~TRT.reverseRelayState(TRT.dropped, TRT.drawn,
      TRT.drawn, TRT.dropped),
  [reverseRelayState]
    TRT.reverseRelayState(TRT.dropped, TRT.drawn,
      TRT.dropped, TRT.drawn)

test_case
  [trackSectionNotPartOfRoute]
    ~TRT.trackSectionPartOfRoute("00", dataRow),
  [trackSectionPartOfRoute]
    TRT.trackSectionPartOfRoute("01", dataRow)

test_case
  [improperRelayState1]
    ~TRT.properRelayState(TRT.drawn, TRT.dropped),
  [improperRelayState2]
    ~TRT.properRelayState(TRT.drawn, TRT.drawn),
  [improperRelayState3]
    ~TRT.properRelayState(TRT.dropped, TRT.dropped),
  [properRelayState]
    TRT.properRelayState(TRT.dropped, TRT.drawn)

/* Well-formed mutually blocking */
value
  trt : TRT.TrainRouteTable =
    [ "1" ↦ dataRowA, "2" ↦ dataRowB ],

  dataRowA : TRT.DataRow =
    TRT.mk_DataRow(
      { "03", "B12" },
      [ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
        "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
      [ "01" ↦ T.plus, "02" ↦ T.plus ] ,

```

```

[ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
  "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
( "A", "A12" ),
( ("01", TRT.dropped), ("02", TRT.drawn)),
( ("02", TRT.dropped), ("01", TRT.drawn)) ),
[ "1" ↦ true, "2" ↦ true ]
),

dataRowB : TRT.DataRow =
  TRT.mk.DataRow(
    { "03", "B12" },
    [ "A" ↦ TRT.gr, "E" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.plus, "02" ↦ T.plus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ( "A", "A12" ),
    ( ("01", TRT.dropped), ("02", TRT.drawn)),
    ( ("02", TRT.dropped), ("01", TRT.drawn)) ),
    [ "1" ↦ true, "2" ↦ true ]
  ),

mb_nwf1 : T.TrainRouteId → Bool =
  [ "1" ↦ false ],
mb_nwf2 : T.TrainRouteId → Bool =
  [ "0" ↦ false, "1" ↦ false ],
mb_nwf3 : T.TrainRouteId → Bool =
  [ "0" ↦ false, "1" ↦ false, "2" ↦ false ],
mb_wf : T.TrainRouteId → Bool =
  [ "1" ↦ false, "2" ↦ false ]

test_case
[ notEveryExistingRoute_few ]
  ~TRT.entryForEveryRoute(mb_nwf1, trt),
[ notEveryExistingRoute_wrong ]
  ~TRT.entryForEveryRoute(mb_nwf2, trt),
[ notEveryExistingRoute_many ]
  ~TRT.entryForEveryRoute(mb_nwf3, trt),
[ everyExistingRoute ]
  TRT.entryForEveryRoute(mb_wf, trt)

```

value

```

stenstrupTRT : TRT.TrainRouteTable =
  [ "2" ↦ dr2_fail, "3" ↦ stenstrup_dr3,
    "5" ↦ stenstrup_dr5, "6" ↦ stenstrup_dr6,
    "7" ↦ stenstrup_dr7, "8" ↦ stenstrup_dr8,
    "9" ↦ stenstrup_dr9, "10" ↦ stenstrup_dr10 ],

dr2_fail : TRT.DataRow =
  TRT.mk.DataRow(
    { "03", "B12" },
    [ "A" ↦ TRT.gr, "E" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.plus, "02" ↦ T.plus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,

```

```

    "04" ↦ false, "03" ↦ true, "B12" ↦ true],
  ( "A", "A12" ),
  (( "01", TRT.dropped), ("02", TRT.drawn)),
  (( "02", TRT.dropped), ("01", TRT.drawn)) ),
  [ "2" ↦ false, "3" ↦ true, "5" ↦ true, "6" ↦ true,
    "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ true ]
),

stenstrup_dr3 : TRT.DataRow =
  TRT.mk.DataRow(
    { "03", "B12" },
    [ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.re,
      "F" ↦ TRT.arbitrary, "G" ↦ TRT.arbitrary, "H" ↦ TRT.re ],
    [ "01" ↦ T.minus, "02" ↦ T.minus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
      "04" ↦ true, "03" ↦ true, "B12" ↦ true ],
    ( "A", "A12" ),
    (( "01", TRT.dropped), ("04", TRT.drawn)),
    (( "04", TRT.dropped), ("01", TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
      "7" ↦ true, "8" ↦ true, "9" ↦ true, "10" ↦ false ]
  ),

stenstrup_dr5 : TRT.DataRow =
  TRT.mk.DataRow(
    { "A12", "01" },
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.gr, "E" ↦ TRT.re,
      "F" ↦ TRT.arbitrary, "G" ↦ TRT.arbitrary, "H" ↦ TRT.re ],
    [ "01" ↦ T.plus, "02" ↦ T.plus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ( "B", "B12" ),
    (( "03", TRT.dropped), ("02", TRT.drawn)),
    (( "02", TRT.dropped), ("03", TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
      "7" ↦ false, "8" ↦ true, "9" ↦ true, "10" ↦ true ]
  ),

stenstrup_dr6 : TRT.DataRow =
  TRT.mk.DataRow(
    { "A12", "01" },
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.gr, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.minus, "02" ↦ T.minus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
      "04" ↦ true, "03" ↦ true, "B12" ↦ true ],
    ( "B", "B12" ),
    (( "03", TRT.dropped), ("04", TRT.drawn)),
    (( "04", TRT.dropped), ("03", TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
      "7" ↦ true, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
  ),

stenstrup_dr7 : TRT.DataRow =
  TRT.mk.DataRow(

```

```

    {
      [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.gr,
        "F" ↦ TRT.re, "G" ↦ TRT.arbitrary, "H" ↦ TRT.arbitrary ],
      [ "01" ↦ T.plus, "02" ↦ T.arbitrary ],
      [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
        "04" ↦ false, "03" ↦ false, "B12" ↦ false ],
      ( "E", "01" ),
      ( ("01", TRT.dropped), ("A12", TRT.drawn)),
      ( ("A12", TRT.dropped), ("01", TRT.drawn)) ),
      [ "2" ↦ true, "3" ↦ true, "5" ↦ false, "6" ↦ true,
        "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ false ]
    ),

  stenstrup_dr8 : TRT.DataRow =
    TRT.mk.DataRow(
      {
        [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.re,
          "F" ↦ TRT.gr, "G" ↦ TRT.arbitrary, "H" ↦ TRT.arbitrary ],
        [ "01" ↦ T.minus, "02" ↦ T.arbitrary ],
        [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
          "04" ↦ false, "03" ↦ false, "B12" ↦ false ],
        ( "F", "01" ),
        ( ("01", TRT.dropped), ("A12", TRT.drawn)),
        ( ("A12", TRT.dropped), ("01", TRT.drawn)) ),
        [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ false,
          "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ false ]
      ),

  stenstrup_dr9 : TRT.DataRow =
    TRT.mk.DataRow(
      {
        [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
          "F" ↦ TRT.arbitrary, "G" ↦ TRT.gr, "H" ↦ TRT.re ],
        [ "01" ↦ T.arbitrary, "02" ↦ T.plus ],
        [ "A12" ↦ false, "01" ↦ false, "02" ↦ false,
          "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
        ( "G", "03" ),
        ( ("03", TRT.dropped), ("B12", TRT.drawn)),
        ( ("B12", TRT.dropped), ("03", TRT.drawn)) ),
        [ "2" ↦ false, "3" ↦ true, "5" ↦ true, "6" ↦ true,
          "7" ↦ false, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
      ),

  stenstrup_dr10 : TRT.DataRow =
    TRT.mk.DataRow(
      {
        [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
          "F" ↦ TRT.arbitrary, "G" ↦ TRT.re, "H" ↦ TRT.gr ],
        [ "01" ↦ T.arbitrary, "02" ↦ T.minus ],
        [ "A12" ↦ false, "01" ↦ false, "02" ↦ false,
          "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
        ( "H", "03" ),
        ( ("03", TRT.dropped), ("B12", TRT.drawn)),
        ( ("B12", TRT.dropped), ("03", TRT.drawn)) ),
        [ "2" ↦ true, "3" ↦ false, "5" ↦ true, "6" ↦ true,

```

```

        "7" ↦ false, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
    )

test_case
[notconflictingIfShareSectionB]
  ~TRT.conflictingWhenShareSection("2", stenstrupTRT),
[conflictingIfShareSection]
  TRT.conflictingWhenShareSection("3", stenstrupTRT)

value
stenstrupTRT2 : TRT.TrainRouteTable =
[ "2" ↦ dr2_fail2, "3" ↦ stenstrup_dr3,
  "5" ↦ stenstrup_dr5, "6" ↦ stenstrup_dr6,
  "7" ↦ stenstrup_dr7, "8" ↦ stenstrup_dr8,
  "9" ↦ stenstrup_dr9, "10" ↦ stenstrup_dr10 ],

dr2_fail2 : TRT.DataRow =
  TRT.mk_DataRow(
    { "03", "B12" },
    [ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.plus, "02" ↦ T.plus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ( "A", "A12" ),
    ( ("01", TRT.dropped), ("02", TRT.drawn)),
    ( ("02", TRT.dropped), ("01", TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
      "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ false ]
  )

test_case
-- Different pos, but not conflicting
[notConflictingWhenDifferentPos]
  ~TRT.conflictingWhenDifferentPos("2", "10", stenstrupTRT2),
-- Same pos
[conflictingWhenDifferentPos_samePos]
  TRT.conflictingWhenDifferentPos("2", "9", stenstrupTRT),
-- Different pos and conflicting
[conflictingWhenDifferentPos]
  TRT.conflictingWhenDifferentPos("2", "10", stenstrupTRT)

end

```

E.5.3 Test of Transition System

```

context: T, TS
scheme Test_TransitionSystem =
  class
    value
      -- TS names

```



```

emptyString : Text = "",
nonemptyString : Text = "test"

test_case -- WF of TS
-- Name
[emptySystemName] ~TS.isWfName(emptyString),
[nonemptySystemName] TS.isWfName(nonemptyString)

-- WF of State
value
bVar1 : TS.Var = TS.mk_Var("t", TS.mk_BoolVal(true)),
bVar2 : TS.Var = TS.mk_Var("hest", TS.mk_BoolVal(true)),
bVar3 : TS.Var = TS.mk_Var("t", TS.mk_BoolVal(false)),

identicalIds : TS.Var-set = {bVar1, bVar3},
uniqueIds : TS.Var-set = {bVar1, bVar2}

test_case
[identicalIds] ~TS.uniqueIds(identicalIds),
[uniqueIds] TS.uniqueIds(uniqueIds)

value
bVar4 : TS.Var = TS.mk_Var("", TS.mk_BoolVal(false)),
bVar5 : TS.Var = TS.mk_Var("0a", TS.mk_BoolVal(false)),
bVar6 : TS.Var = TS.mk_Var("1s", TS.mk_BoolVal(false)),
bVar7 : TS.Var = TS.mk_Var("2d", TS.mk_BoolVal(false)),
bVar8 : TS.Var = TS.mk_Var("3f", TS.mk_BoolVal(false)),
bVar9 : TS.Var = TS.mk_Var("4g", TS.mk_BoolVal(false)),
bVar10 : TS.Var = TS.mk_Var("5h", TS.mk_BoolVal(false)),
bVar11 : TS.Var = TS.mk_Var("6j", TS.mk_BoolVal(false)),
bVar12 : TS.Var = TS.mk_Var("7k", TS.mk_BoolVal(false)),
bVar13 : TS.Var = TS.mk_Var("8l", TS.mk_BoolVal(false)),
bVar14 : TS.Var = TS.mk_Var("9z", TS.mk_BoolVal(false)),

bVar15 : TS.Var = TS.mk_Var("a0", TS.mk_BoolVal(false)),
bVar16 : TS.Var = TS.mk_Var("s1", TS.mk_BoolVal(false)),
bVar17 : TS.Var = TS.mk_Var("d2", TS.mk_BoolVal(false)),
bVar18 : TS.Var = TS.mk_Var("f3", TS.mk_BoolVal(false)),
bVar19 : TS.Var = TS.mk_Var("g4", TS.mk_BoolVal(false)),
bVar20 : TS.Var = TS.mk_Var("h5", TS.mk_BoolVal(false)),
bVar21 : TS.Var = TS.mk_Var("j6", TS.mk_BoolVal(false)),
bVar22 : TS.Var = TS.mk_Var("k7", TS.mk_BoolVal(false)),
bVar23 : TS.Var = TS.mk_Var("l8", TS.mk_BoolVal(false)),
bVar24 : TS.Var = TS.mk_Var("z9", TS.mk_BoolVal(false)),

invalid : TS.Var-set =
{ bVar1, bVar2, bVar3, bVar4, bVar5,
  bVar6, bVar7, bVar8, bVar9, bVar10,
  bVar11, bVar12, bVar13, bVar14 },

valid : TS.Var-set =
{ bVar15, bVar16, bVar17, bVar18, bVar19,
  bVar20, bVar21, bVar22, bVar23, bVar24 }

test_case

```

```

[invalidIds] ~TS.validIds(invalid),
[validIds] TS.validIds(valid)

