Modelling Interlocking Systems for Railway Stations

Marie Le Bliguet
Andreas Andersen Kjær

Kongens Lyngby 2008
IMM-M.Sc.-2008-68

Technical University of Denmark
Department of 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

Interlocking systems are used for ensuring the safety of trains. This master thesis is made in cooperation with Banedanmark and deals with relay-based interlocking systems for railway stations. The goal of this project is to develop a formal method for verifying that such systems guarantee the safety of trains.

By using RSL models of interlocking systems, this thesis deduces an automated procedure for making an RSL-SAL transition system that defines the dynamic behaviour of an interlocking system. Also, the procedure specifies how to auto-generate confidence conditions for the generated transition system, formulated using Linear Temporal Logic (LTL). Finally, a tool for computing such a transition system and its associated confidence conditions is implemented using Java.

Furthermore, this thesis develops patterns for specifying the behaviour of external inputs to an interlocking system (e.g. a rule can define when a train can enter a station), formulated using RSL-SAL. Also, patterns for specifying safety properties are developed using LTL.

Altogether, the tool and patterns define a method that uses the state-space based model checker SAL for verifying that interlocking systems guarantee the safety of trains. The method has successfully been applied to a Danish railway station.

Keywords: Formal methods, Linear Temporal Logic, Model checking, Modelling, Railway interlocking systems, RAISE Specification Language, RSL-SAL, SAL.

Resumé

Sikringsanlæg benyttes til at sikre togenes sikkerhed. Dette kandidatspeciale er lavet i samarbejde med Banedanmark og omhandler relæbaserede stationssikringsanlæg. Formålet med dette projekt er at udvikle en formel metode til verifikation af, at sådanne systemer garanterer togenes sikkerhed.

Ved brug af RSL-modeller af sikringsanlæg, udleder dette speciale en automatiseret procedure til at lave et RSL-SAL transitionssystem, der definerer et sikringsanlægs dynamiske adfærd. Proceduren specificerer også, hvordan konfidensbetingelser for det generede transitionssystem kan udledes automatisk og formuleres i form a Linear Temporal Logic (LTL). Til sidst laves en Java-implementering af et værktøj, der kan generere sådan et transitionssystem og dets tilhørende konfidensbetingelser.

Yderligere udvikler dette kandidatspeciale mønstre (formuleret ved brug af RSL-SAL) for eksterne inputs til et sikringsanlæg (f.eks. kan en regel definere, hvornår et tog kan ankomme til en station). Der udvikles også mønstre for hvordan sikkerhedsegenskaber kan specificeres ved brug af LTL.

Tilsammen definerer værktøjet og mønstrene en metode, der bruger den tilstandsrumbaserede model checker SAL til at verificere at stationssikringsanlæg garanterer togenes sikkerhed. Metoden er succesfuldt blevet anvendt på en dansk jernabanestation.

Nøgleord: Formelle metoder, Jernbanesikringsanlæg, Linear Temporal Logic, Model checking, Modellering, RAISE Specification Language, RSL-SAL, SAL.

Preface

This master thesis was made at the Department of Informatics and Mathematical Modelling, the Technical University of Denmark in partial fulfilment of the requirements for acquiring the M.Sc. degree in engineering.

The thesis was made in cooperation with Banedanmark. The goal of this thesis is to develop a method for model checking that interlocking system guarantee the safety of trains.

The thesis supervisors are Associate Professor Anne E. Haxthausen, Department of Informatics and Mathematical Modelling, Technical University of Denmark, and Kirsten Mark Hansen, Technical Operations and Maintenance, Interlocking Systems, Banedanmark.

Kongens Lyngby, July 2008

Marie Le Bliguet (s060764)   Andreas Andersen Kjær (s032103)

Acknowledgements

The authors of this thesis would like to thank:

Anne E. Haxthausen for showing great interests in this thesis, for useful feedback, and for many valuable inputs and discussions during the project.

Kirsten Mark Hansen for giving a detailed introduction to the domain of this thesis, for many beneficial domain related discussions, and for being available for questions during the project.

Chris George, International Institute for Software Technology, United Nations University, for help related to the RSL-SAL tool and for spending time on e-mail correspondences with the authors during the project.

Troels Andersen Kjær for proofreading the entire report.

Contents

Chapter 1

Introduction

1.1 The background of the project

When a train enters or leaves a railway station, it is important to be sure that it does not derail and does not collide with another train. Therefore, rules have to be made for when a train can enter and leave a station. Like other railway enterprises, the owner of the main part of the Danish railways, *Banedanmark*, uses *interlocking systems* for ensuring that the safety rules are respected. Such systems are deployed for enforcing these rules on the physical objects of the stations. For instance, it must be ensured that the correct signals are displayed to the drivers of the trains.

Most of the interlocking systems in Denmark are *relay- and relay group systems.* These systems have been used for Danish railway stations since the 1950's. Relay- and relay group systems consist of complex electrical circuits that respond to inputs from the *external world.* For instance, such an input can be that a train occupies a track section or when an operator pushes a button. If an operator pushes a button and thereby is telling the interlocking system to authorise a train to enter a station, the interlocking system must only let the train enter the station if it is considered as being safe.

Banedanmark documents relay- and relay group systems using large *relay di-*

agrams. These diagrams can be considered as a snapshot of an interlocking system in a specific state. When responding to the external world, an interlocking system enters new states that are not shown in the *relay diagrams*.

Until now, Banedanmark has verified that relay- and relay group systems work as they are supposed to by inspecting diagrams manually and by testing the systems after deploying them. The inspection of diagrams is done informally without any tool or proof technique. Even though Banedanmark has many years of experience in inspections and tests, this procedure alone cannot fully guarantee that safety rules are enforced by the interlocking systems. There might exist an unusual state of the interlocking system where some safety rules are not enforced by it, making derailing or collision possible.

1.2 The goal of the project

The goal of the project is to introduce a new way of verifying that interlocking systems for railway stations actually enforce safety. A general method to verify safety properties should be developed and afterwards applied to an interlocking system of a railway station. Instead of relying on informal inspections, the method should provide a formal way of verifying that interlocking systems guarantee the safety of trains at a station.

1.3 Main approach to solving the problem

Due to the complexity of interlocking systems, it might be difficult and time-consuming to manually prove the safety of trains at a given station. Therefore, it has been decided to model interlocking systems in such a way that safety properties can be checked by an automated verification tool.

To enable an automated verification of safety properties, one needs to model the complete behaviour of an interlocking system and formulate safety properties for it. As this is not a trivial task, this project introduces a method for obtaining them. The method is illustrated in figure 1.1 (a more detailed method overview will be given in chapter 4).

By examining the documentation of a station including diagrams and station layout, a transition system containing the following items can be formulated:

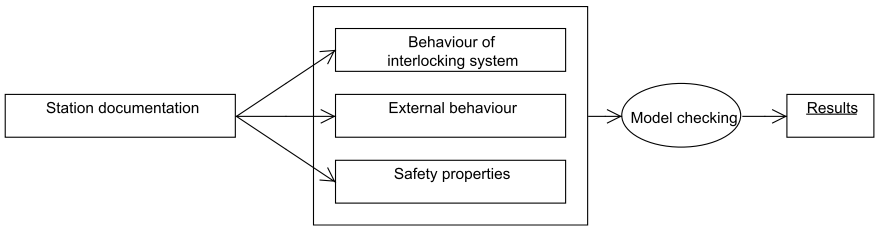


Figure 1.1: The strategy for checking safety properties of an interlocking system

- Behaviour of the interlocking system of the station: rules for how the interlocking system responds to the external world (e.g. rules for what happens when track sections are being occupied).
- External behaviour: rules for how the external world behaves (e.g. rules for when track sections can be occupied).
- Safety properties: properties associated with the transition system that can be checked by a model checker.

When formulating behaviour and safety properties in a transition system, the properties can be verified by a checker. However, formulating such a transition system is not a trivial task. In order to ease the verification process, this project will provide:

- A tool for auto-generating the behaviour of an interlocking system by inspection of diagrams.
- Patterns for how external behaviour can be formulated.
- Patterns for how safety properties can be formulated.

When developing a tool for generating the behaviour of an interlocking system, it is essential to know that the generated models are correct. For this reason, formal methods will be preferred in the modelling phase and in the analysis leading to the formulation of such a generation.

The Raise Specification Language (*RSL*) [9] has been chosen as a modelling language. It allows for the needed abstraction in the early modelling phase and gives the possibility of making concrete specifications later in the project. Also, the advantage of using *RSL* is that it has been extended with the possibility of making transition systems for which properties can be verified by a model checker called *Symbolic Analysis Laboratory* (*SAL*) [2]. By selecting *RSL*, all models can be specified using the same language. Some models can be specified using the ordinary *RSL* features and some can be specified using the new extension, *RSL-SAL* [13] [14].

As detailed in chapter 4, two *RSL* models will be given: an abstract (i.e. an algebraic and property-oriented) model and a concrete (i.e. model-oriented, but still generic) model. When having formulated the concrete model, it will be translated to Java. The Java implementation of the concrete model will be able to take an XML representation of relay diagrams as input and give an *RSL-SAL* transition system as output that represents the behaviour of an interlocking system.

After having auto-generated the behaviour of an interlocking system using the Java program, one can manually add instantiations of patterns for external behaviour and safety properties to it. After that, the model checking tool *SAL* [2] can be used for verifying the properties.

1.4 Chapter overview

This thesis contains the following chapters:

Chapter 2 will introduce the domain of this thesis. The rest of the chapters will refer to the information provided in this chapter.

Chapter 3 will introduce *RSL-SAL* and *Linear Temporal Logics* (*LTL*) that will be used in the rest of the thesis.

Chapter 4 will give an overview of the method developed by this thesis to formulate and verify safety properties of an interlocking system. Also, it will explain the different development steps that are taken towards implementing a tool for generating the behaviour of an interlocking system. An overview of the work presented in chapters 5, 6, 7, 9 and 10 will be given.

Chapter 5 will introduce an abstract *RSL* model of relay diagrams.

Chapter 6 will analyse and deduce how one can use the model given in chapter 5 for computing the behaviour of an interlocking system. On an abstract level, *RSL* functions are defined for computing a transition system that defines how an interlocking system responds to inputs from the external world. Also, confidence conditions of such a transition system are introduced and a function for computing them is presented.

Chapter 7 will give patterns for how the behaviour of the external world can be added to a transition system that contains the behaviour of an interlocking system. Also, patterns for how to formulate safety properties are introduced.

Chapter 8 will apply the method developed in this project to a concrete Danish railway station.

Chapter 9 will make a concrete *RSL* model that implements the models introduced in chapters 5 and 6. This is a step towards an executable program that generates the behaviour of an interlocking system.

Chapter 10 will introduce the design and implementation of the Java program that implements the concrete model given in chapter 9.

Chapter 11 will explain how the Java implementation has been tested.

Chapter 12 will present the conclusions of this thesis. The accomplished work will be summarised and suggestions for future work will be presented.

Appendix A contains the complete specifications introduced by chapters 5 and 6.

Appendix B contains the complete specifications introduced by chapter 9.

Appendix C will explain the content of the CD attached to this thesis.

1.5 Reader assumptions

This report will assume that the reader knows of:

- The most common applicative *RSL* features. However, a reader that knows of functional programming languages like SML and logical quantifiers might be able to understand the report without knowledge of *RSL*.
- Transition systems in general.
- Temporal Logic in general.
- State-space based model checking in general.

- The following UML features: sequence diagrams and especially class diagrams. However, the primary analysis can be understood without knowledge of UML.
- The most common features of the Java language. However, the primary analysis can be understood without knowledge of Java.
- The most common features of Extensible Markup Language (XML). However, the primary analysis can be understood without knowledge of XML.

The reader is not assumed to know of the *RSL-SAL* extension of *RSL* and the language used for specifying model properties, *LTL*. The *RSL-SAL* and *LTL* features that are used are explained in chapter 3.

Chapter 2

Domain description

This chapter will give an introduction to the key concepts of Danish *railway stations* and *interlocking systems* that are relevant for this thesis. In general, most of the available literature on this subject is written in Danish. If the reader wishes to know more about Danish railways and interlocking systems in English, we recommend reading the bachelor thesis *Simulation of Relay Interlocking Systems* [7]. Information on railways and interlocking systems in general can be found in the book *Railway Operation and Control* [12].

Section 2.1 will explain how the track system of the Danish railways is divided.

Section 2.2 will introduce some of the physical objects that are located at Danish railway stations.

Section 2.3 will introduce the concept of interlocking. Banedanmark's approach to interlocking will be explained along with Banedanmark's basic safety goals and different kinds of interlocking systems.

Section 2.4 will explain the concept of *train route based interlocking* that is used by Banedanmark for ensuring safety.

Section 2.5 will introduce the components of *relay- and relay group* interlocking systems.

Section 2.6 will introduce the *diagrams* used for documenting *relay- and relay group* interlocking systems.

Section 2.7 will introduce the operator's panel that can be used for generating inputs to an interlocking system.

Section 2.8 will give an overview of the relations between an interlocking system and the physical objects of a station.

2.1 Division of the Danish track system

As described by Niels E. Jensen and Benny Mølgaard Nielsen [11], the Danish railway track system can be divided into two separate entities: *line blocks* and *stations*. This division is not only physical, but also concerns the way the railways are secured, e.g. how derailing and collisions are prevented. Securing the rails between the stations and at the stations is considered as two different and independent tasks. As the title of this project indicates, taking the tracks between the stations into consideration is beyond the scope of this project. We will only consider the safety of the trains at a station.

2.2 Physical objects

The Danish railways consist of several types of physical objects. This section will describe the objects that are considered relevant for this project.

2.2.1 Track circuits

The railway track layout consists of *track circuits* (also referred to as *track sections*) that can be divided into *linear track circuits* and *points*. Track circuits can be connected at their end points and thereby form a railway network. One can tell whether a track circuit is occupied or free thanks to captors (relays) which will be described later in section 2.5.

2.2.1.1 Linear track circuits

Linear track circuits are, as illustrated in figure 2.1, track circuits with two end points. Therefore, it will be impossible for a linear track circuit to provide new branches in a railway network.

Figure 2.1: A linear track circuit

2.2.1.2 Points

Points have three end points and they will allow two new branches in a railway network, as illustrated in figure 2.2.

Points have three possible positions: a *plus* position, a *minus* position, and an *intermediate* position. The *plus* position allows trains to go in one direction, the *minus* position allows trains to go in the other direction, and the *intermediate* position, which is used when changing from *plus* to *minus* and vice versa, might cause trains to derail. These positions are illustrated in figure 2.3.

According to Kirsten Mark Hansen from Banedanmark, there are two different conventions for deciding which directions correspond to the *plus* position and the *minus* position. In the past, the *plus* position was considered as the one leading the train in the straightest direction through the station. This convention still appears in the literature, but it is no longer used by Banedanmark when documenting new railway networks. The second convention, which is now used by Banedanmark, states that the *plus* position corresponds to right (as seen from the stem side), and that the *minus* direction corresponds to left. From now on, we will use the second convention unless we explicitly state otherwise.

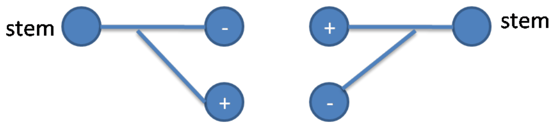


Figure 2.2: Two points following the new convention (plus=right, minus=left)

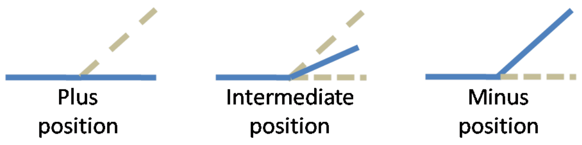


Figure 2.3: A point in the three possible positions

2.2.2 Signals

Signals are used to control the railway traffic. As explained in DSB Baneanlæg [6], there are several kinds of signals. Some signals are used for shunting while other signals are used for controlling the ordinary traffic on the rails.

Taking all the aspects of a signal (i.e. what a signal is displaying) into consideration is not relevant for this project. The scope of this project will only require the knowledge of a few aspects. The relevant ones (as described by Niels E. Jensen and Benny Mølgaard Nielsen [11]) are:

- **Stop**.
  The train must stop in front of the signal. The *stop* aspect will be indicated by a red light, sometimes combined with a yellow light.
- **Drive**.
  The train is allowed to pass the given signal, but the train driver must expect to stop at the next signal. The *drive* aspect is indicated by a green light, sometimes combined with a yellow light.
- **Drive through.**
  The train is allowed to pass the given signal and it should expect *drive* or *drive through* at the next signal. The *drive through* aspect is either indicated by two green lights or a blinking green light.

In general, we will not distinguish between the aspects *drive* and *drive through*. When referring to the *drive* aspect it can mean either of the two aspects.

There are several signal types indicating the role of the signal. For instance, *entrance signals* control the entry of a station by displaying the appropriate aspect and *exit signals* control the exit of a station.

2.2.3 Trains

Trains drive on track sections and can be longer than one track section (they can be up to 800 meters long). Their drivers are supposed to respect the aspects of the signals as explained in 2.2.2.

2.2.4 Representing a station layout

For each station, the physical objects are documented on paper. This section will explain how to read such documentation by giving an example of parts of the documentation from Stenstrup Station.

Stenstrup station is represented in figure 2.4. Trains approaching from Odense arrive from the left side of the station and those approaching from Svendborg arrive from the right side of the station.

2.2.4.1 The track layout

The station is formed by 4 linear track sections (*A12*, *02*, *04*, *B12*) and 2 points (*01* and *02*). Points *01* and *02* enable trains to access the two central track sections. Each point is directly associated with a track section: point *01* with track section *01* and point *02* with track section *03*. If one wishes to know whether point *02* is occupied, one has to check the state of the relay monitoring track section *03*.

2.2.4.2 The signals

There are 6 signals at the station:

- 2 entrance signals: *A* from Odense, *B* from Svendborg. When one of them displays a *drive* aspect, trains are authorised to pass it and enter the station. They both have a distant signal, *a* and *b* respectively. Distant signals will not be considered in this thesis.
- 4 exit signals: *E*, *F*, *G* and *H*. When one of them displays a *drive* aspect, trains are authorised to pass it and leave the station.

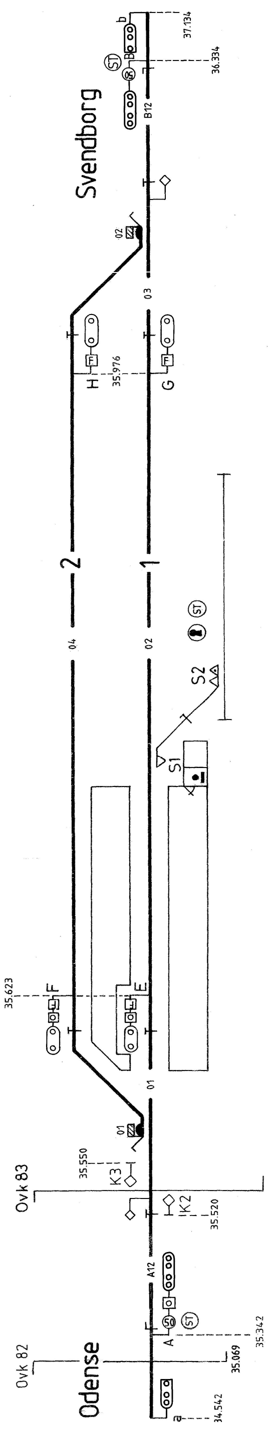


Figure 2.4: Station layout of Stenstrup Station

2.2.4.3 Other elements

S1/S2 is a point that was used to exit the station from the track *02*, but it is no longer regularly used.

There are two level crossings named *ovk82* and *ovk83*.

2.3 Introduction to interlocking

For now, only physical objects have been introduced. This section will introduce the concept of interlocking, i.e. how the safety rules are enforced.

Figure 2.5 shows a general approach to interlocking and the specific one used by Banedanmark. The goal of interlocking is to ensure some basic safety goals. The one specified by Banedanmark will be presented in section 2.3.1. These goals are specified on a high level and there may be several approaches to implementing these goals at a station.

Banedanmark uses *train route based interlocking* to ensure the safety goals. By defining train routes and a train route table for a given station, concrete safety rules are specified for it. An introduction to train route based interlocking will be given in section 2.4.

After having specified safety rules on a more concrete level, a physical implementation of the concrete safety rules is made. This is done by deploying an *interlocking system* that enforces rules on the physical objects at the given station. As it will be explained in section 2.3.2, the most frequently used type of interlocking system in Denmark is *relay-based.*

2.3.1 Basic safety goals

As explained in [4], Banedanmark define their basic safety goals in the following way:

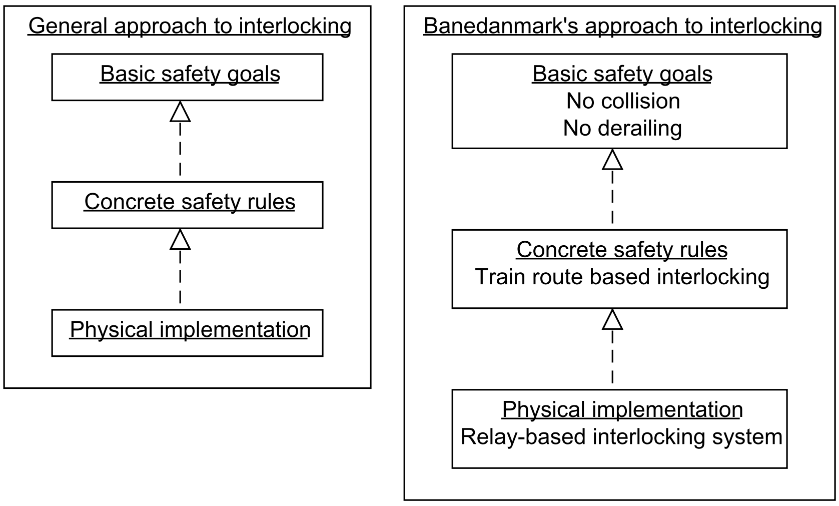


Figure 2.5: A general approach to interlocking and the specific one used by Banedanmark

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

This project will only consider the following basic safety goals:

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

2.3.2 Interlocking systems

Interlocking systems are used to ensure the safety of a station. Basically, the interlocking systems can (as described by DSB Baneanlæg [6]) be divided into the following types:

- **Mechanical systems.** They were equipped with arm signals and used wires for switching the points and setting the signals. They were put into use around 1870 and are not used any more.
- **Electro mechanical systems.** This type of interlocking system was introduced in the 19th century and is operated manually by a local operator [12]. These systems are no longer frequently used by the Danish railways.
- **Relay- and relay group systems.** This is the most frequently used type of interlocking systems in Denmark. It is totally electric and has been used since the 1950's. Further details about this kind of interlocking system will be given later.
- **Electronic systems.** This kind of interlocking system is computer-based. The first "part wise" electronic system was introduced in Denmark in 1977. Even though the electronic systems are the most modern interlocking systems, they are rarely used in Denmark. One of the main reasons is that it can be difficult to get spare parts for such systems in a long-term perspective and they are difficult to modify compared to relay- and relay group systems. For instance, interlocking systems can be used for more than 50 years, but some electronic spare parts are not in production for more than 10 years.

Banedanmark uses primarily relay- and relay group systems, therefore other kinds of interlocking systems are considered as beyond the scope of this project. Relay- and relay group systems will later be described in section 2.5.

As previously mentioned, the Danish railways are divided into line blocks and stations. Likewise, the interlocking systems are divided into interlocking systems for line blocks and interlocking systems for stations. This project will only consider interlocking systems for railway stations.

2.4 Train route based interlocking

As explained in section 2.3, there are several approaches to interlocking. The one used by Banedanmark is train route based. For train route based interlocking, there are two key concepts called *train routes* and *train route tables.*

A train can be allowed to follow a certain route called a *train route.* The rules for when it is safe to let a train follow a specific train route are described by a *train* *route table.* If the rules of the train route table are enforced by an interlocking system, the trains will travel safely. The following sections will describe the concepts of train routes and train route tables. After that, it will be described how the safety requirements are enforced.

2.4.1 Train routes

A *train route* (as described by Henrik W. Karlson and Carsten S. Lundsten [5]) is a route from one location in a railway station to another. Each train route has a *start location*, an *end location*, and some *via locations*, e.g. track circuits that connect the start location and the end location. A signal is linked to the train route, i.e. a train is only allowed to enter a train route if the aspect of a given signal allows it (see section 2.2.2): this signal can be considered as the *entrance signal* of the train route. A train route is said to be *locked* when its points are forced to remain in the positions required for a train to be able to drive through the route. A train route will become *unlocked* (also referred to as *released*) when some specific track circuits have been occupied in a certain order.

Usually train routes are locked and unlocked in the following way:

1. **Setting the points.**
   An operator has to put the points in the correct position.
2. **Locked.**
   If the points are in a legal position and the operator is pushing a specific button, the train route is locked. In this situation, the points cannot be switched until the release of the train route. Once the train route is locked, the operator can release the button.
3. **Setting the entrance signal of the train route to a *drive* aspect.**
   When some conditions (which will be described in the train route table of the station in section 2.4.2) are met, the signal will be switched to a *drive* aspect to allow a train to enter the train route.
4. **Setting the signal to a *stop* aspect.**
   When a train occupies the first track section of the train route, the entrance signal of the train route is switched to a *stop* aspect.
5. **Unlocking initiated.**
   When a train reaches some specific location of the train route, the unlocking of the train route will begin.

6. **Unlocked.**
   When a train reaches the end location of the train route, the train route becomes unlocked.

Train routes can be of two types:

- *Entrance train routes*, to enable trains to enter the station.
- *Exit train routes*, to enable trains to exit the station.

If a train has to go through the station without any stop, an entrance and an exit train route have to be locked at the same time.

Train routes are said to be *conflicting* if they are not allowed to be locked at the same time, e.g. if they have track circuits in common.

2.4.2 Train route tables

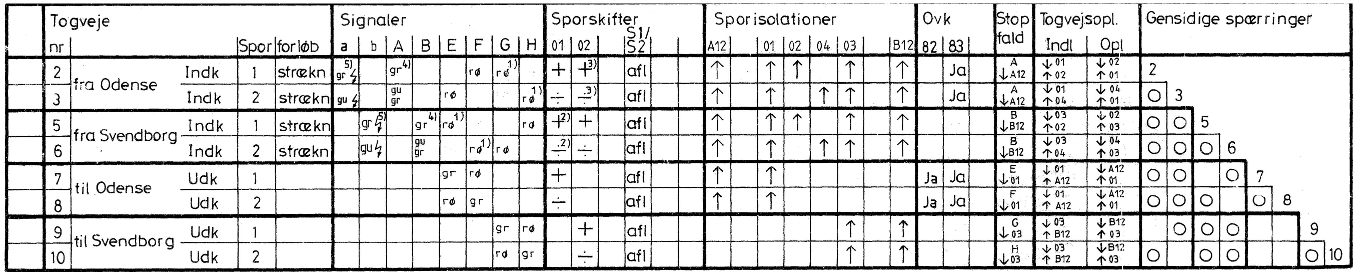
For a given station, all the train routes and the concrete safety rules associated with these are defined in the *train route table* of the station. In this section, the concept of train route tables will be introduced by presenting such a table from Stenstrup Station.

Recall the drawing of Stenstrup station in figure 2.4, page 12 along its explanation in section 2.2.4.

The possible train routes and the possible conflicts between train routes are described by a train route table. Such a train route table for Stenstrup station can be seen in figure 2.6. It contains four couples of train routes:

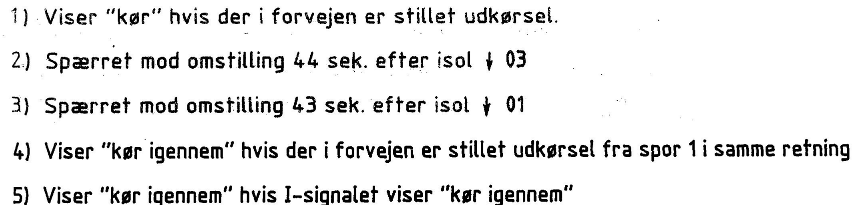
- two entrance routes (no. 2 and 3) from Odense: the first ends at track *02*, the second ends at track *04*,
- two entrance routes (no. 5 and 6) from Svendborg,
- two exit routes (no. 7 and 8) to Odense
- and two exit routes (no. 9 and 10) to Svendborg.

The train route table contains several safety parameters concerning the track sections, the signals, etc. These have to be met in order to guaranty that trains



Togveje					Signaler								Sporskifter			Sporisolationer						Ovk		Stop fald	Togvejsopl.		Gensidige spærringer							
nr			Spor	forløb	a	b	A	B	E	F	G	H	01	02	S1/S2	A12	01	02	04	03	B12	82	83		Indl	Opl								
2	fra Odense	Indk	1	strækn	5) gr ϟ		gr 4)			rø	rø 1)		+	+ 3)	afl	↑	↑	↑		↑	↑		Ja	A ↓A12	↓01 ↑02	↓02 ↑01	2							
3	fra Odense	Indk	2	strækn	gu ϟ		gu gr		rø			rø 1)	÷	÷ 3)	afl	↑	↑		↑	↑	↑		Ja	A ↓A12	↓01 ↑04	↓04 ↑01	○	3						
5	fra Svendborg	Indk	1	strækn		gr ϟ 5)		gr 4)	rø 1)			rø	+ 2)	+	afl	↑	↑	↑		↑	↑			B ↓B12	↓03 ↑02	↓02 ↑03	○	○	5					
6	fra Svendborg	Indk	2	strækn		gu ϟ		gu gr		rø 1)	rø		÷ 2)	÷	afl	↑	↑		↑	↑	↑			B ↓B12	↓03 ↑04	↓04 ↑03	○	○	○	6				
7	til Odense	Udk	1						gr	rø			+		afl	↑	↑					Ja	Ja	E ↓01	↓01 ↑A12	↓A12 ↑01	○	○		○	7			
8	til Odense	Udk	2						rø	gr			÷		afl	↑	↑					Ja	Ja	F ↓01	↓01 ↑A12	↓A12 ↑01	○	○	○		○	8		
9	til Svendborg	Udk	1								gr	rø		+	afl					↑	↑			G ↓03	↓03 ↑B12	↓B12 ↑03		○	○	○			9	
10	til Svendborg	Udk	2								rø	gr		÷	afl					↑	↑			H ↓03	↓03 ↑B12	↓B12 ↑03	○		○	○			○	10

Figure 2.6: The train route table for Stenstrup Station


1) Viser "kør" hvis der i forvejen er stillet udkørsel.

2) Spærret mod omstilling 44 sek. efter isol ↓ 03

3) Spærret mod omstilling 43 sek. efter isol ↓ 01

4) Viser "kør igennem" hvis der i forvejen er stillet udkørsel fra spor 1 i samme retning

5) Viser "kør igennem" hvis I-signalet viser "kør igennem"

Translation:

1. Displays *drive* if there is already a locked and ready exit
2. Cannot be switched until 44 seconds after track circuit ↓ 03
3. Cannot be switched until 43 seconds after track circuit ↓ 01
4. Displays *drive through* if there is already a locked and ready exit from track 1 in the same direction
5. Displays *drive through* if the entrance signal displays *drive through*

Figure 2.7: Notes for Stenstrup train route table in figure 2.6

can travel safely. We will examine each column of the train route table one by one:

- **nr**
  The id of the train route
- **Direction**
  It is indicated in which direction the train will pass through the station, in this case *from* (in Danish "fra") or *to* (in Danish "til") Odense, and *from* or *to* Svenborg
- **Indk/Udk**
  This column states whether the route is an entrance route (Indk) or an exit route (Udk).
- **Spor**
  This column states from which side of the station the train will enter or leave when using the train route: 1 if the train route contains track *02*, 2 if it contains *04*.
- **Forløb**
  This column states whether a safety distance is required (strækn=yes, ⊘=no). For example, route *02* has an end location before track sections

03 and *B12*, but *03* and *B12* are still included in the train route because they must be free in case the train does not stop at track section *02*. For the same reason, point *02* has to be set in the plus position even though the train must stop before it. If it cannot stop, it will not derail because point *02* will be in its plus position.

- **Signaler**
  This part is about the aspects of the signals used by the train route. *gr* means green, *rø* means red and *gu* means yellow. If no aspect of a given signal is indicated for a given train route, it means that the signal is not relevant for the given train route. The signal that is required to display a *drive* aspect is the *entrance signal of the train route*, i.e. if a train enters the train route, it has to pass this signal first. For instance, for route 2, signals *a* and *A* (*A* being the entrance signal of the train route) are supposed to display a *drive* aspect, *F* a *stop* aspect and *G* can display either a *drive* or *stop* aspect, depending if an exit train route is locked at the same time (see note nr. 1 in figure 2.7).
  This columns are used in step 3 of the locking process of a train route (see section 2.4.1).