-- WF of transitions
value
  tinyBoolExp : T.BooleanExp = T.bool(true),

  transition1 : TS.TransitionRule =
    TS.mk_TransitionRule(emptyString, tinyBoolExp, {}),
  transition2 : TS.TransitionRule =
    TS.mk_TransitionRule(nonemptyString, tinyBoolExp, {})

test_case
  [emptyTransitionName] ~TS.validNames({transition1}),
  [nonemptyTransitionName] TS.validNames({transition2})

value
  tinyBoolExp2 : T.BooleanExp = T.bool(false),
  transition3 : TS.TransitionRule =
    TS.mk_TransitionRule(nonemptyString, tinyBoolExp2, {}),
  transition4 : TS.TransitionRule =
    TS.mk_TransitionRule("TR4", tinyBoolExp2, {})

test_case
  [identicalTransitionNames] ~TS.uniqueNames({transition2, transition3}),
  [uniqueTransitionNames] TS.uniqueNames({transition3, transition4})

value
  state : TS.Var-set = {bVar1},
  boolExp : T.BooleanExp = T.literal(TS.id(bVar1)),
  tr : TS.TransitionRule =
    TS.mk_TransitionRule("notok", boolExp, {})

test_case
  [nonExistingIdsInGuard] ~TS.idsInGuardMustBeInState({}, {tr}),
  [existingIdsInGuard] TS.idsInGuardMustBeInState(state, {tr})

value
  assignment : TS.Assignment =
    TS.mk_Assignment(TS.id(bVar1), T.BoolAssign(true)),
  assignments : TS.MultipleAssignment = {assignment},
  transition5 : TS.TransitionRule =
    TS.mk_TransitionRule("TR5", tinyBoolExp, assignments)

test_case
  [emptyAssignment] ~TS.nonemptyAssignment({transition2}),
  [nonemptyAssignment] TS.nonemptyAssignment({transition5})

value
  state1 : TS.Var-set = {bVar1},
  assignment1 : TS.Assignment =
    TS.mk_Assignment(TS.id(bVar2), T.BoolAssign(false)),
  transition1a : TS.TransitionRule =
    TS.mk_TransitionRule("TR1", tinyBoolExp, {assignment1}),
  transitions1 : TS.TransitionRule-set = {transition1a},
  assertions1 : TS.Assertion-set = {},

```

```

ts1 : TS.TransitionSystem =
  TS.mk_TransitionSystem("name", state1, transitions1, assertions1),

state2 : TS.Var-set = {bVar2},
assignment2 : TS.Assignment =
  TS.mk_Assignment(TS.id(bVar2), T.BoolAssign(false)),
transition2a : TS.TransitionRule =
  TS.mk_TransitionRule("TR1", tinyBoolExp, {assignment2}),
transitions2 : TS.TransitionRule-set = {transition2a},
assertions2 : TS.Assertion-set = {},
ts2 : TS.TransitionSystem =
  TS.mk_TransitionSystem("name", state2, transitions2, assertions2)

```

test_case

```

[assignToNonexistingIds] ~TS.assignmentToExistingIds(ts1),
[assignToExistingIds] TS.assignmentToExistingIds(ts2)

```

value

```

assign1 : TS.Assignment =
  TS.mk_Assignment(TS.id(bVar1), T.BoolAssign(true)),
assign2 : TS.Assignment =
  TS.mk_Assignment(TS.id(bVar2), T.BoolAssign(true)),
assign3 : TS.Assignment =
  TS.mk_Assignment(TS.id(bVar1), T.BoolAssign(false)),

tr1 : TS.TransitionRule =
  TS.mk_TransitionRule("TR1", tinyBoolExp, {assign1, assign2}),
tr2 : TS.TransitionRule =
  TS.mk_TransitionRule("TR2", tinyBoolExp, {assign2, assign3}),
tr3 : TS.TransitionRule =
  TS.mk_TransitionRule("TR3", tinyBoolExp, {assign1, assign3})

```

test_case

```

[idUpdMoreThanOnce] ~TS.idUpdOnlyOnce({tr3}),
[idUpdOnlyOnce] TS.idUpdOnlyOnce({tr1, tr2})

```

-- WF of assertions

value

```

tinyLTL : T.LTLformula = T.ltl(TS.id(bVar1)),
assert1 : TS.Assertion = ("'", tinyLTL),
assert2 : TS.Assertion = ("assertA", tinyLTL)

```

test_case

```

[emptyAssertionName] ~TS.nonemptyNames({assert1}),
[nonemptyAssertionName] TS.nonemptyNames({assert2})

```

value

```

tinyLTL2 : T.LTLformula = T.ltl(TS.id(bVar2)),
assert3 : TS.Assertion = ("assertA", tinyLTL2),
assert4 : TS.Assertion = ("assertB", tinyLTL)

```

test_case

```

[identicalAssertionNames] ~TS.uniqueNames({assert2, assert3}),
[uniqueAssertionNames] TS.uniqueNames({assert2, assert4})

```

```

test_case
  [nonexistingVars] ~TS.existingVars({bVar1}, {assert3}),
  [existingVars] TS.existingVars({bVar1}, {assert2})

end

```

E.5.4 Test of Object Relay Associations

context: ORA, SL, TRT

scheme Test_ObjectRelayAssociations =

```

class
  value
    stenstrupSLD : SL.Diagram =
      SL.mk_Diagram(
        "line",
        allLines,
        allPoints,
        allSignals,
        neighbours,
        branchNeighbours,
        pointMachineTrack,
        trackPointMachine,
        signalLocations
      ),

    allLines : SL.TrackId-set = {"A12", "02", "04", "B12"},
    allPoints : SL.TrackId-set = {"01", "03"},
    allSignals : SL.SignalId-set = {"A", "B", "E", "F", "G", "H"},

    neighbours : (SL.TrackId × SL.TrackId)-set =
      { ("line", "A12"), ("A12", "01"),
        ("01", "02"), ("01", "04"),
        ("02", "03"), ("03", "04"),
        ("03", "B12"), ("B12", "line") },

    branchNeighbours : (SL.TrackId × SL.TrackId)  $\multimap$  SL.Branch =
      [ ("01", "02")  $\mapsto$  T.plus,
        ("01", "04")  $\mapsto$  T.minus,
        ("03", "02")  $\mapsto$  T.plus,
        ("03", "04")  $\mapsto$  T.minus ],

    pointMachineTrack : SL.PointMachineId  $\multimap$  SL.TrackId =
      [ "01"  $\mapsto$  "01",
        "02"  $\mapsto$  "03" ],

    trackPointMachine : SL.TrackId  $\multimap$  SL.PointMachineId =
      [ "01"  $\mapsto$  "01",
        "03"  $\mapsto$  "02" ],

    signalLocations : (SL.TrackId × SL.TrackId)  $\multimap$  SL.SignalId =

```

```
[ ("line", "A12") ↦ "A", ("line", "B12") ↦ "B",
  ("02", "01") ↦ "E", ("04", "01") ↦ "F",
  ("02", "03") ↦ "G", ("04", "03") ↦ "H"]
```

value

```
stenstrupTRT : TRT.TrainRouteTable =
[ "2" ↦ dataRow2, "3" ↦ dataRow3,
  "5" ↦ dataRow5, "6" ↦ dataRow6,
  "7" ↦ dataRow7, "8" ↦ dataRow8,
  "9" ↦ dataRow9, "10" ↦ dataRow10 ],

dataRow2 : TRT.DataRow =
TRT.mk_DataRow(
{ "03", "B12" },
[ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
  "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
[ "01" ↦ T.plus, "02" ↦ T.plus ],
[ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
  "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
( "A", "A12" ),
( ("01", TRT.dropped), ("02", TRT.drawn)),
( ("02", TRT.dropped), ("01", TRT.drawn)) ),
[ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
  "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ true ]
),

dataRow3 : TRT.DataRow =
TRT.mk_DataRow(
{ "03", "B12" },
[ "A" ↦ TRT.gr, "B" ↦ TRT.arbitrary, "E" ↦ TRT.re,
  "F" ↦ TRT.arbitrary, "G" ↦ TRT.arbitrary, "H" ↦ TRT.re ],
[ "01" ↦ T.minus, "02" ↦ T.minus ],
[ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
  "04" ↦ true, "03" ↦ true, "B12" ↦ true ],
( "A", "A12" ),
( ("01", TRT.dropped), ("04", TRT.drawn)),
( ("04", TRT.dropped), ("01", TRT.drawn)) ),
[ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
  "7" ↦ true, "8" ↦ true, "9" ↦ true, "10" ↦ false ]
),

dataRow5 : TRT.DataRow =
TRT.mk_DataRow(
{ "A12", "01" },
[ "A" ↦ TRT.arbitrary, "B" ↦ TRT.gr, "E" ↦ TRT.re,
  "F" ↦ TRT.arbitrary, "G" ↦ TRT.arbitrary, "H" ↦ TRT.re ],
[ "01" ↦ T.plus, "02" ↦ T.plus ],
[ "A12" ↦ true, "01" ↦ true, "02" ↦ true,
  "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
( "B", "B12" ),
( ("03", TRT.dropped), ("02", TRT.drawn)),
( ("02", TRT.dropped), ("03", TRT.drawn)) ),
[ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
  "7" ↦ false, "8" ↦ true, "9" ↦ true, "10" ↦ true ]
),
```

```

dataRow6 : TRT.DataRow =
  TRT.mk.DataRow(
    { "A12" , "01" },
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.gr, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.re, "G" ↦ TRT.re, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.minus, "02" ↦ T.minus ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
      "04" ↦ true, "03" ↦ true, "B12" ↦ true ],
    ( "B" , "B12" ),
    ( ("03" , TRT.dropped), ("04" , TRT.drawn)),
    ( ("04" , TRT.dropped), ("03" , TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ true,
      "7" ↦ true, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
  ),

dataRow7 : TRT.DataRow =
  TRT.mk.DataRow(
    {},
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.gr,
      "F" ↦ TRT.re, "G" ↦ TRT.arbitrary, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.plus, "02" ↦ T.arbitrary ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
      "04" ↦ false, "03" ↦ false, "B12" ↦ false ],
    ( "E" , "01" ),
    ( ("01" , TRT.dropped), ("A12" , TRT.drawn)),
    ( ("A12" , TRT.dropped), ("01" , TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ false, "6" ↦ true,
      "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ false ]
  ),

dataRow8 : TRT.DataRow =
  TRT.mk.DataRow(
    {},
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.re,
      "F" ↦ TRT.gr, "G" ↦ TRT.arbitrary, "H" ↦ TRT.arbitrary ],
    [ "01" ↦ T.minus, "02" ↦ T.arbitrary ],
    [ "A12" ↦ true, "01" ↦ true, "02" ↦ false,
      "04" ↦ false, "03" ↦ false, "B12" ↦ false ],
    ( "F" , "01" ),
    ( ("01" , TRT.dropped), ("A12" , TRT.drawn)),
    ( ("A12" , TRT.dropped), ("01" , TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ true, "5" ↦ true, "6" ↦ false,
      "7" ↦ true, "8" ↦ true, "9" ↦ false, "10" ↦ false ]
  ),

dataRow9 : TRT.DataRow =
  TRT.mk.DataRow(
    {},
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.arbitrary, "G" ↦ TRT.gr, "H" ↦ TRT.re ],
    [ "01" ↦ T.arbitrary, "02" ↦ T.plus ],
    [ "A12" ↦ false, "01" ↦ false, "02" ↦ false,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ( "G" , "03" ),

```

```

    ( ("03", TRT.dropped), ("B12", TRT.drawn)),
    ( ("B12", TRT.dropped), ("03", TRT.drawn)) ),
  [ "2" ↦ false, "3" ↦ true, "5" ↦ true, "6" ↦ true,
    "7" ↦ false, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
),

dataRow10 : TRT.DataRow =
  TRT.mk.DataRow(
    {},
    [ "A" ↦ TRT.arbitrary, "B" ↦ TRT.arbitrary, "E" ↦ TRT.arbitrary,
      "F" ↦ TRT.arbitrary, "G" ↦ TRT.re, "H" ↦ TRT.gr ],
    [ "01" ↦ T.arbitrary, "02" ↦ T.minus ],
    [ "A12" ↦ false, "01" ↦ false, "02" ↦ false,
      "04" ↦ false, "03" ↦ true, "B12" ↦ true ],
    ( "H", "03" ),
    ( ("03", TRT.dropped), ("B12", TRT.drawn)),
    ( ("B12", TRT.dropped), ("03", TRT.drawn)) ),
    [ "2" ↦ true, "3" ↦ false, "5" ↦ true, "6" ↦ true,
      "7" ↦ false, "8" ↦ false, "9" ↦ true, "10" ↦ true ]
  ),

stenstrupAssocs : ORA.ObjectRelayAssociations =
  ORA.mk_ObjectRelayAssociations(
    routeRelayAssoc,
    trackRelayAssoc,
    pointRelayAssoc,
    signalRelayAssoc
  ),

routeRelayAssoc : T.TrainRouteId  $\multimap$  T.VarId =
  [ "2" ↦ "ia", "3" ↦ "ia",
    "5" ↦ "ib", "6" ↦ "ib",
    "7" ↦ "ua", "8" ↦ "ua",
    "9" ↦ "ub", "10" ↦ "ub" ],

trackRelayAssoc : SL.TrackId  $\multimap$  T.VarId =
  [ "A12" ↦ "a12", "01" ↦ "t01", "02" ↦ "t02",
    "B12" ↦ "b12", "03" ↦ "t03", "04" ↦ "t04" ],

pointRelayAssoc : SL.PointMachineId  $\times$ 
  SL.Branch  $\multimap$  T.VarId =
  [ ("01", T.plus) ↦ "plus01", ("01", T.minus) ↦ "minus01",
    ("02", T.plus) ↦ "plus02", ("02", T.minus) ↦ "minus02" ],

signalRelayAssoc : SL.SignalId  $\times$  ORA.Lamp  $\multimap$  T.VarId =
  [ ("A", ORA.gr) ↦ "aGreen", ("A", ORA.re) ↦ "aRed",
    ("B", ORA.gr) ↦ "bGreen", ("B", ORA.re) ↦ "bRed",
    ("E", ORA.gr) ↦ "eGreen", ("E", ORA.re) ↦ "eRed",
    ("F", ORA.gr) ↦ "fGreen", ("F", ORA.re) ↦ "fRed",
    ("G", ORA.gr) ↦ "gGreen", ("G", ORA.re) ↦ "gRed",
    ("H", ORA.gr) ↦ "hGreen", ("H", ORA.re) ↦ "hRed" ]

/* Well-fomnedness of ORA */
value
  assocs.fail0A : ORA.ObjectRelayAssociations =

```

```
ORA.mk_ObjectRelayAssociations(  
    routeRelayAssoc_fail0,  
    trackRelayAssoc_fail0,  
    pointRelayAssoc,  
    signalRelayAssoc  
),  
  
assocs_fail0B : ORA.ObjectRelayAssociations =  
    ORA.mk_ObjectRelayAssociations(  
        routeRelayAssoc_fail0,  
        trackRelayAssoc,  
        pointRelayAssoc_fail0,  
        signalRelayAssoc  
    ),  
  
assocs_fail0C : ORA.ObjectRelayAssociations =  
    ORA.mk_ObjectRelayAssociations(  
        routeRelayAssoc_fail0,  
        trackRelayAssoc,  
        pointRelayAssoc,  
        signalRelayAssoc_fail0  
    ),  
  
assocs_fail0D : ORA.ObjectRelayAssociations =  
    ORA.mk_ObjectRelayAssociations(  
        routeRelayAssoc_fail0,  
        trackRelayAssoc_fail0,  
        pointRelayAssoc_fail0,  
        signalRelayAssoc  
    ),  
  
assocs_fail0E : ORA.ObjectRelayAssociations =  
    ORA.mk_ObjectRelayAssociations(  
        routeRelayAssoc_fail0,  
        trackRelayAssoc_fail0,  
        pointRelayAssoc,  
        signalRelayAssoc_fail0  
    ),  
  
assocs_fail0F : ORA.ObjectRelayAssociations =  
    ORA.mk_ObjectRelayAssociations(  
        routeRelayAssoc,  
        trackRelayAssoc_fail0,  
        pointRelayAssoc_fail0,  
        signalRelayAssoc_fail0  
    ),  
  
assocs_fail0G : ORA.ObjectRelayAssociations =  
    ORA.mk_ObjectRelayAssociations(  
        routeRelayAssoc,  
        trackRelayAssoc_fail0,  
        pointRelayAssoc,  
        signalRelayAssoc_fail0  
    ),
```



```

assoc_fail0H : ORA.ObjectRelayAssociations =
  ORA.mk_ObjectRelayAssociations(
    routeRelayAssoc,
    trackRelayAssoc_fail0,
    pointRelayAssoc_fail0,
    signalRelayAssoc_fail0
  ),

routeRelayAssoc_fail0 : T.TrainRouteId  $\rightsquigarrow$  T.VarId =
  [ "2"  $\mapsto$  "ia", "3"  $\mapsto$  "fail",
    "5"  $\mapsto$  "ib", "6"  $\mapsto$  "ib",
    "7"  $\mapsto$  "ua", "8"  $\mapsto$  "ua",
    "9"  $\mapsto$  "ub", "10"  $\mapsto$  "ub" ],

trackRelayAssoc_fail0 : SL.TrackId  $\rightsquigarrow$  T.VarId =
  [ "A12"  $\mapsto$  "a12", "01"  $\mapsto$  "t01", "02"  $\mapsto$  "t02",
    "B12"  $\mapsto$  "b12", "03"  $\mapsto$  "t03", "04"  $\mapsto$  "fail" ],

pointRelayAssoc_fail0 : SL.PointMachineId  $\times$ 
  SL.Branch  $\rightsquigarrow$  T.VarId =
  [ ("01", T.plus)  $\mapsto$  "plus01", ("01", T.minus)  $\mapsto$  "minus01",
    ("02", T.plus)  $\mapsto$  "fail", ("02", T.minus)  $\mapsto$  "minus02" ],

signalRelayAssoc_fail0 : SL.SignalId  $\times$  ORA.Lamp  $\rightsquigarrow$  T.VarId =
  [ ("A", ORA.gr)  $\mapsto$  "aGreen", ("A", ORA.re)  $\mapsto$  "aRed",
    ("B", ORA.gr)  $\mapsto$  "bGreen", ("B", ORA.re)  $\mapsto$  "bRed",
    ("E", ORA.gr)  $\mapsto$  "eGreen", ("E", ORA.re)  $\mapsto$  "fail",
    ("F", ORA.gr)  $\mapsto$  "fGreen", ("F", ORA.re)  $\mapsto$  "fRed",
    ("G", ORA.gr)  $\mapsto$  "gGreen", ("G", ORA.re)  $\mapsto$  "gRed",
    ("H", ORA.gr)  $\mapsto$  "hGreen", ("H", ORA.re)  $\mapsto$  "hRed" ]

```

test_case

```

[noRelayVariableOverlap_failA]
  ~ORA.noRelayVariableOverlap(assoc_fail0A),
[noRelayVariableOverlap_failB]
  ~ORA.noRelayVariableOverlap(assoc_fail0B),
[noRelayVariableOverlap_failC]
  ~ORA.noRelayVariableOverlap(assoc_fail0C),
[noRelayVariableOverlap_failD]
  ~ORA.noRelayVariableOverlap(assoc_fail0D),
[noRelayVariableOverlap_failE]
  ~ORA.noRelayVariableOverlap(assoc_fail0E),
[noRelayVariableOverlap_failF]
  ~ORA.noRelayVariableOverlap(assoc_fail0F),
[noRelayVariableOverlap_failG]
  ~ORA.noRelayVariableOverlap(assoc_fail0G),
[noRelayVariableOverlap_failH]
  ~ORA.noRelayVariableOverlap(assoc_fail0H),
[noRelayVariableOverlap_ok]
  ORA.noRelayVariableOverlap(stenstrupAssocs)

```

/* Well-formedness of route associations */

value

```

assoc_fail1 : ORA.ObjectRelayAssociations =
  ORA.mk_ObjectRelayAssociations(

```

```

        routeRelayAssoc_fail,
        trackRelayAssoc,
        pointRelayAssoc,
        signalRelayAssoc
    ),

    routeRelayAssoc_fail : T.TrainRouteId  $\rightsquigarrow$  T.VarId =
    [ "1"  $\mapsto$  "ia", "2"  $\mapsto$  "ia",
      "3"  $\mapsto$  "ib", "4"  $\mapsto$  "ib",
      "5"  $\mapsto$  "ua", "6"  $\mapsto$  "ua",
      "7"  $\mapsto$  "ub", "8"  $\mapsto$  "ub" ]

test_case
[ routeAssoc_fail ]  $\sim$  ORA.isWfRouteAssoc(stenstrupTRT, assoc_fail1),
[ routeAssoc_ok ] ORA.isWfRouteAssoc(stenstrupTRT, stenstrupAssocs)

/* Well-formedness of track associations */
value
assoc_fail2 : ORA.ObjectRelayAssociations =
    ORA.mk.ObjectRelayAssociations(
        routeRelayAssoc,
        trackRelayAssoc_fail,
        pointRelayAssoc,
        signalRelayAssoc
    ),

    trackRelayAssoc_fail : SL.TrackId  $\rightsquigarrow$  T.VarId =
    [ "A12"  $\mapsto$  "a12", "t01"  $\mapsto$  "t01", "t02"  $\mapsto$  "t02",
      "B12"  $\mapsto$  "b12", "t03"  $\mapsto$  "t03", "t04"  $\mapsto$  "t04" ]

test_case
[ trackAssoc_fail ]  $\sim$  ORA.isWfTrackAssoc(stenstrupSLD, assoc_fail2),
[ trackAssoc_ok ] ORA.isWfTrackAssoc(stenstrupSLD, stenstrupAssocs)

/* Well-formedness of point associations */
value
assoc_fail3 : ORA.ObjectRelayAssociations =
    ORA.mk.ObjectRelayAssociations(
        routeRelayAssoc,
        trackRelayAssoc,
        pointRelayAssoc_fail1,
        signalRelayAssoc
    ),

    pointRelayAssoc_fail1 : SL.PointMachineId  $\times$ 
    SL.Branch  $\rightsquigarrow$  T.VarId =
    [ ("01", T.plus)  $\mapsto$  "plus01", ("02", T.minus)  $\mapsto$  "minus01",
      ("03", T.plus)  $\mapsto$  "plus02", ("04", T.minus)  $\mapsto$  "minus02" ],

    assoc_fail4 : ORA.ObjectRelayAssociations =
    ORA.mk.ObjectRelayAssociations(
        routeRelayAssoc,
        trackRelayAssoc,
        pointRelayAssoc_fail2,
        signalRelayAssoc
    )

```

```

    ),

pointRelayAssoc_fail2 : SL.PointMachineId ×
  SL.Branch  $\overrightarrow{\mapsto}$  T.VarId =
  [ ("01", T.plus)  $\mapsto$  "plus01", ("01", T.minus)  $\mapsto$  "plus01",
    ("02", T.plus)  $\mapsto$  "plus02", ("02", T.minus)  $\mapsto$  "minus02" ],

assocs_fail5 : ORA.ObjectRelayAssociations =
  ORA.mk_ObjectRelayAssociations(
    routeRelayAssoc,
    trackRelayAssoc,
    pointRelayAssoc_fail3,
    signalRelayAssoc
  ),

pointRelayAssoc_fail3 : SL.PointMachineId ×
  SL.Branch  $\overrightarrow{\mapsto}$  T.VarId =
  [ ("01", T.plus)  $\mapsto$  "plus01", ("01", T.minus)  $\mapsto$  "minus01",
    ("02", T.plus)  $\mapsto$  "plus01", ("02", T.minus)  $\mapsto$  "minus02" ]

test_case
  [pointAssoc_existing_fail]
    ~ORA.existingPointMachines(stenstrupSLD, assocs_fail3),
  [pointAssoc_different_fail1]
    ~ORA.differentPointRelays(assocs_fail4),
  [pointAssoc_different_fail2]
    ~ORA.differentPointRelays(assocs_fail5),
  [pointAssoc_ok]
    ORA.isWfPointAssoc(stenstrupSLD, stenstrupAssocs)

/* Well-formedness of signal associations */
value
  assocs_fail6 : ORA.ObjectRelayAssociations =
    ORA.mk_ObjectRelayAssociations(
      routeRelayAssoc,
      trackRelayAssoc,
      pointRelayAssoc,
      signalRelayAssoc_fail1
    ),

signalRelayAssoc_fail1 : SL.SignalId × ORA.Lamp  $\overrightarrow{\mapsto}$  T.VarId =
  [ ("A", ORA.gr)  $\mapsto$  "aGreen", ("A", ORA.re)  $\mapsto$  "aRed",
    ("B", ORA.gr)  $\mapsto$  "bGreen", ("B", ORA.re)  $\mapsto$  "bRed",
    ("C", ORA.gr)  $\mapsto$  "eGreen", ("C", ORA.re)  $\mapsto$  "eRed",
    ("F", ORA.gr)  $\mapsto$  "fGreen", ("F", ORA.re)  $\mapsto$  "fRed",
    ("G", ORA.gr)  $\mapsto$  "gGreen", ("G", ORA.re)  $\mapsto$  "gRed",
    ("H", ORA.gr)  $\mapsto$  "hGreen", ("H", ORA.re)  $\mapsto$  "hRed" ],

assocs_fail7 : ORA.ObjectRelayAssociations =
  ORA.mk_ObjectRelayAssociations(
    routeRelayAssoc,
    trackRelayAssoc,
    pointRelayAssoc,
    signalRelayAssoc_fail2
  ),

```

```

signalRelayAssoc.fail2 : SL.SignalId × ORA.Lamp  $\overrightarrow{m}$  T.VarId =
  [ ("A'", ORA.gr)  $\mapsto$  "aGreen", ("A'", ORA.re)  $\mapsto$  "aGreen",
    ("B'", ORA.gr)  $\mapsto$  "bGreen", ("B'", ORA.re)  $\mapsto$  "bRed",
    ("E'", ORA.gr)  $\mapsto$  "eGreen", ("E'", ORA.re)  $\mapsto$  "eRed",
    ("F'", ORA.gr)  $\mapsto$  "fGreen", ("F'", ORA.re)  $\mapsto$  "fRed",
    ("G'", ORA.gr)  $\mapsto$  "gGreen", ("G'", ORA.re)  $\mapsto$  "gRed",
    ("H'", ORA.gr)  $\mapsto$  "hGreen", ("H'", ORA.re)  $\mapsto$  "hRed" ]

test_case
  [signalAssoc.existing_fail]
    ~ORA.existingSignals(stenstrupSLD, assocs.fail6),
  [signalAssoc.different_fail]
    ~ORA.differentLampRelays(assocs.fail7),
  [signalAssoc.ok]
    ORA.isWfSignalAssoc(stenstrupSLD, stenstrupAssocs)

end

```

E.5.5 Test of Train Movement Associations

context: TMA, SL

scheme Test_TrainMovementAssociations =
class

```

value
  stenstrupSLD : SL.Diagram =
    SL.mk_Diagram(
      "line",
      allLinears,
      allPoints,
      allSignals,
      neighbours,
      branchNeighbours,
      pointMachineTrack,
      trackPointMachine,
      signalLocations
    ),

  allLinears : SL.TrackId-set = {"A12'", "02'", "04'", "B12'"},
  allPoints  : SL.TrackId-set = {"01'", "03'"},
  allSignals : SL.SignalId-set = {"A'", "B'", "E'", "F'", "G'", "H'"},

  neighbours : (SL.TrackId × SL.TrackId)-set =
    { ("line'", "A12'"), ("A12'", "01'"),
      ("01'", "02'"), ("01'", "04'"),
      ("02'", "03'"), ("03'", "04'"),
      ("03'", "B12'"), ("B12'", "line'") },

  branchNeighbours : (SL.TrackId × SL.TrackId)  $\overrightarrow{m}$  SL.Branch =
    [ ("01'", "02'")  $\mapsto$  T.plus,

```