- **Sporskifter**
  The position of the points is specified in this column. It can be + if the point has to be in the plus position, **-** if the point has to be in the minus position or empty if the point is not required to be in a certain position. Since the interlocking system of the station is relatively old, the station documentation uses the former convention for naming the position of the points, meaning that the plus position is leading the train in the straightest direction through the station (see section 2.2.1.1). *S1/S2* is an exit track that is not used regularly so the point has to be locked (*afl*) for all the train routes.
  This section is used in steps 1 and 2 of the locking process of a train route (see section 2.4.1). If the points are not in the position indicated by the train route table, the locking process is aborted.

- **Sporisolationer**
  $\uparrow$ means that the specific track circuit needs to be free before switching the entrance signal of the train route to a *drive* aspect. The arrows represent the state of the relay associated to a track section (see section 2.5 for further explanations about relays): $\uparrow$ corresponds to a drawn relay, that means that the track is free. For instance, for route 2, *A12*, *01*, *02*, *03*, and *B12* have to be free.
  This section is used in step 3 of the locking process of a train route (see section 2.4.1).

- **Ovk**
  There are two level crossings at the Odense end of the station. If the cell

corresponding to a level crossing contains "Ja", it means that this level crossing must be safe to be crossed by a train.
This section is used in step 3 of the locking process of a train route (see section 2.4.1).

- **Stop fald**
  It is the condition for when the entrance signal of the train route should switch from a *drive* to a *stop* aspect. For instance, for route 2, signal *A* must change when track section *A12* becomes occupied (↓A12). Usually the signal is changed after that the train driver has passed the first signal. In that way, it is avoided that the driver sees the red signal before passing the signal.
  This section is used in step 4 of the locking process of a train route (see section 2.4.1).

- **Togvejsopl**
  This specifies the conditions for releasing the train route. It consists of two states that have to occur in a certain order. For instance, for route 2, at some point track circuit *01* has to be occupied and *02* free, then, at a later time, *01* has to be free and *02* occupied. When these conditions are met, route 2 can be released.
  This section is used in steps 5 and 6 of the locking process of a train route (see section 2.4.1).

- **Gensidige spærringer**
  It is the set of the conflicting routes, which must not be locked at the same time. If there is a "O" symbol, the two routes are conflicting. For instance, route 2 has a conflict with every other route except route 9.

2.4.3 Enforcing the basic safety requirements

As described in section 2.3.1, this project will consider the two basic safety goals: *trains must not collide* and *trains must not derail.* By enforcing concrete rules extracted from a train route table, the basic safety goals are indirectly implemented in the following way:

- *Trains must not collide.*
  A train collision will be avoided by the fact that two conflicting train routes cannot be locked at the same time, that a signal can only become green if a train route related to this signal is locked, and that all the safety requirements of the train route table for the given train route are met. As the points are fixed while the train route is locked, a train cannot leave the route it is following.

- *Trains must not derail.*
  A train derailment will be avoided by checking that the points of a train route are in a legal position when locking this train route, by preventing these points to change when the train route is still locked, and by enforcing that a train route can only be released when a train has crossed every point of the train route.

2.5 Relay- and relay group systems

As explained in section 2.3, the rules of train route tables are physically implemented by interlocking systems. This section will introduce the kind of interlocking systems that is considered by this thesis, *relay- and relay group systems*, and its components.

The *relay- and relay group systems*, mentioned in section 2.3.2, are electrical circuits that contain relays, buttons, wires, power supplies, fuses, lamps, and resistors. Some of these components are only used for electrical purposes, but they will have no implication on the logics of the interlocking systems. The following section will give an introduction to the most relevant components from a logical point of view, *relays* and *buttons*. If the reader wishes to know more about the physical details, we recommend reading the bachelor thesis *Simulation of Relay Interlocking Systems* [7].

2.5.1 Relays

A relay is a component that can be in two states:

- Drawn / Up, graphically represented by $\uparrow$
- Dropped / Down, graphically represented by $\downarrow$

A relay has a set of contacts. A given contact of a relay can be in two states:

- Closed: current propagates through the contact.
- Open: current cannot pass the contact.

The contacts of a relay are grouped into upper contacts and lower contacts. The upper contacts of a relay are closed when the relay is drawn, otherwise they are open. The lower contacts of a relay are closed when the relay is dropped, otherwise they are open.

Drawing one relay might enable current in some parts of the circuit by having the upper contacts closed. At the same time, it might disconnect the current in some other parts of the circuit by having the lower contacts open.

There are two kinds of relays, *regular relays* and *steel core relays.* The following sections will describe these.

2.5.1.1 Regular Relays

Regular relays have two pins (they can be considered as sockets for wires) and each of them is connected to at least one wire. One regular relay can be in two states:

- Dropped when no current is propagating through the relay, making the relay demagnetised.
- Drawn when current is propagating through the relay, making the relay magnetised.

Regular relays can be used as sensors for detecting the state of a physical object. Some of the usages of a regular relay are:

- To detect if a track circuit is occupied or free:
  Each track circuit is linked to one regular relay: if the track circuit is free, the relay is drawn. If the track circuit is occupied, the relay is dropped.
- To detect the position of a point:
  Two regular relays are linked to each point: the first one will be drawn when the point is in the plus position and dropped otherwise. The second one will be drawn when the point is in the minus position and dropped otherwise.
- To detect the current aspect of a signal:
  A regular relay is linked to each lamp of the circuit. If the lamp is on, the relay is drawn. If not, it is dropped.

2.5.1.2 Steel Core Relays

A steel core relay is different from a regular relay because it is capable of maintaining its state (drawn or dropped) even though no current is propagating through it.

It has three pins, each of them connected to at least one wire. The two upper pins are for receiving current from a positive pole. These can be considered as possible electrical inputs to the component and are called “up” and “down”. The third pin is used for having an electrical output, e.g. to send the current to a negative pole.

- if there is current between the first input (up) and the output, the relay will be drawn.
- if there is current between the second input (down) and the output, the relay will be dropped.
- if there is no current between one of the two inputs and the output, the relay stays in its current state.

The relay must not receive current from both inputs at the same time and no current can propagate from input to input through the relay.

A steel core relay can be used for storing information. For instance, when a train route is locked, the steel core relay that is used to store that information will be dropped. It will stay that way until the conditions for the release of the train route are met. After that, the relay will be drawn and stay drawn until the train route is locked again.

2.5.2 Buttons

A button is a component that is able to disconnect current in a specific wire. A button can be in two states:

- Pushed: current is allowed to propagate through the wire.
- Released: current is not allowed to propagate through the wire.

As explained in section 2.7, buttons are controlled by an operator. In that way, buttons can be used for generating input to an interlocking system.

2.6 Diagrams

Relay- and relay group interlocking systems are, as described in section 2.5, electrical circuits consisting of different components. *Diagrams* are used by Banedanmark to describe interlocking systems. A diagram can be considered as a snapshot of the electrical circuit, showing the *normal state* (defined in the next paragraph) of an interlocking system or a part of it. An example of diagram can be seen in figure 2.8. Diagram signatures of components will be introduced in the next section.

The *normal state* of an interlocking system is defined by the following properties:

- Current is applied to the system.
- All points are in the plus position.
- All track sections are free.
- No train route is locked.
- All signals display stop aspects.
- All buttons are released.

Because a diagram is a static snapshot of an interlocking system, it does not explicitly describe the behaviour of the system. However, the diagrams contain enough information to deduce the behaviour of the systems.

In this section, we will first look into the signatures of components of the diagrams and then we will present an example of the behaviour of an interlocking system.

2.6.1 Signatures of relay diagrams

In a diagram one can encounter signatures of several kinds of components. These are connected by wires, drawn as black lines. With a few exceptions, all the signatures will be connected by at least two wires and can be associated with pin numbers.

The following sections will describe the most essential signatures of a diagram.

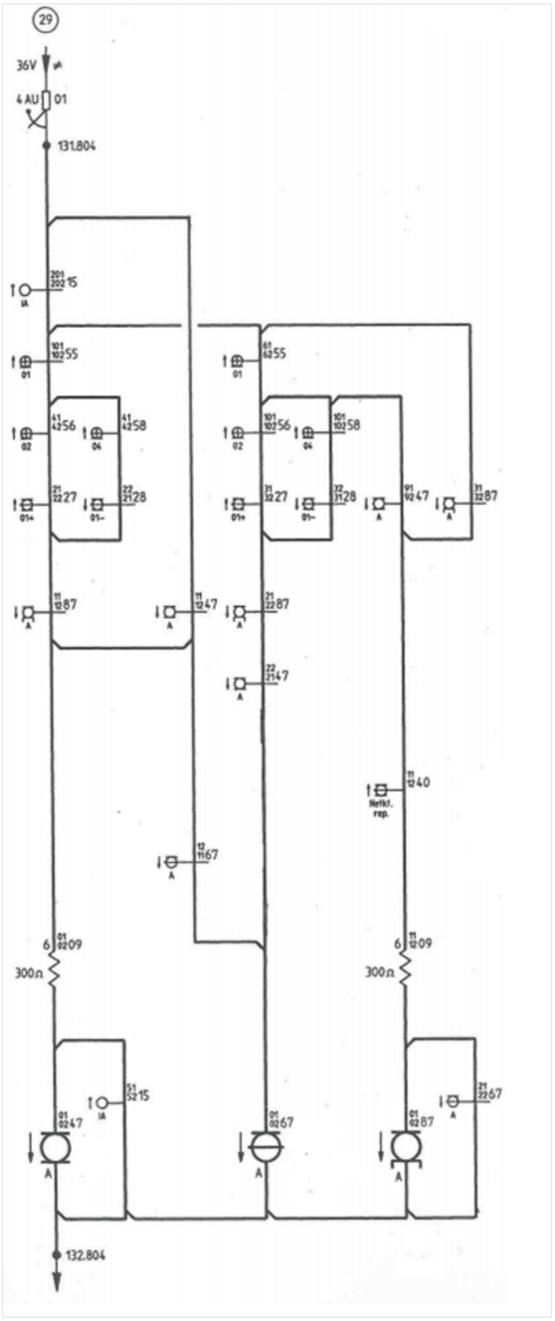


Figure 2.8: An example of a diagram

2.6.1.1 Power Supplies

The power supplies are the origin of the current. From a more electrical point of view, it can be seen as a positive pole. They are drawn in diagrams as in figure 2.9. Their voltage and the type of current (AC $\sim$ or DC $\neq$) are indicated near them.

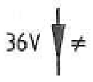


Figure 2.9: A power supply

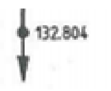


Figure 2.10: A negative pole

2.6.1.2 Negative pole

The negative pole is not a physical component, but is represented in diagrams by an arrow like in figure 2.10.

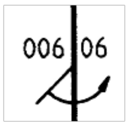


Figure 2.11: A button

2.6.1.3 Buttons

As described in sections 2.5.2 and 2.7, buttons are used for interacting with an interlocking system and are located on an operator's panel. An example of the signature of a button is shown in diagrams as seen in figure 2.11. One can find a button on the operator's panel thanks to the coordinates given by the diagram (here x=006, y=06).

The signature of a given button can only occur once in a diagram. As the buttons are released in the normal state, the signature in figure 2.11 indicates that current cannot propagate through the part of the circuit where the signature occurs.

2.6.1.4 Regular Relays

Figure 2.12 shows a the signature of a regular relay in a diagram. It contains the following information:

- Where the relay is physically located:
  In figure 2.12, the id of the relay, 87, makes it possible to find it in a physical relay room. In some cases, another notation is used to indicate more directly the location of a relay in the relay room: (level number, field number). In that way, the locations of the relays can be seen as a coordinate system.
- The two pin numbers that are connected to the circuit:
  This gives another indication about the physical location of the relay: at one address (level number, field number), there can be two relays, a left position and a right position. In this case, the pin numbers (here 01 and 02) indicate that the relay is at the left position. If the relay had been at the right position, the pin numbers would have been 03 and 04.
- The state of the relay in the normal state:
  The arrow to the left of the relay shown in figure 2.12 shows the state of the relay in the normal state of the interlocking system. In this case, the relay is dropped in the normal state, because the arrow is pointing downwards. If the arrow pointed upwards, the relay would be drawn.
- The role of a relay can be indicated by its signature:
  The relay in figure 2.12 is used in the unlocking process and the relay in figure 2.13 monitors a signal. One can find more about the meaning of the different relay signatures in [16].

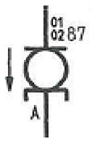


Figure 2.12: A regular relay (helping the unlocking process)

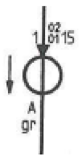


Figure 2.13: A regular relay (monitoring a signal)

2.6.1.5 Steel Core Relays

The signature of a steel core relay is shown in figure 2.14. Steel core relays are the only components that have three pins. The id of the steel core relay in the figure is 15. The output is on pin 02. The input that will draw the relay is on pin 01 and the input that will drop the relay is on pin 11. Pin 12 is not accessible.

Like regular relays, the state of a steel core relay is indicated by the arrow on the left of the relay. The only difference is that the end of the arrow is a black dot. In figure 2.14, the steel core relay is drawn, because the arrow is pointing upwards.

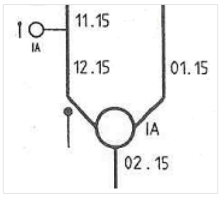


Figure 2.14: A steel core relay

2.6.1.6 Contact

A contact is like a switch that is ruled by a relay. For the contact in figure 2.15, relay 47 must be dropped in order to have the contact closed. The contact is linked to pins 91 and 92 of relay 47. As the normal state of 47 is dropped, the normal state of this contact is closed and current can propagate through it. As soon as relay 47 will be drawn, the contact will be open and the current will not be able to propagate through it.

Figure 2.16 shows a contact that is open in the normal state. The associated relay must be drawn in order to have the contact closed.

2.6.1.7 Other signatures

In diagrams, one can find other signatures that are not relevant for this project. These signatures are mainly used for security and physical reasons and they do

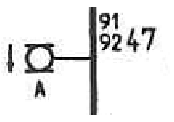


Figure 2.15: A closed contact

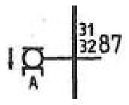


Figure 2.16: An open contact

not have any direct influence on the normal behaviour of an interlocking system.

- Fuses
  A fuse as seen in figure 2.17 can cut current in case of an emergency or over-current.
- Resistors
  The value and the location of the resistor in figure 2.18 are indicated on the side of it.
- Lamps
  Lamps are used in the physical signals at the stations. In diagrams, the lamps will look like the one shown in figure 2.19. The colour of the lamp is indicated by its signature (*gr* for green, *rø* for red, and *gu* for yellow). A regular relay is always linked to a lamp in order to monitor it and enable the operator to know whether the lamp is on or off.

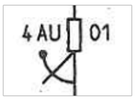


Figure 2.17: A fuse

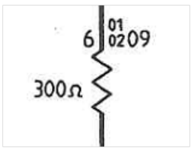


Figure 2.18: A resistor

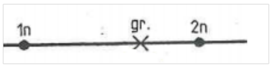


Figure 2.19: A lamp

2.6.2 An example of the behaviour of an interlocking system

We have now considered the structure of static relay diagrams. We will now present an example of how relay changes can happen in an interlocking system.

If one can follow a wire from a positive to a negative pole without meeting an open contact or a released button, it means that current can propagate through that wire. In other words, if there is a path from plus to minus that does not contain an open contact or a released button, current propagates through that conductive path.

If one follows the current in the normal state of the interlocking system (step 0) in figure 2.20, one can see that there are no conductive paths: each of the four possible paths is interrupted by an open contact or a released button. So they are not conductive and no relay is drawn.

When the button of the circuit is pushed (step 1), there are still four possible paths starting from the plus pole, but now one of them (shown by a continuous arrow) can reach the minus pole, so it is conductive.

As current propagates through relay 3-7 in step 1, it is drawn in step 2. Therefore, a contact controlled by relay 3-7 is closed, opening a new path from plus to minus.

Finally, in step 3, relay 3-3 is drawn thanks to the path opened in step 2.

An extension of the presented scenario is on the attached CD (see appendix C.2).

2.7 The Operator's Panel

The operator's panel is an interface to an interlocking system. It makes it possible for an operator to interact with the interlocking system and see the current state of the station, e.g. the state of the points, the track sections, etc.

An operator's panel is a physical object that can be found at a station. Figure 2.21 shows the operator's panel from Birkerød Station and figure 2.22 shows a drawing of the operator's panel for Birkerød Station.

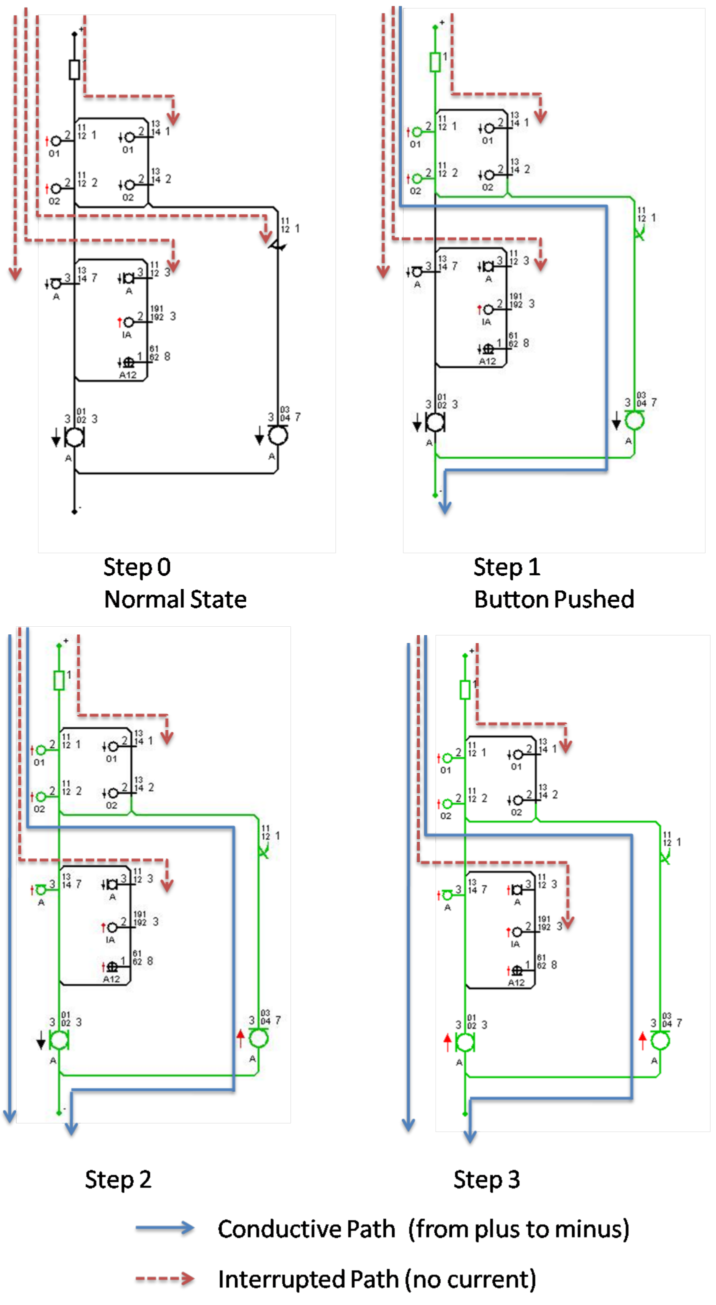

Step 0
Normal State
Step 1
Button Pushed
Step 2
Step 3
Conductive Path (from plus to minus)
Interrupted Path (no current)

Figure 2.20: The representation of a circuit and the possible paths for the current

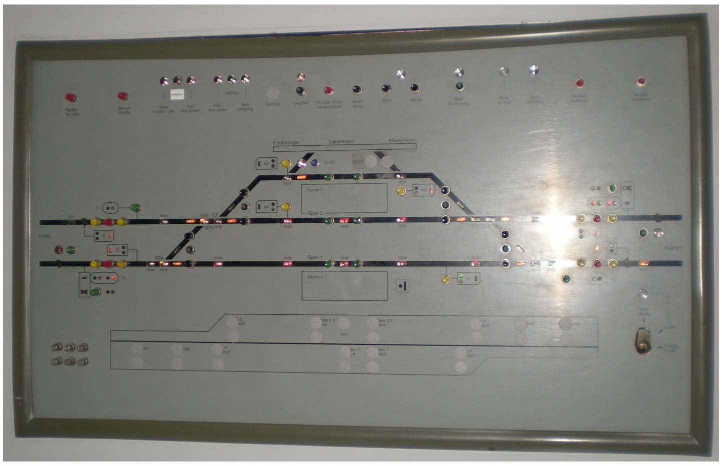


Figure 2.21: An operator's panel - Birkerød Station

The following is shown on an operator's panel:

- The geography of the station
  The organisation of the track sections, points, and signals is shown on a static drawing of the station.
- The buttons
  Thanks to them, the operator can interact with the interlocking system. Buttons are used to request the system to change the position of the points, to lock train routes, or in case of emergency.
- Information about the state of the station
  The panel is linked to the interlocking system such that it can show the current state of the station. The state of the track sections are indicated by lights on the panel: if the light is green, the track is free and a train route containing this track is locked. If the light is red, the track is occupied. If the light is off, the track is free and no train route containing this track is locked. There are also indications of the state of the signals, the position of the points, etc.

Nowadays, most of the actions that can be performed from an operator's panel are done automatically from a central that can be far from the station. In this project, we will consider any operator actions as if they were originated from the operator's panel.

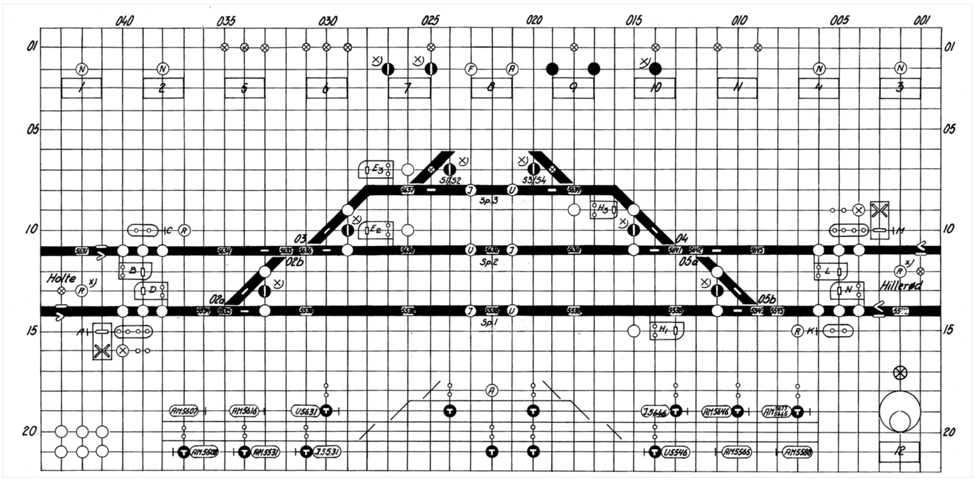


Figure 2.22: The diagram of the operator's panel - Birkerød Station

2.8 Interaction overview

This chapter introduced the physical objects of a station, an operator's panel, and interlocking systems. As previously explained, this project only considers relay- and relay group interlocking systems that enforce the safety goals at a station by controlling some physical objects of the station.

The relationship between the different elements of a station can be seen in figure 2.23.

The following physical objects are controlled by an interlocking system:

- **Signals**. The rules specified in the train route table for the signals must be enforced by the interlocking system. A *drive* aspect must only be displayed when it is considered as being safe.

An interlocking system knows the state of:

- **Track sections**. When a track circuit (a linear track circuit or a point) is occupied (by a train or due to some other physical reason), the associated track relay will be dropped and the interlocking system can therefore know that the track circuit is occupied.

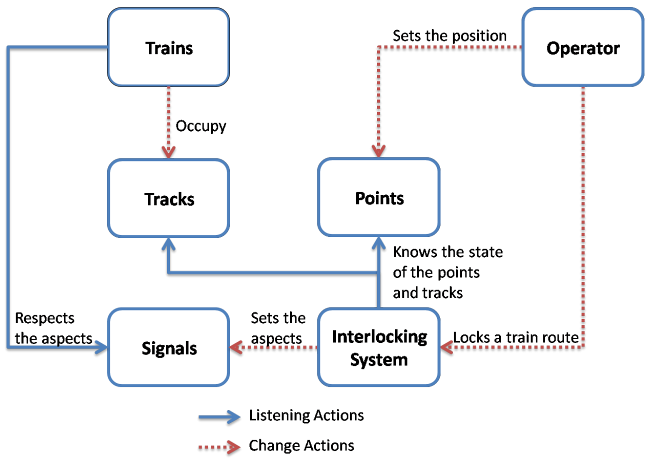


Figure 2.23: Relationships between the different elements of a station

- **Buttons**. When an operator wants to authorise a train to enter or exit the station, he or she has to lock a specific train route. The operator can push buttons for initiating such processes. However, an interlocking system is only allowed to lock a given train route when it is considered as being safe, e.g. conflicting train routes are not supposed to be locked at the same time.
- **Points**. For each point there are two point relays. Thanks to them, an interlocking system knows the position of a given point.

Note:
Points are only supposed to be switched when it is considered as being safe. In the real world, this is enforced by interlocking systems. The mechanism for controlling points is implemented in the same way of other functionalities of an interlocking system, using relays and buttons.
The circuits that are enforcing safety rules on points are relatively complex. Therefore, even though one could apply the principles presented in this thesis to model point control, modelling the control of points performed by an interlocking system is considered as being beyond the scope of this project. From now on, we assume that points can be switched by an operator when some conditions are fulfilled (see section 7.1.2).

Chapter 3

Introduction to RSL-SAL and LTL

This chapter will introduce *RSL-SAL* and *Linear-Time Temporal Logic*. A user guide for these languages is available on the internet [8].

As described by Juan Ignacio Perna and Chris George in [13] and [14], the *RSL* language has been extended such that the following declarations are possible within an *RSL* scheme:

```
class
  transition_system
    /∗ Specification of a transition system ∗/
      ...

  ltl_assertion
    /∗ Specification of properties that
       can be checked for a transition system ∗/
      ...

end
```

- *transition_system* is used to specify a state transition system within an

RSL scheme.

- *ltl_assertion* enables the possibility of specifying system properties for a state transition system within an *RSL* scheme using *Linear-Time Temporal Logic.*

In 2006, Juan Perna extended the RSLTC tool[1] such that it can convert from *RSL* to *Symbolic Analysis Laboratory* (*SAL*). This tool is capable of converting *RSL* specifications to a form that can be interpreted by a state space based model checker *SAL*[2]. After the conversion, the SAL tool can check whether the properties specified within *ltl_assertion* are valid for a transition system defined in *transition_system.* The model checker will either indicate that a property is valid or give a counter example.

The following sections give further details on the two kinds of declarations, *transition_system* and *ltl_assertion.*

3.1 Transition systems

The following is an example of a transition system specified using *RSL-SAL*:

```
scheme X =
  class
    transition_system
      /* The name of the transition system*/
      [TS]

      local
        /* Declaration of the initial state*/
        myVarInt : Int := 0,
        myVarBool : Bool := false
        in

        /* Declaration of transition rules */
        [rule1] myVarBool → myVarBool′ = false, myVarInt′ = 0
        ▯
        [rule2] myVarInt = 0 → myVarBool′ = true, myVarInt′ = 1
      end
```

[1]The newest version can be obtained on the internet [1].
[2]Further information on SAL can be found on the official SAL homepage [2].

end

The declarations after **local** specify the variables within the transition system and their value in its *initial state*. After **in**, transition rules are defined. The rules can be written on the following form:

[optionalName] guard → multipleAssignment

The rules specify how the current state of a transition system can be changed by taking transitions. If Boolean expression *guard* is true in a given state, the state can be changed by applying the multiple assignment of the transition. In that way, a new state is obtained. For instance, *rule1* defines that if *myVarBool* is true, then a new state can be obtained where *myVarBool* is false and *myVarInt* is 0. If a multiple assignment of a transition rule does not assign a new value to a given variable, the variable will not change when taking a transition defined by the rule.

All the possible states of a transition system are usually referred to as the *state space*. As seen in the state transition diagram in figure 3.1, the state space of *TS* contains two states. In the initial state, the guard of *rule2* is true and makes a transition possible to a new state. In the new state, only the guard of *rule1* is true. This rule enables a transition back to the initial state.

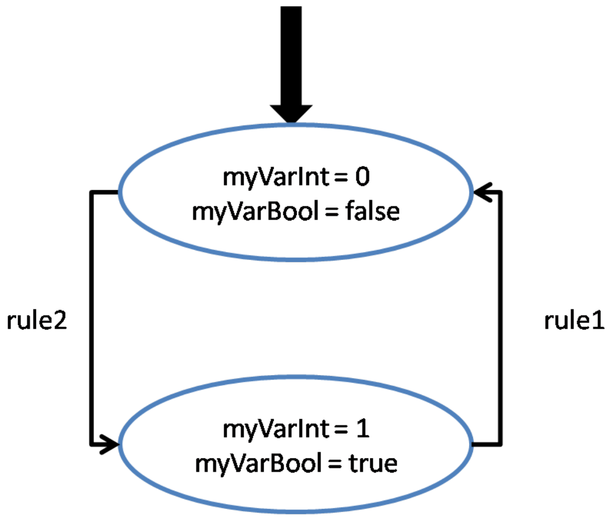


Figure 3.1: The state transition diagram of the transition system *TS*.

3.1.1 Allowed *RSL* constructs within an *RSL-SAL* transition system

Not all *RSL* constructs within a *transition_system* can be translated to *SAL*. The next sections will explain which constructs that can or cannot be used for translation from *RSL* to *SAL*.

3.1.1.1 Types

The following *RSL* types are allowed by the translator:

- **Bool**
- **Int**.
- **Nat**.
- **Variant types**.
- **Sets** of the form **T−set** are accepted if the type **T** is accepted.
- **Maps** are accepted, but they must be deterministic.

The following types are not translatable:

- **Sort types**
- **Union types**
- **Product types**
- **Lists** are neither accepted by *SAL* nor by the translator.

3.1.1.2 Functions

Some *RSL* functions can be used inside a transition system. In general, concrete *RSL* functions are accepted by the translator, but functions defined axiomatically and functions defined implicitly are not translatable. The translated functions must not be partial. If a function is undefined for a given value, one should add a precondition stating that the function must not be applied to this value.

The translated functions are neither allowed to be recursive nor iterative.

3.1.1.3 Operators

Some *RSL* operators are not allowed by the translation tool. For instance, the *hd* operator is not allowed.

3.1.1.4 Comprehended expressions

Comprehended expressions are not translatable to *SAL*.

3.1.1.5 Case expressions

Case expressions for the accepted types are allowed by the translator. If one wishes to write a case expression with a product (*case* $a \times b$ *of*), one should use a nested case expression, e.g.:

```
f : Int × Int → Int
f(a,b) ≡
   case a of
      _ →
         case b of
            _ → 0
         end
   end
```

3.1.1.6 Axioms

Axioms cannot be translated from *RSL* to *SAL*.

3.2 Linear-Time Temporal Logic assertions

The properties that are checked by the *SAL* tool are expressed using *Linear-Time Temporal Logic*, also known as *LTL*. Such properties are called *assertions* and they can be valid or invalid for a given transition system. If a property is valid, it means that it is true for all the possible traces of the studied transition system.

LTL assertions are basically Boolean expressions combined with some operators that allow references to future states. The following is an example of an *LTL* assertion that refers to the above transition system *TS*:

ltl_assertion
[assertionName] TS $\vdash$ myVarInt = 0

The assertion states that *myVarInt* equals 0 in the initial state of *TS*. If one wants to refer to future states (when using *SAL*, the future includes the current state), the following operators are possible:

- **G** means globally true. G(p) expresses that the *LTL* expression *p* must be satisfied by any future state.
- **F** means eventually true. F(p) expresses that the *LTL* expression *p* must be true in some future state.
- **X** means true in the next state. X(p) expresses that the *LTL* expression *p* must be true in the next state.
- **U** means strong until. U(p,q) expresses that *p* must remain true until *q* is true and that *q* is eventually true.
- **W** means weak until. W(p,q) expresses that *p* must remain true until *q* is true, but *q* is not required to be eventually true.
- **R** means release. R(p,q) expresses that *q* remains true until after a state where *p* is true. When *p* is true, it "releases" *q*. If *p* does not become true, *q* must remain true forever.

The meaning of these operators is illustrated by figure 3.2.

The LTL operators can be combined. For instance, *G(F(p))* expresses that *p* must be true over and over again. Another example is *X(X(p))* which expresses that *p* is true in the second state after the current state.

In general, the usual *RSL* operators for Boolean expressions like the ones for negation, conjunction, disjunction and implication can be used in the *LTL* expressions.