```

("01", "04") ↦ T.minus,
("03", "02") ↦ T.plus,
("03", "04") ↦ T.minus ],

pointMachineTrack : SL.PointMachineId  $\overrightarrow{\mapsto}$  SL.TrackId =
[ "01" ↦ "01",
  "02" ↦ "03" ],

trackPointMachine : SL.TrackId  $\overrightarrow{\mapsto}$  SL.PointMachineId =
[ "01" ↦ "01",
  "03" ↦ "02" ],

signalLocations : (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  SL.SignalId =
[ ("line", "A12") ↦ "A", ("line", "B12") ↦ "B",
  ("02", "01") ↦ "E", ("04", "01") ↦ "F",
  ("02", "03") ↦ "G", ("04", "03") ↦ "H" ],

stenstrupTMA : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars,
    counterVars,
    directionFwdVars,
    directionBwdVars,
    directions
  ),

connectionVars : (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  T.VarId =
[ ("A12", "01") ↦ "con_A12_01",
  ("01", "02") ↦ "con_01_02",
  ("01", "04") ↦ "con_01_04",
  ("02", "03") ↦ "con_02_03",
  ("03", "04") ↦ "con_03_04",
  ("03", "B12") ↦ "con_03_B12"
],

counterVars : SL.TrackId  $\overrightarrow{\mapsto}$  T.VarId =
[ "A12" ↦ "cnt_A12", "B12" ↦ "cnt_B12",
  "01" ↦ "cnt_01", "02" ↦ "cnt_02",
  "03" ↦ "cnt_03", "04" ↦ "cnt_04"
],

directionFwdVars : SL.TrackId  $\overrightarrow{\mapsto}$  T.VarId =
[ "A12" ↦ "fwd_A12", "B12" ↦ "fwd_B12",
  "01" ↦ "fwd_01", "02" ↦ "fwd_02",
  "03" ↦ "fwd_03", "04" ↦ "fwd_04"
],

directionBwdVars : SL.TrackId  $\overrightarrow{\mapsto}$  T.VarId =
[ "A12" ↦ "bwd_A12", "B12" ↦ "bwd_B12",
  "01" ↦ "bwd_01", "02" ↦ "bwd_02",
  "03" ↦ "bwd_03", "04" ↦ "bwd_04"
],

directions : (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  T.Direction =
[ ("line", "A12") ↦ T.fwd, ("A12", "line") ↦ T.bwd,

```

$$\begin{aligned}
& ("A12", "01") \mapsto T.fwd, ("01", "A12") \mapsto T.bwd, \\
& ("01", "02") \mapsto T.fwd, ("02", "01") \mapsto T.bwd, \\
& ("01", "04") \mapsto T.fwd, ("04", "01") \mapsto T.bwd, \\
& ("02", "03") \mapsto T.fwd, ("03", "02") \mapsto T.bwd, \\
& ("03", "04") \mapsto T.fwd, ("04", "03") \mapsto T.bwd, \\
& ("03", "B12") \mapsto T.fwd, ("B12", "03") \mapsto T.bwd, \\
& ("B12", "line") \mapsto T.fwd, ("line", "B12") \mapsto T.bwd]
\end{aligned}$$

value

```

tma_fail0A : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars.fail0,
    counterVars.fail0,
    directionFwdVars,
    directionBwdVars,
    directions
  ),

tma_fail0B : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars.fail0,
    counterVars,
    directionFwdVars,
    directionBwdVars.fail0,
    directions
  ),

tma_fail0C : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars,
    counterVars.fail0,
    directionFwdVars.fail0,
    directionBwdVars,
    directions
  ),

tma_fail0D : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars,
    counterVars.fail0,
    directionFwdVars,
    directionBwdVars.fail0,
    directions
  ),

tma_fail0E : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars,
    counterVars,
    directionFwdVars.fail0,
    directionBwdVars.fail0,
    directions
  ),

tma_fail0F : TMA.TrainMovementAssociations =

```

```

TMA.mk_TrainMovementAssociations(
  connectionVars_fail0,
  counterVars_fail0,
  directionFwdVars_fail0,
  directionBwdVars,
  directions
),

tma_fail0G : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars_fail0,
    counterVars_fail0,
    directionFwdVars,
    directionBwdVars_fail0,
    directions
  ),

tma_fail0H : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars_fail0,
    counterVars,
    directionFwdVars_fail0,
    directionBwdVars_fail0,
    directions
  ),

tma_fail0I : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars,
    counterVars_fail0,
    directionFwdVars_fail0,
    directionBwdVars_fail0,
    directions
  ),

connectionVars_fail0 : (SL.TrackId × SL.TrackId)  $\mapsto$  T.VarId =
  [ ("A12", "01")  $\mapsto$  "fail",
    ("01", "02")  $\mapsto$  "con_01_02",
    ("01", "04")  $\mapsto$  "con_01_04",
    ("02", "03")  $\mapsto$  "con_02_03",
    ("03", "04")  $\mapsto$  "con_03_04",
    ("03", "B12")  $\mapsto$  "con_03_B12"
  ],

counterVars_fail0 : SL.TrackId  $\mapsto$  T.VarId =
  [ "A12"  $\mapsto$  "cnt_A12", "B12"  $\mapsto$  "fail",
    "01"  $\mapsto$  "cnt_01", "02"  $\mapsto$  "cnt_02",
    "03"  $\mapsto$  "cnt_03", "04"  $\mapsto$  "cnt_04"
  ],

directionFwdVars_fail0 : SL.TrackId  $\mapsto$  T.VarId =
  [ "A12"  $\mapsto$  "fwd_A12", "B12"  $\mapsto$  "fwd_B12",
    "01"  $\mapsto$  "fail", "02"  $\mapsto$  "fwd_02",
    "03"  $\mapsto$  "fwd_03", "04"  $\mapsto$  "fwd_04"
  ],

```

```

directionBwdVars_fail0 : SL.TrackId  $\overline{\mapsto}$  T.VarId =
  [ "A12"  $\mapsto$  "bwd_A12", "B12"  $\mapsto$  "bwd_B12",
    "01"  $\mapsto$  "bwd_01", "02"  $\mapsto$  "bwd_02",
    "03"  $\mapsto$  "bwd_03", "04"  $\mapsto$  "fail"
  ]

test_case
  [ noRelayVariableOverlap_failA ]
     $\sim$ TMA.noRelayVariableOverlap(tma_fail0A),
  [ noRelayVariableOverlap_failB ]
     $\sim$ TMA.noRelayVariableOverlap(tma_fail0B),
  [ noRelayVariableOverlap_failC ]
     $\sim$ TMA.noRelayVariableOverlap(tma_fail0C),
  [ noRelayVariableOverlap_failD ]
     $\sim$ TMA.noRelayVariableOverlap(tma_fail0D),
  [ noRelayVariableOverlap_failE ]
     $\sim$ TMA.noRelayVariableOverlap(tma_fail0E),
  [ noRelayVariableOverlap_failF ]
     $\sim$ TMA.noRelayVariableOverlap(tma_fail0F),
  [ noRelayVariableOverlap_failG ]
     $\sim$ TMA.noRelayVariableOverlap(tma_fail0G),
  [ noRelayVariableOverlap_failH ]
     $\sim$ TMA.noRelayVariableOverlap(tma_fail0H),
  [ noRelayVariableOverlap_failI ]
     $\sim$ TMA.noRelayVariableOverlap(tma_fail0I),
  [ noRelayVariableOverlap_ok ]
    TMA.noRelayVariableOverlap(stenstrupTMA)

/* Connection Vars */
value
  tma_fail1 : TMA.TrainMovementAssociations =
    TMA.mk_TrainMovementAssociations(
      connectionVars_fail1A,
      counterVars,
      directionFwdVars,
      directionBwdVars,
      directions
    ),

  connectionVars_fail1A : (SL.TrackId  $\times$  SL.TrackId)  $\overline{\mapsto}$  T.VarId =
    [ ("A12", "B12")  $\mapsto$  "con_A12_01",
      ("01", "02")  $\mapsto$  "con_01_02",
      ("01", "04")  $\mapsto$  "con_01_04",
      ("02", "03")  $\mapsto$  "con_02_03",
      ("03", "04")  $\mapsto$  "con_03_04",
      ("03", "B12")  $\mapsto$  "con_03_B12"
    ],

  connectionVars_fail1B : (SL.TrackId  $\times$  SL.TrackId)  $\overline{\mapsto}$  T.VarId =
    [ ("A12", "01")  $\mapsto$  "con_A12_01",
      ("01", "02")  $\mapsto$  "same",
      ("01", "04")  $\mapsto$  "con_01_04",
      ("02", "03")  $\mapsto$  "con_02_03",
      ("03", "04")  $\mapsto$  "same",
    ]

```



```

    ("03", "B12") ↦ "con_03_B12"
  ]

test_case
  [conVarsValidDomain_fail]
    ~TMA.conVarsValidDomain(stenstrupSLD, tma_fail1),
  [conVarsValidRange_fail]
    ~TMA.conVarsValidRange(connectionVars_fail1B),
  [connectionVars]
    TMA.isWfConnectionVars(stenstrupSLD, stenstrupTMA)

/* Counter Vars */
value
  tma_fail2 : TMA.TrainMovementAssociations =
    TMA.mk_TrainMovementAssociations(
      connectionVars,
      counterVars_fail2A,
      directionFwdVars,
      directionBwdVars,
      directions
    ),

  counterVars_fail2A : SL.TrackId  $\overrightarrow{m}$  T.VarId =
    [ "A12" ↦ "cnt_A12", "B12" ↦ "cnt_B12",
      "01" ↦ "cnt_01", "02" ↦ "cnt_02",
      "03" ↦ "cnt_03", "04" ↦ "cnt_04",
      "line" ↦ "cnt_line"
    ],

  counterVars_fail2B : SL.TrackId  $\overrightarrow{m}$  T.VarId =
    [ "A12" ↦ "same", "B12" ↦ "cnt_B12",
      "01" ↦ "cnt_01", "02" ↦ "same",
      "03" ↦ "cnt_03", "04" ↦ "cnt_04"
    ]

test_case
  [cntVarsValidDomain_fail]
    ~TMA.cntVarsValidDomain(stenstrupSLD, tma_fail2),
  [cntVarsValidRange]
    ~TMA.cntVarsValidRange(counterVars_fail2B),
  [counterVars]
    TMA.isWfCounterVars(stenstrupSLD, stenstrupTMA)

/* Fwd Direction Vars */
value
  tma_fail3 : TMA.TrainMovementAssociations =
    TMA.mk_TrainMovementAssociations(
      connectionVars,
      counterVars,
      directionFwdVars_fail3A,
      directionBwdVars,
      directions
    ),

```

```

directionFwdVars_fail3A : SL.TrackId  $\rightsquigarrow$  T.VarId =
  [ "A12"  $\mapsto$  "fwd_A12", "B12"  $\mapsto$  "fwd_B12",
    "01"  $\mapsto$  "fwd_01", "02"  $\mapsto$  "fwd_02",
    "03"  $\mapsto$  "fwd_03", "04"  $\mapsto$  "fwd_04",
    "line"  $\mapsto$  "fwd_line"
  ],

directionFwdVars_fail3B : SL.TrackId  $\rightsquigarrow$  T.VarId =
  [ "A12"  $\mapsto$  "same", "B12"  $\mapsto$  "fwd_B12",
    "01"  $\mapsto$  "fwd_01", "02"  $\mapsto$  "same",
    "03"  $\mapsto$  "fwd_03", "04"  $\mapsto$  "fwd_04"
  ]

test_case
[ fwdVarsValidDomain_fail ]
  ~TMA.fwdVarsValidDomain(stenstrupSLD, tma_fail3),
[ fwdVarsValidRange_fail ]
  ~TMA.fwdVarsValidRange(directionFwdVars_fail3B),
[ directionFwdVars ]
  TMA.isWfDirectionFwdVars(stenstrupSLD, stenstrupTMA)

/* Bwd Direction Vars */
value
  tma_fail4 : TMA.TrainMovementAssociations =
    TMA.mk_TrainMovementAssociations(
      connectionVars,
      counterVars,
      directionFwdVars,
      directionBwdVars_fail4A,
      directions
    ),

  directionBwdVars_fail4A : SL.TrackId  $\rightsquigarrow$  T.VarId =
    [ "A12"  $\mapsto$  "bwd_A12", "B12"  $\mapsto$  "bwd_B12",
      "01"  $\mapsto$  "bwd_01", "02"  $\mapsto$  "bwd_02",
      "03"  $\mapsto$  "bwd_03", "04"  $\mapsto$  "bwd_04",
      "line"  $\mapsto$  "bwd_line"
    ],

  directionBwdVars_fail4B : SL.TrackId  $\rightsquigarrow$  T.VarId =
    [ "A12"  $\mapsto$  "bwd_A12", "B12"  $\mapsto$  "same",
      "01"  $\mapsto$  "bwd_01", "02"  $\mapsto$  "bwd_02",
      "03"  $\mapsto$  "same", "04"  $\mapsto$  "bwd_04"
    ]

test_case
[ bwdVarsValidDomain_fail ]
  ~TMA.bwdVarsValidDomain(stenstrupSLD, tma_fail4),
[ bwdVarsValidRange_fail ]
  ~TMA.bwdVarsValidRange(directionBwdVars_fail4B),
[ directionBwdVars ]
  TMA.isWfDirectionBwdVars(stenstrupSLD, stenstrupTMA)

/* Directions */
value

```

```

tma_fail5 : TMA.TrainMovementAssociations =
  TMA.mk_TrainMovementAssociations(
    connectionVars,
    counterVars,
    directionFwdVars,
    directionBwdVars,
    directions_fail5A
  ),

directions_fail5A : (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  T.Direction =
  [ ("line", "A12")  $\mapsto$  T.fwd, ("A12", "line")  $\mapsto$  T.bwd,
    ("A12", "01")  $\mapsto$  T.fwd, ("01", "A12")  $\mapsto$  T.bwd,
    ("01", "02")  $\mapsto$  T.fwd, ("02", "01")  $\mapsto$  T.bwd,
    ("01", "04")  $\mapsto$  T.fwd, ("04", "01")  $\mapsto$  T.bwd,
    ("02", "03")  $\mapsto$  T.fwd, ("03", "02")  $\mapsto$  T.bwd,
    ("03", "04")  $\mapsto$  T.fwd, ("04", "03")  $\mapsto$  T.bwd,
    ("03", "B12")  $\mapsto$  T.fwd, ("B12", "03")  $\mapsto$  T.bwd,
    ("B12", "line")  $\mapsto$  T.fwd, ("line", "B12")  $\mapsto$  T.bwd,
    ("02", "04")  $\mapsto$  T.fwd, ("04", "02")  $\mapsto$  T.bwd ],

directions_fail5B : (SL.TrackId × SL.TrackId)  $\overrightarrow{\mapsto}$  T.Direction =
  [ ("line", "A12")  $\mapsto$  T.fwd, ("A12", "line")  $\mapsto$  T.fwd,
    ("A12", "01")  $\mapsto$  T.fwd, ("01", "A12")  $\mapsto$  T.bwd,
    ("01", "02")  $\mapsto$  T.fwd, ("02", "01")  $\mapsto$  T.bwd,
    ("01", "04")  $\mapsto$  T.fwd, ("04", "01")  $\mapsto$  T.bwd,
    ("02", "03")  $\mapsto$  T.fwd, ("03", "02")  $\mapsto$  T.bwd,
    ("03", "04")  $\mapsto$  T.fwd, ("04", "03")  $\mapsto$  T.bwd,
    ("03", "B12")  $\mapsto$  T.fwd, ("B12", "03")  $\mapsto$  T.bwd,
    ("B12", "line")  $\mapsto$  T.fwd, ("line", "B12")  $\mapsto$  T.bwd ],

test_case
[ directionsValidDomain_fail ]
  ~TMA.directionsValidDomain(stenstrupSLD, tma_fail5),
[ bothDirections_fail ]
  ~TMA.bothDirections(directions_fail5B),
[ directions ]
  TMA.isWfDirections(stenstrupSLD, stenstrupTMA)

end

```


APPENDIX F

Output

F.1 Output of Generator

```
[isWfStationLayoutDiagram] true
[isWfTrainRouteTable] true
[isWfInterlockingPlan] true
[isWfObjectRelayAssociations] true
[generate_transitionSystem] "
transition_system
```

```
[InterlockingSystem]
```

```
local
fwd_04 : Bool := false,
bwd_04 : Bool := false,
fwd_02 : Bool := false,
bwd_02 : Bool := false,
fwd_B12 : Bool := false,
bwd_B12 : Bool := false,
fwd_A12 : Bool := false,
bwd_A12 : Bool := false,
fwd_03 : Bool := false,
```

```

bwd_03 : Bool := false,
fwd_01 : Bool := false,
bwd_01 : Bool := false,
con_01_04 : Bool := false,
cnt_04 : Nat := 0,
con_02_03 : Bool := false,
cnt_02 : Nat := 0,
con_01_02 : Bool := false,
cnt_B12 : Nat := 0,
con_03_04 : Bool := false,
cnt_A12 : Nat := 0,
con_A12_01 : Bool := false,
cnt_03 : Nat := 0,
con_03_B12 : Bool := false,
cnt_01 : Nat := 0

in

[change_direction_at_04_towards_03]
hGreen /\ ~(con_01_04) /\ ~(hRed) /\
idle /\ fwd_04 /\ cnt_04 = 1 ==>
fwd_04' = false, bwd_04' = true
[=]
[plusToIntermediate01]
idle /\ ib /\ plus01 /\ ua /\ t01 /\ ia ==>
plus01' = false, idle' = false
[=]
[change_direction_at_02_towards_01]
eGreen /\ ~(con_02_03) /\ ~(eRed) /\
idle /\ bwd_02 /\ cnt_02 = 1 ==>
bwd_02' = false, fwd_02' = true
[=]
[intermediateToPlus01]
idle /\ ib /\ ~(plus01) /\ ua /\
~(minus01) /\ ia /\ t01 ==>
plus01' = true, idle' = false
[=]
[change_direction_at_04_towards_01]
fGreen /\ ~(con_03_04) /\ ~(fRed) /\
idle /\ bwd_04 /\ cnt_04 = 1 ==>
bwd_04' = false, fwd_04' = true
[=]
[minusToIntermediate01]
idle /\ ib /\ minus01 /\ ua /\ t01 /\ ia ==>

```

```

minus01' = false, idle' = false
[=]
[change_direction_at_02_towards_03]
gGreen /\ ~(con_01_02) /\ ~(gRed) /\
idle /\ fwd_02 /\ cnt_02 = 1 ==>
fwd_02' = false, bwd_02' = true
[=]
[intermediateToMinus01]
idle /\ ib /\ ~(plus01) /\ ua /\
~(minus01) /\ ia /\ t01 ==>
minus01' = true, idle' = false
[=]
[leave_B12_to_line]
idle /\ ~(con_03_B12) /\ cnt_B12 = 1 /\ fwd_B12 ==>
fwd_B12' = false, idle' = false,
cnt_B12' = cnt_B12 - 1, b12' = true
[=]
[intermediateToMinus03]
idle /\ ub /\ ~(plus02) /\ ib /\
~(minus02) /\ ia /\ t03 ==>
minus02' = true, idle' = false
[=]
[leave_A12_to_line]
idle /\ ~(con_A12_01) /\ cnt_A12 = 1 /\ bwd_A12 ==>
bwd_A12' = false, idle' = false,
cnt_A12' = cnt_A12 - 1, a12' = true
[=]
[minusToIntermediate03]
idle /\ ub /\ minus02 /\ ib /\ t03 /\ ia ==>
minus02' = false, idle' = false
[=]
[enter_B12_from_line]
idle /\ bGreen /\ ~(bRed) ==>
bwd_B12' = true, idle' = false,
b12' = false, cnt_B12' = cnt_B12 + 1
[=]
[intermediateToPlus03]
idle /\ ub /\ ~(plus02) /\ ib /\
~(minus02) /\ ia /\ t03 ==>
plus02' = true, idle' = false
[=]
[enter_A12_from_line]
idle /\ aGreen /\ ~(aRed) ==>
fwd_A12' = true, idle' = false,

```

```

a12' = false, cnt_A12' = cnt_A12 + 1
[=]
[plusToIntermediate03]
idle /\ ub /\ plus02 /\ ib /\ t03 /\ ia ==>
plus02' = false, idle' = false
[=]
[enter_03_from_B12]
~(con_03_B12) /\ idle /\ bwd_B12 /\ cnt_B12 = 1 ==>
bwd_03' = true, idle' = false,
con_03_B12' = true, cnt_03' = cnt_03 + 1, t03' = false
[=]
[pushButton_b03306]
idle ==>
b03306' = true, idle' = false
[=]
[enter_B12_from_03]
~(con_03_B12) /\ idle /\ fwd_03 /\ cnt_03 = 1 ==>
fwd_B12' = true, idle' = false,
con_03_B12' = true, cnt_B12' = cnt_B12 + 1, b12' = false
[=]
[pushButton_b00406]
idle ==>
b00406' = true, idle' = false
[=]
[enter_A12_from_01]
~(con_A12_01) /\ idle /\ bwd_01 /\ cnt_01 = 1 ==>
bwd_A12' = true, idle' = false,
con_A12_01' = true, cnt_A12' = cnt_A12 + 1, a12' = false
[=]
[pushButton_b00606]
idle ==>
b00606' = true, idle' = false
[=]
[enter_01_from_A12]
~(con_A12_01) /\ idle /\ fwd_A12 /\ cnt_A12 = 1 ==>
fwd_01' = true, idle' = false,
con_A12_01' = true, cnt_01' = cnt_01 + 1, t01' = false
[=]
[pushButton_b03106]
idle ==>
b03106' = true, idle' = false
[=]
[enter_03_from_04]
idle /\ hGreen /\ cnt_04 = 1 /\

```



```

~(hRed) /\ fwd_04 /\ minus02 /\ ~(con_03_04) ==>
fwd_03' = true, idle' = false,
con_03_04' = true, cnt_03' = cnt_03 + 1, t03' = false
[=]
[leave_01_to_04]
idle /\ ~(con_A12_01) /\ fwd_01 /\
~(con_01_02) /\ con_01_04 /\ minus01 /\ cnt_01 = 1 ==>
fwd_01' = false, idle' = false,
con_01_04' = false, t01' = true, cnt_01' = cnt_01 - 1
[=]
[enter_03_from_04_derail]
idle /\ hGreen /\ cnt_04 = 1 /\
~(hRed) /\ fwd_04 /\ ~(minus02) /\ ~(con_03_04) ==>
con_03_04' = true, idle' = false,
t03' = false, cnt_03' = cnt_03 + 1
[=]
[leave_04_to_01]
idle /\ ~(con_03_04) /\
bwd_04 /\ cnt_04 = 1 /\ con_01_04 ==>
bwd_04' = false, idle' = false,
con_01_04' = false, t04' = true, cnt_04' = cnt_04 - 1
[=]
[enter_04_from_03]
minus02 /\ idle /\
~(con_03_04) /\ cnt_03 = 1 /\ bwd_03 ==>
bwd_04' = true, idle' = false,
con_03_04' = true, cnt_04' = cnt_04 + 1, t04' = false
[=]
[leave_03_to_02]
idle /\ ~(con_03_B12) /\ bwd_03 /\
~(con_03_04) /\ con_02_03 /\ plus02 /\ cnt_03 = 1 ==>
bwd_03' = false, idle' = false,
con_02_03' = false, t03' = true, cnt_03' = cnt_03 - 1
[=]
[enter_01_from_02]
idle /\ eGreen /\ cnt_02 = 1 /\
~(eRed) /\ bwd_02 /\ plus01 /\ ~(con_01_02) ==>
bwd_01' = true, idle' = false,
con_01_02' = true, cnt_01' = cnt_01 + 1, t01' = false
[=]
[leave_02_to_03]
idle /\ ~(con_01_02) /\
fwd_02 /\ cnt_02 = 1 /\ con_02_03 ==>
fwd_02' = false, idle' = false,

```

```

con_02_03' = false, t02' = true, cnt_02' = cnt_02 - 1
[=]
[enter_01_from_02_derail]
idle /\ eGreen /\ cnt_02 = 1 /\
~(eRed) /\ bwd_02 /\ ~(plus01) /\ ~(con_01_02) ==>
con_01_02' = true, idle' = false,
t01' = false, cnt_01' = cnt_01 + 1
[=]
[leave_02_to_01]
idle /\ ~(con_02_03) /\
bwd_02 /\ cnt_02 = 1 /\ con_01_02 ==>
bwd_02' = false, idle' = false,
con_01_02' = false, t02' = true, cnt_02' = cnt_02 - 1
[=]
[enter_02_from_01]
plus01 /\ idle /\
~(con_01_02) /\ cnt_01 = 1 /\ fwd_01 ==>
fwd_02' = true, idle' = false,
con_01_02' = true, cnt_02' = cnt_02 + 1, t02' = false
[=]
[leave_01_to_02]
idle /\ ~(con_A12_01) /\ fwd_01 /\
~(con_01_04) /\ con_01_02 /\ plus01 /\ cnt_01 = 1 ==>
fwd_01' = false, idle' = false,
con_01_02' = false, t01' = true, cnt_01' = cnt_01 - 1
[=]
[enter_02_from_03]
plus02 /\ idle /\
~(con_02_03) /\ cnt_03 = 1 /\ bwd_03 ==>
bwd_02' = true, idle' = false,
con_02_03' = true, cnt_02' = cnt_02 + 1, t02' = false
[=]
[leave_04_to_03]
idle /\ ~(con_01_04) /\
fwd_04 /\ cnt_04 = 1 /\ con_03_04 ==>
fwd_04' = false, idle' = false,
con_03_04' = false, t04' = true, cnt_04' = cnt_04 - 1
[=]
[enter_03_from_02_derail]
idle /\ gGreen /\ cnt_02 = 1 /\
~(gRed) /\ fwd_02 /\ ~(plus02) /\ ~(con_02_03) ==>
con_02_03' = true, idle' = false,
t03' = false, cnt_03' = cnt_03 + 1
[=]

```

```

[leave_03_to_04]
idle /\ ~(con_03_B12) /\ bwd_03 /\
~(con_02_03) /\ con_03_04 /\ minus02 /\ cnt_03 = 1 ==>
bwd_03' = false, idle' = false,
con_03_04' = false, t03' = true, cnt_03' = cnt_03 - 1
[=]
[enter_03_from_02]
idle /\ gGreen /\ cnt_02 = 1 /\
~(gRed) /\ fwd_02 /\ plus02 /\ ~(con_02_03) ==>
fwd_03' = true, idle' = false,
con_02_03' = true, cnt_03' = cnt_03 + 1, t03' = false
[=]
[leave_01_to_A12]
idle /\ ~(con_01_04) /\ bwd_01 /\
~(con_01_02) /\ con_A12_01 /\ cnt_01 = 1 ==>
bwd_01' = false, idle' = false,
con_A12_01' = false, t01' = true, cnt_01' = cnt_01 - 1
[=]
[enter_04_from_01]
minus01 /\ idle /\
~(con_01_04) /\ cnt_01 = 1 /\ fwd_01 ==>
fwd_04' = true, idle' = false,
con_01_04' = true, cnt_04' = cnt_04 + 1, t04' = false
[=]
[leave_A12_to_01]
cnt_A12 = 1 /\ idle /\ con_A12_01 /\ fwd_A12 ==>
fwd_A12' = false, idle' = false,
con_A12_01' = false, a12' = true, cnt_A12' = cnt_A12 - 1
[=]
[enter_01_from_04_derail]
idle /\ fGreen /\ cnt_04 = 1 /\
~(fRed) /\ bwd_04 /\ ~(minus01) /\ ~(con_01_04) ==>
con_01_04' = true, idle' = false,
t01' = false, cnt_01' = cnt_01 + 1
[=]
[leave_B12_to_03]
cnt_B12 = 1 /\ idle /\ con_03_B12 /\ bwd_B12 ==>
bwd_B12' = false, idle' = false,
con_03_B12' = false, b12' = true, cnt_B12' = cnt_B12 - 1
[=]
[enter_01_from_04]
idle /\ fGreen /\ cnt_04 = 1 /\
~(fRed) /\ bwd_04 /\ minus01 /\ ~(con_01_04) ==>
bwd_01' = true, idle' = false,

```