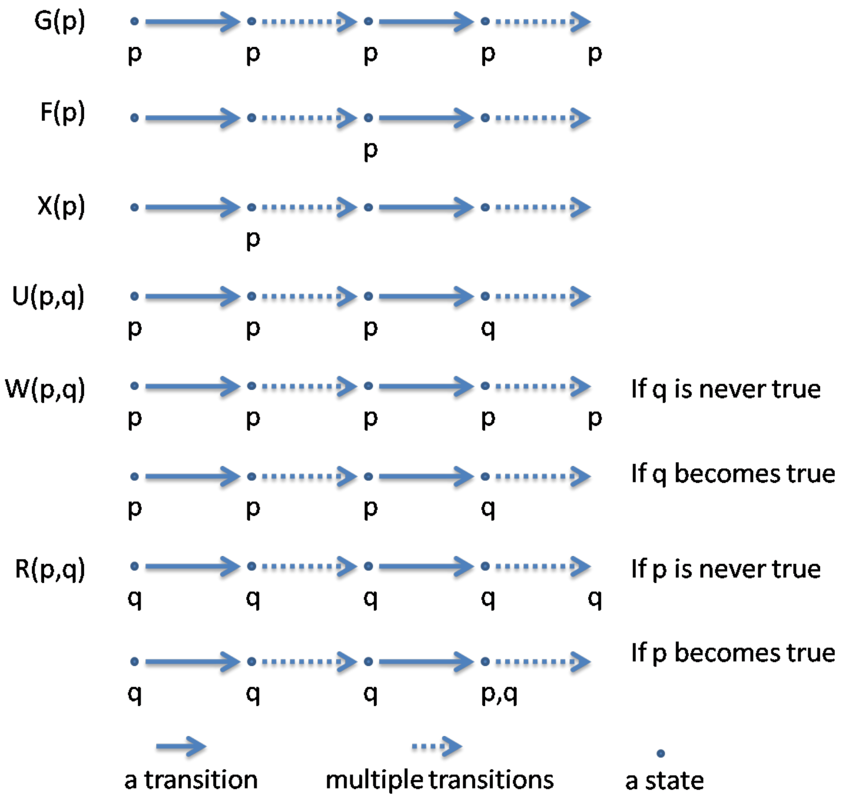


Figure 3.2: Description of the LTL operators

Further information on LTL can be found in *Logic in Computer Science* by Michael Huth and Mark Ryan [10].

Chapter 4

Method overview

Chapter 2 introduced the domain of this thesis and chapter 3 introduced the language *RSL-SAL*. This chapter will give an overview of the approach to model relay-based interlocking systems and verify properties related to them.

Section 4.1 will give an overview of the method that is developed in this project for formulating and verifying safety properties for an interlocking system using *RSL-SAL*.

Section 4.2 will describe the development steps that will be taken during this report in order to develop a tool that is used as part of the method described in section 4.1.

4.1 The verification method defined by this thesis

As mentioned in chapter 2, Banedanmark uses diagrams to describe the normal state of an interlocking system. When checking safety properties, it is not enough to consider a single state like the normal state. One must prove that safety is guaranteed in every possible state of an interlocking system. Proving properties by hand for a single interlocking system might be difficult due to the high number of possible states. Also, a manually-based proof process might be time consuming.

For these reasons, it is decided to automate the proving process by using the *SAL* state-based model checking tool. The advantage of model checking is that if one can describe the behaviour of an interlocking system in terms of a transition system, the proofs can be done automatically.

However, obtaining a transition system that describes the behaviour of an interlocking system and formulating safety properties are not trivial tasks. This project will provide a method to do so, illustrated in figure 4.1.

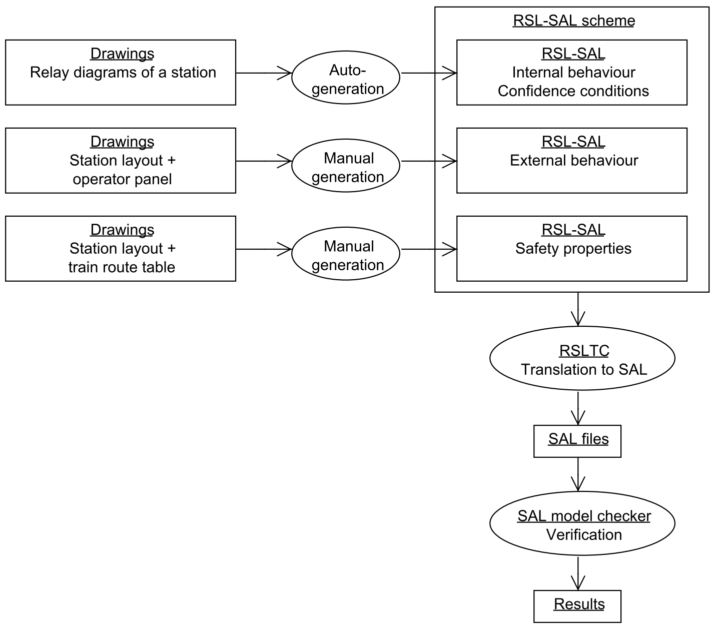


Figure 4.1: The verification method defined by this thesis for checking safety properties of an interlocking system

The main idea behind this method is that one should be able to start from the existing documentation for a station and its interlocking system and use it for formulating everything needed for the verification process. During the process, the following three steps are made for generating an *RSL-SAL scheme* that will be used when verifying properties:

- *The first step* when making an *RSL-SAL scheme* is to make a transition system that contains the internal behaviour of an interlocking system and some specific *confidence conditions*. The internal behaviour defines how an interlocking system responds to actions performed in the external world, e.g. when track sections are being occupied or freed and buttons are being pushed or released. *Confidence conditions* are conditions that do not relate to the safety of trains at a station, but that specify desired diagram properties that must be true. If a confidence condition is invalid, the model of the internal behaviour is not sound.

  As interlocking systems are documented by diagrams, it is decided to use them as a base to generate the internal behaviour and the confidence conditions. Doing this manually is neither a quick nor a trivial task. Therefore, it has been decided to provide a tool to perform such a generation. An overview of how the tool was developed will be given in the following section.

- *The second step* when making an *RSL-SAL scheme* is to extend the autogenerated transition system containing the internal behaviour of an interlocking system with external behaviour. The *external behaviour* is a collection of rules for when track sections can be occupied, when buttons can be pushed, etc. Any verification of properties is then sound under the assumptions made when specifying the external behaviour. The external behaviour of a given station can be generated by analysing the layout of this station and its operator panel. In section 7.1, patterns for generating different kinds of external behaviour are introduced. One could think of making a tool for generating external behaviour, but doing so is considered as being beyond the scope of this project.

- *The third step* is to formulate and add *safety properties* to the *RSL-SAL scheme* using *LTL*. Some safety properties may be derived directly from a train route table while others are more directly related to the overall safety goals. Again, one might make a tool for generating safety properties, but it is decided to limit this part of the project to providing patterns for formulating such properties. Such patterns can be found in section 7.2.

When having generated the internal behaviour, the confidence conditions, the external behaviour, and the safety properties of a given interlocking system, one has a complete *RSL-SAL scheme* that can be used for checking the properties of this interlocking system. The *RSL-SAL scheme* can then be translated to *SAL* using the *RSLTC* tool. After that, the *SAL* model checker can be used for checking the safety properties and for each property it will either report that the property is valid or it will give a counter-example for it.

An application of the method for checking *safety properties* of an interlocking system can be found in chapter 8.

4.2 Development of a tool for generating transition systems

This section will give details on how the tool provided for generating a transition system that describes the internal behaviour of an interlocking system will be developed.

There are different approaches to obtaining the behaviour of an interlocking system. Section 6.2 will explain two distinct approaches for making a transition system that describes the behaviour of an interlocking system. In that given section, it is decided to generate a specific transition system for a specific interlocking system.

Figure 4.2 gives an overview of a how different development steps are taken in order to obtain a tool for doing this. Initially, everything is specified on an abstract level using *RSL*. After that, a concrete version of the abstract specification is introduced. Finally, a Java implementation of the concrete specification is made.

In order to analyse an interlocking system described by diagrams, an abstract syntax for them is introduced. This is done by defining an abstract (i.e. algebraic and property-oriented) model of a collection of diagrams, called *static interlocking system* (see the block labelled A in figure 4.2). Details on a *static interlocking system* is given in chapter 5.

After that, chapter 6 will analyse and deduce how a transition system describing the internal behaviour of a static interlocking system and the associated confidence conditions should be computed. The end result of the chapter is abstract generator functions (see the block labelled B in figure 4.2) that take an instance of the model introduced in A and produces a transition system and its associated confidence conditions, E. E is *RSL* abstract syntaxes for an *RSL-SAL* transition system and *LTL* assertions.

When having done the necessary analysis on an abstract level, a development step is taken in chapter 9 towards a program that can do the transformation. The chapter introduces concrete (i.e. model-oriented, but still generic) *RSL* models of A and B: C is a concrete model that implements A and D is a concrete model that implements B.

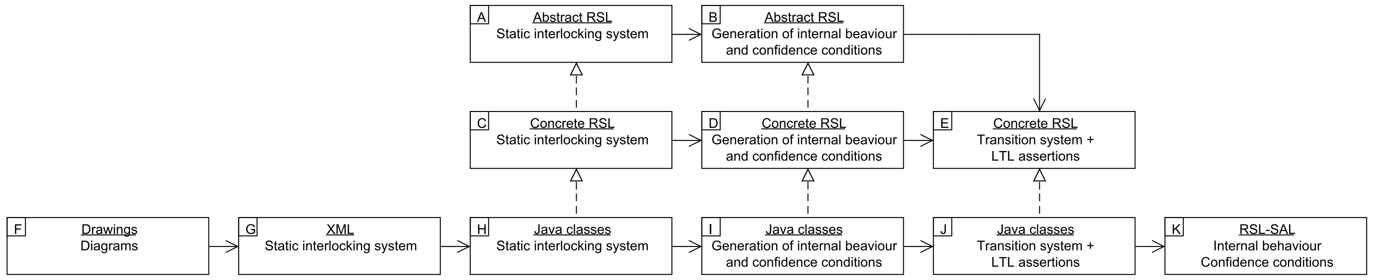

A Abstract RSL
Static interlocking system
B Abstract RSL
Generation of internal beaviour and confidence conditions
C Concrete RSL
Static interlocking system
D Concrete RSL
Generation of internal beaviour and confidence conditions
E Concrete RSL
Transition system + LTL assertions
F Drawings
Diagrams
G XML
Static interlocking system
H Java classes
Static interlocking system
I Java classes
Generation of internal beaviour and confidence conditions
J Java classes
Transition system + LTL assertions
K RSL-SAL
Internal behaviour
Confidence conditions

Figure 4.2: An overview of the development steps towards an *RSL-SAL* transition system containing the behaviour of interlocking systems.

In order to provide a tool that can be executed on most of the modern operating systems, it is decided to implement the concrete model using Java. The design of the Java implementation and the Java implementation itself are introduced in chapter 10.

Java implementations of C, D, and E are made: H, I, and J respectively. However, this is not enough for enabling the step shown in figure 4.1 where diagrams are converted directly to an *RSL-SAL scheme*: the tool must be able to take diagrams as an input and give an *RSL-SAL scheme* as output using concrete *RSL-SAL* syntax.

To be able to take diagrams as an input, an XML version of a *static interlocking system* is introduced, G. One can translate the diagrams of an interlocking system, F, to an XML version of a *static interlocking system*, G. A parser will then be responsible for converting the XML to H such that I can convert it to J. One might imagine that a graphical user interface can be made to draw diagrams and convert them to XML. However, this is considered as being beyond the scope of this project. Therefore, the conversion must be done manually.

In order to translate the data represented by J to an *RSL-SAL* scheme, K, an unparser is introduced as part of the Java implementation.

The parser and the unparser are both detailed in chapter 10.

Now, having explained the approach to this project, we are ready to begin the modelling process.

Chapter 5

Abstract model of relay diagrams

Chapters 2 and 3 introduced the domain, *RSL-SAL*, and *Linear-Time Temporal Logic*. After that, chapter 4 detailed the method developed by this thesis for formulating and verifying safety properties for interlocking systems. It was explained that the auto-generation of a transition system that contains the internal behaviour of an interlocking system will both be modelled on an abstract and on a concrete level before a Java implementation of the generation is made.

As explained in chapter 2, an interlocking system can be described by diagrams that are snapshots of it in its normal state. The purpose of this chapter is to introduce an abstract *RSL* model of such diagrams using sorts and abstract observer functions. As diagrams do not model the behaviour of interlocking systems, this chapter will only model interlocking systems in a static state.

The model of diagrams will then be used in chapter 6 for analysing and deducing how one can define functions that generate an *RSL-SAL* transition system containing the behaviour of an interlocking system.

Section 5.1 of this chapter will explain some assumptions about the modelled diagrams. After that, an *RSL* model for diagrams will be introduced in section 5.2. The complete *RSL* model will not be presented in this chapter. If the

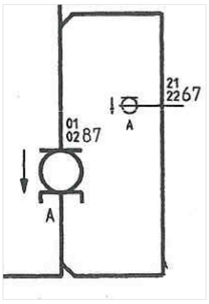


Figure 5.1: Shunt situation

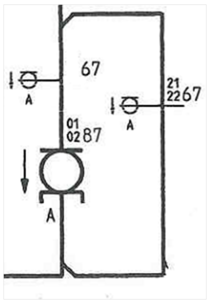


Figure 5.2: The same behaviour without the use of shunting

reader wishes to read the whole abstract model together with its associated transformation functions, it can be found in appendix A.

5.1 Assumptions about the modelled diagrams

This section will introduce assumptions about the modelled diagrams. Every modelled diagram is expected to fulfil the assumptions of this section.

In the circuits represented by diagrams, shunting can be used for dropping relays. In figure 5.1, when the contact ruled by relay 67 is closed, relay 87 is dropped due to a phenomenon called *shunting*.

When the contact is closed, the branch containing the relay has a much higher resistance than the branch that is only containing a contact. In that situation, the current will not go through the branch containing the relay because of its relatively high resistance compared to the second branch.

In this project, we do not want to model resistance. Therefore, we will assume that diagrams are transformed such that they do not include shunting situations. The diagram part in figure 5.1 can be replaced with a behavioural equivalent diagram part that can be seen in figure 5.2. By adding a specific contact, the relay will now be dropped thanks to the added contact instead of being dropped due to shunting.

From now on, we will assume that similar modifications are done to every diagram in order to avoid the use of shunting.

5.2 Modelling diagrams

This section will introduce an abstract RSL model of relay diagrams. Only the most important details of the model will be explained here. The whole model can be seen in appendix A together with the introduced schemes in chapter 6.

The abstract *RSL* model consists of the following schemes:

- **Types**. The purpose of this scheme is to represent the common types and the common auxiliary functions. The complete scheme can be found in appendix A.1, page 209.
- **Diagrams**. The purpose of this scheme is to describe what diagrams look like using an abstract type for a diagram and observer, auxiliary, and well-formed functions. The complete scheme can be found in appendix A.2, page 210.
- **StaticInterlockingSystem**. The purpose of this scheme is to describe what an interlocking system looks like using abstract types for diagrams and interlocking systems, and observer, auxiliary, and well-formed functions. The complete scheme can be found in appendix A.3, page 216.

In order to enable references between the schemes, the following global objects are introduced:

- **T**, an instance of *Types.*
- **D**, an instance of *Diagrams.*
- **SIS**, an instance of *StaticInterlockingSystems.*

The following sections will give a more detailed description of the mentioned *RSL* schemes.

5.2.1 Types

This section will explain parts of the *Types* scheme that includes common elements of the abstract *RSL* model. The purpose of the abstract model is not to give concrete types for the components in a diagram. Instead, an identifier for a component in a *Diagram* or a *StaticInterlockingSystem* is introduced as a sort type:

type
  Id

For representing the state of a relay, a variant type is introduced. As seen in the following declaration, a relay can either be up or down (i.e. drawn or dropped):

type
  State == up | down

Further types and functions are included in the *Types* scheme, but these are only relevant when extracting the behaviour of an interlocking system from a *StaticInterlockingSystem*. Therefore, the explanation of these functions will not be given until chapter 6.

5.2.2 Diagrams

This section will explain the *Diagrams* scheme that models diagrams, but not a collection of diagrams. For representing a diagram, the following sort type is introduced:

type
  Diagram

5.2.2.1 Identifying components

The components of a diagram are represented by values in the *Id* type from the *Types* scheme. Observer functions are introduced in order to identify which type of component an identifier represents. Examples of such observer functions are:

value
  isPlus : T.Id × Diagram → **Bool**,
  isMinus : T.Id × Diagram → **Bool**

isPlus indicates whether a given *Id* is a positive pole in a given *Diagram* and *isMinus* indicates whether a given id is a negative pole in a given *Diagram*. Similar observer functions of the same type are introduced in order to test whether an *Id* is representing a regular relay, a steel relay, a contact, a button, or a junction in a given diagram. The names of these functions are *isRegularRelay*, *isSteelRelay*, *isContact*, *isButton*, and *isJunction* respectively.

Junctions are not components of the real diagram circuits. However, diagrams have a general way of introducing branches. Branches in a circuit represented by a diagram are either introduced by steel core relays or by letting three wires meet. When three wires meet in a diagram, we will from now refer to this as a *junction*. Two *junctions* can be seen in figure 5.3.

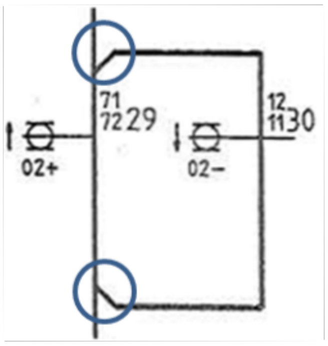


Figure 5.3: Two junctions marked by circles.

Junctions will be used in the model of the diagrams for introducing branches. This will allow for setting up rules about the number of neighbours. For instance, one can make rules for allowing regular relays to be connected to exactly two other components in the model of a diagram. If a given relay is connected to several components inside the real diagram, one can connect this relay to a junction and then connect the junction to the other components.

A junction is only supposed to introduce one branch in a network. If one wishes to introduce several branches at the same time, several junctions can be connected in series.

As one might have noticed, there is no function for detecting whether a component is a lamp, a fuse, a resistor, or a power source that were introduced in section 2.6.1. Concerning the lamps, each of them is monitored by a relay. Therefore, they are not needed as part of the model. If one wants to know the state of a given lamp, one can consider the relay that monitors the lamp. As we do not model electrical phenomenons like resistance, resistors, fuses, and the power sources are not included in the model.

5.2.2.2 Connecting components

As previously explained in section 2.6.1, the components in a diagram have pins that can be considered as sockets. Components are connected from pin to pin by wires. If one wants to make a model that includes all the information of the diagrams, it is necessary to model the pins.

If pins were included in the model, the components represented by *Ids* would be related to their pins and two components would be connected if two of their pins were connected. Such information would be needed if the components were asymmetric, e.g. if the direction of the current going through the component were important. However, most of the components of the diagrams are symmetrical. For instance, a regular relay can be drawn if there is current through it, but where the current is coming from does not matter. The only asymmetric component is the steel core relay. Further information on it is presented in the next section.

As most components are symmetric, there is no need to include pin information in each component. It would introduce extra information to the model that is not required to extract the behaviour of interlocking systems.

Therefore, it is chosen not to include the information about pins in the definition of components. Instead, it is chosen to introduce a neighbour relation between two identifiers representing two components in a diagram:

value
  areNeighbours : T.Id $\times$ T.Id $\times$ Diagram $\rightarrow$ **Bool**

The function tests whether the two components represented by the two *Ids* are neighbours in the *Diagram*. If two components are neighbours, it corresponds to having a wire between them inside the *Diagram*. For instance, the function is supposed to return true when being applied to the identifiers of the two contacts shown in figure 5.4 and the *Diagram* that contains them. The contacts in figure 5.3 are not considered as being neighbours, but they are neighbours to the junctions.

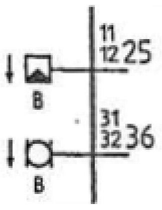


Figure 5.4: Two contacts that are neighbours.

5.2.2.3 Extra steel core relay information

Most components conduct current from one neighbour to another neighbour when all the conditions for having current through the component are fulfilled (e.g if the component is a button, it has to be pushed, if it is a contact, it has to be closed, if it is a regular relay or a junction, it is always conductive). However, as previously mentioned in section 2.5.1.2, a steel core relay cannot conduct current between all its neighbours. For instance, the steel core relay in figure 5.5 is only capable of conducting current from up to minus and down to minus, but not from up to down.

If current is conducted from up to minus, the relay will be drawn and if current is conducted from down to minus, the relay will be dropped. When no current is applied to the steel core relay, it will maintain its state.

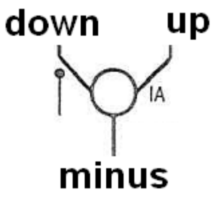


Figure 5.5: A steel core relay with named neighbours

Therefore, the *areNeighbours* function is not enough for specifying how steel relays are connected to other components in a *Diagram*. In order to add the needed extra information, it is decided to add the following observer functions to the model:

value
 upRelation : T.Id × Diagram $\xrightarrow{\sim}$ T.Id,

downRelation : T.Id × Diagram $\overset{\sim}{\to}$ T.Id,
minusRelation : T.Id × Diagram $\overset{\sim}{\to}$ T.Id

These functions are only supposed to be applied to *Ids* representing steel core relays in the given *Diagram*. When applied to the *Id* of a steel core relay, the functions are supposed to return an *Id* of a neighbour to the steel core relay in a *Diagram*. The functions will, respectively, give the neighbour connected to the up part, the neighbour connected to the down part, and the neighbour connected to the minus part of the steel core relay.

5.2.2.4 Representing a relay state

The state of a relay in the normal state of an interlocking system is represented in relay diagrams. Therefore, the following observer function is introduced for observing the state of a relay identified by an *Id* in a given *Diagram*:

value
relayState : T.Id × Diagram $\overset{\sim}{\to}$ T.State

5.2.2.5 Representing contact information

As previously mentioned, diagrams contain the following information about contacts:

- The id of the relay that rules the contact, i.e. opens or closes it when changing state.
- The state of this relay in the normal state of the interlocking system.
- The state of the contact, i.e. open or closed in the normal state of the interlocking system.

As the state of the relay that rules the contact is already represented by the *relayState* function, it would be redundant to add that information. The following observer functions are enough for adding the necessary information about contacts:

value
relayIdForContact : T.Id × Diagram $\stackrel{\sim}{\rightarrow}$ T.Id,
relayStateForContact : T.Id × Diagram $\stackrel{\sim}{\rightarrow}$ T.State

relayIdForContact will return the *Id* of the relay that rules the contact and *relayStateForContact* will return the state of the relay required for the contact to be closed. Whether the contact is open or closed can then be extracted from the model by considering the state of the relay that rules the contact.

5.2.2.6 State of buttons

As buttons are released in the normal state of an interlocking system, the state of a button is not given by any observer function. The normal state of a button is implicitly *released*.

5.2.2.7 Well-formed diagrams

For now, diagrams have been considered, but no constraints have been introduced on the observer functions. From now on, we will refer to diagrams that follow the conventions used by Banedanmark as *well-formed Diagrams*.

The following axiomatic function is introduced for testing whether a *Diagram* is well-formed:

value
isWfDiagram : Diagram → **Bool**

axiom
∀ d : Diagram • isWfDiagram(d) ⇒
okNeighbourRelation(d) ∧ okNumberOfNeighbours(d) ∧
twoPoles(d) ∧ noIdOverlaps(d) ∧
okSteelRelayRelations(d)

The axiom underspecifies the function in the sense that the defined constraints must hold for every well-formed *Diagram*. However, further constraints can be added when refining the specification.

okNeighbourRelation checks that:

- An *Id* cannot be neighbour to itself.
- The *areNeighbours* function is symmetric.

okNumberOfNeighbours checks that:

- A plus and a minus in a *Diagram* both have at least one neighbour.
- Contacts, regular relays, and buttons have exactly 2 neighbours.
- Steel core relays and junctions have exactly 3 neighbours.

twoPoles checks that:

- There are exactly one plus and one minus in a *Diagram*.

Note: Real diagram can contain several positive and negative poles. This is reflected in the model by allowing an unlimited number of neighbours for a plus and a minus. Instead of adding more than one positive pole or negative pole, one can add extra neighbours to an already existing pole.

noIdOverlaps checks that:

- At most one of the observer functions *isPlus*, *isMinus*, *isRegularRelay*, *isSteelRelay*, *isContact*, *isButton*, and *isJunction* can be true for a given *Id* in a given *Diagram*.

okSteelRelayRelations checks that:

- For every *Id* that is a steel core relay in a given *Diagram*, the set of the returned *Ids* from *upRelation*, *downRelation*, and *minusRelation* applied to the *Id* of the steel core relay is equal to the set of all the neighbours of the given *Id* in the given *Diagram*.

5.2.3 StaticInterlockingSystem

Scheme *StaticInterlockingSystem* is responsible for handling a collection of *Diagrams* that represents a snapshot of a given interlocking system in its normal state. For representing the whole interlocking system, the following sort type is introduced:

type
  StaticInterlockingSystem

For getting the *Diagrams* of a *StaticInterlockingSystem*, the following observer function is introduced:

value
  diagrams : StaticInterlockingSystem → D.Diagram-**set**

The diagrams contain relays that are part of the circuits of the interlocking system. However, some relays are not ruled by the circuits of the diagrams. These *external relays* are ruled by the external world and will be used as input to the interlocking system. For instance, a track relay will be dropped when a train occupies its associated track circuit and drawn when the associated track circuit is free. In general, if a relay rules at least one contact in a diagram of an interlocking system, but does not occur in one of the diagrams of the interlocking system, it is an *external relay*.

As contacts of the external relays appear in the diagrams, it is necessary to have information about the external relays of the interlocking system. For getting the *Ids* of the external relays in a *StaticInterlockingSystem*, the following observer function is introduced:

value
  externalRelayIds : StaticInterlockingSystem → T.Id-**set**

Because external relays can have different states in the normal state of the system, the initial state of each external relay must be specified. It is therefore decided to introduce the following observer function for determining the state of an external relay:

value
  externalRelayState : T.Id × StaticInterlockingSystem $\xrightarrow{\sim}$ T.State

5.2.3.1 Well-formed StaticInterlockingSystem

As for *Diagrams*, it is necessary to assume that a *StaticInterlockingSystem* fulfils the conventions used by Banedanmark. For checking that, an axiomatic well-formed function for *StaticInterlockingSystems* is introduced:

```
value
  isWfStaticInterlockingSystem : StaticInterlockingSystem → Bool

axiom
  ∀ sis : StaticInterlockingSystem •
  isWfStaticInterlockingSystem(sis) ⇒
    (∀ d : D.Diagram •
        d ∈ diagrams(sis) ⇒ D.isWfDiagram(d)) ∧
    uniqueIds(sis) ∧ contactsHaveRelays(sis)
```

The axiom underspecifies the function in the sense that the defined constraints must hold for every well-formed *StaticInterlockingSystem*. However, further constraints can be added when refining the specification.

The quantified expression checks that:

- All the *Diagrams* of the *StaticInterlockingSystem* are well-formed.

uniqueIds checks that:

- An *Id* is never used more than once in the sense that two elements in the interlocking system (either in the diagrams or in the external relays) cannot have the same *Id*.

contactsHaveRelays checks that:

- Each contact is ruled by a relay that exists in some part of the *StaticInterlockingSystem*.

5.2.3.2 The normal state properties

As previously mentioned in section 2.6, an interlocking system has a normal state in which some specific properties are fulfilled. Checking that an interlocking system is in the normal state is closely related to modelling the behaviour of an interlocking system. Therefore, checking that the diagrams represent an interlocking system in its normal state will be treated in section 6.5.4.1.

CHAPTER 6

Towards a behavioural semantics of relay diagrams

Chapter 5 dealt with modelling the static structure of relay diagrams on an abstract level and introduced *RSL* schemes *Types*, *Diagrams*, and *StaticInterlockingSystem*. Also, the following sorts were introduced: *Id* for identifying components inside a relay diagram, *Diagram* for representing a single relay diagram, and *StaticInterlockingSystem* for representing a collection of *Diagrams* and external relays.

As explained in chapter 4, generator functions will be formulated for converting the static model of interlocking systems presented in chapter 5 to an *RSL-SAL* transition system and *LTL* assertions expressing confidence conditions for it. The generated transition system models the behaviour of the interlocking system. For the same transition system, the confidence conditions can be used for verifying that it specifies the complete behaviour of this interlocking system and respects some desired properties that are not related to safety.

This chapter will analyse how such functions can be formulated on an abstract level. This is done by introducing abstract syntax for transition systems and *LTL* assertions (block E, figure 4.2, page 48) and generator functions (block B, figure 4.2).

Most of the focus will be on the circuits of the relay diagrams. Rules will be introduced for drawing and dropping relays, but general rules for how the external world behaves (e.g. when buttons can be pushed and track relays can be occupied) will not be introduced in this chapter. Only the problems related to interactions with buttons and track sections will be discussed. Patterns for external behaviour are described in chapter 7.

The following main sections are included in this chapter:

- Section 6.1 will explain how paths can be used to decide whether a relay can be dropped or drawn.
- Section 6.2 will discuss two different approaches for making a transition system that models the behaviour of an interlocking system. It will be decided to generate specific transition systems based on specific *StaticInterlockingSystems* instead of making a general transition system that could be used for every *StaticInterlockingSystem.*
- Section 6.3 will deduce formal conditions for when a single relay can be dropped or drawn.
- Section 6.4 will give an informal discussion of problems related to the chosen model in section 6.2. Confidence conditions will be introduced for proving that the problems do not exist for a specific transition system.
- Section 6.5 will give an informal discussion of problems related to modelling interactions between interlocking systems and the external world. Further confidence conditions will be introduced.
- Section 6.6 will, based on the discussions in sections 6.2, 6.3, 6.4, and 6.5, give a formal description of how a transition system that describes the behaviour of a *StaticInterlockingSystem* can be generated.
- Section 6.7 will, based on the discussions in sections 6.4 and 6.5, give a formal description of how to generate the confidence conditions that must be verified in order to know if the generated transition system is sound.

6.1 The notion of paths in diagrams

Diagrams are, as previously mentioned in section 2.6, snapshots of interlocking systems in the normal state. When modelling the behaviour of an interlocking system, the model is no longer static and will be able to enter different states.

This section will informally describe how to decide when a relay of a diagram is allowed to change its state based on the current state of the interlocking system.

In order to decide whether a given relay can be dropped or drawn, it is necessary to introduce a way to decide whether there is current through a given part of the circuit. As explained in section 5.1, it is assumed that the modelled diagrams do not use resistance for shunting down relays. With that assumption in mind, it is possible to use the concept of a *path* to decide whether there is current through a relay.

In this context, a *path* can be considered as a list of *Ids* representing components contained by a given *Diagram* such that:

- its first element is the positive pole of the *Diagram*.
- its last element is the negative pole of the *Diagram*.
- path(1) is neighbour with path(2), path(2) is neighbour with path(3),..., path(n-1) is neighbour with path(n), where n is the length of the path.

In principle, there can be an unlimited number of paths in a *Diagram*. Therefore, it is decided only to consider simple paths, i.e. paths without repetition of elements in a *Diagram*. From now on, when referring to a path, we mean a simple path without repetition of elements.

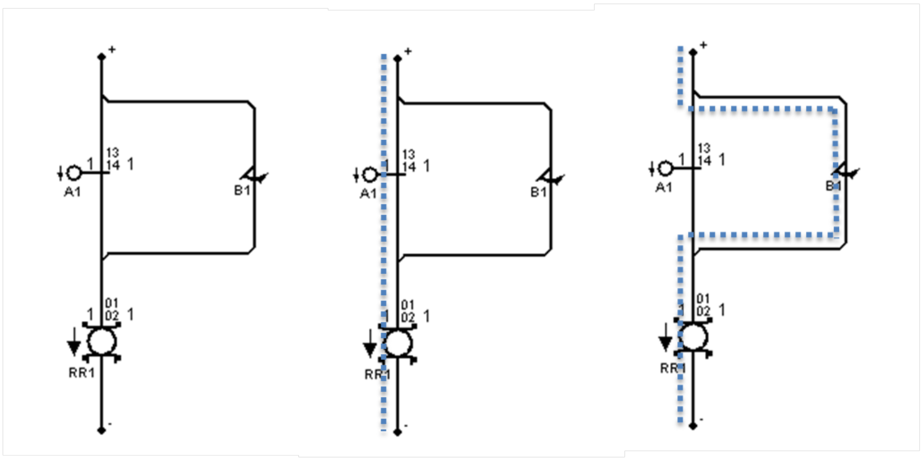


Figure 6.1: A diagram with two simple paths from plus to minus. The paths are indicated by the dotted lines.

6.1.1 The notion of conductive paths

Having a path containing a given regular relay is not enough to know whether the relay should be drawn or dropped. To know that, one needs to know if there is current propagating through the path.

Current will propagate through all the components of a path if and only if all the contacts inside the path are closed and all the buttons inside the path are pushed. From now on, we will refer to a path as being *conductive* if and only if current propagates through the path.

If there is a regular relay inside a conductive path, the relay can be drawn in the next state. On the contrary, if no conductive path goes through a regular relay, it can be dropped in the next state.

Whether a path is conductive or not will depend on the state of the interlocking system. Therefore, for each state of the interlocking system, one should consider every path through a given regular relay in order to decide whether it can be dropped or drawn in the next state of the system. For instance, in the diagram shown in figure 6.1, no path is conductive because the contact ruled by relay *A1* is closed and button *B1* is released. However, if *A1* is drawn or *B1* is pushed at some point, there is a conductive path through relay *RR1* and it can be drawn.

6.1.2 Paths containing steel core relays

Recall section 5.2.2.3 that explains how extra observer functions have been added to the abstract model for providing information on steel core relays. As explained in the same section, even though a steel core relay has three neighbours, current cannot pass through it to all its neighbours. Current can pass through a steel core relay from the neighbour given by the *upRelation* observer function to the neighbour given by *minusRelation*. Also, current can pass through a steel core relay from the neighbour given by *downRelation* to the neighbour given by *minusRelation*. However, current cannot pass through a steel core relay between the neighbours given by *upRelation* and *downRelation*.

This implies that some of the paths through a steel core relay cannot be conductive by definition. From now on, such paths will be considered as being illegal. Figure 6.2 shows a part of an impossible path through a steel core relay from *up* to *down*. Current cannot go this way and the paths containing this sequence of components should never be considered.

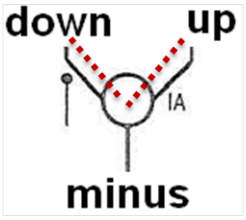


Figure 6.2: A part of an illegal path through a steel core relay that is indicated by the dotted line.

Figure 6.3 shows two parts of legal paths through a steel core relay. A steel core relay can then be drawn in the next state if there is a conductive path through it that contains both its neighbour given by *upRelation* and *minusRelation*. In the same way, a steel core relay can be dropped in the next state if there is a conductive path through it that contains its neighbours given by *downRelation* and *minusRelation*. If there is no conductive path through a steel core relay, the steel core relay should keep its current state in the next state of the interlocking system.

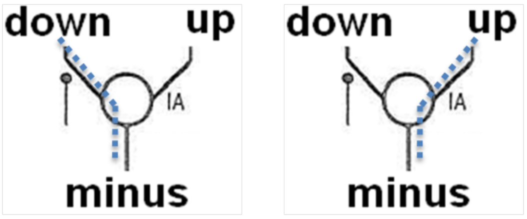


Figure 6.3: Two parts of legal paths through a steel core relay. The paths are indicated by the dotted lines.

6.2 Different approaches to a behavioural semantics of relay diagrams

We have now informally introduced the concept of a path in a *Diagram* of a *StaticInterlockingSystem*. If all the contacts in the path are closed and all the buttons in the path are pushed in the current state of the interlocking system, there is current through the components that are in the path.

This section will discuss two different approaches to modelling the behavioural semantics of relay diagrams. The first approach will try making a general transition system that can be used for modelling the behaviour of an arbitrary *StaticInterlockingSystem*. The second approach will discuss how to make a specific transition system that models the behaviour of a specific *StaticInterlockingSystem*. At the end of the section, one of the approaches is selected to be used for the rest of the project.

6.2.1 A general semantics of relay diagrams

This section will present an idea for how to make a general *RSL-SAL* transition system that can be used for modelling the behaviour an arbitrary, well-formed *StaticInterlockingSystem* (introduced in chapter 5) that is given to the *RSL-SAL* scheme of the transition system as a parameter.

Suppose a type *State* is given to represent the state of an interlocking system:

type
  State

State will contain every information needed for knowing whether the relays are drawn or dropped, the contact are closed or open, and the button are pushed or released.

When the current state of an interlocking system and the static structure given by the *StaticInterlockingSystem* type are given, it will be possible to decide whether there is a conductive path through some part of the circuit or not. Therefore, it will also be possible to decide whether a relay can be drawn or dropped by examining the current *State* of the interlocking system.

The observer functions *canDraw* and *canDrop* are therefore introduced for making such decisions:

value
  canDraw :
    T.Id $\times$ State $\times$ SIS.StaticInterlockingSystem $\overset{\sim}{\to}$ **Bool**,
  canDrop :
    T.Id $\times$ State $\times$ SIS.StaticInterlockingSystem $\overset{\sim}{\to}$ **Bool**

For instance, *canDraw* is assumed to return true if the relay identified by the

Id can be drawn in the current *State* of the *StaticInterlockingSystem*, otherwise it is assumed to return false.

Furthermore, generator functions for drawing and dropping a relay are introduced:

```
value
   draw : T.Id × State $\xrightarrow{\sim}$ State,

   drop : T.Id × State $\xrightarrow{\sim}$ State
```

For instance, *draw* is assumed to return a new state where the relay identified by *Id* is drawn. Every other variable of the transition system is assumed to keep its value in the new state.

Also, a generator function for computing the initial state based on a *StaticInterlockingSystem* is introduced:

```
value
   makeInitialState : SIS.StaticInterlockingSystem $\xrightarrow{\sim}$ State
```

After having presented these functions, a transition system that is able to model the behaviour of an arbitrary *StaticInterlockingSystem* can be introduced:

```
context: SIS, T
scheme SISSemantics(
     SISContainer :
         class
             value
                 sis : SIS.StaticInterlockingSystem
         end) =
    class
        type
            State

        value
            canDraw :
                T.Id × State × SIS.StaticInterlockingSystem $\xrightarrow{\sim}$ Bool,
            canDrop :
                T.Id × State × SIS.StaticInterlockingSystem $\overset{\sim}{\to}$ **Bool**,

            draw : T.Id × State $\overset{\sim}{\to}$ State,

            drop : T.Id × State $\overset{\sim}{\to}$ State,

            makeInitialState : SIS.StaticInterlockingSystem $\overset{\sim}{\to}$ State

        transition_system
            [ InterlockingSystem ]
            local
                state : State := makeInitialState(SISContainer.sis)
            in
                ([] id : T.Id •
                   [ drawRelay ]
                   id ∈ SIS.allRelayIds(SISContainer.sis) ∧
                   canDraw(id, state, SISContainer.sis) →
                     state′ = draw(id, state))
                []
                ([] id : T.Id •
                   [ dropRelay ]
                   id ∈ SIS.allRelayIds(SISContainer.sis) ∧
                   canDrop(id, state, SISContainer.sis) →
                     state′ = drop(id, state))
            end
    end
```

The transition rule *drawRelay* can draw the relay of the interlocking system identified by *id* if the current state permits to draw it. In the same way, *dropRelay* can drop the relay of the interlocking system identified by *id* if the current state permits to drop it.

6.2.2 A specific semantics of relay diagrams

This section will present the idea of making a specific transition system that models the behaviour of a specific *StaticInterlockingSystem*. The idea is to examine a *StaticInterlockingSystem* and then generate a specific transition system that models the behaviour of it.

If a transition of the transition system introduced in the previous section was taken, it would either draw or drop a single relay. Instead of making two general rules like in the previous section, one can introduce specific rules for drawing and dropping specific relays. For instance, one could, for each relay, make one rule for drawing it and one rule for dropping it.

The state of relays can be modelled as Boolean variables where true is equivalent to drawn and false is equivalent to dropped. Similarly, buttons can be modelled as Boolean variables where true is equivalent to pushed and false is equivalent to released.

Recall figure 6.1, page 64. A transition system that models the behaviour of relay *RR1* is:

```
transition_system
  [InterlockingSystem]
  local
    RR1 := false,
    A1 := false,
    B1 := false,
    /* Other variables */
    ...
  in
    [drawRR1]
      $\sim$RR1 $\wedge$ (A1 $\vee$ B1) $\rightarrow$ RR1$'$ = true
    $\square$
    [dropRR1]
      RR1 $\wedge$ $\sim$(A1 $\vee$ B1) $\rightarrow$ RR1$'$ = false
    $\square$
    /* Other transition rules */
    ...
  end
```

As the Boolean variables are false, relays *RR1* and *A1* are dropped in the initial state and button *B1* is released.

The transition rule *drawRR1* specifies that *RR1* can be drawn if:

1. *RR1* is dropped.
2. *A1* is drawn or *B1* is pushed. This corresponds to the condition for having current through *RR1*.

The transition rule *dropRR1* states that *RR1* can be dropped if:

1. *RR1* is drawn.
2. *A1* is dropped and *B1* is released. This corresponds to the condition for not having current through *RR1*.

In more general terms: suppose the *Diagrams* of a *StaticInterlockingSystem* contain *N* relays. A transition system that will describe its behaviour will be on the following form:

```
transition_system
  [ InterlockingSystem ]
  local
    /* X1 is either true or false */
    R1 := X1,
    ...
    /* XN is either true or false */
    RN := XN,
    /* Declarations of other variables */
    ...
  in
    [ drawR1 ]
      canDrawR1 → R1′ = true
    ▯
    [ dropR1 ]
      canDropR1 → R1′ = false
    ▯
    ...
    [ drawRN ]
      canDrawRN → RN′ = true
    ▯
    [ dropRN ]
      canDropRN → RN′ = false
    ▯
    /* Other transition rules */
    ...
  end
```

When making such an *interleaving model* for a specific *StaticInterlockingSystem*, the difficult parts is to calculate the guards of each transition rule. If one wishes to calculate the guards, one has to examine all the possible paths in each *Diagram* of the *StaticInterlockingSystem*. For instance, the condition for drawing a regular relay is a condition that evaluates to true if and only if there is a conductive path through it.

Also, it is necessary to prove that the interleavings of relay changes also include the states that can be obtained by drawing and dropping relays concurrently. Therefore, it is also necessary to introduce some *confidence conditions* (formulated as *LTL* assertions) that can be used for verifying that this is actually the case.

Specifying such a transition system for a specific *StaticInterlockingSystem* manually is not a trivial task. Therefore, this approach should be supported by functions that, given a *StaticInterlockingSystem*, automatically generate a transition system and its confidence conditions. Such a generation is illustrated in figure 6.4.

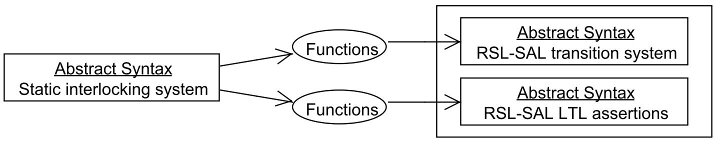


Figure 6.4: Conversion from a *StaticInterlockingSystem* to a transition system and its confidence conditions.

The idea is to formulate an abstract syntax for an *RSL-SAL* transition system and for the necessary *LTL* expressions. Generator functions to convert an abstract syntax for relay diagrams (i.e. a *StaticInterlockingSystem* and its contained *Diagrams*) to abstract syntax of a transition system and confidence conditions can then be specified.

The functions shown in figure 6.4 correspond to the box marked *B* in figure 4.2, page 48. The abstract syntaxes for an *RSL-SAL* transition system and its confidence conditions correspond to the box marked *E*.

6.2.3 Conclusion

In this section, two approaches to obtaining the behaviour of an interlocking system described by the type *StaticInterlockingSystem* have been presented.

There are two central problems related to the *SISSemantics* scheme introduced in section 6.2.1. First of all, as explained in chapter 3, *RSL-SAL* supports neither recursive nor iterative functions. For a regular relay, *canDraw* and *canDrop* will have to examine whether a conductive path exists through it. For a steel core relay, the functions will have to examine whether a conductive path exists trough a specific part of it. Therefore, it seems difficult to formulate these functions without using recursion or iteration inside *SISSemantics*, making it impossible to translate the scheme to *SAL*.

The second central problem is that maps in *RSL-SAL* seem to be quite inefficient in terms of verification time. Our experience with this data structure is that it takes a relatively long time to verify simple properties when using maps, even if the state space is small. As lists and recursive data types are not available in *RSL-SAL*, sets are the only alternative to maps. Formulating a completely set-based version of a *StaticInterlockingSystem* seems to be difficult in an *RSL-SAL* context, especially because it is impossible to iterate through sets (as explained in chapter 3, recursion, the *hd* set operator, and comprehended expressions are not allowed). Therefore, using maps seems to be unavoidable.

With these facts in mind, it is decided to use the approach introduced in section 6.2.2: generator functions will be specified for transforming a *StaticInterlockingSystem* to a transition system and its confidence conditions. Generating the transition system before translating it to *RSL-SAL* allows the use of recursive functions. Therefore, it is possible to use functions that would have been impossible to use in *SISSemantics*. The drawback of making a specific transition system is that one needs to generate a transition system each time one wants the behaviour of a new *StaticInterlockingSystem*. However, with the limitations of *RSL-SAL* in mind, it seems unlikely that one can succeed in using the first solution.

The rest of this chapter will discuss how to generate a specific transition system and its confidence conditions based on a *StaticInterlockingSystem*.

6.3 Conditions for drawing and dropping a single relay

In the previous section, it was decided to specify functions that can generate a transition system that models the behavioural semantics of a *StaticInterlockingSystem*.

The state of the transition system will contain:

- A Boolean variable for each button. True is equivalent to pushed and false is equivalent to released.
- A Boolean variable for each relay. True is equivalent to drawn and false is equivalent to dropped.

For each relay *R*, two rules will be introduced, a transition rule for drawing it and a transition rule for dropping it:

[drawR]
canDrawR → R′ = **true**
▯
[dropR]
canDropR → R′ = **false**

The purpose of this section is to consider how to obtain the guards of such rules on an abstract level. This will be done by adding further functionality to the abstract model introduced in chapter 5. The *RSL* schemes introduced in this chapter can be found in appendix A together with the schemes of chapter 5.

6.3.1 Types

Section 5.2.1 introduced the *Type* scheme (see appendix A.1, page 209) that contains some of the common types and auxiliary functions. In order to introduce an abstract syntax for the *RSL* type *Bool*, the following variant type has been added to the *Types* scheme:

```
type
  BooleanExp ==
    and(a : BooleanExp-set) |
    or(o : BooleanExp-set) |
    neg(n : BooleanExp) |
    literal(id : Id)
```

and contains a set of *BooleanExp*. It should be interpreted as having conjunctions between all the members of the set. Therefore, the expression should be interpreted as true if and only if all the members of the set *a* are evaluated as true or the set is empty. Otherwise, it should be interpreted as false.

The only difference between *and* and *or* is that *or* corresponds to having disjunctions between the members of the set *o* instead of conjunctions. Therefore, the *or* expression should be interpreted as true if and only if there exists a member of the *o* set that is interpreted as true or the set is empty. Otherwise it should be interpreted as false.

neg corresponds to having the negation of a Boolean expression. *neg* should be interpreted as true if and only if the inner expression, *n*, is interpreted as false. Otherwise *neg* should be interpreted as false.

literal contains an *Id* that is of the same data type as the one used for identifying components inside a *StaticInterlockingSystem*. If *id* of a *literal* equals an *Id* of a relay, it should be interpreted as true if and only if the relay is drawn. Otherwise it should be interpreted as false. A *literal* should be interpreted in the same fashion for buttons. If *id* of a *literal* equals the *Id* of a button, it should be interpreted as true if and only if the button is pushed. Otherwise it should be interpreted as false.

6.3.2 Pathfinding

Section 6.1 informally introduced the concept of a path. It is now time to make a more formal description of what a path is inside a *Diagram* of a *StaticInterlockingSystem*.

To make an *RSL* description of a path, a scheme called *Pathfinding* (see appendix A.4, page 218) and a global object *PF* referring to the functionality of *Pathfinding* have both been added to the abstract model. The major functionality of *Pathfinding* will be explained here.

The concept of paths was introduced in section 6.1. A path can be formulated as an *Id*-list:

type
  Path = T.Id*

To decide whether a path is legal or not, the following well-formed function is introduced:

```
value
  isWfPath : Path × D.Diagram $\xrightarrow{\sim}$ Bool
  isWfPath(p, d) ≡
    noDuplicates(p) ∧
    len p ≥ 2 ∧
    D.isPlus(p(1), d) ∧ D.isMinus(p(len p), d) ∧
    (∀ n : Nat •
        n ∈ (inds p \ {len p}) ∧
        D.areNeighbours(p(n), p(n + 1), d)) ∧
    noSteelRelayProblem(p, d)
  pre D.isWfDiagram(d)
```

The function *noDuplicates* checks that a path does not contain duplicates. Also, it is checked that a path contains at least two elements and that the first *Id* of a path represents a positive pole and the last *Id* of a path represents a negative pole. The quantified expression inside the well-formed function checks that the components of a path are neighbours in a way such that *p(1)* and *p(2)* are neighbours, *p(2)* and *p(3)* are neighbours, ..., and p(n-1) and p(n) are neighbours, where n is the length of the path. *noSteelRelayProblem* checks that, for all steel relays in the path, the *Ids* given by the functions *upRelation* and *downRelation* are not both inside the same path.

In order to know whether a path contains a set of *Ids* representing components, the following function is introduced:

```
value
  isPathFor : Path × T.Id-set → Bool
  isPathFor(p, ids) ≡ ids ⊆ elems p
```

By combining the above functions, it is possible to specify a function that gives all the well-formed paths in a *Diagram* that contain a given set of *Ids*:

```
value
  allPathsFor : T.Id-set × D.Diagram $\xrightarrow{\sim}$ Path-set
  allPathsFor(ids, d) ≡
    {p | p : Path • isWfPath(p, d) ∧ isPathFor(p, ids)}
  pre D.isWfDiagram(d) ∧ ids ⊆ D.allIds(d)
```

Note: The function does not tell how to compute the paths inside a *Diagram*. How to do this will be explained in section 9.5 that gives a more concrete version of the *Pathfinding* scheme.

6.3.3 Conditionfinding

In order to compute the conditions for drawing and dropping relays, a scheme called *Conditionfinding* (see appendix A.5, page 219) and a global object of it, *CF*, have been added to the abstract model. The following sections will introduce the content of this scheme.

6.3.3.1 Path conductivity

As previously explained in section 6.1, current can propagate through the components of a path if the buttons of the path are pushed and the contacts of the path are closed.

The following function is supposed to take the *Id* of a button or a contact and the *Diagram* that contains the given button or contact as argument. The function returns a *BooleanExp* that will be interpreted as true if the given contact is closed or the given button is pushed:

```
value
  isConducting : T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
  isConducting(id, d) ≡
    if D.isButton(id, d) then
      /* A pushed button is conductive */
      T.literal(id)
    else
      /* The conductivity of a contact */
      /* depends on the relay that is ruling it */
      case D.relayStateForContact(id, d) of
        T.up →
        /* The contact is conductive if the relay is drawn */
          T.literal(D.relayIdForContact(id, d)),
        T.down →
        /* The contact is conductive if the relay is dropped */
          T.neg(T.literal(D.relayIdForContact(id, d)))
      end
    end
  pre
    D.isWfDiagram(d) ∧
    (D.isButton(id, d) ∨ D.isContact(id, d))
```

After having introduced this function, it is possible to make a function that gives a *BooleanExp* that will be interpreted as true if and only if there is current through a given path. In other words, the function will give an expression that is true if all the contacts are closed and all the buttons are pushed:

```
value
  isConducting : PF.Path × D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
  isConducting(p, d) ≡
    T.and(
       {isConducting(id, d) |
         id : T.Id •
            id ∈ elems p ∧
            (D.isButton(id, d) ∨ D.isContact(id, d))})
  pre D.isWfDiagram(d) ∧ PF.isWfPath(p, d)
```

6.3.3.2 Conditions for having current through a regular relay

When having a function that gives the condition for having current through a path and a function that gives all the paths through a set of *Ids* inside a *Diagram*, the combination of these can be used for making conditions for having current through a regular relay.

The following function gives the condition for having current through a regular relay:

```
value
  currentThroughRegularRelay :
    T.Id × D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
  currentThroughRegularRelay(id, d) ≡
    T.or(
       {isConducting(p, d) |
         p : PF.Path • p ∈ PF.allPathsFor({id}, d)})
  pre D.isWfDiagram(d) ∧ D.isRegularRelay(id, d)
```

The *or* expression contains all the conductivity conditions for all the possible paths through the given regular relay. This part of the expression will therefore be true in a given state if there exists at least one conductive path through the relay.

In the same way, it is possible to make a function that gives the condition for not having current through a regular relay:

```
value
  noCurrentThroughRegularRelay :
    T.Id × D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
  noCurrentThroughRegularRelay(id, d) ≡
    T.neg(currentThroughRegularRelay(id, d))
  pre D.isWfDiagram(d) ∧ D.isRegularRelay(id, d)
```

The function returns the negation of the condition for having current through the relay.

6.3.3.3 Conditions for having drawing and dropping current through a steel core relay

As explained in section 6.1.2, steel core relays can be dropped in the next state when having current through a part of a steel core relay and drawn in the next state when having current through another part of the steel core relay.

The following function gives an expression that is true if and only if there is a current through the part of the relay that makes it draw:

```
value
  drawingCurrentThroughSteelRelay :
    T.Id × D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
  drawingCurrentThroughSteelRelay(id, d) ≡
    T.or(
        {isConducting(p, d) |
          p : PF.Path •
            p ∈
              PF.allPathsFor(
                  {D.upRelation(id, d), id,
                    D.minusRelation(id, d)}, d)})
  pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d)
```

In the same way, the following function will give an expression that is true if and only if there is a current through that part of the relay that makes it drop:

```
value
  droppingCurrentThroughSteelRelay :
    T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
  droppingCurrentThroughSteelRelay(id, d) ≡
    T.or(
       {isConducting(p, d) |
         p : PF.Path •
           p ∈
           PF.allPathsFor(
                {D.downRelation(id, d), id,
                  D.minusRelation(id, d)}, d)})
  pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d)
```

6.3.3.4 Conditions for drawing and dropping regular relays

With a function for knowing when there is a current through a regular relay, it can also be determined when it can be drawn. Having current through a regular relay is not enough as a condition for drawing it, because a relay can only be drawn if it is not already drawn.

A function that gives the condition for drawing a regular relay is then:

```
value
  canDrawRegularRelay :
    T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
  canDrawRegularRelay(id, d) ≡
    T.and(
       {T.neg(T.literal(id)),
         currentThroughRegularRelay(id, d)})
  pre D.isWfDiagram(d) ∧ D.isRegularRelay(id, d)
```

For instance, for relay *r1*, if the condition for drawing it is *draw_r1* (using concrete *RSL-SAL* syntax), the condition for drawing *r1* is:
$\sim r1 \wedge draw\_r1$

Similarly, a regular relay can be dropped if there is no current through it and it is drawn:

value

```
canDropRegularRelay :
   T.Id × D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
canDropRegularRelay(id, d) ≡
   T.and(
       {T.literal(id),
         noCurrentThroughRegularRelay(id, d)})
pre D.isWfDiagram(d) ∧ D.isRegularRelay(id, d)
```

6.3.3.5 Conditions for drawing and dropping steel core relays

A steel core relay can be drawn when a drawing current propagates through it and it is dropped. The following function gives the condition for drawing a steel core relay:

```
value
   canDrawSteelRelay :
      T.Id × D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
   canDrawSteelRelay(id, d) ≡
      T.and(
          {T.neg(T.literal(id)),
            drawingCurrentThroughSteelRelay(id, d)})
   pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d)
```

Similarly a steel core relay can be dropped if there is a dropping current through it and it is drawn:

```
value
   canDropSteelRelay :
      T.Id × D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
   canDropSteelRelay(id, d) ≡
      T.and(
          {T.literal(id),
            droppingCurrentThroughSteelRelay(id, d)})
   pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d)
```

6.4 Confidence conditions for the transition system

As previously mentioned, it is decided to make a transition system that includes two transition rules for each relay: a rule for dropping it and a rule for drawing it. The guards of these rules can be computed in terms of abstract syntax by the functions introduced in section 6.3.

However, we have not discussed the problems related to only drawing and dropping a single relay at a time. The danger is that such a model might not include the states that can be obtained by changing relay states concurrently.

Also, we have not discussed a specific problem related to steel core relays: there should never be a dropping and a drawing current through a steel core relay at the same time.

This section will introduce confidence conditions that can be used for proving that such problems do not exist.

6.4.1 Steel core relay issue

As previously described in section 2.5.1.2, a steel core relay must not have a drawing current and a dropping current through it in the same state. The previously defined conditions for drawing and dropping a steel core relay cannot be true at the same time because a relay can only be dropped when it is drawn and it can only be drawn when it is dropped. However, the two conditions for having current through the two parts of the steel core relay can be true at the same time. Such a situation is not desired by Banedanmark and it is therefore relevant to prove that it does not happen.

Suppose *c1* is the condition for having a drawing current through a given steel core relay and *c2* is the condition for having a dropping current through the same steel core relay (*c1* and *c2* can be computed by the functions *drawingCurrentThroughSteelRelay* and *droppingCurrentThroughSteelRelay* given in section 6.3 and afterwards be converted from abstract syntax to concrete *RSL-SAL* syntax). Then an *LTL* assertion expressing that *c1* and *c2* are never true in the same state is:

ltl_assertion
  [mutualExclusionC1C2] InterlockingSystem $\vdash$ G($\sim$(c1 $\wedge$ c2))

In order to be sure that steel core relays behave in a consistent manner, it is decided that a similar property should be verified for each steel core relay.

6.4.2 Concurrency issues

When only dropping or drawing one relay at a time, we ignore the fact that events can happen concurrently. It might only happen rarely, but one cannot dismiss the possibility of a situation where several relays change their state at the exact same time. In *RSL-SAL*, the only way of modelling this would be to introduce other transition rules. For instance, one could introduce a transition rule for drawing a relay *relay1*, a transition rule for drawing a relay *relay2*, and a transition for drawing them at the same time. However, following this principle would lead to a lot of extra transition rules. Each time several events can happen concurrently, there should be a transition for handling this. Introducing such extra rules would lead to the following issues:

- The number of transition rules would be greatly increased.
- The introduction of transition rules for concurrent relay changes would lead to extra static analysis before making the transition system. It would be necessary to decide which guards can be true at the same time. Doing this statically might not be a trivial task.
- The introduction of a lot of extra rules would lead to an increased running time when model checking. The *SAL* verification tool would have to evaluate a lot of extra guards when building the state space.
- The introduction of extra guards would probably make it difficult for a human reader to understand the transition system.

Instead of adding extra transitions, it would be better to prove that the order of the transitions does not matter. Verifying this will include proving that there is no race between transitions (i.e. it should be proved that taking one transition cannot falsify the guard of another transition). This can be proved by checking that if a guard is true at one point, it remains true until the transition protected by the guard is taken. In reality, it corresponds to that if a relay can change its state, it will change its state before the physical condition for changing it is no longer existing.

Verifying this is especially relevant because Banedanmark desires determinism: a relay that can change state should change state even though other relay changes occur.

By verifying this, Banedanmark can avoid situations where an interlocking system seems to be working properly, but suddenly one day a new sequence of relay changes takes place, making the interlocking system enter an unsafe state. This might happen because some relays react slower than they used to.

The *interleaving model* introduced in section 6.2.2 contains exactly two transition rules for changing the state of a relay. Also, each transition rule can only change the state of a single relay. Therefore, the transition rules can be grouped into pairs such that each pair is mutually assignment disjoint with every other pair. As each pair contains a rule for drawing and a rule for dropping the same relay, the guards of the rules cannot be true at the same time. This means that if one proves that there is no race between the transitions, one also proves that the model will include the states that can be obtained by taking two transitions concurrently.

One can ensure that the order does not matter by verifying that if a guard becomes true, then it will remain true until the corresponding transition is taken. If the condition for drawing a relay is fulfilled, it will remain true until the relay is drawn.

Assume that a guard, *g1*, contains all the conditions needed for drawing *relay1*. These conditions are given by the function *canDrawRegularRelay* or the function *canDrawSteelRelay* defined in section 6.3 (assuming that the abstract syntax is converted to concrete syntax). The following *LTL* expression will then correspond to expressing that *g1* remains true until *relay1* is drawn:

ltl_assertion
[g1UntilDrawn] InterlockingSystem ⊢
    G(g1 ⇒ X(∼g1 ⇒ relay1))

The LTL expression should be interpreted in the following way:

It holds for all states: if *g1* is true in one state, the following will hold for the next state: If g1 is false, then *relay1* is drawn. Therefore, when *g1* becomes true in one state, it must remain true until *relay1* is drawn.

An equivalent pattern can be used when dropping the relays. In the following LTL expression, it is assumed that *g2* is the guard for dropping *relay1*:

ltl_assertion
[g2UntilDropped] InterlockingSystem ⊢
    G(g2 ⇒ X(∼g2 ⇒ ∼relay1))

The two *LTL* expressions for *relay1* are sufficient for verifying that there will not be any concurrency issues related to drawing and dropping *relay1*. However, if one wishes to ensure that racing conditions do not exist and the transition system can obtain the states that can be obtained by changing relay states concurrently, one must verify equivalent *LTL* assertions for all relays. All the *LTL* expressions for verifying that no concurrency issues exist can then be considered as confidence conditions for a transition system containing two rules for each relay, one for drawing it and one for dropping it. If one of the expressions is invalid, the model will not cover the case where some specific relay changes happen at the same time. That is, the interleaving model will not be completely sound if one of the confidence conditions is invalid.

It can therefore be concluded that the above pattern for detecting concurrency issues must be introduced for every transition that is capable of drawing and dropping a relay.

6.4.3 Conclusion

We have now studied an approach where we focus on the logics of interlocking systems instead of modelling the propagation of current. Section 6.3 described how to obtain the following conditions based on the *Diagrams* of a *StaticInterlockingSystem*:

- A condition for when it is possible to draw a relay, $g_{up,r}$ (given by *canDrawRegularRelay* or *canDrawSteelRelay*, converted to concrete *RSL-SAL* syntax).
- A condition for when it is possible to drop a relay, $g_{down,r}$ (given by *canDropRegularRelay* or *canDropSteelRelay*, converted to concrete *RSL-SAL* syntax).

A transition system can be formulated by using these principles:

- Every relay and button are modelled as a Boolean variable. For relays, true means drawn and false means dropped. For buttons, true means pushed and false means released.
- The initial value of the Boolean variables will correspond to the normal state of the *StaticInterlockingSystem*.
- For each relay r of a diagram, two transition rules are added:

– one that can draw r when $g_{up,r}$ is true

$$\text{gUpr} \rightarrow \text{r}' = \textbf{true},$$

– one that can drop r when $g_{down,r}$ is true

$$\text{gDownr} \rightarrow \text{r}' = \textbf{false},$$

- In *RSL-SAL*, only one transition can be taken at a time. In order to be sure that this transition system also includes the states that can be obtained by drawing or dropping relays concurrently, it is necessary to verify that there are no racing conditions. In other words, if the condition for drawing a relay is true, it must remain true until the relay is drawn. The same should hold for conditions for dropping a relay. If such a condition is true, it must remain true until the relay is dropped.

  If the above transitions are defined, the associated assertions for checking that the order does not matter are:

  ltl_assertion
  [relayUpConcurrency] InterlockingSystem $\vdash$
  G(gUpr $\Rightarrow$ X($\sim$gUpr $\Rightarrow$ r))
  [relayDownConcurrency] InterlockingSystem $\vdash$
  G(gDownr $\Rightarrow$ X($\sim$gDownr $\Rightarrow$ $\sim$r))

- Also, it has been decided to verify that there is never a drawing current and a dropping current through a steel core relay in the same state. Suppose *c1* and *c2* are computed by the functions *drawingCurrentThroughSteelRelay* and *droppingCurrentThroughSteelRelay* given in section 6.3 (assuming that abstract syntax is converted to concrete *RSL-SAL* syntax). The following assertion will then express that *c1* and *c2* are never true at the same time:

  ltl_assertion
  [mutualExclusionC1C2] InterlockingSystem $\vdash$ G($\sim$(c1 $\wedge$ c2))

6.5 Interaction with the environment

We have previously focused on how to model diagrams, but the modelling of the interaction with the environment has been postponed. The following section will look deeper into this topic.

An interlocking system can receive different inputs from different sources in the external world. Examples of such sources are:

- **Buttons** that can be pushed or released.
- **Track relays** that are dropped or drawn when a given track circuit becomes occupied or free, respectively.

Like the other relays, track relays can either be drawn or dropped, and buttons can be pushed or released. Therefore, they can be modelled as Boolean variables. If a button is pushed or the track monitored by a track relay is free, the corresponding variable is true. If a button is released or the track circuit monitored by a track relay is occupied, the corresponding variable is false.