```

con_01_04' = true, cnt_01' = cnt_01 + 1, t01' = false
[=]
[leave_03_to_B12]
idle /\ ~(con_02_03) /\ fwd_03 /\
~(con_03_04) /\ con_03_B12 /\ cnt_03 = 1 ==>
fwd_03' = false, idle' = false,
con_03_B12' = false, t03' = true, cnt_03' = cnt_03 - 1

end

ltl_assertion

[no_derailing_03_02] InterlockingSystem |-
G(con_02_03 => plus02),

[train_route_release_7] InterlockingSystem |-
G(ua /\ X(plus01 /\ ~(ua) /\ F(ua)) =>
X(U(~(ua), ~(t01) /\ a12 /\ ~(ua) /\
X(U(~(ua), ~(ua) /\ ~(a12) /\ t01))))),

[no_derailing_01_04] InterlockingSystem |-
G(con_01_04 => minus01),

[train_route_release_6] InterlockingSystem |-
G(ib /\ X(minus02 /\ minus01 /\ ~(ib) /\ F(ib)) =>
X(U(~(ib), ~(t03) /\ t04 /\ ~(ib) /\
X(U(~(ib), ~(ib) /\ ~(t04) /\ t03))))),

[no_derailing_03_04] InterlockingSystem |-
G(con_03_04 => minus02),

[train_route_release_8] InterlockingSystem |-
G(ua /\ X(minus01 /\ ~(ua) /\ F(ua)) =>
X(U(~(ua), ~(t01) /\ a12 /\ ~(ua) /\
X(U(~(ua), ~(ua) /\ ~(a12) /\ t01))))),

[no_derailing_01_02] InterlockingSystem |-
G(con_01_02 => plus01),

[train_route_release_5] InterlockingSystem |-
G(ib /\ X(plus02 /\ plus01 /\ ~(ib) /\ F(ib)) =>
X(U(~(ib), ~(t03) /\ t02 /\ ~(ib) /\
X(U(~(ib), ~(ib) /\ ~(t02) /\ t03))))),

```

```
[points_locked_when_occupied_03] InterlockingSystem |-
G(cnt_03 >= 1 => plus02 \/ minus02),
```

```
[train_route_release_9] InterlockingSystem |-
G(ub /\ X(plus02 /\ ~(ub) /\ F(ub)) =>
X(U(~(ub), ~(t03) /\ b12 /\ ~(ub) /\
X(U(~(ub), ~(ub) /\ ~(b12) /\ t03))))),
```

```
[points_locked_when_occupied_01] InterlockingSystem |-
G(cnt_01 >= 1 => plus01 \/ minus01),
```

```
[train_route_release_3] InterlockingSystem |-
G(ia /\ X(minus02 /\ minus01 /\ ~(ia) /\ F(ia)) =>
X(U(~(ia), ~(t01) /\ t04 /\ ~(ia) /\
X(U(~(ia), ~(ia) /\ ~(t04) /\ t01))))),
```

```
[no_collision_A12] InterlockingSystem |-
G(cnt_A12 < 2),
```

```
[train_route_release_10] InterlockingSystem |-
G(ub /\ X(minus02 /\ ~(ub) /\ F(ub)) =>
X(U(~(ub), ~(t03) /\ b12 /\ ~(ub) /\
X(U(~(ub), ~(ub) /\ ~(b12) /\ t03))))),
```

```
[no_collision_01] InterlockingSystem |-
G(cnt_01 < 2),
```

```
[train_route_release_2] InterlockingSystem |-
G(ia /\ X(plus02 /\ plus01 /\ ~(ia) /\ F(ia)) =>
X(U(~(ia), ~(t01) /\ t02 /\ ~(ia) /\
X(U(~(ia), ~(ia) /\ ~(t02) /\ t01))))),
```

```
[no_collision_02] InterlockingSystem |-
G(cnt_02 < 2),
```

```
[signal_release_7] InterlockingSystem |-
G(idle /\ ~(t01) => eRed),
```

```
[no_collision_03] InterlockingSystem |-
G(cnt_03 < 2),
```

```
[signal_release_6] InterlockingSystem |-
G(idle /\ ~(b12) => bRed),
```

```
[no_collision_04] InterlockingSystem |-
G(cnt_04 < 2),

[signal_release_8] InterlockingSystem |-
G(idle /\ ~(t01) => fRed),

[no_collision_B12] InterlockingSystem |-
G(cnt_B12 < 2),

[signal_release_5] InterlockingSystem |-
G(idle /\ ~(b12) => bRed),

[points_configuration_01] InterlockingSystem |-
G(idle => ~(plus01 /\ minus01)),

[signal_release_9] InterlockingSystem |-
G(idle /\ ~(t03) => gRed),

[points_configuration_03] InterlockingSystem |-
G(idle => ~(plus02 /\ minus02)),

[signal_release_3] InterlockingSystem |-
G(idle /\ ~(a12) => aRed),

[free_B12] InterlockingSystem |-
G(cnt_B12 = 0 => b12),

[signal_release_10] InterlockingSystem |-
G(idle /\ ~(t03) => hRed),

[free_B12_] InterlockingSystem |-
G(b12 => cnt_B12 = 0),

[signal_release_2] InterlockingSystem |-
G(idle /\ ~(a12) => aRed),

[free_04_] InterlockingSystem |-
G(t04 => cnt_04 = 0),

[point_position_ua] InterlockingSystem |-
G(~(ua) => plus01 \/ minus01),

[free_04] InterlockingSystem |-
G(cnt_04 = 0 => t04),
```

```
[point_position_ib] InterlockingSystem |-
G(~(ib) => plus02 /\ plus01 \/ minus02 /\ minus01),

[free_03_] InterlockingSystem |-
G(t03 => cnt_03 = 0),

[point_position_ub] InterlockingSystem |-
G(~(ub) => plus02 \/ minus02),

[free_03] InterlockingSystem |-
G(cnt_03 = 0 => t03),

[point_position_ia] InterlockingSystem |-
G(~(ia) => plus02 /\ plus01 \/ minus02 /\ minus01),

[free_02_] InterlockingSystem |-
G(t02 => cnt_02 = 0),

[signal_E] InterlockingSystem |-
G(idle => ~(eGreen /\ eRed)),

[free_02] InterlockingSystem |-
G(cnt_02 = 0 => t02),

[signal_F] InterlockingSystem |-
G(idle => ~(fGreen /\ fRed)),

[free_01_] InterlockingSystem |-
G(t01 => cnt_01 = 0),

[signal_B] InterlockingSystem |-
G(idle => ~(bGreen /\ bRed)),

[free_01] InterlockingSystem |-
G(cnt_01 = 0 => t01),

[signal_G] InterlockingSystem |-
G(idle => ~(gGreen /\ gRed)),

[free_A12_] InterlockingSystem |-
G(a12 => cnt_A12 = 0),

[signal_A] InterlockingSystem |-
```

```

G(idle => ~(aGreen /\ aRed)),

[free_A12] InterlockingSystem |-
G(cnt_A12 = 0 => a12),

[signal_H] InterlockingSystem |-
G(idle => ~(hGreen /\ hRed)),

[occ_A12] InterlockingSystem |-
G(cnt_A12 > 0 => ~(a12)),

[signal_exp_H] InterlockingSystem |-
G(idle /\ hGreen =>
~(ub) /\ minus02 /\ b12 /\ gRed /\ t03)),

[occ_A12_] InterlockingSystem |-
G(~(a12) => cnt_A12 > 0),

[signal_exp_A] InterlockingSystem |-
G(idle /\ aGreen =>
(a12 /\ b12 /\ t03 /\ ~(ia) /\ t01 /\ plus01 /\
t02 /\ plus02 /\ (fRed /\ gRed \/ (plus02 /\
~(ub) /\ hRed /\ gGreen)) /\ fRed) \/ (a12 /\
b12 /\ t03 /\ ~(ia) /\ t01 /\ minus01 /\ t04 /\
minus02 /\ (eRed /\ hRed \/ (minus02 /\ ~(ub) /\
gRed /\ hGreen)) /\ eRed)),

[occ_01] InterlockingSystem |-
G(cnt_01 > 0 => ~(t01)),

[signal_exp_G] InterlockingSystem |-
G(idle /\ gGreen =>
~(ub) /\ plus02 /\ b12 /\ hRed /\ t03)),

[occ_01_] InterlockingSystem |-
G(~(t01) => cnt_01 > 0),

[signal_exp_B] InterlockingSystem |-
G(idle /\ bGreen =>
(a12 /\ b12 /\ t03 /\ ~(ib) /\ t01 /\ plus01 /\
t02 /\ plus02 /\ (eRed /\ hRed \/ (plus01 /\
~(ua) /\ fRed /\ eGreen)) /\ hRed) \/ (a12 /\
b12 /\ t03 /\ ~(ib) /\ t01 /\ minus01 /\ t04 /\
minus02 /\ (fRed /\ gRed \/ (minus01 /\

```



```

~(ua) /\ eRed /\ fGreen)) /\ gRed)),

[occ_02] InterlockingSystem |-
G(cnt_02 > 0 => ~(t02)),

[signal_exp_F] InterlockingSystem |-
G(idle /\ fGreen =>
~(ua) /\ minus01 /\ t01 /\ eRed /\ a12)),

[occ_02_] InterlockingSystem |-
G(~(t02) => cnt_02 > 0),

[signal_exp_E] InterlockingSystem |-
G(idle /\ eGreen =>
~(ua) /\ plus01 /\ t01 /\ fRed /\ a12)),

[occ_03] InterlockingSystem |-
G(cnt_03 > 0 => ~(t03)),

[conflicting_route_2] InterlockingSystem |-
G(plus01 /\ plus02 /\ ~(ia) =>
~(plus01 /\ ~(ua)) /\ ~(minus01 /\ minus02 /\ ~(ia)) /\
~(plus01 /\ plus02 /\ ~(ib)) /\ ~(minus01 /\ ~(ua)) /\
~(minus01 /\ minus02 /\ ~(ib)) /\ ~(minus02 /\ ~(ub))),

[occ_03_] InterlockingSystem |-
G(~(t03) => cnt_03 > 0),

[conflicting_route_10] InterlockingSystem |-
G(minus02 /\ ~(ub) =>
~(plus01 /\ plus02 /\ ~(ib)) /\ ~(plus02 /\ ~(ub)) /\
~(minus01 /\ minus02 /\ ~(ib)) /\ ~(plus01 /\ plus02 /\
~(ia))),

[occ_04] InterlockingSystem |-
G(cnt_04 > 0 => ~(t04)),

[conflicting_route_3] InterlockingSystem |-
G(minus01 /\ minus02 /\ ~(ia) =>
~(plus01 /\ ~(ua)) /\ ~(minus01 /\ ~(ua)) /\
~(plus01 /\ plus02 /\ ~(ib)) /\
~(plus01 /\ plus02 /\ ~(ia)) /\
~(minus01 /\ minus02 /\ ~(ib)) /\ ~(plus02 /\ ~(ub))),

```

```

[occ_04_] InterlockingSystem |-
G(~(t04) => cnt_04 > 0),

[conflicting_route_9] InterlockingSystem |-
G(plus02 /\ ~(ub) =>
~(plus01 /\ plus02 /\ ~(ib)) /\
~(minus01 /\ minus02 /\ ~(ia)) /\
~(minus01 /\ minus02 /\ ~(ib)) /\
~(minus02 /\ ~(ub))),

[occ_B12_] InterlockingSystem |-
G(~(b12) => cnt_B12 > 0),

[conflicting_route_5] InterlockingSystem |-
G(plus01 /\ plus02 /\ ~(ib) =>
~(minus01 /\ minus02 /\ ~(ia)) /\ ~(plus02 /\ ~(ub)) /\
~(minus01 /\ ~(ua)) /\ ~(plus01 /\ plus02 /\ ~(ia)) /\
~(minus01 /\ minus02 /\ ~(ib)) /\ ~(minus02 /\ ~(ub))),

[occ_B12] InterlockingSystem |-
G(cnt_B12 > 0 => ~(b12)),

[conflicting_route_8] InterlockingSystem |-
G(minus01 /\ ~(ua) =>
~(minus01 /\ minus02 /\ ~(ia)) /\
~(plus01 /\ ~(ua)) /\
~(plus01 /\ plus02 /\ ~(ib)) /\
~(plus01 /\ plus02 /\ ~(ia))),

[direction_A12] InterlockingSystem |-
G(fwd_A12 \/ bwd_A12 => cnt_A12 > 0),

[conflicting_route_6] InterlockingSystem |-
G(minus01 /\ minus02 /\ ~(ib) =>
~(minus01 /\ minus02 /\ ~(ia)) /\
~(plus02 /\ ~(ub)) /\ ~(plus01 /\ ~(ua)) /\
~(plus01 /\ plus02 /\ ~(ia)) /\
~(plus01 /\ plus02 /\ ~(ib)) /\ ~(minus02 /\ ~(ub))),

[direction_01] InterlockingSystem |-
G(fwd_01 \/ bwd_01 => cnt_01 > 0),

[conflicting_route_7] InterlockingSystem |-
G(plus01 /\ ~(ua) =>

```

```

~(minus01 /\ minus02 /\ ~(ia)) /\
~(plus01 /\ plus02 /\ ~(ia)) /\
~(minus01 /\ minus02 /\ ~(ib)) /\
~(minus01 /\ ~(ua))),