However, it is not possible to apply the previous deduced rules for dropping and drawing relays on buttons and track relays. For instance, a button can in reality be pushed or released at any time.

For pushing and releasing a button, one might introduce a transition rule similar to the one shown in the following transition system:

```
transition_system
   [ InterlockingSystem ]
   local
      button1 : Bool := false,
      ...
   in
      true → button1′ = ∼button1
      $\Box$
      ...
   end
```

In the above transition system, the button can always change its state. Similarly, one might introduce a rule for when to occupy track sections.

6.5.1 Timing issues

The above representation of track relays and buttons is not enough for making a transition system that models the behaviour of an interlocking system. In general, one of the major problems is that *RSL-SAL* does not have the notion of time.

For instance, if one takes a closer look at the above transition system that contains *button1*, one can identify the following possible trace of the system:

0. button1 = false
1. button1 = true
2. button1 = false
3. button1 = true
4. button1 = false
5. ...

In the given trace, the transition system does nothing else than changing the state of the button. Even if other transitions than the one changing the state of the button are possible, the above trace will still exist. This behaviour is considered as being unrealistic because of the high speed of the propagation of the current in the circuits of interlocking systems. It seems unlikely that relay changes occur slower than the release of a button, especially because the number of relays changes following an event in the external world (e.g. when a button becomes pushed or a track becomes occupied) is usually low.

If time had been available in *RSL-SAL*, this issue could have been solved by enforcing that a button has to be pushed for a minimum period of time, offering the relays of the interlocking system a chance to respond. One could think of adding a clock to the systems, but that would require exact measurement of how fast a relay changes its state. Also, introducing a clock would greatly increase the number of states, making it likely that one would encounter a state-space explosion. Finally, adding clocks to a language that does not have a notion of time might be difficult. Instead, one could think of making a model in an environment like UPPAAL [3] that provides clocks as a language construct.

Also, the above trace would introduce troubles when verifying the confidence conditions introduced in section 6.4. For instance, if a guard for drawing a relay depends on having a certain button pushed, the guard can change from being true to false and false to true without taking the transition.

Another problem related to this totally random behaviour is that it causes an explosion of the number of possible states. For instance, if a button can change in every state, the total number of states will be multiplied by 2. The size of the state space might for instance make verification infeasible as a consequence of having many buttons.

6.5.2 Solution to the timing issues

In general, the above issue can be summarised as: the inputs from the external world can happen faster than the interlocking system can respond to them, leading to an unrealistic starvation of the possible relay changes. As the notion of time does not exist in *RSL-SAL*, it is needed to ensure that the relays are able to respond to the inputs.

Taking a transition can be interpreted as an event, meaning that something happens in the external world or in the circuit of the interlocking system. In general, transitions can be split into two kinds of events:

- **Internal events**: they correspond to something happening inside the electrical circuit. For example, when a relay is drawn, it is an internal event.
- **External events**: they happen when something generates an input to the interlocking system.

Examples of such external events are:

- The state of a button is changed.
- The state of a track relay is changed (when something occupies or frees a track).

The idea is to assume that external events are slower than internal events. This can be enforced by rejecting external events until no internal event is possible. It can be formulated like:

- if there is a possible internal transition: no external transitions can be taken.
- if there is no possible internal transition: external transitions can be taken.

From now on, we will refer to an interlocking system as being *idle* if the system is waiting for an external event. Also, we will refer to the system as being *busy* if internal events are possible. Figure 6.5 shows how external events cannot happen when the system is busy.

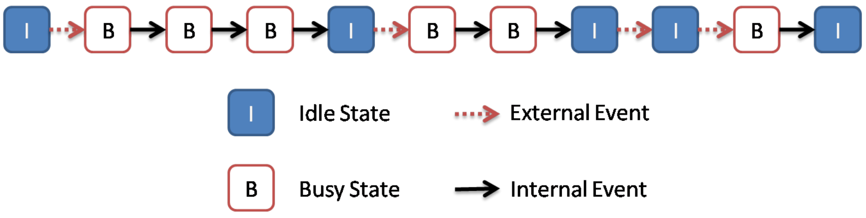


Figure 6.5: A possible chain of transitions

Even though we consider it as being realistic to assume that the external world is slower than the electrical circuits, one should have in mind that the verification will only be sound under this given assumption.

We have not assumed anything yet about how the track relays are updated and buttons are pushed, but we have decided that it will happen slower than the current propagates.

Further assumptions about external behaviour will be introduced in chapter 7.

6.5.3 How to introduce the concept of idle and busy in a transition system

For now, we have introduced the concept of internal and external events and it has been decided that a system can either be idle or busy. However, how to introduce this in an *RSL-SAL* context is still not considered. This section will discuss how to do it.

Consider the following transition system where *gi1* and *gi2* are guards for two internal events and *ge1* and *ge2* are guards for two external events:

```
transition_system
   [InterlockingSystem]
   local
      ...
```

```
  in
     /* Rules for internal events*/
     gi1 → ...
     ⌷
     gi2 → ...
     ⌷
     /* Rules for external events */
     ge1 → ...
     ⌷
     ge2 → ...
  end
```

The criteria for having an idle system is that no internal event can happen. In this particular case, it can be formulated like:

ExternalOK $\equiv\sim$(gi1 $\vee$ gi2)

The following sub-sections will explain two different solutions for how to enforce that external transition rules are only taken when no internal transitions are possible. We will refer to *ExternalOK* as the condition for not having any possible internal transition.

6.5.3.1 Extension of the guards with *ExternalOK*

A solution to distinguishing between internal and external events is to introduce *ExternalOK* as part of the guard of every external event. Applying this principle to the above transition system will give the following:

```
transition_system
  [InterlockingSystem]
  local
     ...
  in
     /* Rules for internal events*/
     gi1 → ...
     ⌷
     gi2 → ...
     ⌷
     /* Rules for external events */
```

```
    ∼(gi1 ∨ gi2) ∧ ge1 → ...
    []
    ∼(gi1 ∨ gi2) ∧ ge2 → ...
  end
```

For small transition systems, this might be an acceptable solution, but for systems with many external transitions, it would be inefficient. In each state, the SAL tool would have to examine the extension of the guard for all the external transition rules. This might be expensive when dealing with large interlocking systems.

6.5.3.2 Introduction of an *idle* variable

Instead of extending the guards of each external transition rule with *ExternalOK*, we decided to introduce an artificial Boolean variable called *idle* and an artificial transition rule to decide when the system is idle. The idea is to extend all the guards of external transition rules with the *idle* variable. The variable must be true in order to take the transition, i.e the system has to be idle to allow an external event. When an external transition is taken, the *idle* variable is set to false. The only way of setting it to true is to take the artificial transition whose guard consists of *ExternalOK*. Applying this principle to the above transition system will give the following:

```
transition_system
  [InterlockingSystem]
  local
    /* true if the system is idle */
    idle : Boolean := false
    ...
  in
    /* Rules for internal events */
    gi1 → ...
    []
    gi2 → ...
    []
    /* Rules for external events */
    idle ∧ ge1 → idle′ = false, ...
    []
    idle ∧ ge2 → idle′ = false, ...
    []
    /* Transition for making the system idle */
    ~idle ∧ ~(gi1 ∨ gi2) → idle′= true
  end
```

The benefit of this solution is that the condition for being idle will only be checked one time in each non-idle state. Due to the constraints on the *idle* variable, the state space will not be increased a lot by adding it. Because of that, we select this solution to distinguish between internal and external events and thereby solve the competition issue between these two kinds of events.

6.5.4 Properties related to the *idle* variable

After having introduced the *idle* variable, it is possible to formulate some desired system properties in terms of *LTL* assertions using *idle*. The following sections will introduce these properties.

6.5.4.1 Initial state of the *idle* variable

As described in section 2.6, the diagrams or an interlocking system present the interlocking system as it is in its normal state. In our context, the initial state of a transition system will correspond to the normal state of an interlocking system. In the normal state of an interlocking system, no relay changes are possible unless something happens in the external world. If not, it means that at least one diagram of the interlocking system contains an error. This corresponds to having the conditions for being idle fulfilled in the initial state. For reasons of consistency, this property should be verified. To do this, there are two possible solutions:

- The idea of the first solution is to set *idle* to true in the initial state and verify that the conditions for being idle are fulfilled in the initial state. For the above transition system, the *LTL* assertion for verifying this would look like:

  ```
  ltl_assertion
    [InitIdle] InterlockingSystem ⊢ ~(gi1 ∨ gi2)
  ```

- The second idea is having *idle* = false in the initial state and verify that *idle* is true in the next state, i.e. the only possible transition in the initial state is the one that makes the system idle. In *LTL*, it can be expressed like this:

ltl_assertion
  [InitIdle] InterlockingSystem $\vdash$ X(idle)

The advantage of the second solution is that the *LTL* assertion is the same for all transition systems and can be applied to all transition systems that use the *idle* variable. On the contrary, the first solution requires a specific *LTL* assertion for a specific transition system. Therefore, the second solution is chosen.

6.5.4.2 No internal cycle

Another desired property of interlocking systems is that a chain reaction of internal relay changes always ends in an idle state. For instance, if an input is received from the external world, the interlocking system should respond to it and at some point find another idle state where no change will happen until a new input is received from the external world. Thereby, it is verified that unrealistic starvation of events of the external world is avoided. In our context, this is equivalent to having an idle system over and over again. In *LTL*, it can be formulated like:

ltl_assertion
  [AlwaysEventuallyIdle] InterlockingSystem $\vdash$ G(F(idle))

The formula states that the system will always eventually be idle.

In general, both of the above properties should be verified for every transition system in order to detect possible errors in the diagrams of the interlocking system.

6.5.5 Conclusion

We have now discussed how to integrate the interaction with the external world. The main problem was that the notion of time does not exist in *RSL-SAL*. Therefore, it has been decided to distinguish between *internal events* and *external events*. An internal event is something that happens inside the electrical circuit of an interlocking systems and an external event is something that happens in the external world. The main assumption is that external events cannot happen if internal events are possible. This corresponds to assuming that changes happen faster in the circuit than in the external world.

To do that, it has been decided that a transition system can either be idle or busy. In a state where the system is idle, no internal events are possible and the transition system is ready to accept external events. If the system is busy, only internal events will be possible. To enforce this, the following concepts have been introduced:

- A Boolean variable called *idle*. If the variable is true, the system is idle. Otherwise the system is busy.
- *idle* is initialised to false.
- External events cannot happen when *idle* is false.
- When an external event happens, *idle* is set to false.
- A transition is made for setting *idle* to true. This transition can only be taken if no internal transition is possible.

Suppose we have a transition system containing the following internal transition rules:

$$\begin{array}{l} \text{guard1} \rightarrow \text{relay1}' = \textbf{true} \\ \Box \\ \text{guard2} \rightarrow \text{relay1}' = \textbf{false} \\ \Box \\ \text{guard3} \rightarrow \text{relay2}' = \textbf{true} \\ \Box \\ \text{guard4} \rightarrow \text{relay2}' = \textbf{false} \end{array}$$

The transition rule for setting *idle* to true will be:

$$\sim\!\text{idle} \wedge \sim (\text{guard1} \vee \text{guard2} \vee \text{guard3} \vee \text{guard4}) \rightarrow \text{idle}' = \textbf{true}$$

Suppose the following external transitions are possible:

$$\text{extguard1} \rightarrow \text{extRelay1}' = \textbf{true}$$

$\square$
extguard2 $\rightarrow$ extRelay1$'$ = **false**
$\square$
extguard3 $\rightarrow$ extRelay2$'$ = **true**
$\square$
extguard4 $\rightarrow$ extRelay2$'$ = **false**

Then *idle* must be added as part of the guard and it must be set to false when taking one of the transitions:

idle $\wedge$ extguard1 $\rightarrow$ idle$'$ = **false**, extRelay1$'$ = **true**
$\square$
idle $\wedge$ extguard2 $\rightarrow$ idle$'$ = **false**, extRelay1$'$ = **false**
$\square$
idle $\wedge$ extguard3 $\rightarrow$ idle$'$ = **false**, extRelay2$'$= **true**
$\square$
idle $\wedge$ extguard4 $\rightarrow$ idle$'$ = **false**, extRelay2$'$ = **false**

In order to check that no relay change is possible in the initial state, one can check that *idle* is true in the state after the initial state:

ltl_assertion
[InitIdle] InterlockingSystem $\vdash$ X(idle)

In order to ensure that a sequence of internal transitions terminates, it is checked that the transition system always will be *idle* at some point in the future:

ltl_assertion
[AlwaysEventuallyIdle] InterlockingSystem $\vdash$ G(F(idle))

6.6 Abstract generation of behavioural semantics

In the previous sections we have discussed different approaches for how to make a transition system that models the behaviour of a *StaticInterlockingSystem.* It has been decided to make a transition system consisting of one Boolean variable for each relay and one Boolean variable for each button. In this context, *true* is equivalent to drawn or pushed while *false* is equivalent to dropped or released. Furthermore, it has been decided to make one transition rule per relay for drawing it and one transition rule per relay for dropping it. How to obtain these rules is specified in section 6.3.

In addition to that, section 6.5 introduced the concept of *idle*. The system should be set to *idle* if and only if no internal event is possible.

The purpose of this section is to specify a function that, given a *StaticInterlockingSystem*, can generate a transition system that models the dynamic behaviour of the given interlocking system. The transition system will be represented using abstract syntax.

The following sections will only present parts of *RSL* schemes. The schemes can be viewed in their full length in appendix A that contains the abstract *RSL* model.

6.6.1 *RSL* abstract syntax for *RSL-SAL* transition systems

As previously mentioned in chapter 3, *RSL-SAL* contains the possibility of introducing several data types like maps and sets. As previously decided, the only data type we are going to need is Boolean. In the same way, it is only necessary to assign the values true or false to variables. The assigned values do not need to be evaluations of Boolean expressions.

In order to introduce abstract syntax for the *RSL* type *Bool* in a transition system, the following variant type has been added to the *Types* scheme (see appendix A.1, page 209):

```
type
  Boolean == True | False
```

We will now introduce the *TransitionSystem* scheme of the abstract model in appendix A.6, page 222. To introduce the abstract syntax for a variable declaration in the initial state of a transition system, the following short record type has been introduced:

```
type
   Var ::
      id : T.Id
      val : T.Boolean
```

id is the variable name which is of the same type as the identifiers for components inside a *StaticInterlockingSystem*. It will therefore be possible to use the *Ids* from a *StaticInterlockingSystem* as variable names in the transition system. *val* is the value of the given *Var* in the initial state.

The variables will be used in the state of the transition system. In order to define transition rules, it is necessary to define assignments. The short record type definition of an assignment is:

```
type
   Assignment ::
      id : T.Id
      assign : T.Boolean
```

id is supposed to be equal to the *id* of a *Var* inside the state of the transition system such that *assign* is assigned to that variable in the next state of the transition system.

RSL-SAL allows multiple assignments as part of a transition rule. In order to introduce multiple assignments, the following data type is introduced:

```
type
   MultipleAssignment = Assignment*
```

One could have chosen a set of assignments, but a list better reflects the fact that assignments in *RSL-SAL* are specified in an ordered, comma separated sequence.

Section 6.3 introduced how to compute rules for dropping and drawing relays. Such rules are formulated in terms of the data type *BooleanExp* that includes *literals* with *Ids*. Therefore, *BooleanExp* can be used directly as guard inside a transition rule. A type for transition rules that includes a guard and a *MultipleAssignment* is:

type

```
  TransitionRule ::
    guard : T.BooleanExp
    assignments : MultipleAssignment
```

After having introduced *Var* and *TransitionRule*, it is possible to make a type for a transition system:

```
type
  TransitionSystem ::
    state : Var*
    transitionRules : TransitionRule*
```

One could have chosen sets instead of lists, but we prefer having the transition system as close as possible to how a real *RSL-SAL* transition system is specified.

6.6.1.1 Well-formed transition systems

A data type for transition systems has now been introduced. However, it has not been specified what a well-formed transition system is. In our context, a well-formed *TransitionSystem* is a system that corresponds to a real *RSL-SAL* transition system that can be type-checked.

The following well-formed function has been introduced for testing whether a *TransitionSystem* is well-formed:

```
value
  isWfTransitionSystem : TransitionSystem → Bool
  isWfTransitionSystem(ts) ≡
    isWfState(state(ts)) ∧
    areWfTransitionRules(state(ts), transitionRules(ts))
```

isWfState checks that the variable declarations in the state use different *Ids*. *areWfTransitionRules* checks that:

- The identifiers used inside the guards are defined in variables of the state.
- Each multiple assignment contains minimum one assignment.

- If identifiers are referred to by assignments, the identifiers are defined in the state.
- A multiple assignment cannot contain more than one assignment for a given identifier.

6.6.2 Transformation to a transition system

A *TransitionSystem* has now been defined. Therefore, it is possible to define a transformation from a *StaticInterlockingSystem* to a *TransitionSystem.* To do that, scheme *StaticInterlockingSystemToTransitionSystem* has been introduced in the abstract *RSL* model in appendix A.7, page 224. The scheme defines the following function to make the behavioural semantics of a *StaticInterlockingSystem*:

value
  makeBehaviouralSemantics :
    SIS.StaticInterlockingSystem $\overset{\sim}{\rightarrow}$ TS.TransitionSystem
  makeBehaviouralSemantics(sis) **as** ts **post**
    TS.isWfTransitionSystem(ts) $\wedge$ stateRel(sis, ts) $\wedge$
    transitionRel(sis, ts)
  pre SIS.isWfStaticInterlockingSystem(sis)

The function takes a well-formed *StaticInterlockingSystem* as argument and gives back a well-formed *TransitionSystem.*

The function *stateRel* checks that the following properties hold:

- For each relay, the initial state of *sis* will contain a variable with the same *Id* as the relay and an initial value that corresponds to the state of the relay in the *StaticInterlockingSystem. true* corresponds to drawn and *false* corresponds to dropped.
- For each button, the initial state of *sis* will contain a variable with the same *Id* as the button and *false* as initial value.
- A variable *idle* is defined in the state and is initialised to false.
- No other variables are defined in the state.

The function *transitionRel* checks that the following properties hold:

- For each relay, the transition rules will contain one rule for drawing it and one rule for dropping it. The guards of the rules will correspond to the ones defined in section 6.3.
- A transition rule is defined for setting *idle* to true if no other transition is possible. The guard of the transition corresponds to the one specified in section 6.5.
- No other transition rules are defined.

6.7 Abstract generation of confidence conditions

It has now been specified how to obtain a transition system that models the behaviour of a *StaticInterlockingSystem.* As informally explained in sections 6.4 and 6.5, there are some *LTL* assertions that express confidence conditions of the generated transition system. This section will give a more formal description of these confidence conditions on an abstract level.

First, *RSL* abstract syntax for *LTL* assertions will be introduced. Then, an implicit function for generating the confidence conditions will be defined.

6.7.1 *RSL* abstract syntax of *LTL* assertions

In order to model *LTL* assertions in *RSL*, the following type has been added to the *Types* scheme of the abstract model shown in appendix A.1, page 209:

```
type
  LTLassertion ==
    G(g : LTLassertion) |
    F(f : LTLassertion) |
    X(x : LTLassertion) |
    Imply(lhs : LTLassertion, rhs : LTLassertion) |
    B(b : BooleanExp)
```

The variant type represents the necessary *LTL* operators for expressing the conditions introduced in sections 6.4 and 6.5. The constructors and destructors should be interpreted in the following way:

- *G(g)* means that *g* is globally true.

- *F(f)* means that f is eventually true.
- *X(x)* means that x is true in the next state.
- *Imply(lhs,rhs)* means that *lhs* implies *rhs*.
- *B(b)* means that the BooleanExp b is true.

For more information on *LTL* operators, see chapter 3.

6.7.2 *LTL* generation

It is now time to specify the generation of *LTL* assertions of the confidence conditions. Scheme *StaticInterlockingSystemToConfidenceConditions* is introduced for taking care of the conversion (see the complete scheme in appendix A.8, page 230). The following will introduce the central functions of *StaticInterlockingSystemToConfidenceConditions*:

For specifying the assertions described in sections 6.4 and 6.5, two generator functions are introduced.

The first function takes two Boolean expressions and generates an assertion specifying that the two expressions are never true in the same state:

```
value
  mutualExclusionLTL :
    T.BooleanExp × T.BooleanExp → T.LTLassertion
  mutualExclusionLTL(b1, b2) ≡
        T.G(T.B(T.neg(T.and({b1, b2}))))
```

The method gives abstract syntax corresponding to the formula given in section 6.4.1. When applied to *b1* and *b2*, the function gives abstract syntax corresponding to the following assertion:

```
ltl_assertion
  [mutualExclusionB1B2] InterlockingSystem ⊢ G(∼(b1 ∧ b2))
```

The function will be useful when specifying that the conditions for having a drawing current and a dropping current through a steel core relay are never true at the same time.

The second function takes an *Id* and two Boolean expressions as argument and gives a list of two *LTLassertions*:

```
value
   trueUntilChangeLTL :
      T.Id × T.BooleanExp × T.BooleanExp →
         T.LTLassertion*
   trueUntilChangeLTL(id, upGuard, downGuard) ≡
      ⟨T.G(
            T.Imply(
                  T.B(upGuard),
                  T.X(
                      T.Imply(
                            T.B(T.neg(upGuard)),
                            T.B(T.literal(id)))))),
         T.G(
            T.Imply(
                  T.B(downGuard),
                  T.X(
                      T.Imply(
                            T.B(T.neg(downGuard)),
                            T.B(T.neg(T.literal(id)))))))⟩
```

The function gives abstract syntax corresponding to the assertions specified in section 6.4.2. When being applied to *id*, *upGuard*, and *downGuard*, the function gives a list containing abstract syntax that corresponds to the following concrete syntax:

```
ltl_assertion
[upGuardUntilDrawn] InterlockingSystem ⊢
   G(upGuard ⇒ X(∼upGuard ⇒ id)),

[downGuardUntilDropped] InterlockingSystem ⊢
   G(downGuard ⇒ X(∼downGuard ⇒ ∼id))
```

It is now possible to specify an implicit function that gives all the conditions introduced in section 6.4 and section 6.5 (*SIStoTS* is an object of *StaticInterlockingSystemToTransitionSystem* and *SIStoTS.idle* is an underspecified value):

value

```
makeConfidenceConditions :
   SIS.StaticInterlockingSystem $\overset{\sim}{\to}$ T.LTLassertion$^*$
makeConfidenceConditions(sis) as cc post
   /* G(F(idle)) */
   T.G(T.F(T.B(T.literal(TS.id(SIStoTS.idle))))) $\in$
      elems cc $\wedge$
   /* X(idle) */
   T.X(T.B(T.literal(TS.id(SIStoTS.idle)))) $\in$
      elems cc $\wedge$
   /* For regular relays: */
   /* G(canDraw $\Rightarrow$ X($\sim$canDraw $\Rightarrow$ id)) */
   /* G(canDrop $\Rightarrow$ X($\sim$canDrop $\Rightarrow$ $\sim$id)) */
   ($\forall$ d : D.Diagram, id : T.Id •
      d $\in$ SIS.diagrams(sis) $\wedge$
      id $\in$ SIS.internalRelayIds(sis) $\wedge$
      D.isRegularRelay(id, d) $\Rightarrow$
         let
            canDraw = CF.canDrawRegularRelay(id, d),
            canDrop = CF.canDropRegularRelay(id, d)
         in
            elems trueUntilChangeLTL(id, canDraw, canDrop) $\subseteq$
               elems cc
         end) $\wedge$
   /* For steel core relays: */
   /* G(canDraw $\Rightarrow$ X($\sim$canDraw $\Rightarrow$ id)) */
   /* G(canDrop $\Rightarrow$ X($\sim$canDrop $\Rightarrow$ $\sim$id)) */
   /* G($\sim$(drawingCurrent $\wedge$ droppingCurrent)) */
   ($\forall$ d : D.Diagram, id : T.Id •
      d $\in$ SIS.diagrams(sis) $\wedge$
      id $\in$ SIS.internalRelayIds(sis) $\wedge$
      D.isSteelRelay(id, d) $\Rightarrow$
         let
            canDraw = CF.canDrawSteelRelay(id, d),
            canDrop = CF.canDropSteelRelay(id, d),
            drawingCurrent =
               CF.drawingCurrentThroughSteelRelay(id, d),
            droppingCurrent =
               CF.droppingCurrentThroughSteelRelay(id, d)
         in
            elems trueUntilChangeLTL(id, canDraw, canDrop) $\subseteq$
               elems cc $\wedge$
            mutualExclusionLTL(
                  droppingCurrent, drawingCurrent) $\in$
            **elems** cc
        **end**)
  **pre** SIS.isWfStaticInterlockingSystem(sis)
```

The first part of the post-conditions checks that the following assertions are generated:

- G(F(idle)), the system is always eventually idle
- X(idle), the system is idle in the state after the initial state

The second part of the post-conditions checks that assertions verifying the following are generated:

- For all relays: if the condition for drawing the relay is true, it remains true until the relay is drawn. If the condition for dropping the relay is true, it remains true until the relay is dropped.
- For all steel core relays: there can never be current through both parts of a steel core relay at the same time.

Note: The function allows the generation of additional confidence conditions. We do not intent to add others, but doing so would neither change the semantics of the associated transition system nor change the semantics of the necessary confidence conditions.

Chapter 7

Patterns for external behaviour and safety properties

Chapter 6 introduced the behaviour of a *StaticInterlockingSystem* in terms of a transition system. This chapter will present the next two steps of the method presented in section 4.1: developping patterns for the external behaviour for a station (i.e. rules will be stated for occupation of track sections, switching points, and pushing and releasing buttons) and patterns for safety properties to be verified for a transition system that includes both the internal and external behaviours, based on the documentation of the station.

Figure 7.1 illustrates how the internal relays of an interlocking system interact with the external world. The internal relays listen to the external world in the sense that pushing or releasing a button, occupying or freeing a track circuit, and changing a point might enable relay changes in the circuits of the interlocking system.

Our model of the internal behaviour responds to the external world by letting the guards of the transition rules refer directly to the state of the buttons, point relays, and track relays. Any change of their state might enable some transitions and disable other transitions.

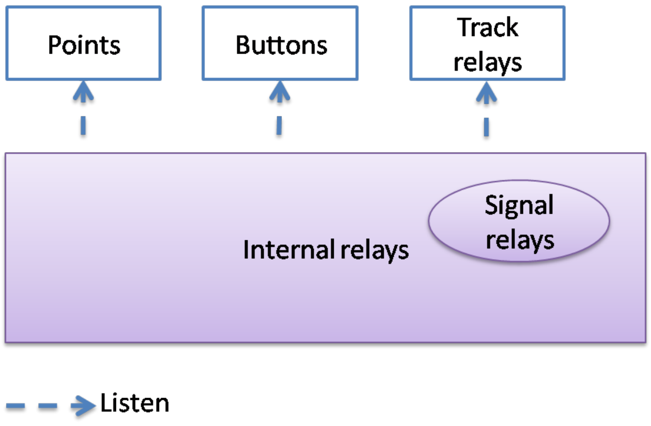


Figure 7.1: Interaction between internal relays and the external world

If a transition system, as the ones described in chapter 6, does only define the internal behaviour, it cannot be used for verifying safety properties. Because the track sections will never be occupied, the buttons will never be pushed, and the points will never be changed, the transition system will remain in an idle state. Therefore, the state space will only include two states: the normal state defined by the diagrams with *idle* equal to false and the same normal state with *idle* equal to true. In order to verify safety properties, external behaviour must be introduced such that the behaviour defined in chapter 6 can respond to the external world.

Section 7.1 will introduce patterns for adding external behaviour to a transition system containing the internal behaviour. After that, section 7.2 will introduce patterns for safety properties for transition systems containing both internal and external behaviour. The focus will both be on basic safety goals like having *no collision* and *no derailing* and be on safety properties derived from a train route table.

7.1 Patterns for external behaviour

As the internal behaviour of a *StaticInterlockingSystem* is defined by an *RSL-SAL* transition system, the only way of introducing external behaviour is adding transition rules for the external behaviour inside the transition system containing the internal behaviour.

In chapter 6 it was decided to have transition rules for the external behaviour on the following form:

$$[\,\text{externalRule}\,]\ \text{idle} \wedge \text{eg} \rightarrow \text{idle}' = \textbf{false}, \text{externalAssignments...}$$

where *idle* is true if and only if no rule for the internal behaviour is possible. *eg* is the condition for taking this transition that modifies the external world and *externalAssignments* are modifications of the external world.

Figure 7.2 illustrates how rules can be introduced for the external behaviour. By listening to the status of the points, the track relays, the buttons, and some of the internal relays of the diagrams (e.g. signal relays), the external transition rules define when it is possible to modify the external world.

In reality, several external events might happen at the same time. For instance, one might push a button at the exact same moment as a track relay goes down. However, combining external behaviour like having one transition for pushing a given button, one transition for occupying a given track, and one transition for doing both at the same time will lead to a high number of rules for the external transitions. It is therefore decided to ignore the fact that external events can happen concurrently. After changing the state of a point, a track relay, or a button, the relays of the circuits of the interlocking system will have time to respond to the event before new events can happen. This assumption excludes some system states that can happen in the reality, but obtaining such states in the reality is considered as being unlikely.

The following sections will discuss how to introduce behaviour for track relays, points, and buttons.

7.1.1 Patterns for track relay behaviour

This section will discuss two different approaches to changing the status of track relays. The two approaches can be characterised as *random track relay behaviour* and *ordered track relay behaviour*.

The following sections will look deeper into the two approaches.

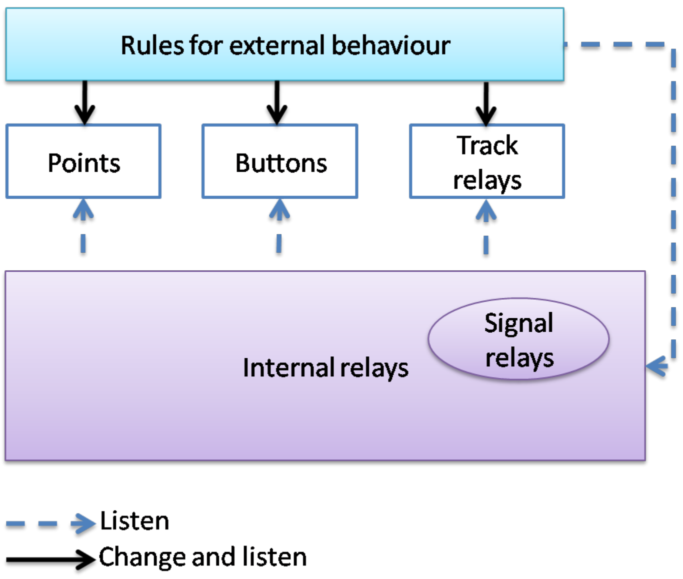


Figure 7.2: Interaction between internal relays and the external world using rules for external behaviour

7.1.1.1 Random track relay behaviour

Track relays monitor the status of the track circuits of the station. For instance, a track relay is down if there is a train on the track circuit it is linked to. However, having trains occupying track sections is not the only way of changing tracks relays. If a piece of metal falls on a track circuit, it might short-circuit the current running through the track circuit, causing the track relay to drop. With this in mind, one can argue that the behaviour of the track relays does not only depend on train movements: a track relay can go down any time, even though a train does not occupy the track circuit monitored by it.

The consequence of this observation is *random track relay behaviour*. When the system is idle, any track relay can change its state. A transition rule for changing the state of a track relay t might be:

$$[\,\text{randomT}\,]\ \text{idle} \rightarrow \text{idle}' = \textbf{false},\ \text{t}' = {\sim}\text{t}$$

Similar transitions can be introduced for every track relay of a station.

7.1.1.2 Ordered track relay behaviour

The advantage of having a random track relay behaviour is that it will include states where abnormal things happen, like a piece of metal falling down on a track section. The disadvantage is that it does not model train movements. Proving that trains neither derail nor collide will therefore be indirect compared to proving that two trains do never occupy the same track circuit at the same time.

If one wishes to include train movements inside the transition system, assumptions about trains must be introduced. Trains may have different lengths, implying that the number of track circuits a train is able to occupy at the same time varies from train to train.

In this section, we will assume the following properties about trains:

- Trains can at most occupy two track circuits at a time.
- Trains follow the track layout while moving.
- Trains respect the aspects displayed by the signals (see section 2.2.2 for more details about signals).
- Trains in general follow a direction while driving through a station. At some track circuits, a train might be able to change direction, but this will not apply to all the track sections. Shunting is not modelled.
- Trains follow the points, i.e. drive in the direction of which the points are directed.

Track relays are not enough to represent train movements because they do not indicate any direction. Therefore, movement rules for trains must be able to access and modify information on the direction of a train. There are at least two different ways of storing this information. One approach is to represent trains inside the transition system. A train might have a position and a direction. Another approach is to model the positions of trains. Instead of representing a specific train occupying a specific track, one can represent that there is a train on a given track in a given direction without knowing which train.

There are two main advantages of the second approach:

- The first advantage is that it will give a smaller state space: assume that we use the first approach and have two trains, *train1* and *train2*.

Furthermore, assume we have track sections *track1* and *track2*. In one state of the system, *train1* might be on *track1* and *train2* on *track2*. In another state of the system, *train2* might be on *track1* and *train1* on *track2*. The two states will have the same effect on the interlocking system and will both be in the state space. The second approach will not include such equivalent states, because it does not distinguish between *train1* and *train2*.

- The second advantage is that the second approach does not have a limited number of trains.

Therefore, the rest of this section will refer to the second approach. The strategy of it is illustrated in figure 7.3. The movement rules will use the position of the points and the status of the signal relays combined with the current *train positions* (i.e. information on where trains are located on the track circuits and their direction, together with the track relays) for deciding the next possible movement of a train. When a train moves, the train positions will be updated.

The patterns for the movement rules will be given in the following order: first rules for linear track circuits, then rules for points.

Movements on linear track circuits:
Consider figure 7.4 which shows an artificial station with an entrance signal *A*, an exit signal *B*, and two track sections, *T1* and *T2*, monitored by track relays *t1* and *t2* respectively. For that station, one can introduce the following variables indicating train positions for movements from A to B:

T1_AtoB : Bool := **false**,
T1_T2_AtoB : Bool := **false**,
T2_AtoB : Bool := **false**

Having T1_AtoB = **true** will indicate that there is a train on track circuit *T1* and it is moving from *A* to *B*. Similar, having T1_T2_AtoB = **true** will indicate that a train is on both *T1* and *T2* and is moving from *A* to *B*.

If trains were allowed to move from *B* to *A*, one could introduce the similar variables:

T2_BtoA : Bool := **false**,
T2_T1_BtoA : Bool := **false**,
T1_BtoA : Bool := **false**

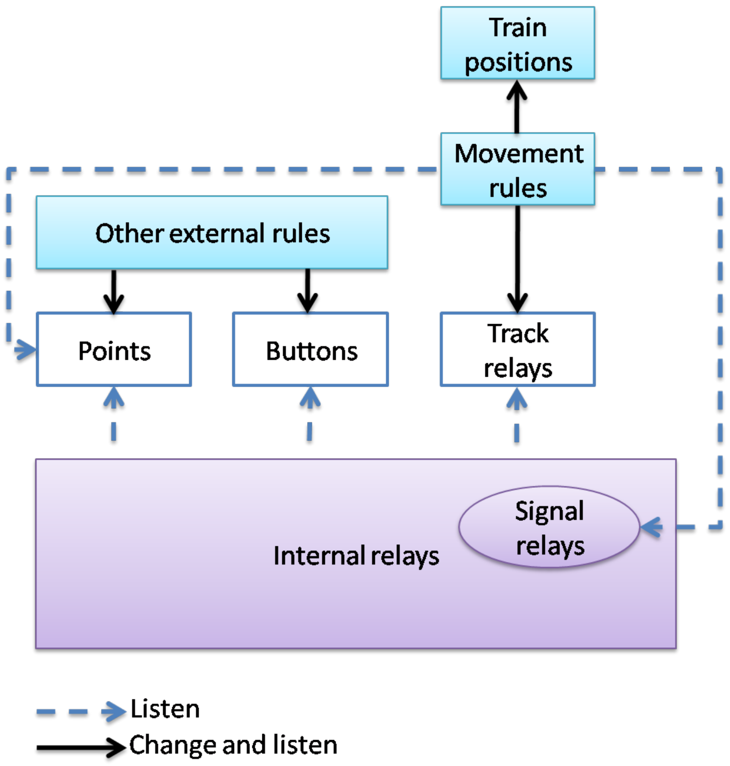


Figure 7.3: Interaction between internal relays and the external world using train movement rules

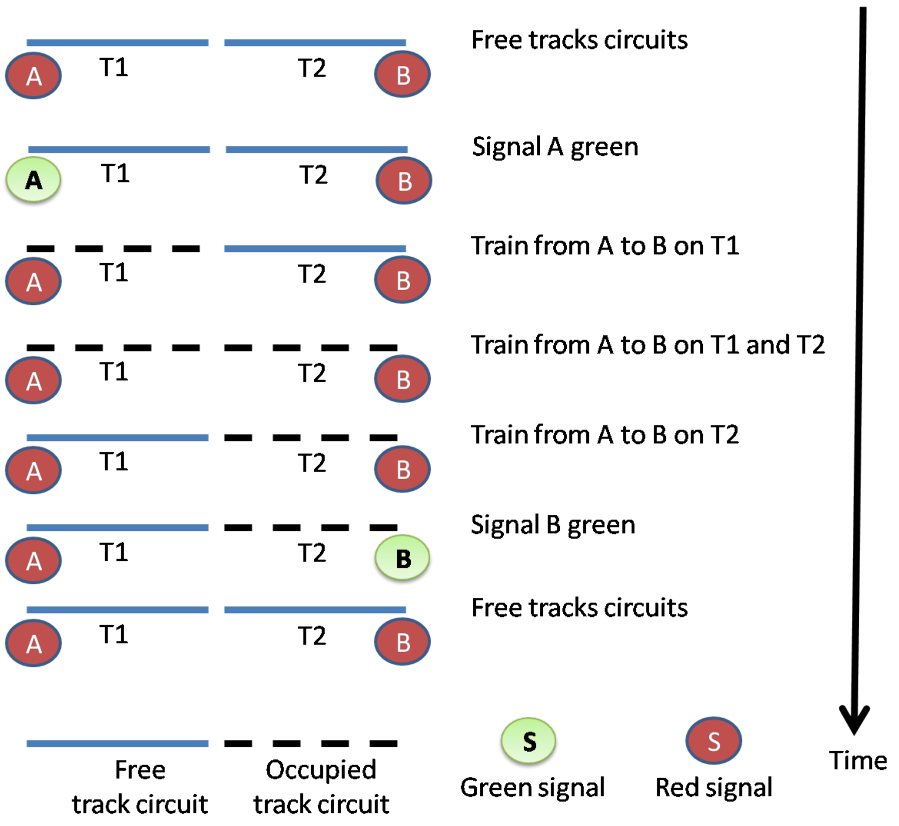


Figure 7.4: An ordered track relay behaviour for an artificial station. Signal *A* is an entrance signal and signal *B* is an exit signal for the station.

For instance, having T1_BtoA = **true** would indicate that a train is on *T1* and is moving from *B* to *A*.

In figure 7.4, the track circuits are initially free, as they are supposed to be in the normal state. A rule for letting a train enter the station from *A* could be:

```
[ insertTrainA ]
   idle ∧ aGreen →
      idle′ = false, T1_AtoB′ = true, t1′ = false
```

The rule inserts a train if signal *A* is green (i.e. signal *A* displays a *drive* aspect). It updates the train position *T1_AtoB* and makes track relay *t1* drop by setting *t1* to false. If there is already a train in the position *T1_AtoB*, the new train will not be taken into consideration because the variable can only tell that there is a train, but not how many trains there are. However, the situation will be equivalent to having a collision and one has to verify that this situation never happens (see section 7.2.1.1).

The rest of the rules for train movements from *A* to *B* are:

```
[ t1AtoB ]
   idle ∧ T1_AtoB →
    idle′ = false, T1_AtoB′ = false, T1T2_AtoB = true, t2 = false
[]
[ t1t2AtoB ]
   idle ∧ T1T2_AtoB →
    idle′ = false, T1T2_AtoB′ = false, T2_AtoB = true, t1 = true
[]
[ t2AtoB ]
   idle ∧ T2_AtoB ∧ bGreen →
    idle′ = false, T2_AtoB′ = false, t2 = true
```

t1AtoB and *t1t2AtoB* make the train move towards *B* and occupy and free track circuits corresponding to the pattern shown in figure 7.4. *t2AtoB* makes the train leave the station if signal *B* is green. Similar rules can be made for movements in the other direction. This would, however, require the introduction of an entrance signal and an exit signal for movements from *B* to *A*.

Movements on points:
If a station includes points, the train movement rules should be able to send the

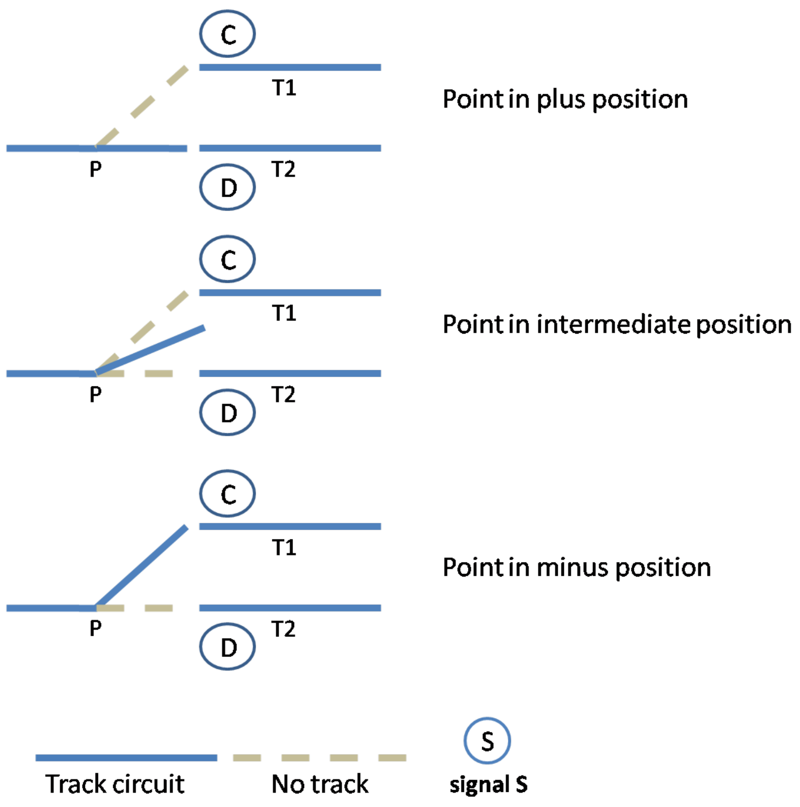


Figure 7.5: The possible positions of a points and two signals associated with it. For trains to enter the point from *T1* and *T2*, the signals *C* and *D* must be green respectively.

train in different directions depending on the position of the points. An example of a point that is linked to two linear track circuits can be seen in figure 7.5.

When a train approaches point *P* from the left, three things can happen:

- If the point is in the plus position, the point will direct the train to *T2*.
- If the point is in the minus position, the point will direct the train to *T1*.
- If the point is in the intermediate position, the train will derail.

The movements from *P* to *T1* and *T2* there can therefore be represented by the following position variables:

- *P_right*, indicating that the train is on *P* and is moving towards *T1* or *T2*.
- *P_T1_right*, indicating that the train is on *P* and on *T1* and is moving towards *T1*.
- *P_T2_right*, indicating that the train is on *P* and on *T2* and is moving towards *T2*.
- *T1_right*, indicating that the train is on *T1* and is moving away from *P*.
- *T2_right*, indicating that the train is on *T2* and is moving away from *P*.

There is no reason for representing the intermediate position: if a train is on a point that is in the intermediate position, a safety property should detect that a derailing is happening (see section 8.3.1.2).

Similar train positions can be constructed for the movements from *T1* and *T2* to *P*.

Suppose *plusP* and *minusP* are the relays that are drawn if and only if *P* is in the plus position and minus position respectively. Also, suppose that *p*, *t1*, and *t2* are the track relays that monitor track circuits *P*, *T1*, and *T2* respectively. The movement rules for movements from left to right can then be defined as:

```
[P_right_plus]
  idle ∧ plusP ∧ P_right →
    idle′ = false,
    P_right′ = false,
    P_T2_right′ = **true**,
    t2′ = **false**
▯
[P_right_minus]
  idle ∧ minusP ∧ P_right →
    idle′ = **false**,
    P_right′ = **false**,
    P_T1_right′ = **true**,
    t1′ = **false**
▯
[P_T1_right]
  idle ∧ P_T1_right →
    idle′ = **false**,
    P_T1_right′ = **false**,
    T1_right′ = **true**,
    p′ = **false**
▯
[P_T2_right]
  idle ∧ P_T2_right →
    idle′ = **false**,
    P_T2_right′ = **false**,
    T2_right′ = **true**,
    p′ = **false**
```

When a train is moving from *T1* and *T2* towards *P*, the train will either occupy *P* or derail if *P* is not in the required position. Therefore, the movement rules in that direction should not consider the position of *P*. However, they must consider the signals that allow the train to enter a point.

Suppose a signal relay *cGreen* is drawn when signal *C* displays a *drive* aspect. Also, suppose *T1_left* represents that there is a train on *T1* driving towards *P* and that *T1_P_left* represents that there is a train on *P* and *T1* driving away from *T1*. A movement rule that allows a train to enter *P* from *T1* can then be formulated in the following way:

```
[T1_left]
    idle ∧ T1_left ∧ cGreen →
        idle′ = **false**,
        T1_left′ = **false**,
        T1_P_left′ = **true**,
        p′ = **false**
```

7.1.1.3 Conclusion

In this section two kinds of track relay behaviour were introduced:

- *Random track relay behaviour*: Any track circuit can be occupied or free at any time.
- *Ordered track relay behaviour*: Track circuits are occupied by trains following movement rules. These rules are based on assumptions about trains, e.g. that a train can at most occupy two track circuits at a time.

Due to the simplicity of the rules for *random track relay behaviour* it can easily be applied to any station and will include situations where a track circuit appears to be occupied even though there is no train at the station.

Ordered track relay behaviour requires analysis of the track layout before formulating the movement rules and does not include the states where track circuits are not occupied by trains. However, *ordered track relay behaviour* allows for direct verification of some of the basic safety goals described in section 2.3.1: trains do not collide and trains do not derail.

In general, it must be concluded that *random track relay behaviour* will cover every possible combination of track relay changes while *ordered track relay behaviour* only covers a subset of these. This subset can be increased by adding trains with different lengths to the system. However, doing so might not scale very well, because movement variables should be added for each possible train length. This will increase the state space, the number of variables, and the number of transition rules.

One should also consider the length of the track circuits of a station in order to increase realism. For instance, it may not be realistic that a train at some point only occupies a point.

Therefore, a *random track relay behaviour* might be preferable even though the safety properties related to it are not directly related to the basic safety goals.

7.1.2 Patterns for point behaviour

This section will present patterns for point behaviour. First, assumptions will be stated about the point behaviour and then the patterns will be formulated.

7.1.2.1 Assumptions about point behaviour

Real interlocking systems are supposed to enforce that the position of points can only be changed when it is considered as being safe. Point control is done using circuits containing relays, implying that the functions developed in chapter 6 can be used for generating the behaviour of point control from the diagrams describing its circuits. However, hardware implementation of point control can be quite complex compared to implementations of locking and unlocking of train routes. In this project, we will therefore not look deeper into real hardware implementation of point control, even though the functions defined in chapter 6 could be used for handling it.

In reality, a point should never be changed when a train route that includes the point is locked or when the point is occupied. In this section, we assume that points do not malfunction and that the rules for changing points are obeyed. The condition for changing a point must therefore express that such routes are not locked and that the point is not occupied when the given point is changed.

Usually, switching a point is initiated by pushing buttons. Therefore, it is decided to treat the changes of a point as external events. In other words, points can be changed when the system is *idle* and when changing a point, *idle* should be set to false.

7.1.2.2 Formulation of patterns

The purpose of this section is to introduce a model abstraction from the real point control. As described in section 2.2.1.2, points can be in a *minus position*, a *plus position*, and an *intermediate position*. For each point, there are two relays:

- A relay that is drawn if and only if the given point is in the plus position.
- A relay that is drawn if and only if the given point is in the minus position.

Neither of the two relays are drawn if the point is in the intermediate position.

Suppose we have a point P with relay $plusP$ indicating whether the point is in the plus position and relay $minusP$ indicating whether the point is in the minus position.

Suppose that a Boolean expression, *gP*, expresses the fact that no train routes that includes *P* is locked and that *P* is free. In the real world, the given point should never be changed when *gP* is false. Therefore, one could actually prove that this actually is enforced by the interlocking system. However, as we assume that the rules for changing the points are obeyed, we can use *gP* as part of the guard for changing the points.

The following transition rules defines the behaviour of *P*:

$$
\begin{array}{l}
[\,\text{intermediateToMinus}\,] \\
\quad \text{idle} \wedge \text{gP} \wedge \sim\!\text{plusP} \wedge \sim\!\text{minusP} \rightarrow \text{idle}' = \mathbf{false}, \text{minusP}' = \mathbf{true} \\
[\!] \\
[\,\text{minusToIntermediate}\,] \\
\quad \text{idle} \wedge \text{gP} \wedge \sim\!\text{plusP} \wedge \text{minusP} \rightarrow \text{idle}' = \mathbf{false}, \text{minusP}' = \mathbf{false} \\
[\!] \\
[\,\text{intermediateToPlus}\,] \\
\quad \text{idle} \wedge \text{gP} \wedge \sim\!\text{plusP} \wedge \sim\!\text{minusP} \rightarrow \text{idle}' = \mathbf{false}, \text{plusP}' = \mathbf{true} \\
[\!] \\
[\,\text{plusToIntermediate}\,] \\
\quad \text{idle} \wedge \text{gP} \wedge \text{plusP} \wedge \sim\!\text{minusP} \rightarrow \text{idle}' = \mathbf{false}, \text{plusP}' = \mathbf{false}
\end{array}
$$

The rules allow to switch *P* from the intermediate position to the plus position or the minus position and from the plus position or the minus position to the intermediate position, as shown in figure 7.6. When defining behaviour for each point, one can introduce similar rules for every point if one does not want to model the circuits that control the points.

7.1.3 Patterns for button behaviour

This section will present patterns for button behaviour. First, assumptions will be stated about the button behaviour and then the patterns will be formulated.

7.1.3.1 Assumptions about button behaviour

In reality, buttons can be pushed or released any time, which in our context will correspond to allowing changing the state of a button when the system is idle as well as not idle. However, as explained in section 6.5, we will only allow the changes of a button when the system is idle.

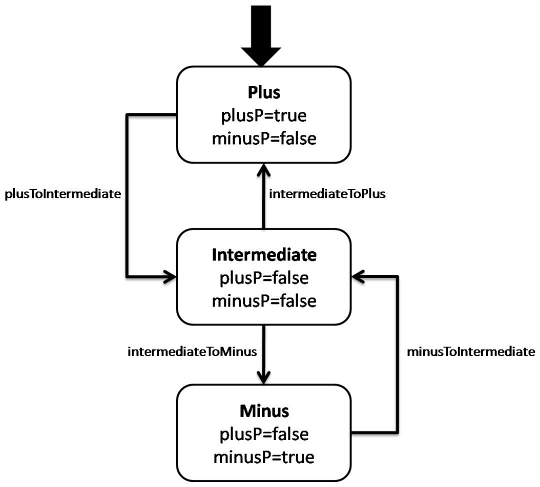


Figure 7.6: State transition diagram of point P, with the behaviour described in section 7.1.2

Also, it is in reality possible to have several buttons pushed at the same time. However, there is a problem related to having several buttons pushed at the same time. If one tries to lock conflicting train routes at the same time by having several buttons pushed at the same time, it might lead to a competition between the locking of the train routes. Only one of these train routes can be locked at the same time, so the timing of the electrical circuits will decide which train route there will be locked. From a logical point of view, the end result of the locking procedure will be non-deterministic.

Non-determinism when drawing and dropping will contradict one of the confidence conditions introduced in section 6.4. It was decided to verify that if the condition for drawing or dropping a relay is true, it remains true until the relay has changed its state. Competition between the locking of several train routes will imply that some relays for locking the conflicting train routes can change state, but when one of the train routes eventually becomes locked, some of the relays cannot change their state any more due to the fact that the given train route was locked. In that way, the introduced confidence condition in 6.4 will be invalid for the involved relays.

As this type of confidence condition is important for the soundness of the verification of safety properties, it is decided not to include the competition between several buttons as part of the external behaviour. It is therefore decided to assume that only one button can be pushed at a time.

Another problem occurs when a button is kept pushed too long. Usually interlocking systems have an emergency release procedure for unlocking train routes in case a button is pushed too long. However, in this project, we do not model such a procedure. This implies that if a button remains pushed, it may prevent a train route from being unlocked even if all the conditions for unlocking it are true. Therefore, it is necessary to enforce that a button is not pushed too long.

This can be done by releasing every button before setting *idle* to true combined with pushing only one button at a time, i.e. when the system is in an idle state, all buttons are released.

7.1.3.2 Formulation of patterns

Suppose we have a system with two buttons, *B1* and *B2*. The initialisation of these buttons in an auto-generated transition system would be:

$\mathrm{B1} : \mathrm{Bool} := \mathbf{false},$
$\mathrm{B2} : \mathrm{Bool} := \mathbf{false}$

Transition rules for pushing the two buttons are:

$[\,\mathrm{pushB1}\,]\ \mathrm{idle} \rightarrow \mathrm{idle}' = \mathbf{false},\ \mathrm{B1}' = \mathbf{true}$
$[\!]$
$[\,\mathrm{pushB2}\,]\ \mathrm{idle} \rightarrow \mathrm{idle}' = \mathbf{false},\ \mathrm{B2}' = \mathbf{true}$

The two guards do not contain $\sim B1$ or $\sim B2$ because one of the assumptions of this button behaviour is that the system cannot be idle if a button is pushed. If one wishes to apply another kind of behaviour, one should not forget to add them to the guards.

Now, the only problem is how to release the button that is pushed before setting *idle* to true. One might be tempted to release it when *idle* is set to true. However, releasing a button might enable some internal transitions, meaning that *idle* is not supposed to be true any more. Releasing a button and setting the system in an idle state must therefore happen in several steps:

1. When no internal transition is possible, the button that is pushed is released. This might enable further internal transitions.

2. When no internal transition is possible, *idle* is set to true.

Enforcing this can be done by modifying the transition rule for setting *idle* to true such that it can be taken twice before setting *idle* to true. Suppose *ExternalOK* is the requirement described in section 6.5.3.1 for setting *idle* to true.

The *idle* transition described by section 6.5 is then on the following form:

$$[\,\mathrm{setIdle}\,]\ {\sim}\mathrm{idle} \wedge \mathrm{ExternalOK} \rightarrow \mathrm{idle}' = \mathbf{true}$$

In a system with two buttons, *B1* and *B2*, the *idle* transition can then be modified in the following way:

$$[\,\mathrm{setIdle}\,]\ {\sim}\mathrm{idle} \wedge \mathrm{ExternalOK} \rightarrow \mathrm{idle}' = {\sim}(\mathrm{B1} \vee \mathrm{B2}),\ \mathrm{B1}' = \mathbf{false},\ \mathrm{B2}' = \mathbf{false}$$

If no internal transition is possible, *setIdle* will release the buttons. Furthermore, *idle* will be set to true if and only if all the buttons are released in the current state. When *setIdle* is taken, there will be two possible scenarios:

1. If every button is released, *idle* is set to true.
2. If a button is pushed, it will be released, but *idle* will not be set to true. The next time *setIdle* is taken, all the buttons are released and *idle* is set to true.

An example of a chain of transitions under the introduced button behaviour can be seen in figure 7.7.

Similar button behaviour can be introduced for an arbitrary transition system by treating every button in the same way as *B1* and *B2*. In that way, competition between the locking of different train routes is avoided.

Note: It will still be possible to try locking a train route *R2* when a conflicting train route *R1* is locked. When the conflicting train route *R1* is locked, the buttons for locking this train route will be released and a new button (like the one initiating the locking of *R2*) can be pushed.

In the real world, this behaviour corresponds to the fact that the operator must push a button until the actions enabled by it have taken place and that he or she

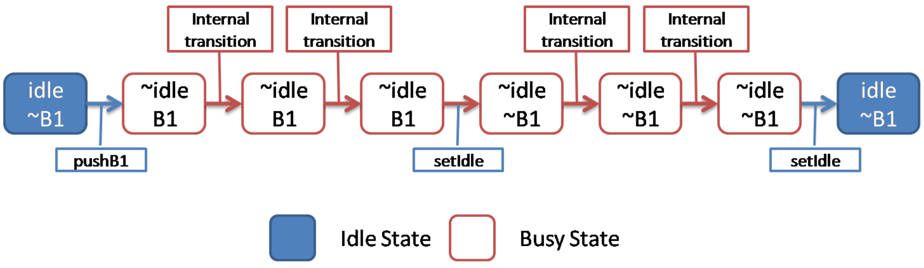


Figure 7.7: A possible chain of transitions with the button behaviour described in section 7.1.3

afterwards releases the button in time such that the system can respond to this release before anything else happens. One can avoid making this assumption by modelling the emergency release procedure, using the diagrams describing it. The assumption is only introduced to limit the scope of this project.

The second assumption requires that the operator only pushes one button at a time, which does not require any knowledge of the internal state of the interlocking system. If one wishes to avoid this assumption, one should ensure that no race between the locking of train routes takes place.

7.2 Patterns for safety properties

After having analysed how to add external behaviour to a transition system containing rules for the internal behaviour of an interlocking system, it is now time to introduce patterns for safety properties that should be proven to hold for the complete transition system.

First of all, it is important to prove the basic safety goals described in section 2.3.1:

- Trains do not collide.
- Trains do not derail.

Secondly, it can be proved that the implementation requirements specified in a train route table are respected. Proving these will (as explained later) turn out to be an indirect way of proving the basic safety goals.

As described in section 7.1, safety can be verified under different assumptions about the external behaviour. In this section, we will assume having the point behaviour and the button behaviour described in section 7.1. For the track relay behaviour, safety properties can be verified using:

- Random track relay behaviour.
- Ordered track relay behaviour.

For these kinds of track relay behaviour, we will use the assumptions described in 7.1. As ordered track relay behaviour uses train positions, verification of the basic safety goals can be more direct for this kind of track relay behaviour. Therefore, the first of the following sections will describe how to express the basic safety goals in terms of *LTL* assertions. Afterwards, safety properties that apply to both random track relay behaviour and ordered track relay behaviour will be introduced.

7.2.1 Patterns for basic safety properties

With the random track relay behaviour, it is not possible to distinguish between trains and other objects that occupy a track section. Therefore, if one wants direct proofs of the basic safety goals, it is necessary to use ordered track relay behaviour, as specified in section 7.1.1.2.

Recall the station shown in figure 7.4, page 113. For that station, the following track positions were introduced:

T1_AtoB : Bool := **false**,
T1_T2_AtoB : Bool := **false**,
T2_AtoB : Bool := **false**,
T2_BtoA : Bool := **false**,
T2_T1_BtoA : Bool := **false**,
T1_BtoA : Bool := **false**

7.2.1.1 No collision

This section will formulate *LTL* assertions for checking that collisions do not take place.

When expressing the next *LTL* assertions, the following function will be used for extracting a *Nat* from a *Bool* expression such that *1* corresponds to *true* and *0* corresponds to *false*:

```
value
   v : Bool → Nat
   v(b) ≡ if b then 1 else 0 end
```

A collision on track section *T1* has occurred if there is more than one train on *T1*.

The following *LTL* assertion checks that at most one train is on track section *T1* at a time:

```
ltl_assertion
   [ noCollisionOnT1 ] InterlockingSystem ⊢
      G(v(T1_AtoB) + v(T1T2_AtoB) + v(T1_BtoA) + v(T2T1_BtoA) ≤ 1)
```

This is not sufficient for checking that no collision takes place on *T1*. If *T1_AtoB* is true and the signal *A* is green, a new train can enter the station without changing the train position variables. The transition rule for inserting the new train will set *T1_AtoB* to true. As *T1_AtoB* is already true, the change will not be detected by the above assertion and thereby not be detected as a collision.

To check that such a collision does not take place, the following *LTL* assertion is introduced:

```
ltl_assertion
   [ noTrainOnT1WhenAGreen ] InterlockingSystem ⊢
      G(idle ∧ aGreen ⇒
      v(T1_AtoB) + v(T1T2_AtoB) + v(T1_BtoA) + v(T2T1_BtoA) = 0)
```

The *LTL* expression states that it holds for every state that if the system is *idle* and signal *A* is green, then track section *T1* is empty. The check for having *idle* must be added in order to give the system a chance to respond on having *T1* occupied. Signal *A* is supposed to be green to allow a train to enter *T1* (driving towards *B*). Then, when a train enters *T1*, the system is supposed to respond to that during a sequence of non-idle states. Finally, when *idle* becomes true, the system has responded and signal *A* is not supposed to be green any more.

Similar properties can be introduced for *T2*. In fact, they would be equivalent to the ones defined for *T1* with a few modifications of the used variables in the expressions.

For a real station, one will have to analyse the track layout when formulating similar properties for a specific station.

7.2.1.2 No derailing

This section will formulate *LTL* assertions to check that derailing does not take place.

Consider point *P* in figure 7.8. When trains approach *P* from *T0*, they cannot derail if the point is either in the plus position or in the minus position and remains in that position while the train occupies it.

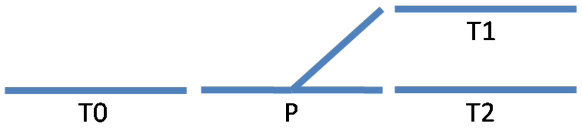


Figure 7.8: A point *P* and the track sections next to it.

Suppose that *P* has relays *plusP* and *minusP* for indicating its position. Furthermore, suppose that *p* is the name of the track relay that monitors whether there is a train on *P*. The following expression will check that *P* is either in the plus position or the minus position when *P* is occupied:

ltl_assertion
  [plusOrMinusPositionWhenP] InterlockingSystem $\vdash$
    $\mathrm{G}(\sim\mathrm{p} \Rightarrow \mathrm{plusP} \vee \mathrm{minusP})$

The expression states that if there is a train on *P*, *P* cannot be in the intermediate position. As *P* needs to enter the intermediate position on the way from the plus position to the minus position or the other way around, the expression will also ensure that the position of the point is not changed when a train is on it.

When ensuring this, derailing cannot happen when trains are approaching *P* from *T0*. If a train reaches *P*, the point will remain in either the plus position or the minus position until the train has left the track. However, when a train approaches *P* from either *T1* or *T2*, this property is not sufficient to ensure that derailing does not take place. For instance, if *P* is in a position such that *P* and *T1* are connected and a train approaches *P* from *T2*, a derailing will take place.

Consider figure 7.9 that shows different occupations of *P*, *T1*, and *T2*. When a train is on *P* and *T1*, the point must connect *P* and *T1*. Similarly, if a train occupies *P* and *T2*, *P* must connect *P* and *T2*.

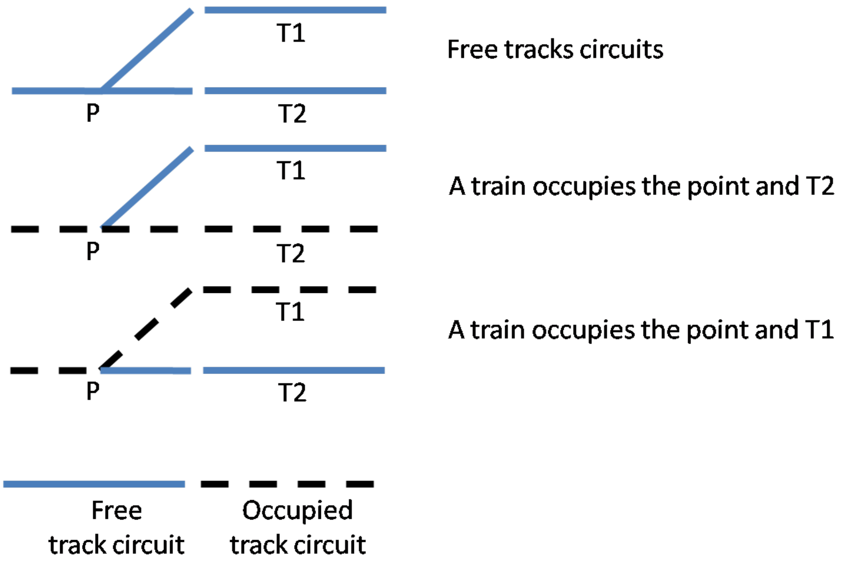


Figure 7.9: Different ways of occupying a point and the track sections next to it.