[direction_02] InterlockingSystem |-
G(fwd_02 \/ bwd_02 => cnt_02 > 0),

[con_01_04] InterlockingSystem |-
G(con_01_04 => cnt_01 > 0 /\ cnt_04 > 0),

[direction_03] InterlockingSystem |-
G(fwd_03 \/ bwd_03 => cnt_03 > 0),

[con_02_03] InterlockingSystem |-
G(con_02_03 => cnt_02 > 0 /\ cnt_03 > 0),

[direction_04] InterlockingSystem |-
G(fwd_04 \/ bwd_04 => cnt_04 > 0),

[con_01_02] InterlockingSystem |-
G(con_01_02 => cnt_01 > 0 /\ cnt_02 > 0),

[direction_B12] InterlockingSystem |-
G(fwd_B12 \/ bwd_B12 => cnt_B12 > 0),

[con_03_04] InterlockingSystem |-
G(con_03_04 => cnt_03 > 0 /\ cnt_04 > 0),

[con_03_B12] InterlockingSystem |-
G(con_03_B12 => cnt_03 > 0 /\ cnt_B12 > 0),

[con_A12_01] InterlockingSystem |-
G(con_A12_01 => cnt_A12 > 0 /\ cnt_01 > 0)
"

```

F.2 Output of Model Checker

Summary:

The assertion 'initIdle' located at
[Context: combined, line(5055), column(0)] is valid.

The assertion 'alwaysEventuallyIdle' located at [Context: combined, line(5056), column(0)] is valid.
The assertion 'gk_gkAG_downConcurrency' located at [Context: combined, line(5057), column(0)] is valid.
The assertion 'gk_gkAG_upConcurrency' located at [Context: combined, line(5076), column(0)] is valid.
The assertion 'gk_gkBE_downConcurrency' located at [Context: combined, line(5095), column(0)] is valid.
The assertion 'gk_gkBE_upConcurrency' located at [Context: combined, line(5114), column(0)] is valid.
The assertion 'lockingIA1_r35_downConcurrency' located at [Context: combined, line(5133), column(0)] is valid.
The assertion 'lockingIA1_r35_upConcurrency' located at [Context: combined, line(5204), column(0)] is valid.
The assertion 'lockingIA1_r44_downConcurrency' located at [Context: combined, line(5275), column(0)] is valid.
The assertion 'lockingIA1_r44_upConcurrency' located at [Context: combined, line(5294), column(0)] is valid.
The assertion 'lockingIA2_ia_downConcurrency' located at [Context: combined, line(5313), column(0)] is valid.
The assertion 'lockingIA2_ia_mutualExclusion' located at [Context: combined, line(5324), column(0)] is valid.
The assertion 'lockingIA2_ia_upConcurrency' located at [Context: combined, line(5335), column(0)] is valid.
The assertion 'lockingIA2_iadub_downConcurrency' located at [Context: combined, line(5352), column(0)] is valid.
The assertion 'lockingIA2_iadub_mutualExclusion' located at [Context: combined, line(5361), column(0)] is valid.
The assertion 'lockingIA2_iadub_upConcurrency' located at [Context: combined, line(5370), column(0)] is valid.
The assertion 'lockingIB1_r36_downConcurrency' located at [Context: combined, line(5385), column(0)] is valid.
The assertion 'lockingIB1_r36_upConcurrency' located at [Context: combined, line(5456), column(0)] is valid.
The assertion 'lockingIB1_r45_downConcurrency' located at [Context: combined, line(5527), column(0)] is valid.
The assertion 'lockingIB1_r45_upConcurrency' located at [Context: combined, line(5546), column(0)] is valid.
The assertion 'lockingIB2_ib_downConcurrency' located at [Context: combined, line(5565), column(0)] is valid.
The assertion 'lockingIB2_ib_mutualExclusion' located at [Context: combined, line(5576), column(0)] is valid.
The assertion 'lockingIB2_ib_upConcurrency' located at [Context: combined, line(5587), column(0)] is valid.

The assertion 'lockingIB2_ibdub_downConcurrency' located at [Context: combined, line(5604), column(0)] is valid.
The assertion 'lockingIB2_ibdub_mutualExclusion' located at [Context: combined, line(5613), column(0)] is valid.
The assertion 'lockingIB2_ibdub_upConcurrency' located at [Context: combined, line(5622), column(0)] is valid.
The assertion 'lockingUA1_ef111_downConcurrency' located at [Context: combined, line(5637), column(0)] is valid.
The assertion 'lockingUA1_ef111_upConcurrency' located at [Context: combined, line(5684), column(0)] is valid.
The assertion 'lockingUA1_ef411_downConcurrency' located at [Context: combined, line(5731), column(0)] is valid.
The assertion 'lockingUA1_ef411_upConcurrency' located at [Context: combined, line(5746), column(0)] is valid.
The assertion 'lockingUA2_ua_downConcurrency' located at [Context: combined, line(5761), column(0)] is valid.
The assertion 'lockingUA2_ua_mutualExclusion' located at [Context: combined, line(5770), column(0)] is valid.
The assertion 'lockingUA2_ua_upConcurrency' located at [Context: combined, line(5790), column(0)] is valid.
The assertion 'lockingUA2_uadub_downConcurrency' located at [Context: combined, line(5827), column(0)] is valid.
The assertion 'lockingUA2_uadub_mutualExclusion' located at [Context: combined, line(5834), column(0)] is valid.
The assertion 'lockingUA2_uadub_upConcurrency' located at [Context: combined, line(5851), column(0)] is valid.
The assertion 'lockingUB1_gh111_downConcurrency' located at [Context: combined, line(5884), column(0)] is valid.
The assertion 'lockingUB1_gh111_upConcurrency' located at [Context: combined, line(5931), column(0)] is valid.
The assertion 'lockingUB1_gh411_downConcurrency' located at [Context: combined, line(5978), column(0)] is valid.
The assertion 'lockingUB1_gh411_upConcurrency' located at [Context: combined, line(5993), column(0)] is valid.
The assertion 'lockingUB2_ub_downConcurrency' located at [Context: combined, line(6008), column(0)] is valid.
The assertion 'lockingUB2_ub_mutualExclusion' located at [Context: combined, line(6017), column(0)] is valid.
The assertion 'lockingUB2_ub_upConcurrency' located at [Context: combined, line(6029), column(0)] is valid.
The assertion 'lockingUB2_ubdub_downConcurrency' located at [Context: combined, line(6050), column(0)] is valid.
The assertion 'lockingUB2_ubdub_mutualExclusion' located at [Context: combined, line(6057), column(0)] is valid.

The assertion 'lockingUB2_ubdub_upConcurrency' located at [Context: combined, line(6067), column(0)] is valid.
The assertion 'releaseIA_r47_downConcurrency' located at [Context: combined, line(6086), column(0)] is valid.
The assertion 'releaseIA_r47_upConcurrency' located at [Context: combined, line(6145), column(0)] is valid.
The assertion 'releaseIA_r67_downConcurrency' located at [Context: combined, line(6204), column(0)] is valid.
The assertion 'releaseIA_r67_upConcurrency' located at [Context: combined, line(6263), column(0)] is valid.
The assertion 'releaseIA_r87_downConcurrency' located at [Context: combined, line(6322), column(0)] is valid.
The assertion 'releaseIA_r87_upConcurrency' located at [Context: combined, line(6379), column(0)] is valid.
The assertion 'releaseIB_r50_downConcurrency' located at [Context: combined, line(6436), column(0)] is valid.
The assertion 'releaseIB_r50_upConcurrency' located at [Context: combined, line(6495), column(0)] is valid.
The assertion 'releaseIB_r70_downConcurrency' located at [Context: combined, line(6554), column(0)] is valid.
The assertion 'releaseIB_r70_upConcurrency' located at [Context: combined, line(6613), column(0)] is valid.
The assertion 'releaseIB_r90_downConcurrency' located at [Context: combined, line(6672), column(0)] is valid.
The assertion 'releaseIB_r90_upConcurrency' located at [Context: combined, line(6729), column(0)] is valid.
The assertion 'releaseUA_r48_downConcurrency' located at [Context: combined, line(6786), column(0)] is valid.
The assertion 'releaseUA_r48_upConcurrency' located at [Context: combined, line(6807), column(0)] is valid.
The assertion 'releaseUA_r88_downConcurrency' located at [Context: combined, line(6828), column(0)] is valid.
The assertion 'releaseUA_r88_upConcurrency' located at [Context: combined, line(6855), column(0)] is valid.
The assertion 'releaseUB_r49_downConcurrency' located at [Context: combined, line(6882), column(0)] is valid.
The assertion 'releaseUB_r49_upConcurrency' located at [Context: combined, line(6903), column(0)] is valid.
The assertion 'releaseUB_r89_downConcurrency' located at [Context: combined, line(6924), column(0)] is valid.
The assertion 'releaseUB_r89_upConcurrency' located at [Context: combined, line(6951), column(0)] is valid.
The assertion 'sasb_sa12_downConcurrency' located at [Context: combined, line(6978), column(0)] is valid.

The assertion 'sasb_sa12_upConcurrency' located at [Context: combined, line(6991), column(0)] is valid.
The assertion 'sasb_sb12_downConcurrency' located at [Context: combined, line(7004), column(0)] is valid.
The assertion 'sasb_sb12_upConcurrency' located at [Context: combined, line(7017), column(0)] is valid.
The assertion 'signalA_aGreen2_downConcurrency' located at [Context: combined, line(7030), column(0)] is valid.
The assertion 'signalA_aGreen2_upConcurrency' located at [Context: combined, line(7211), column(0)] is valid.
The assertion 'signalA_aGreen_downConcurrency' located at [Context: combined, line(7392), column(0)] is valid.
The assertion 'signalA_aGreen_upConcurrency' located at [Context: combined, line(7699), column(0)] is valid.
The assertion 'signalA_aRed_downConcurrency' located at [Context: combined, line(8006), column(0)] is valid.
The assertion 'signalA_aRed_upConcurrency' located at [Context: combined, line(8013), column(0)] is valid.
The assertion 'signalA_aYellow_downConcurrency' located at [Context: combined, line(8020), column(0)] is valid.
The assertion 'signalA_aYellow_upConcurrency' located at [Context: combined, line(8031), column(0)] is valid.
The assertion 'signalB_bGreen2_downConcurrency' located at [Context: combined, line(8042), column(0)] is valid.
The assertion 'signalB_bGreen2_upConcurrency' located at [Context: combined, line(8235), column(0)] is valid.
The assertion 'signalB_bGreen_downConcurrency' located at [Context: combined, line(8428), column(0)] is valid.
The assertion 'signalB_bGreen_upConcurrency' located at [Context: combined, line(8759), column(0)] is valid.
The assertion 'signalB_bRed_downConcurrency' located at [Context: combined, line(9090), column(0)] is valid.
The assertion 'signalB_bRed_upConcurrency' located at [Context: combined, line(9097), column(0)] is valid.
The assertion 'signalB_bYellow_downConcurrency' located at [Context: combined, line(9104), column(0)] is valid.
The assertion 'signalB_bYellow_upConcurrency' located at [Context: combined, line(9115), column(0)] is valid.
The assertion 'signalEF_eGreen_downConcurrency' located at [Context: combined, line(9126), column(0)] is valid.
The assertion 'signalEF_eGreen_upConcurrency' located at [Context: combined, line(9143), column(0)] is valid.
The assertion 'signalEF_eRed_downConcurrency' located at [Context: combined, line(9160), column(0)] is valid.

The assertion 'signalEF_eRed_upConcurrency' located at [Context: combined, line(9165), column(0)] is valid.
The assertion 'signalEF_fGreen_downConcurrency' located at [Context: combined, line(9170), column(0)] is valid.
The assertion 'signalEF_fGreen_upConcurrency' located at [Context: combined, line(9187), column(0)] is valid.
The assertion 'signalEF_fRed_downConcurrency' located at [Context: combined, line(9204), column(0)] is valid.
The assertion 'signalEF_fRed_upConcurrency' located at [Context: combined, line(9209), column(0)] is valid.
The assertion 'signalGH_gGreen_downConcurrency' located at [Context: combined, line(9214), column(0)] is valid.
The assertion 'signalGH_gGreen_upConcurrency' located at [Context: combined, line(9231), column(0)] is valid.
The assertion 'signalGH_gRed_downConcurrency' located at [Context: combined, line(9248), column(0)] is valid.
The assertion 'signalGH_gRed_upConcurrency' located at [Context: combined, line(9253), column(0)] is valid.
The assertion 'signalGH_hGreen_downConcurrency' located at [Context: combined, line(9258), column(0)] is valid.
The assertion 'signalGH_hGreen_upConcurrency' located at [Context: combined, line(9275), column(0)] is valid.
The assertion 'signalGH_hRed_downConcurrency' located at [Context: combined, line(9292), column(0)] is valid.
The assertion 'signalGH_hRed_upConcurrency' located at [Context: combined, line(9297), column(0)] is valid.
The assertion 'no_derailing_03_02' located at [Context: combined, line(9302), column(0)] is valid.
The assertion 'train_route_release_7' located at [Context: combined, line(9304), column(0)] is valid.
The assertion 'no_derailing_01_04' located at [Context: combined, line(9314), column(0)] is valid.
The assertion 'train_route_release_6' located at [Context: combined, line(9316), column(0)] is valid.
The assertion 'no_derailing_03_04' located at [Context: combined, line(9327), column(0)] is valid.
The assertion 'train_route_release_8' located at [Context: combined, line(9329), column(0)] is valid.
The assertion 'no_derailing_01_02' located at [Context: combined, line(9339), column(0)] is valid.
The assertion 'train_route_release_5' located at [Context: combined, line(9341), column(0)] is valid.
The assertion 'points_locked_when_occupied_03' located at [Context: combined, line(9352), column(0)] is valid.

The assertion 'train_route_release_9' located at
[Context: combined, line(9355), column(0)] is valid.
The assertion 'points_locked_when_occupied_01' located at
[Context: combined, line(9365), column(0)] is valid.
The assertion 'train_route_release_3' located at
[Context: combined, line(9368), column(0)] is valid.
The assertion 'no_collision_A12' located at
[Context: combined, line(9379), column(0)] is valid.
The assertion 'train_route_release_10' located at
[Context: combined, line(9380), column(0)] is valid.
The assertion 'no_collision_01' located at
[Context: combined, line(9390), column(0)] is valid.
The assertion 'train_route_release_2' located at
[Context: combined, line(9391), column(0)] is valid.
The assertion 'no_collision_02' located at
[Context: combined, line(9402), column(0)] is valid.
The assertion 'signal_release_7' located at
[Context: combined, line(9403), column(0)] is valid.
The assertion 'no_collision_03' located at
[Context: combined, line(9406), column(0)] is valid.
The assertion 'signal_release_6' located at
[Context: combined, line(9407), column(0)] is valid.
The assertion 'no_collision_04' located at
[Context: combined, line(9410), column(0)] is valid.
The assertion 'signal_release_8' located at
[Context: combined, line(9411), column(0)] is valid.
The assertion 'no_collision_B12' located at
[Context: combined, line(9414), column(0)] is valid.
The assertion 'signal_release_5' located at
[Context: combined, line(9415), column(0)] is valid.
The assertion 'points_configuration_01' located at
[Context: combined, line(9418), column(0)] is valid.
The assertion 'signal_release_9' located at
[Context: combined, line(9421), column(0)] is valid.
The assertion 'points_configuration_03' located at
[Context: combined, line(9424), column(0)] is valid.
The assertion 'signal_release_3' located at
[Context: combined, line(9427), column(0)] is valid.
The assertion 'free_B12' located at
[Context: combined, line(9430), column(0)] is valid.
The assertion 'signal_release_10' located at
[Context: combined, line(9432), column(0)] is valid.
The assertion 'free_B12_' located at
[Context: combined, line(9435), column(0)] is valid.

The assertion 'signal_release_2' located at
[Context: combined, line(9437), column(0)] is valid.
The assertion 'free_04_' located at
[Context: combined, line(9440), column(0)] is valid.
The assertion 'point_position_ua' located at
[Context: combined, line(9442), column(0)] is valid.
The assertion 'free_04' located at
[Context: combined, line(9445), column(0)] is valid.
The assertion 'point_position_ib' located at
[Context: combined, line(9447), column(0)] is valid.
The assertion 'free_03_' located at
[Context: combined, line(9452), column(0)] is valid.
The assertion 'point_position_ub' located at
[Context: combined, line(9454), column(0)] is valid.
The assertion 'free_03' located at
[Context: combined, line(9457), column(0)] is valid.
The assertion 'point_position_ia' located at
[Context: combined, line(9459), column(0)] is valid.
The assertion 'free_02_' located at
[Context: combined, line(9464), column(0)] is valid.
The assertion 'signal_E' located at
[Context: combined, line(9466), column(0)] is valid.
The assertion 'free_02' located at
[Context: combined, line(9469), column(0)] is valid.
The assertion 'signal_F' located at
[Context: combined, line(9471), column(0)] is valid.
The assertion 'free_01_' located at
[Context: combined, line(9474), column(0)] is valid.
The assertion 'signal_B' located at
[Context: combined, line(9476), column(0)] is valid.
The assertion 'free_01' located at
[Context: combined, line(9479), column(0)] is valid.
The assertion 'signal_G' located at
[Context: combined, line(9481), column(0)] is valid.
The assertion 'free_A12_' located at
[Context: combined, line(9484), column(0)] is valid.
The assertion 'signal_A' located at
[Context: combined, line(9486), column(0)] is valid.
The assertion 'free_A12' located at
[Context: combined, line(9489), column(0)] is valid.
The assertion 'signal_H' located at
[Context: combined, line(9491), column(0)] is valid.
The assertion 'occ_A12' located at
[Context: combined, line(9494), column(0)] is valid.

The assertion 'signal_exp_H' located at
[Context: combined, line(9496), column(0)] is valid.
The assertion 'occ_A12_' located at
[Context: combined, line(9503), column(0)] is valid.
The assertion 'signal_exp_A' located at
[Context: combined, line(9505), column(0)] is valid.
The assertion 'occ_01' located at
[Context: combined, line(9537), column(0)] is valid.
The assertion 'signal_exp_G' located at
[Context: combined, line(9539), column(0)] is valid.
The assertion 'occ_01_' located at
[Context: combined, line(9546), column(0)] is valid.
The assertion 'signal_exp_B' located at
[Context: combined, line(9548), column(0)] is valid.
The assertion 'occ_02' located at
[Context: combined, line(9580), column(0)] is valid.
The assertion 'signal_exp_F' located at
[Context: combined, line(9582), column(0)] is valid.
The assertion 'occ_02_' located at
[Context: combined, line(9589), column(0)] is valid.
The assertion 'signal_exp_E' located at
[Context: combined, line(9591), column(0)] is valid.
The assertion 'occ_03' located at
[Context: combined, line(9598), column(0)] is valid.
The assertion 'conflicting_route_2' located at
[Context: combined, line(9600), column(0)] is valid.
The assertion 'occ_03_' located at
[Context: combined, line(9618), column(0)] is valid.
The assertion 'conflicting_route_10' located at
[Context: combined, line(9620), column(0)] is valid.
The assertion 'occ_04' located at
[Context: combined, line(9633), column(0)] is valid.
The assertion 'conflicting_route_3' located at
[Context: combined, line(9635), column(0)] is valid.
The assertion 'occ_04_' located at
[Context: combined, line(9653), column(0)] is valid.
The assertion 'conflicting_route_9' located at
[Context: combined, line(9655), column(0)] is valid.
The assertion 'occ_B12_' located at
[Context: combined, line(9668), column(0)] is valid.
The assertion 'conflicting_route_5' located at
[Context: combined, line(9670), column(0)] is valid.
The assertion 'occ_B12' located at
[Context: combined, line(9688), column(0)] is valid.

The assertion 'conflicting_route_8' located at
[Context: combined, line(9690), column(0)] is valid.
The assertion 'direction_A12' located at
[Context: combined, line(9703), column(0)] is valid.
The assertion 'conflicting_route_6' located at
[Context: combined, line(9706), column(0)] is valid.
The assertion 'direction_01' located at
[Context: combined, line(9724), column(0)] is valid.
The assertion 'conflicting_route_7' located at
[Context: combined, line(9727), column(0)] is valid.
The assertion 'direction_02' located at
[Context: combined, line(9740), column(0)] is valid.
The assertion 'con_01_04' located at
[Context: combined, line(9743), column(0)] is valid.
The assertion 'direction_03' located at
[Context: combined, line(9746), column(0)] is valid.
The assertion 'con_02_03' located at
[Context: combined, line(9749), column(0)] is valid.
The assertion 'direction_04' located at
[Context: combined, line(9752), column(0)] is valid.
The assertion 'con_01_02' located at
[Context: combined, line(9755), column(0)] is valid.
The assertion 'direction_B12' located at
[Context: combined, line(9758), column(0)] is valid.
The assertion 'con_03_04' located at
[Context: combined, line(9761), column(0)] is valid.
The assertion 'con_03_B12' located at
[Context: combined, line(9764), column(0)] is valid.
The assertion 'con_A12_01' located at
[Context: combined, line(9767), column(0)] is valid.

F.3 Output of Tests

F.3.1 Station Layout Diagram