Suppose that having *plusP* drawn indicates that *P* is connected to *T2* and that *P_T2_fwd* and *T2_P_back* are the position variables for representing that a train occupies *P* and *T2*, going from *P* to *T2* or from *T2* to *P* respectively. Then, the following expression will check chat *P* must be in the plus position when there is a train on both *P* and *T2*:

ltl_assertion
[no_derailing_P_t01] InterlockingSystem $\vdash$
$\quad$ G((v(P_T2_fwd) + v(T2_P_back) $\geq$ 1) $\Rightarrow$ plusP)

If *minusP* indicates that *P* is connected to *T1* and that *P_T1_fwd* and *T1_P_back* are the position variables representing that a train occupies *P* and *T1*, going from *P* to *T1* or from *T1* to *P* respectively, the similar expression for *T1* and *P* is:

ltl_assertion
[no_derailing_P_t01] InterlockingSystem $\vdash$
    G((v(P_T1_fwd) + v(T1_P_back) $\geq$ 1) $\Rightarrow$ minusP)

When model checking the above assertions for *P* using *ordered track relay behaviour*, one can ensure that trains do not derail. Similar expressions should be introduced for each point of a station in order to make sure that derailing does not take place in the context of *ordered track relay behaviour.*

7.2.2 Patterns for safety properties extracted from the train route table

This section will focus on safety properties that can be checked for both *random* and *ordered track relay behaviour.* These properties are derived from train route tables. After having introduced the properties, it will be analysed whether these properties ensure safety. The properties will be formulated in such a way that the train positions from *ordered track relay behaviour* are not required.

Further information on train routes and train route tables can be found in section 2.4.

7.2.2.1 Locking of conflicting routes

As mentioned in section 2.4, a train route table specifies the routes that are considered as being conflicting. Therefore, it is relevant to verify that conflicting train routes are never locked at the same time.

Having a train route locked is usually indicated by having some points in a certain position combined with having a steel core relay in a certain state. Therefore, one can tell whether a given train route is locked by examining the state of the interlocking system.

Suppose that *L1* is true when a specific train route, *TR1*, is locked. Furthermore, assume that train route *TR2* is locked when *L2* is true and train route *TR3* is locked when *L3* is true. That *TR2* and *TR3* are never locked when *TR1* is locked can then be formulated in the following way:

ltl_assertion
  [notT2orT3WhenT1] InterlockingSystem $\vdash$
    $G(L1 \Rightarrow \sim(L2 \vee L3))$

7.2.2.2 Signals

Train route tables specify that some conditions need to be fulfilled when changing the entrance signal of a train route to a *drive* aspect. These conditions can be found in a train route table. For instance, when an entrance signal allows a train to approach a station, the condition for letting a train enter one of the routes that go past the signal must be fulfilled. These conditions are:

- A train route that uses the train as an entrance signal is locked.
- The points are in the legal position for the locked train route.
- The track sections specified for the locked train route are free.
- The signals display the aspects specified for the locked train route.

For a given station, assume that we have a signal *A* and two entrances routes, *TR1* and *TR2*, that go past *A*. Suppose that *CTR1* and *CTR2* are the conditions for entering *TR1* and *TR2* respectively.

If *aGreen* is the relay that is drawn when *A* allows trains to enter the station, the following assertion expresses that *A* will only allow trains to enter the station when the conditions for doing so are fulfilled:

ltl_assertion
  [enterFromAWhenSafe] InterlockingSystem $\vdash$
    $G(\mathrm{idle} \wedge \mathrm{aGreen} \Rightarrow \mathrm{CTR1} \vee \mathrm{CTR2})$

The formula states that if signal *A* is green in an idle state, one of the conditions *CTR1* and *CTR2* is true.

7.2.2.3 Stop fald

A train route table specifies when the entrance signal of a given train route should stop displaying a *drive* aspect. Suppose that signal *A* from the previous section must stop displaying a *drive* aspect when track T (monitored by track relay t) is occupied. The following assertion expresses that the signal stops displaying a *drive* aspect when the given track is occupied:

ltl_assertion
 [stopFallA] InterlockingSystem $\vdash$
  G(idle $\wedge$ ∼t $\Rightarrow$ ∼aGreen)

It is necessary to add *idle* on the left-hand side of implication in order give the system time to respond to the occupation of T.

7.2.2.4 Point positions

It is relevant to verify that if a train route is locked, the points of this train route are in the position specified for it.

Suppose L is the condition for having a train route locked and CP is true if the points associated to the given train route are in the acceptable positions defined by the train route table for the given train route. The following assertion will then express that the points are in the right position when the route is locked:

ltl_assertion
 [pointsInPositionWhenLocked] InterlockingSystem $\vdash$
  G(L $\Rightarrow$ CP)

In that way, a train following the train route will not derail if the route remains locked until the train has passed the point. However, the formula in itself is not enough to ensure that derailing does not take place. If the route is unlocked before a given train has passed the point, one cannot be sure that the point does not switch position before the train has passed it.

7.2.2.5 Train route release

As mentioned before, if a train is following a train route and the train route is unlocked before the train reaches the end of it, a point of the train route might be switched before the given train has passed it. In other words, if the train route is unlocked too early, derailing might happen. This section will introduce patterns that can be used for model checking that a given train route remains locked long enough.

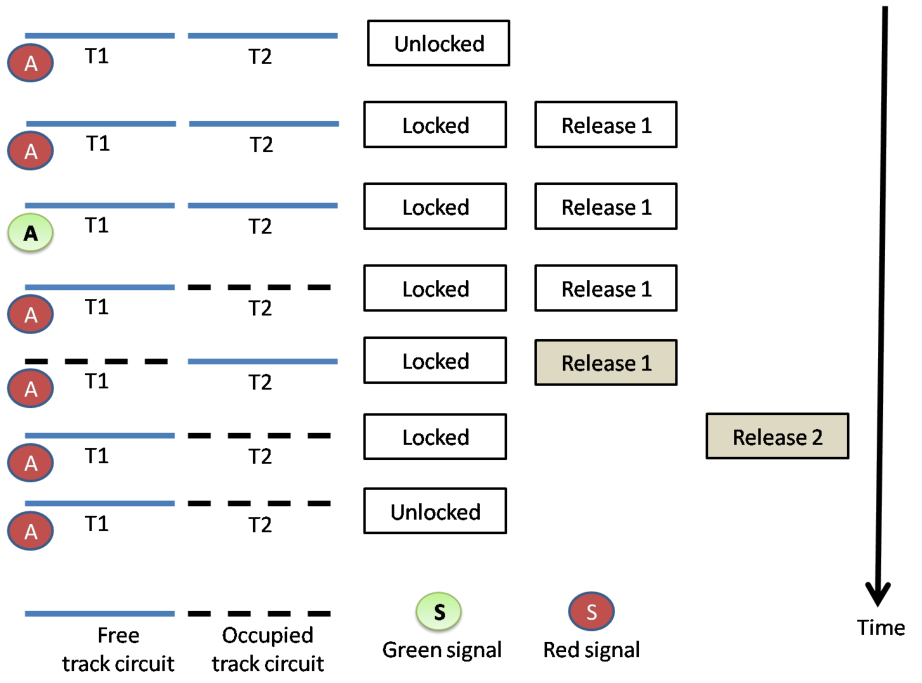


Figure 7.10: Scenario: locking of a train route under random track relay behaviour. The conditions *Release 1* and *Release 2* are used to check that the route is not released too early.

A train route table establishes when a train route is to be released. Usually rules for releasing train routes specify conditions on the occupation of two track circuits. Suppose the involved track circuits are *T1* and *T2*. The following states can then be used to specify the rule for unlocking the train route:

- *State 1*: *T1* is occupied and *T2* is free.
- *State 2*: *T2* is occupied and *T1* is free.

An example of a rule can then be that a given train route must not be unlocked unless *state 1* occurs at some point and that *state 2* occurs at some point later before the route is unlocked.

A scenario using random track relay behaviour in which a train route becomes locked and unlocked can be seen in figure 7.10. As seen in the scenario, the route remains locked until *state 1* and *state 2* have happened in the order specified by the rule. The pattern *state 2* eventually followed by *state 1* is observed, but the release does not happen before the pattern *state 1* eventually followed by *state 2* is observed.

For checking that the unlocking does not happen too early, we will introduce two *LTL* assertions.

Suppose *L* is the condition for having the train route in figure 7.10 locked and *t1* and *t2* are the track relays monitoring *T1* and *T2*.

The following assertions can then specify that the train route is not unlocked before *state 1* has occurred:

ltl_assertion
[release1] InterlockingSystem ⊢
    G(∼L ∧ X(L) ⇒ X(W(L, L ∧ ∼t1 ∧ t2)))

release1 is illustrated in figure 7.10. When the train route is unlocked in the current state and locked in the next state, it should hold that, from the next state, the route must remain locked until *state 1* occurs and that the route is still locked in that state. The boxes marked *Release 1* indicate that the weak until operator ensures that the route remains locked until *state 1.*

The second assertion is:

ltl_assertion
[release2] InterlockingSystem ⊢
    G(L ∧ X(∼L) ⇒ t1 ∧ ∼t2)

release2 is also illustrated in figure 7.10. When the train route is locked in the current state and unlocked in the next state, we have *state 2.* The box marked *Release 2* indicates if the assertion applies to the state before the route is unlocked.

Together the two rules specify that if the train route becomes locked at some point, *state 1* must happen before the unlocking of the route and that if the unlocking happens, we have *state 2.*

7.2.2.6 Indirect verification of basic safety goals

We have now introduced *LTL* assertions for expressing that:

1. Conflicting train routes are not locked at the same time.
2. The signals can only allow the trains to pass them if the conditions for doing so are true. Such conditions are given by the train route tables. Some signals must display some specific aspects, some points need to be in a specific position, some track circuits need to be free, combined with having a train route locked.
3. A signal displaying a *drive* aspect changes to *stop* aspect when a specific track section is occupied.
4. The points are in the required position by a train route as long as the train route is locked.
5. A train route is not released too early.

One can notice that property 3 is included in property 2. Property 2 states among other things that a signal cannot display a *drive* aspect if a track in the train route is occupied. This includes the track which is referred to by property 3.

Neither of these properties will alone ensure the overall basic safety goals. The interesting question is then whether a combination of these ensure that neither *collision* nor *derailing* take place.

To ensure that collisions do not take place, the following needs to be verified:

- **Conflicting train routes cannot be locked at the same time**: property 1 will ensure this.
- **That a train can only enter or leave a station when it is considered as being safe**: property 2 will ensure that. By verifying that property for every signal, several trains will not be allowed to enter conflicting train routes at the same time.
- **That a train entering a train route will follow it**: for instance, if the points are changed when a train is using its train route, this train will be sent away from it. In order to enforce this requirement, property 4 and 5 are needed. As these properties state that the points will remain in their

position when a train route is locked and the train route is not released too early (i.e. before all the points of the train route were passed), it means that the train cannot leave its route.

By ensuring that two conflicting train routes cannot be locked at the same time, that a train is only allowed to enter or leave a station when it is safe and that it has a route to follow, combined with ensuring that the train will actually follow its route until it reaches the end of it, one can be sure that collisions do not take place.

Also, the combination of 4 and 5 ensures that no derailing takes place. As the points are in the required position when a given train route is locked and a given train route cannot be released too early, derailing cannot happen.

In other words, properties 1, 2, 4, and 5 are enough for ensuring the basic safety goals. If one verifies these properties for every point, signal, and train route, one can verify that the basic safety goals are ensured by a given interlocking system.

7.3 Conclusion

In this chapter, patterns were introduced for adding external behaviour (i.e. behaviours of points, track relays, and buttons) to a transition system that models the internal behaviour of an interlocking system (see chapter 6).

For points and buttons, only one version of behaviour is suggested. However, for track relays, two kinds of behaviour have been introduced:

- *Random track relay behaviour* (see section 7.1.1.1), where a track section can become occupied or freed when the system is *idle*.
- *Ordered track relay behaviour* (see section 7.1.1.2), where track sections are occupied by trains only.

Two kind of patterns for *LTL* assertions that specify safety properties have been introduced:

- Patterns for checking the basic safety goals(section 7.2.1).
  They can be integrated with a scheme containing a transition system that

models internal behaviour and ordered track relay behaviour. These assertions check directly whether a collision or a derailing is possible in this transition system.

- Patterns for safety properties extracted from the train route table of the station (section 7.2.2).
  They can be integrated with a scheme containing a transition system that models internal behaviour, button behaviour, point behaviour, and either random or ordered track relay behaviour and used for model checking that the requirements expressed by the train route table are respected.

It has been concluded that the basic safety goals can be proved indirectly by proving the properties derived from the train route table. Therefore, one can verify the basic safety goals under random track relay behaviour when making the following assumptions:

- Only one external event can happen at a time.
- An external event can only occur when the system is idle.
- Trains respect the aspects displayed by signals.
- Points cannot be switched when they are occupied.
- Points cannot be switched when a train route containing them is locked.
- Only one button can be pushed at a time.
- All buttons have to be released before any other external transition can be taken.

When using the patterns provided for ordered track relay behaviour, the following extra assumption are made:

- Trains are not longer than two track sections.
- Trains can only change direction at some specific track sections.

One can try adding trains with different lengths to ordered track relay behaviour, but as mentioned in section 7.1.1.3, ordered track relay behaviour will not scale easily. Scaling it will lead to a high number of position variables and a larger state space.

It is possible to indirectly prove the basic safety goals using random track relay behaviour. This behaviour includes more states possible in the real world than ordered track relay behaviour and is therefore considered as being stronger when model checking safety properties. However, in chapter 8, we will still apply both kinds of patterns to a concrete railway station.

Chapter 8

Application: Stenstrup Station

Relay diagrams were modelled in chapters 5 and functions for generating the internal behaviour of an interlocking system were introduced in chapter 6. A model-oriented refinement of these will be given in chapter 9 and a Java implementation of it will be presented in chapter 10.

When having the Java program and the patterns for external behaviour and safety properties that were introduced in chapter 7, the method for verifying safety properties for a concrete station is complete.

The purpose of this chapter is to apply the method to a Danish railway station by using the Java program and by instantiating the patterns for external behaviour and safety properties. Stenstrup Station is a good choice for this experiment because it is a small station that contains all the physical objects introduced in chapter 2.

First section 8.1 will introduce Stenstrup Station. After that, in sections 8.2 and 8.3, we use the developed method to derive internal and external behaviours as well as safety properties for this station.

In section 8.4, the verification results of the *RSL-SAL* transition system representing Stenstrup will be presented and finally, in section 8.5, conclusions are made based on the results.

8.1 Introduction to Stenstrup Station

The layout (see figure 2.4, page 12) and train route table (see figure 2.6, page 18) of Stenstrup Station have already been introduced and explained in sections 2.2.4 and 2.4.2. This section will describe Stenstrup on a more technical level: which diagrams are used to represent the interlocking system of the station and how the relays found in these diagrams are linked to the physical objects they monitor.

8.1.1 The diagrams

The interlocking system of Stenstrup Station is represented by 18 diagrams:

- 3 for each train route couple (locking and releasing).
- 1 for the signal *A*.
- 1 for the signal *B*.
- 1 for the signals *E* and *F*.
- 1 for the signals *G* and *H*.
- 2 for auxiliary relays.

These diagrams have been transformed before translating them to XML:

- The transformed diagrams fulfil the assumptions written in the section 5.1.
- Some extra lamps and their associated relays are removed (the ones whose name ends by *.rs*). They are simply a copy of the main lamp and have exactly the same behaviour.
- The part of the diagrams taking care of the emergency release has been removed.
- The contacts ruled by external relays that are neither linked to points nor track sections are removed.

These external relays (and their associated contacts) could have been kept, but in that case, their behaviour should have been studied carefully to obtain the transition rules that define it. It was chosen to limit the scope of the project to external behaviour of only track relays, points, and buttons. Therefore, some auxiliary external relays (like the ones for level crossings *ovk82* and *ovk83*) are removed. One has to be careful not to remove contacts that are needed for ensuring the safety of trains at a station. Removing such contacts might lead to invalid assertions when checking confidence conditions and safety properties.

As explained in appendix C.4.1, one can find the original diagrams of Stenstrup as well as an explanation of the performed modifications on the attached CD.

These diagrams contain:

- 38 regular relays
- 8 steel core relays
- 4 buttons

8.1.2 Components necessary when specifying external behaviour and safety properties

When formulating the external behaviour, one needs to give specific behaviour to the external relays and the buttons. Also, when specifying safety properties, it is necessary to refer to some specific relays.

Table 8.1 shows how certain relays of the diagrams indicate the state of a physical object at the station or a train route. One can refer to this table later when specifying the external behaviour and the safety properties.

Table 8.2 shows the purpose of the buttons that are on the operator's panel.

8.2 The behaviour of Stenstrup station

8.2.1 The internal behaviour

The *RSL* scheme containing the internal behaviour and the associated confidence conditions was computed from an XML representation of the diagrams of

Type	Name of the relays	Physical objects/train route	Behaviour
Track relays	*t01*, *t02*, *t03*, *t04*, *a12*, *b12*	Track sections 01, 02, 03, 04, A12 and B12	If 01 is occupied then the relay *t01* is dropped, etc.
Point relays	*plus01*, *minus01*	Point 01	*plus01*/*minus01* is drawn only when point 01 is in plus/minus position.
Point relays	*plus02*, *minus02*	Point 02	*plus02*/*minus02* is drawn only when point 02 is in plus/minus position.
Signal relays	*xGreen*, *xRed*, *xYellow*	Signal X(=A,B,E,F,G or H)	If the signal X displays its *drive* aspect, then the relay *xGreen* is drawn
Signal relays	*xGreen2*	Signal X(=A or B)	If the signal X displays its *drive* aspect, then the relay *xGreen2* is drawn
Locking relays	*ia*, *iadub*	trains routes 2 or 3	If 2 or 3 is locked, the relay *ia* and *iadub* are dropped.
Locking relays	*ib*, *ibdub*	trains routes 5 or 6	If 5 or 6 is locked, the relay *ib* and *ibdub* are dropped.
Locking relays	*ua*, *uadub*	trains routes 7 or 8	If 7 or 8 is locked, the relay *ua* and *uadub* are dropped.
Locking relays	*ub*, *ubdub*	trains routes 9 or 10	If 9 or 10 is locked, the relay *ub* and *ubdub* is dropped.

Table 8.1: The role of the relays found in the diagrams of Stenstrup

Name	Related to train routes
b00406	7, 8
b00606	2, 3
b03106	5, 6
b03306	9, 10

Table 8.2: The role of the buttons of the operator's panel of Stenstrup: when a button is pushed, the locking process of one of the related train routes (depending on the position of the points) begins.

Stenstrup by the program introduced in chapter 10 and it can be found on the attached CD (see appendix C.4.4).

8.2.2 The external behaviour

Now that the internal behaviour of the interlocking system of Stentrup Station is described by a transition system, external behaviour has to be added, following the patterns introduced in section 7.1. There are three categories of external behaviour that need to be added: one for track relays, one for buttons, and one for points. This will be described in further details in the following sections.

The resulting transition systems can be found on the attached CD (see appendix C.4.2).

8.2.2.1 The track relays

As mentioned in section 7.1.1 there are two ways of modelling track relay behaviour: *random track relay behaviour* and *ordered track relay behaviour*. The following will explain how these are formulated for Stenstrup Station:

Random track relay behaviour The behaviour will be formulated by instantiating the patterns given in section 7.1.1.1, page 109.

At Stenstrup, there are 6 track sections: *A12*, *B12*, *01*, *02*, *03*, and *04*. When using a random track relay behaviour, each relay linked to these track sections will be ruled by one transition rule that can be taken each time the system is idle.

From Odense	From Svendborg	Both directions
a12_fwd	b12_back	t02_fwd_back
a12_t01_fwd	b12_t03_back	t04_fwd_back
t01_fwd	t03_back	
t01_t02_fwd	t03_t02_back	
t01_t04_fwd	t03_t04_back	
t02_t03_fwd	t02_t01_back	
t04_t03_fwd	t04_t01_back	
t03_fwd	t01_back	
t03_b12_fwd	t01_a12_back	
b12_fwd	a12_back	

Table 8.3: Position variables for Stenstrup

For example, as track section *A12* is linked to relay *a12*, the transition rule for changing the state of the relay is:

$[\,\text{randomA12}\,]\ {\sim}\text{idle} \rightarrow \text{idle}' = \textbf{false},\ \text{a12}' = {\sim}\text{a12}$

Ordered track relay behaviour The behaviour will be formulated by instantiating the patterns given in section 7.1.1.2, page 110.

Position variables are needed to represent the trains on the track. As previously explained, we assume that a train can at most occupy two track sections at a time. Due to this assumption, a train position will define a direction and the occupation of either a single track or two neighbour tracks.

The only exception is train positions for track sections *02* and *04*. When trains occupy these, they are allowed to change direction. E.g. they can arrive from Odense, move to track section *02*, stop, and leave the station towards Odense. Therefore, there is no need to have a direction for trains when they are on these track sections.

All the position variables necessary to describe an ordered track relay behaviour for Stenstrup can be found in table 8.3. All these variables are initialized to false because there is no train at the station in the *normal state* of the interlocking system. In order to allow trains to enter and to exit the station, transition rules are added. Examples of the formulated transition rules for train movements are:

- **For a train to enter the station:**

When entering from Odense, the train has to wait for signal *A* to become green. When it is green, track section *A12* becomes occupied (i.e. track relay *a12* is dropped) and the position variable *a12_fwd* is set to true.

```
[ insertTrainA ]
   idle ∧ aGreen →
   idle′ = **false**, a12_fwd′ = **true**, a12′ = **false**
```

There is a similar rule for entering the station from Svendborg.

- **For a train to occupy the next track section:**
  When a train is on *A12* and coming from Odense, the next step is to move to track section *01*. After the transition is taken, track section *01* will become occupied, *a12_fwd* will be set to false and *a12_t01_fwd* to true.

  ```
  [ a12Fwd ]
     idle ∧ a12_fwd →
     idle′ = false,
     a12_fwd′ = false,
     a12_t01_fwd′ = true,
     t01′ = false
  ```

- **For a train to leave a track section:**
  When a train is on *A12* and *01* and coming from Odense, the next step is to leave track section *A12*. After this transition, track section *A12* will become free, *a12_t01fwd* will be set to false and *t01_fwd* will be set to true.

  ```
  [ a12t01Fwd ]
     idle ∧ a12_t01_fwd →
     idle′ = false,
     a12_t01_fwd′ = false,
     t01_fwd′ = true,
     a12′ = true
  ```

- **For a train to cross a point:**
  Stenstrup is using the old point convention described in section 2.2.1.2. When a train is on a point, the next track section that will be occupied by the train will depend on the position of the point. If a train is on track section *01* coming from Odense (*t01_fwd* = true), there are two possible transition rules:

  – if point *01* is in the plus position

[t01plusFwd]
idle ∧ plus01 ∧ t01_fwd →
idle′ = **false**,
t01_fwd′ = **false**,
t01_t02_fwd′ = **true**,
t02′ = **false**

– if point *01* is in the minus position

[t01minusFwd]
idle ∧ minus01 ∧ t01_fwd →
idle′ = **false**,
t01_fwd′ = **false**,
t01_t04_fwd′ = **true**,
t04′ = **false**

8.2.2.2 The points

This section will instantiate the patterns for point behaviour given in section 7.1.2, page 118.

As previously mentioned, Stenstrup has two points, *01* and *02*. As seen in section 7.1.2, each point has three positions: plus, intermediate and minus. Two relays are linked to each point. The position of a point is detected by these relays. For example, point *01* is linked to relay *plus01* that is only drawn when the point is in the plus position and to relay *minus01* that is only drawn when the point is in the minus position. So point *01* is in:

- the plus position if *plus01* is up and *minus01* is dropped,
- the minus position if *plus01* is down and *minus01* is drawn,
- the intermediate position if both *plus01* and *minus01* are dropped.

A point can only be changed when no train route containing it is locked and when it is free (see section 7.1.2). For point *01*, three couples of train routes use it. Whether they are locked is shown by the state of three steel core relays *ia*, *ib* and *ua*. The occupation of the point is shown by relay *t01*. The interlocking system has to be in an idle state when changing a point. The transition rules for point *01* will be on the following form:

idle ∧ ia ∧ ib ∧ ua ∧ t01 ∧ ... → ...

For each point there are four transition rules for switching it from intermediate to plus, intermediate to minus, plus to intermediate, and minus to intermediate. The state of the point is part of the transition guard and, when the transition is taken, all the point relays are updated to their new value. The following is an example of the transition rule for switching point *01* from its intermediate position to its plus position:

[intermediateToPlus1]
$\quad$ idle $\wedge$ ia $\wedge$ ib $\wedge$ ua $\wedge$ t01 $\wedge$ $\sim$plus01 $\wedge$ $\sim$minus01 $\quad\rightarrow$
$\quad$ idle$'$ = **false**, plus01$'$ = **true**

8.2.2.3 The buttons

This section will instantiate the patterns for button behaviour given in section 7.1.2, page 118.

At Stenstrup, there are 4 buttons that initiate the locking of train routes. As decided in section 7.1.3, only one button can be pushed at a time, and once all the internal consequences of pushing a button have occurred, the button is released. Then, when all the consequences of this release are finished, the system is idle.

The following has to be added to the transition system in order to include the proper button behaviour:

- one transition rule for each button to push it. Here is the transition rule for the button *b00406*:

  [pushButton_b00406]
  idle $\rightarrow$ idle$'$ = **false**, b00406$'$ = **true**

- the *idle* transition rule is changed to:

  [setIdle]
  $\qquad$ gIdle $\rightarrow$
  $\qquad$ idle$'$ = $\sim$(b00406 $\vee$ b00606 $\vee$ b03106 $\vee$ b03306),
  $\qquad$ b00406$'$ = **false**,
  $\qquad$ b00606$'$ = **false**,
  $\qquad$ b03106$'$ = **false**,
  $\qquad$ b03306$'$ = **false**

where *gIdle* is the original guard of the *setIdle* transition rule auto-generated by the Java program.

8.3 Safety properties

Now that the internal and external behaviours of Stenstrup are formalized by a complete transition system, it is time to introduce the safety properties that have to be respected by the transition system. As explained in section 7.2, there are two categories of properties. The first directly expresses the basic safety goals using ordered track relay behaviour. The second expresses the basic safety goals by proving the properties of the train route table. The second category can be used for both kinds of track relay behaviour.

As explained in appendix C.4.3, all the properties can be found on the attached CD.

8.3.1 Basic safety properties

The following sections give examples of how the patterns introduced in section 7.2.1 can be applied to Stenstrup station, when assuming an ordered track relay behaviour.

8.3.1.1 No collision

The pattern for specifying that no collision takes place (see section 7.2.1.1, page 125) will now be instantiated.

To check that there is no possible collision, one can use the position variables introduced for the behaviour of the track relays. The principle of this is explained in section 7.2.1.1: a track section can only be occupied by one train at a time, i.e. for track *01*, at most one of the position variables *a12_t01_fwd*, *t01_fwd*, *t01_t02_fwd*, *t01_t04_fwd*, *t02_t01_back*, *t04_t01_back*, *t01_back* and *t01_a12_back* can be true in one state. The corresponding assertion is (using the function v defined in section 7.2.1.1):

ltl_assertion
[no_collision_t01] InterlockingSystem $\vdash$

G(
  v(a12_t01_fwd) + v(t01_fwd) + v(t01_t02_fwd) +
  v(t01_t04_fwd) + v(t02_t01_back)+ v(t04_t01_back) +
  v(t01_back) + v(t01_a12_back) $\leq$ 1),

8.3.1.2 No derailing

The pattern for specifying that no collision takes place (see section 7.2.1.2, page 127) will now be instantiated.

To verify that derailing does not take place on point *01*, one has to check that:

- when the point is occupied, it is not in its intermediate position. The occupation of the point is monitored by track relay *t01*.

  ltl_assertion
  [no_derailing_t01] InterlockingSystem $\vdash$
    G($\sim$t01 $\Rightarrow$ plus01 $\vee$ minus01),

- when the point and track section *04* are occupied by a train, the point is in the *minus* position.

  ltl_assertion
  [no_derailing_t01t04] InterlockingSystem $\vdash$
    G(
      (v(t01_t04_fwd) + v(t04_t01_back) $\geq$ 1) $\Rightarrow$
       minus01),

- when the point and track section *02* are occupied by a train, the point is in the *plus* position.

  ltl_assertion
  [no_derailing_t01t02] InterlockingSystem $\vdash$
    G(
      (v(t01_t02_fwd) + v(t02_t01_back) $\geq$ 1) $\Rightarrow$
       plus01),

8.3.2 Safety properties extracted from the train route table

The second approach to verifying that the basic safety goals are implemented by the interlocking system is to use the requirements of the train route table specific to Stenstrup. The following sections give examples of how the patterns introduced in section 7.2.2 can be applied to Stenstrup Station. This will indirectly ensure that there is no possibility of collision or derailing and can be done under the two behaviours introduced in section 7.1.1, because the resulting assertions will not depend on train positions.

8.3.2.1 Locking of conflicting routes

The pattern for specifying that conflicting train routes are not locked at the same time (see section 7.2.2.1, page 129) will now be instantiated.

In the train route table shown in figure 2.6, page 18, the conflicting train routes are indicated one by one. The locking of train routes *2* and *3* is indicated by the steel core relay *iadub*, the locking of *5* and *6* by *ibdub*, the locking of *7* and *8* by *uadub*, and the locking of *9* and *10* by *ubdub*. For instance, if we look at train routes *2* and *3*, they are conflicting with train routes *5*,*6*,*7*, and *8*. That means that *iadub* should not be dropped at the same time as *ibdub* or *uadub*. *2* is also in conflict with *10*, and *3* is in conflict with *9*. However, *2* and *10* cannot be locked at the same time because they require different positions for point *02*. The same goes for *3* and *9*, and for *2* and *3*.

The following assertion checks that if either train route *2* or *3* is locked, none of their conflicting train routes are locked.

ltl_assertion
[conflict_locking_ia] InterlockingSystem $\vdash$
    G($\sim$iadub $\Rightarrow$ ibdub $\wedge$ uadub)

8.3.2.2 Signals

The pattern for specifying that the signals behave as specified in the train route table (see section 7.2.2.2, page 130) will now be instantiated.

There is one assertion for each signal that is extracted from the train route table. The following paragraph specifies the assertion for signal A.

Signal A must only display a *drive* aspect (indicated by a green light) if either train route 2 or 3 is locked. The conditions from train route *2* are: signal F is red, signal G is red or green, both points are in the plus position, and track sections *A12*, *01*, *02*, *03*, and *B12* are free. These requirements are specified in columns "Signaler", "Sporskifter", and "Sporisolationer" of the train route table. The ones for train route *3* can be deduced in the same way. Finally, A must not be green if neither train route *2* nor train route *3* is locked. So the assertion for signal A is:

ltl_assertion
[signalA] InterlockingSystem $\vdash$
G(idle $\wedge$ aGreen $\Rightarrow$
    $\sim$ia $\wedge$ a12 $\wedge$ t01 $\wedge$ t03 $\wedge$ b12 $\wedge$
    ((t02 $\wedge$ plus01 $\wedge$ plus02 $\wedge$ fRed $\wedge$ (gRed $\vee$ gGreen))$\vee$
    (t04 $\wedge$ minus01 $\wedge$ minus02 $\wedge$ eRed $\wedge$ (hRed $\vee$ hGreen)))),

i.e. when signal A is green and there are no more possible internal transitions, train route *2* or *3* must be locked and the conditions concerning points, signals, and track sections for the locked train route must be valid.

8.3.2.3 Stop Fall

The pattern for specifying that the a signal stops displaying a *drive* aspect when a train is on a specific track section (see section 7.2.2.3, page 130) will now be instantiated.

The train route table specifies when each signal has to change from its *drive* aspect to its *stop* aspect. There is one assertion for each signal.

Two train routes, *5* and *6*, use signal B as an entrance signal, i.e. when signal B displays a *drive* aspect, a train is allowed to enter one of the routes. Both express the same condition: signal B must not display a *drive* aspect when track section *B12* is occupied.

ltl_assertion
  [stopfallB]
    InterlockingSystem $\vdash$

$$G(\text{idle} \wedge \sim\text{b12} \Rightarrow \sim\text{bGreen}),$$

i.e. if track section *B12* is occupied, in the next state, either the system still has some possible internal transitions or signal *B* has stopped displaying a *drive* aspect.

8.3.2.4 Point positions

The pattern for specifying that a point is in its required position when a train route is locked (see section 7.2.2.4, page 131) will now be instantiated.

When locking a train route, the position of the points has to be checked. For Stenstrup Station, when locking an entrance train route from Odense (shown by the steel core relay *ia*), there are two possible combinations of positions: points *01* and *02* should be in the plus position (shown by relays *plus01* and *plus02*), or they should be in the minus position (shown by relays *minus01* and *minus02*).

ltl_assertion
[locking_points_ia] InterlockingSystem ⊢
  G(∼ia ⇒
    (plus01 ∧ plus02) ∨ (minus01 ∧ minus02)),

i.e. if an entrance train route from Odense is locked, the points are either both in their plus position or both in their minus position.