```
[isWfIdentifiers_fail] true  
[isWfIdentifiers_succeed] true  
[neighbours_OnlySections_fail] true  
[neighbours_OnlySections_succeed] true  
[neighbours_EverySection_fail] true
```

```
[neighbours_EverySection_succeed] true
[neighbours_Irreflexive_linear_fail] true
[neighbours_Irreflexive_points_fail] true
[neighbours_Antisymmetrical_fail] true
[neighbours_Antisymmetrical_succeed] true
[linears_1or2Neighbours_fail_1] true
[linears_1or2Neighbours_fail_2] true
[linears_1or2Neighbours_succeed_1] true
[linears_1or2Neighbours_succeed_2] true
[points_3Neighbours_fail_1] true
[points_3Neighbours_fail_2] true
[points_3Neighbours_succeed] true
[pointsNotNeighbours_fail] true
[pointsNotNeighbours_succeed] true
[branchNeighbours_areNeighbours_fail] true
[branchNeighbours_areNeighbours_succeed] true
[branchNeighbours_FirstIsPoints_fail] true
[branchNeighbours_FirstIsPoints_succeed] true
[branchNeighbours_everyPoints_fail] true
[branchNeighbours_everyPoints_succeed] true
[signalLocation_UsesExistingSignals_fail] true
[signalLocation_UsesExistingSignals_succeed] true
[signalsArePlacedAtValtIdNeighbours_fail] true
[signalsArePlacedAtValtIdNeighbours_succeed] true
[pointMachines_areSymmetrical_fail1] true
[pointMachines_areSymmetrical_fail2] true
[pointMachines_areSymmetrical_fail3] true
[pointMachines_areSymmetrical_succeed] true
[pointMachines_ExistingTrackSections_fail] true
[pointMachines_ExistingTrackSections_succeed] true
```

F.3.2 Train Route Table

```
[noOverlap] true
[overlapNotPartofRoute] true
[overlapPartofRoute] true
[overlapNotConnected] true
[overlapConnectedA] true
[overlapConnectedB] true
[overlapNotEOR] true
[overlapEOR] true
[nonExistingSignals] true
```

```
[existingSignals] true
[stopEntry] true
[proceedEntry] true
[nonexistingPointMachine] true
[existingPointMachines] true
[pointMachinesNotInTRT] true
[pointMachineNotInTRT] true
[pointMachinesInTRT] true
[pointsInRouteLocked_fail] true
[pointsInRouteLocked_success] true
[pointsNotInCorrectPos] true
[pointsInCorrectPos] true
[nonexistingTrackSections] true
[tooFewTrackSections] true
[tooManyTrackSections] true
[existingTrackSections] true
[emptyTrainDetection] true
[noSectionInRoute] true
[atLeastOneSectionInRoute] true
[notConnectedRoute] true
[connectedRoute] true
[justOneBranchNeighbour_fail] true
[justOneBranchNeighbour] true
[nonexistingSignal] true
[existingSignal] true
[nonexistingTrackSection] true
[existingTrackSection] true
[signalNotFollowedByTrackSection] true
[signalFollowedByTrackSection] true
[notReverseTrackSections] true
[reverseTrackSections] true
[notReverseRelayState] true
[reverseRelayState] true
[trackSectionNotPartOfRoute] true
[trackSectionPartOfRoute] true
[improperRelayState1] true
[improperRelayState2] true
[improperRelayState3] true
[properRelayState] true
[notEveryExistingRoute_few] true
[notEveryExistingRoute_wrong] true
[notEveryExistingRoute_many] true
[everyExistingRoute] true
[notconflictingIfShareSectionB] true
```

```
[conflictingIfShareSection] true
[notConflictingWhenDifferentPos] true
[conflictingWhenDifferentPos_samePos] true
[conflictingWhenDifferentPos] true
```

F.3.3 Transition System

```
[emptySystemName] true
[nonemptySystemName] true
[identicalIds] true
[uniqueIds] true
[invalidIds] true
[validIds] true
[emptyTransitionName] true
[nonemptyTransitionName] true
[identicalTransitionNames] true
[uniqueTransitionNames] true
[nonExistingIdsInGuard] true
[existingIdsInGuard] true
[emptyAssignment] true
[nonemptyAssignment] true
[assignToNonexistingIds] true
[assignToExistingIds] true
[idUpdMoreThanOnce] true
[idUpdOnlyOnce] true
[emptyAssertionName] true
[nonemptyAssertionName] true
[identicalAssertionNames] true
[uniqueAssertionNames] true
[nonexistingVars] true
[existingVars] true
```

F.3.4 Object Relay Associations

```
[noRelayVariableOverlap_failA] true
[noRelayVariableOverlap_failB] true
[noRelayVariableOverlap_failC] true
[noRelayVariableOverlap_failD] true
[noRelayVariableOverlap_failE] true
[noRelayVariableOverlap_failF] true
```

```
[noRelayVariableOverlap_failG] true
[noRelayVariableOverlap_failH] true
[noRelayVariableOverlap_ok] true
[routeAssoc_fail] true
[routeAssoc_ok] true
[trackAssoc_fail] true
[trackAssoc_ok] true
[pointAssoc_existing_fail] true
[pointAssoc_different_fail1] true
[pointAssoc_different_fail2] true
[pointAssoc_ok] true
[signalAssoc_existing_fail] true
[signalAssoc_different_fail] true
[signalAssoc_ok] true
```

F.3.5 Train Movement Associations

```
[noRelayVariableOverlap_failA] true
[noRelayVariableOverlap_failB] true
[noRelayVariableOverlap_failC] true
[noRelayVariableOverlap_failD] true
[noRelayVariableOverlap_failE] true
[noRelayVariableOverlap_failF] true
[noRelayVariableOverlap_failG] true
[noRelayVariableOverlap_failH] true
[noRelayVariableOverlap_failI] true
[noRelayVariableOverlap_ok] true
[conVarsValidDomain_fail] true
[conVarsValidRange_fail] true
[connectionVars] true
[cntVarsValidDomain_fail] true
[cntVarsValidRange] true
[counterVars] true
[fwdVarsValidDomain_fail] true
[fwdVarsValidRange_fail] true
[directionFwdVars] true
[bwdVarsValidDomain_fail] true
[bwdVarsValidRange_fail] true
[directionBwdVars] true
[directionsValidDomain_fail] true
[bothDirections_fail] true
[directions] true
```