8.3.2.5 Train route release

The pattern for specifying that a train route is not released too early (see section 7.2.2.5, page 131) will now be instantiated.

For train route *5* and *6* (monitored by steel core relay *ib*), the 2 assertions expressing that the train routes are not released too early are:

ltl_assertion
[releaseIB1] InterlockingSystem ⊢
  G(ib ∧ X(∼ib) ⇒

X(W(∼ib, ∼ib ∧ ∼t03 ∧ ((t02 ∧ plus01) ∨ (t04 ∧ minus01))))),
[releaseIB2] InterlockingSystem ⊢
G(∼ib ∧ X(ib) ⇒ (t03 ∧ ((∼t02 ∧ plus01) ∨ (∼t04 ∧ minus01))))

i.e.

- for *releaseIB1*: if *ib* is drawn in the current state and dropped in the next state, *ib* will stay dropped until the first condition given by the train route table is fulfilled (*03* is occupied and either track section *02* is free and point *02* is in the *plus* position, or track section *04* is free and point *02* is in the *minus* position)
- for *releaseIB2*: if *ib* is dropped in one state and drawn in the next one, the second condition given by the train route table is fulfilled (*03* is free and either track section *02* is occupied and point *02* is in the *plus* position, or track section *04* is occupied and point *02* is in the *minus* position).

8.4 Results for Stenstrup

Now that the transition system is complete and all the safety properties are defined, the properties can be verified. The results will be presented in the following sections. As it was written before, there are two different transition systems: one using *ordered track relay behaviour* and one using *random track relay behaviour.*

8.4.1 Test setup

The properties were verified using the following configuration:

Software:

- **Operating system:** Ubuntu 8.04, Kernel Linux 2.6.24-19-generic
- **RSLTC version:** 2.5-1
- **SAL version:** 3.0

Hardware:

- **CPU:** Intel Core 2 Duo E6850, 3.00 GHz, 4 MB L2 cache
- **Motherboard:** Asus G33, mATX, Socket 775, P5K-VM
- **RAM:** Corsair XMS Xtreme, 4GB, PC6400, DDR2 TWIN2X4-6400C5

(SAL 3.0 only takes advantage of a single CPU kernel at a time.)

8.4.2 Results for ordered track relay behaviour

The file used for this verification can be found on the attached CD (as explained in appendix C.4.4). It also contains the train position variables as defined in section 8.2.2.1. There are three types of assertions:

- 102 confidence conditions linked to the internal behaviour.
  All the assertions are valid, i.e. there is no concurrency issue and no problems related to steel core relays.
  Verification time: 1 hour 7 minutes and 19 seconds.
  Approximated memory usage: 249 MB.
- 12 assertions checking directly that there is no possible collision and no possible derailing.
  All the assertions are valid.
  Verification time: 20 seconds.
  Approximated memory usage: 74 MB.
- 28 assertions extracted from Stenstrup's train route table.
  All the assertions are valid, the train route table is respected.
  Verification time: 3 minutes and 28 seconds.
  Approximated memory usage: 116 MB.

8.4.3 Results for random track relay behaviour

The file used for this verification can be found on the attached CD (as explained in appendix C.4.4). It contains the random track relay behaviour as defined in section 8.2.2.1. Under this behaviour, only two categories of assertions are checked:

- 102 confidence conditions linked to the internal behaviour.
  All the assertions are valid, i.e. there is no concurrency issue and no

problems related to steel core relays.
Verification time: 26 minutes and 57 seconds.
Approximated memory usage: 150 MB.

- 28 assertions extracted from Stenstrup's train route table.
  All the assertions are valid, i.e. the train route table is respected.
  Verification time: 1 minutes and 46 seconds.
  Approximated memory usage: 117 MB.

8.5 Conclusion

In this chapter all the principles seen in chapters 6 and 7 were put in practice for a Danish station, Stenstrup. The process starts from the diagrams, the track layout, the operator's panel and the train route table of the station provided by Banedanmark. From these, one can extract the internal behaviour from the diagrams and create the two possible external behaviours from the track layout of the station. Finally, one can write the different assertions that will be model-checked.

All the assertions of the resulting transitions systems are valid, i.e. that train traffic at Stenstrup Station can be considered as being safe under the assumptions written in section 7.3.

This chapter has demonstrated that the principles introduced in the previous chapters are applicable to a concrete station. Stenstrup is a regular station that contains all the elements that one can find at any other Danish station. The only possible limitation of the method is the state space. One cannot know whether the memory usage becomes too high when applying the method to larger stations.

Another conclusion is that random track relay behaviour is not only the most general compared to ordered track relay behaviour. It is also more efficient in terms of running time and the memory usage is lower when verifying confidence conditions. If one introduces trains with different lengths under ordered track relay behaviour, the state space (implying a higher memory usage) and verification time would increase for this kind of behaviour.

With this in mind and when knowing that the properties derived from the train route table together covers the basic safety goals, we conclude that random track relay behaviour is preferable when model-checking safety properties.

Chapter 9

Concrete model of relay diagrams

As explained in chapter 4, relay diagrams and the conversion of these to an *RSL-SAL* transition system and its confidence conditions are modelled on different levels.

Chapter 5 introduced an abstract model (i.e. algebraic and property-oriented) for relay diagrams. Furthermore, chapter 6 introduced generator functions for transforming abstract syntax for relay diagrams to abstract syntaxes for *RSL-SAL* transition systems and *LTL* assertions. The generated transition system models the behaviour of the interlocking system. For the same transition system, *LTL* assertions define confidence conditions that can be used for verifying that it specifies the complete behaviour of the interlocking system and respects some desired properties that are not related to safety.

The purpose of this chapter is to introduce a concrete (model-oriented, but still generic) version of the models and generator functions specified in chapters 5 and 6. The goal is to give a model where the functionalities are specified on a level where they can be translated to another programming language by hand.

When the specifications presented by this chapter are translated to another language, the resulting program must be able to read diagrams of an interlocking system in a certain format and afterwards be able to make a file containing an *RSL* scheme with the transition system that models the behaviour of the interlocking system and the confidence conditions. How the parsing of the diagrams works and how the creation of the final *RSL* scheme is done will not be considered in this chapter. However, the *RSL* schemes can still be made in a way such that it is easy to generate a *StaticInterlockingSystem* based on a text file.

Using a *StaticInterlockingSystem* that is specified such that it is close to the layout of a text file might not be efficient when doing computation. Therefore, it seems difficult to make a model that fits both the need for the computation of transition systems and confidence conditions and the need for being close to the format of a text file. Instead of making a compromise, it is chosen to introduce two different models and a translation from one of the models to the other.

The first model will be given by the *StaticInterlockingSystemL* and *DiagramsL* schemes. It will be based on lists and will therefore be close to what one can specify in a text file.

The second model will be given by the *StaticInterlockingSystem* and *Diagrams* schemes and use map-based computation.

The overall strategy of the concrete implementation, including a conversion from an instance of list-based model to an instance of map-based model, can be seen in figure 9.1.

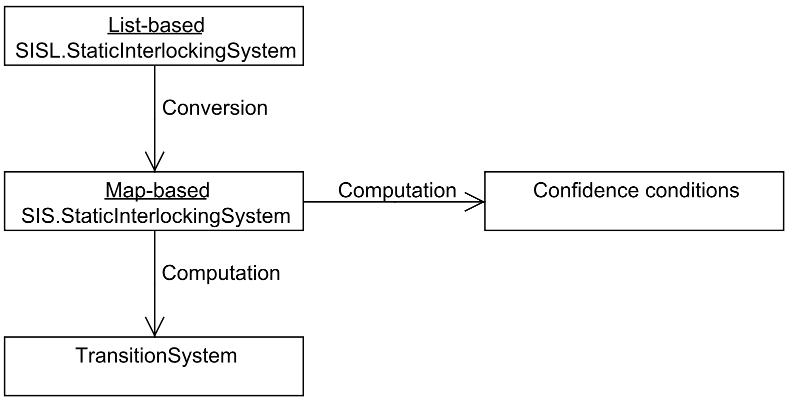


Figure 9.1: The overall strategy for the concrete computation of transition systems and confidence conditions.

Section 9.1 will introduce common types that are used by both the list-based model and the map-based model. They are used to contain the information used for implementing the observer functions.

Section 9.2 will introduce a model based on lists, section 9.3 will introduce a model based on maps, and section 9.4 will introduce a conversion from the list-based model to the map-based model.

Section 9.5 will introduce a concrete way of computing the conditions (introduced in section 6.3) for when a relay can be drawn, dropped, etc. Especially, it will describe how to find all the paths through a component inside a *Diagram*.

Section 9.6 will introduce a concrete version of the function specified in section 6.6 for computing the behaviour of a *StaticInterlockingSystem* specified in section 9.3. Afterwards, section 9.7 will introduce a concrete implementation of the function given in section 6.7 for generating confidence conditions of a transition system.

9.1 Types

Chapter 5 introduced an abstract syntax for relay diagrams. The components were identified by type *Id*, but concrete representations of components were not introduced. This section will explain the necessary changes of the *Types* scheme of the abstract model (see appendix A.1, page 209) when taking a step towards a concrete model. A complete version of the concrete *Types* scheme presented in this chapter can be found in appendix B.1, page 233.

In the concrete model, an *Id* will have the type *Text*:

```
type
  Id = Text
```

In that way, the names of the components from the diagrams can directly be used as identifiers.

The observer functions of the concrete models introduced in the following sections should be concrete. To store the information needed by the concrete versions of the observer functions, it is decided to introduce concrete types for each kind of component that is represented by the abstract model.

Short record types *RegularRelay*, *SteelRelay*, *ExternalRelay*, *Button*, *Pole*, *Contact*, and *Junction* are introduced for that purpose. An example of such a type is:

```
type
   RegularRelay ::
      getId : Id
      getInitState : State
```

The destructor *getId* gives the *Id* of the given regular relay and *getInitState* gives its initial state.

Another example of the introduced types is *Pole*:

```
type
   Pole :: getId : Id
```

The destructor *getId* gives the *Id* of the given pole.

For convenience, a union type for representing a component inside a *Diagram* is introduced:

```
type
   Component =
      Pole | Junction | RegularRelay | SteelRelay |
      Button | Contact
```

As external relays are not present in a *Diagram*, *ExternalRelay* is not part of the union type.

For convenience, a function named *idFromComponent* is introduced to extract the *Id* of a given component:

```
value
   idFromComponent : Component → Id
   idFromComponent(c) ≡ ...
```

The introduced type *Pole* does not store information on whether a given *Pole* is positive or negative. Therefore, the following type is introduced to contain a positive pole and a negative pole:

```
type
    Poles ::
        plus : Pole
        minus : Pole
```

In order to represent relations between components inside a *Diagram*, the following type is introduced:

```
type
    Edge = Id × Id
```

9.2 List-based model

As previously explained, the idea behind introducing a list-based concrete version of the abstract model introduced in chapter 5 is to make a model close to the format of a text file. Later on, it will be possible to parse data from a text file and use this data for making an instance of the list-based model.

Scheme *DiagramsL* (see appendix B.2, page 236) containing a list-based version of a *Diagram* is therefore introduced. The concrete version *DiagramL* is on the following form:

```
scheme DiagramsL =
    class
        type
            /∗ the components of a diagram ∗/
            Components ::
                getPoles : T.Poles
                getContacts : T.Contact*
                getButtons : T.Button*
                getRegularRelays : T.RegularRelay*
                getSteelRelays : T.SteelRelay*
                getJunctions : T.Junction*,
            /∗ A diagram corresponding to one circuit ∗/
```

```
      Diagram ::
         /∗ All the components in the diagram ∗/
         getComponents : Components
         /∗ All the edges between the components in the
         diagram∗/
         getEdges : T.Edge*

   value
      /∗ observer functions and auxiliary functions ∗/
      ...
end
```

For each component type inside a *Diagram* (except poles), there is a list containing all the components of the given type. For the poles, a *Diagram* contains one positive pole and one negative pole.

The list of *Edges* in a *Diagram* specifies the neighbour relation between components such that if there is an *Edge* between two components, they are neighbours.

In order to implement the *Diagrams* scheme of the abstract model, every observer function of it is implemented using the above data structure. An example of such a function is the one telling whether a given *Id* is a positive pole inside a Diagram:

```
value
   isPlus : T.Id × Diagram → Bool
   isPlus(id, d) ≡
      T.getId(T.plus(getPoles(getComponents(d)))) = id
```

The well-formed function of the abstract *Diagram* scheme given in appendix A.2, page 210 was axiomaticly defined in the following way:

```
value
   isWfDiagram : Diagram → Bool
axiom
   ∀ d : Diagram • isWfDiagram(d) ⇒ c1
```

Where *c1* consists of several constraints on the *Diagrams*. This leads to making a concrete well-formed function on the following form:

```
value
```

```
    isWfDiagram : Diagram → **Bool**
    isWfDiagram(d) ≡ c1 ∧ c2
```

Where c2 expresses the following constraints on a list-based diagram:

- The list of *Edges* does not contain duplicates. If (id1,id2) is an *Edge*, (id2,id1) is not an *Edge*.
- An *Edge* can only contain ids of components that are defined in the *Diagram*. If (id1,id2) is an *Edge* of one diagram, id1 and id2 are the ids of two components of this *Diagram*.
- For each list of components, the following holds: the list does not contain duplicates and two components inside a list cannot have the same *Id*.

To represent a list-based *StaticInterlockingSystem*, scheme *StaticInterlockingSystemL* is introduced. It contains the following type for a *StaticInterlockingSystem* where *DL* is an object of *DiagramL*:

```
**type**
    StaticInterlockingSystem ::
        getDiagrams : DL.Diagram*
        getExternal : T.ExternalRelay*
```

Again, concrete versions of the observer functions are introduced. The *StaticInterlockingSystem* scheme of the abstract model was defined in the following way:

```
**value**
    isWfStaticInterlockingSystem : Diagram → **Bool**
**axiom**
    ∀ sis : StaticInterlockingSystem •
        isWfStaticInterlockingSystem(sis) ⇒ c1
```

Where *c1* expresses constraints on a *StaticInterlockingSystem.* This leads to defining a concrete well-formed function for the list-based *StaticInterlockingSystem* in the following way:

```
**value**
    isWfStaticInterlockingSystem(sis) ≡ c1 ∧ c2
```

c2 expresses the following additional constraints:

- The *Diagram-list* does not contain duplicates.
- The *ExternalRelay-list* does not contain duplicates and two external relays do not have the same *Id*.

The *DiagramsL* and *StaticInterlockingSystemL* schemes statically implement all the functionality of the abstract model. All the types and functions from the abstract model are present and the well-formed constraints from the abstract model are also enforced on the concrete model. One could also prove that the new schemes implement *Diagrams* and *StaticInterlockingSystem* of the abstract model respectively. However, doing so is considered as being beyond the scope of this project.

9.3 Map-based model

This section will introduce a map-based implementation of the *StaticInterlockingSystem* and *Diagrams* schemes(see appendix A.2, page 210) from the abstract model in chapter 5. The purpose is to make a model that is more efficient when performing computation than the list-based model introduced in section 9.2.

A scheme called *Diagrams* (see appendix B.4, page 247) is introduced in order to implement the functionality of the *Diagrams* scheme of the abstract model. In the concrete model, scheme *Diagrams* is on the following form:

```
scheme Diagrams =
  class
    type
      /* poles can be plus or minus */
      PoleType == plus | minus,
      Diagram ::
        getComponentMap : T.Id $\overrightarrow{m}$ T.Component
        getEdges : T.Edge-set
        /* Information on the type of the poles */
        getPoleType : T.Id $\overrightarrow{m}$ PoleType

    value
      /* observer functions and auxiliary functions */
      ...
    end
```

A *Diagram* has one map for representing all the components. The relation between components inside a *Diagram* are represented as a set of *Edges*.

As additional information, a map is defined for storing the type of each *Pole*, using *PoleType*.

Like in *DiagramsL*, the observer functions of the *Diagram* scheme in the abstract model are implemented. The well-formed function for a *Diagram* contains the same constraints as in the underspecified axiom of the abstract model and adds further constraints:

- Each *Id* in the domain of the component map is mapped to a component with the same *Id*.
- If an *Edge* (id1,id2) is defined, (id2,id1) is not defined in the *Edge*-set.
- An *Edge* can only contain ids of components that are defined in the diagram. If (id1,id2) is an *Edge* of one diagram, id1 and id2 are the ids of two components of this *Diagram*.
- Each *Pole* defined in the range of the component map, has its *Id* defined in the domain of the map for storing the pole types.

Scheme *StaticInterlockingSystem* is added to the concrete map-based model for implementing the *StaticInterlockingSystem* of the abstract model. The following type is defined for representing a *StaticInterlockingSystem*:

```
type
    StaticInterlockingSystem ::
        getDiagrams : D.Diagram-set
        getExternal : T.Id →m T.ExternalRelay
```

Besides having a set of *Diagrams*, a map is defined inside the *StaticInterlockingSystem* to represent external relays. Again, the observer functions from the concrete model are implemented in a concrete manner. The well-formed function is almost the same as in the underspecified axiom in the abstract model. The only extra constraint is:

- The *Ids* inside the domain of the map containing the external relays are mapped to *ExternalRelays* with the same name.

The schemes introduced in this section have statically the same functionality as the ones defined in the abstract model and the constraints defined in the well-formed functions of the abstract model are also defined by the well-formed functions in the concrete schemes. However, it is considered as being beyond the scope of this project to prove that the schemes implement the ones from the abstract model.

9.4 Conversion from list-based model to map-based model

As previously explained, it should be possible to convert an instance of the list-based model defined in section 9.2 of a *StaticInterlockingSystem* to an instance of the map-based model defined in section 9.3.

To do the conversion, scheme *StaticInterlockingSystemConversion* (see appendix B.6, page 256) is introduced. The scheme defines a function with the following signature where *SISL* and *SIS* are objects of the concrete models *StaticInterlockingSystemL* and *StaticInterlockingSystem* respectively:

```
value
  convertStaticInterlockingSystem :
    SISL.StaticInterlockingSystem $\overset{\sim}{\to}$ SIS.StaticInterlockingSystem
      convertStaticInterlockingSystem(sisl) ≡ ...
    pre SISL.isWfStaticInterlockingSystem(sisl)
```

The function is explicitly defined and uses the concrete auxiliary function *convertDiagram* for converting a *DiagramL* (on the list form) to a *Diagram* (on the map form). Both functions require that their arguments are well-formed.

After the conversion, the observer functions of the map-based model will behave exactly as they did in the list-based model. One could prove this, but establishing the proof is beyond the scope of this project. An example of a relation between list-based *Diagrams* and the map-based *Diagrams* resulting from the conversion is:

```
∀ d : DL.Diagram, id : Id • DL.isWfDiagram(d) ⇒
  DL.isPlus(id, d) ≡ D.isPlus(id, convertDiagram(d))
```

(*DL* and *D* are, respectively, objects of the list-based *Diagrams* scheme and the map-based *Diagrams* scheme).

The expression states that, for each list-based *Diagram*, if the list-based diagram is well-formed, the list-based version of *isPlus* applied to the *Diagram* and an arbitrary *Id* behaves like the map-based version of *isPlus* applied to the converted *Diagram* and the same *Id*.

In general, if a well-formed, list-based *StaticInterlockingSystem* is converted to a map-based *StaticInterlockingSystem*, the map-based representation will be well-formed:

```
∀ sis : SISL.StaticInterlockingSystem •
   SISL.isWfStaticInterlockingSystem(sis) ⇒
      SIS.isWfStaticInterlockingSystem(convertStaticInterlockingSystem(sis))
```

This is illustrated in figure 9.2. Figure 9.3 illustrates that no conversion takes place if the list-based *StaticInterlockingSystem* is not well-formed.

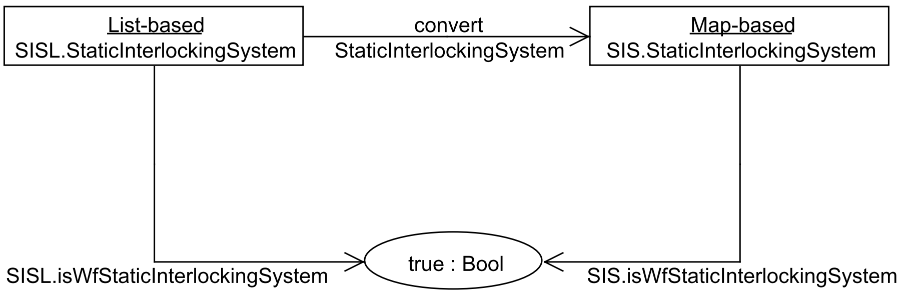

List-based
SISL.StaticInterlockingSystem
convert
StaticInterlockingSystem
Map-based
SIS.StaticInterlockingSystem
SISL.isWfStaticInterlockingSystem
true : Bool
SIS.isWfStaticInterlockingSystem

Figure 9.2: The list-based *StaticInterlockingSystem* is well-formed. After the conversion, the generated map-based *StaticInterlockingSystem* is also well-formed.

One could make a similar conversion from a map-based implementation to a list-based implementation. However, maps do not specify any order. Therefore, if one has a list-based model, converts it to a map-based model, and then converts it back again, the lists will not necessarily have the same order as before.

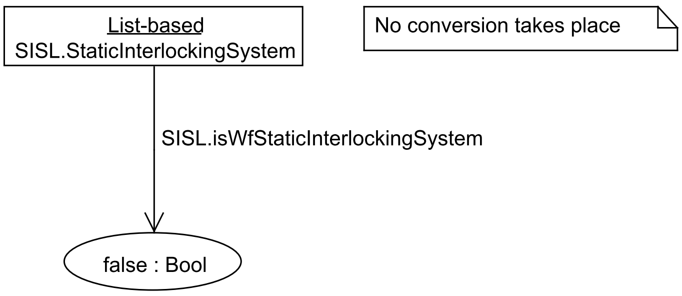

List-based
SISL.StaticInterlockingSystem
No conversion takes place
SISL.isWfStaticInterlockingSystem
false : Bool

Figure 9.3: The list-based *StaticInterlockingSystem* is not well-formed. The precondition of the conversion function is false and no conversion takes place.

9.5 Concrete condition- and pathfinding

The purpose of this section is to make a concrete model of the *Conditionfinding* scheme for the map-based implementation of a *StaticInterlockingSystem*. Actually, the *Conditionfinding* scheme of the abstract model (see appendix A.5, page 219) is concrete enough for being reused in the concrete model. However, the abstract *Pathfinding* scheme (see appendix A.4, page 218) used by it is not concrete enough because it contains the following function definition:

value
  allPathsFor : T.Id-**set** $\times$ D.Diagram $\xrightarrow{\sim}$ Path-**set**
  allPathsFor(ids, d) $\equiv$
    {p | p : Path • isWfPath(p, d) $\wedge$ isPathFor(p, ids)}
  pre D.isWfDiagram(d) $\wedge$ ids $\subseteq$ D.allIds(d)

The problem is that the function does not specify how to compute the paths. Therefore, we need to introduce functions to do this task.

In the concrete model of *Pathfinding* (given in appendix B.7, page 258), the above function is replaced with the following one:

value
  allPathsFor : T.Id-**set** $\times$ D.Diagram $\xrightarrow{\sim}$ Path-**set**
  allPathsFor(ids, d) $\equiv$

```
    {p | p : Path •
    p ∈ makePathsBetweenPoles(d) ∧
    isPathFor(p, ids)}
  pre D.isWfDiagram(d) ∧ ids ⊆ D.allIds(d)
```

The function *makePathsBetweenPoles* computes every simple path in the *Diagram d* starting with the positive pole and ending with the negative pole.

The following algorithm (presented using pseudo-code) was inspired by an algorithm presented by Skiena in [15]. The backbone of the implemented algorithm is two mutual recursive functions, *makePathsBetweenComponents* and *extendPath*.

```
makePathsBetweenComponents : D.Diagram $\xrightarrow{\sim}$ Path-set
makePathsBetweenComponents(endComponent, currentPath, d)
  if(currentPath(len currentPath) = endComponent)
        then
           {currentPath}
        else
             extendPath(endComponent,
                neighbours of currentPath(len currentPath), currentPath, d)
        end
   end
```

```
extendPath : T.Id × T.Id-set × Path × D.Diagram $\xrightarrow{\sim}$ Path-set
extendPath(endComponent, neighboursToBeVisited, currentPath, d)
  if neighboursToBeVisited = {}
     then {}
     else
        let
           nextComponent = an element of neighboursToBeVisited
           nextPath = currentPath ⌢ ⟨ nextComponent ⟩
        in
           if(nextPath is legal)
              then
                 makePathsBetweenComponents(endComponent, nextPath, d)
              else {}
           end
           ∪
           extendPath(endComponent,
              neighboursToBeVisited$\setminus$\{nextComponent\}, currentPath, d)
        end
  end
```

makePathsBetweenPoles initiates the process by calling *makePathsBetweenComponents* in the following way:

```
makePathsBetweenPoles : D.Diagram $\xrightarrow{\sim}$ Path-set
makePathsBetweenPoles(d) $\equiv$
    makePathsBetweenComponents(negative pole in d, $\langle$positive pole in d$\rangle$, d)
pre D.isWfDiagram(d)
```

The arguments are given to *makePathsBetweenComponents* such that:

- *endComponent* equals the negative pole of the diagram d.
- *currentPath* equals a list containing only the positive pole of the diagram d.
- *d* equals the diagram containing the two poles.

makePathsBetweenComponents will then give the set of paths from the positive pole to the negative pole inside *d*.

makePathsBetweenComponents works as follows:

- If the last component of *currentPath* is *endComponent*, it will return the set containing only *currentPath*.
- Otherwise, it will compute the *Id* set representing the neighbours of the last *Id* in *currentPath* and apply it to *extendPath* together with *endComponent*, *currentPath*, and *d*. In this case, *makePathsBetweenComponents* will return the result given by *extendPath*.

extendPath works as follows:

- If the set *neighboursToBeVisited* is empty, it will return the empty set.
- Otherwise, it will take a random *Id* from the set *neighboursToBeVisited*. If *currentPath* extended with the chosen neighbour is a legal path, the result from *makePathsBetweenComponents* applied to *currentPath* extended

with the random element will be included in the result from *extendPath*. *extendPath* will then make a recursive call to itself with the random element removed from the set of *neighboursToBeVisited*. By adding a recursive call to the result, *extendPath* will give every legal extension of *currentPath* with a single element from the set *neighboursToBeVisited*.

9.6 Concrete generation of a transition system

The purpose of this section is to introduce a concrete version of the *makeBehaviouralSemantics* function given in scheme *StaticInterlockingSystemToTransitionSystem* of the abstract model (see in appendix A.7, page 224).

The concrete version of scheme *StaticInterlockingSystemToTransitionSystem* can be found in appendix B.10, page 266. The scheme will be responsible for converting a map-based *StaticInterlockingSystem* to abstract syntax of a *TransitionSystem*. The abstract syntax of an *RSL-SAL TransitionSystem* can be found in scheme *TransitionSystem* (see appendix B.9, page 265) and is identical with the one defined in the abstract model.

The following function is made for replacing the implicit *makeBehaviouralSemantics* function from the abstract model:

```
value
  makeBehaviouralSemantics :
    SIS.StaticInterlockingSystem ~→ TS.TransitionSystem
  makeBehaviouralSemantics(sis) ≡
    TS.mk_TransitionSystem(makeState(sis), makeTransitionRules(sis))
  pre SIS.isWfStaticInterlockingSystem(sis)
```

The function *makeState* is made such that it exactly gives a state corresponding to what is defined in the implicit version of *makeBehaviouralSemantics*. In the same way, the function *makeTransitionRules* computes the transition rule such that the post-condition from the abstract model is fulfilled.

Proving that the post-condition from the abstract model holds for the new model is, however, considered as being beyond the scope of this project.

9.7 Concrete generation of confidence conditions

As for scheme *StaticInterlockingSystemToTransitionSystem*, a concrete version of *StaticInterlockingSystemToConfidenceConditions* from the abstract model (that is given in appendix A.8, page 230) is introduced.

The new version of scheme *StaticInterlockingSystemToConfidenceConditions* (see appendix B.11, page 272) contains a concrete version of the function *makeConfidenceConditions*. *makeConfidenceConditions* makes a list of confidence conditions such that the post-condition of the same method in the abstract model is fulfilled. In the abstract model, the post-condition was underspecified in the sense that it defined that some conditions needed to be included, but it did not forbid adding other conditions than the specified ones. On the contrary, the concrete version is made in a way such that all the conditions needed to fulfil the post-conditions will be included in the result, but no other conditions will be included.

makeConfidenceConditions uses auxiliary functions together with functions defined in *Conditionfinding*.

Proving that the post-condition from the abstract model holds for the new model is considered as being beyond the scope of this project.

Chapter 10

Java design and implementation

As explained in the method overview given in chapter 4, the concrete model presented in chapter 9 is to be converted to a Java implementation. The implementation will then work as a tool for auto-generating the behaviour of an interlocking system and the associated confidence conditions. This chapter will present the Java design and the implementation of the concrete model.

Java has been chosen for several reasons:

- Java is platform-independent. As *RSL-SAL* and *SAL* are available for both Windows and Linux, it will be possible to work with the generated transition systems using different operating systems. Therefore, the program for generating transition systems should work on both platforms.
- Java supports graphical user interfaces, making it possible to extend the implementation presented in this chapter with such an interface in the future.

In general, the Java implementation will be close to the concrete model, but some *RSL* language constructs like variant types and quantifiers are not supported by Java. Also, Java allows for using an object-oriented style which, in some cases, might be preferable. Therefore, there will be minor differences between the Java implementation and the concrete model.

On one hand, one could also think of switching to a more iterative style in order to obtain efficiency. On the other hand, for this application, correctness is more important than efficiency. Therefore, it is chosen to implement the functions of the concrete model using its recursive style.

Besides implementing the schemes from the concrete model, a parser and an unparser will be introduced such that data can be parsed from a stream or a text file, interpreted as a *StaticInterlockingSystem*, converted to a *TransitionSystem*, and unparsed to concrete *RSL-SAL* syntax.

Section 10.1 will present an overview of the design and the implementation using UML.

Section 10.2 will explain how the concrete *RSL* model given in chapter 9 is implemented. UML diagrams will be used for explaining the structure of the Java classes.

Section 10.3 will explain the choice of a text format that can be used for expressing a text based *StaticInterlockingSystem*. The chosen text format will be XML.

Section 10.4 will explain how a parser is made to process the XML format given in section 10.3. Also, an unparser is introduced for unparsing a *TransitionSystem* to an *RSL-SAL* scheme.

In general, we will not aim to fulfil the UML 1.X or 2.X standards. The purpose of the UML diagrams is to help the reader without giving too many details. Therefore, only the necessary details will be shown.

As explained in appendix C.5, the Java classes and an executable compilation for Java 1.6.X can be found on the attached CD. An API generated by the Java-doc tool can be found on the same CD.

Because proofs related to implementation relations are considered as being beyond the scope of this project, we will not prove implementation relations between the concrete model and the Java implementation.

10.1 Overview

The purpose of this section is to give an overview of the Java design and implementation.

10.1.1 Implementation relations

Figure 10.1 explains the relation between the functions and data types of the *RSL* schemes and the Java classes.

Classes containing static methods are introduced for doing the conversion from a *StaticInterlockingSystem* to a *TransitionSystem*. By using static methods, the functional programming style from *RSL* is kept. Schemes *Pathfinding*, *Conditionfinding*, *StaticInterlockingSystemToTransitionSystem*, *StaticInterlockingSystemToConfidenceConditions*, and *StaticInterlockingSystemConversion* define generator functions. For each of these schemes, a class has been introduced that defines static methods that are equivalent to the ones defined in the scheme.

Schemes *StaticInterlockingSystem*, *Diagrams*, *StaticInterlockingSystemL*, and *DiagramsL* both define data types and some functions that work on these types. For representing the types and the functions, classes that can be instantiated are introduced. E.g. a class for representing the *StaticInterlockingSystem* scheme will both contain the methods of the scheme and private fields for representing the *StaticInterlockingSystem* data type. Instead of taking a *StaticInterlockingSystem* as argument, the methods of this class will be applied directly to the private fields of an instance of a *StaticInterlockingSystem*.

The *Types* scheme contains multiple data types. In general, for each of these types, a class is introduced. These classes are located in the package called *types*. However, in order to represent the *LTLassertion* type, sub-package *types.ltl* is introduced. This package contains multiple classes for implementing the given type. Also, the sub-package called *types.bool* contains multiple classes in order to represent the type *BooleanExp*.

10.1.2 Computational overview

The sequence diagram in figure 10.2 gives an overview of how the generation of an *RSL-SAL* transition system takes place when everything goes well, i.e. if the parsed *StaticInterlockingSystem* is well-formed and no I/O error occurs. The

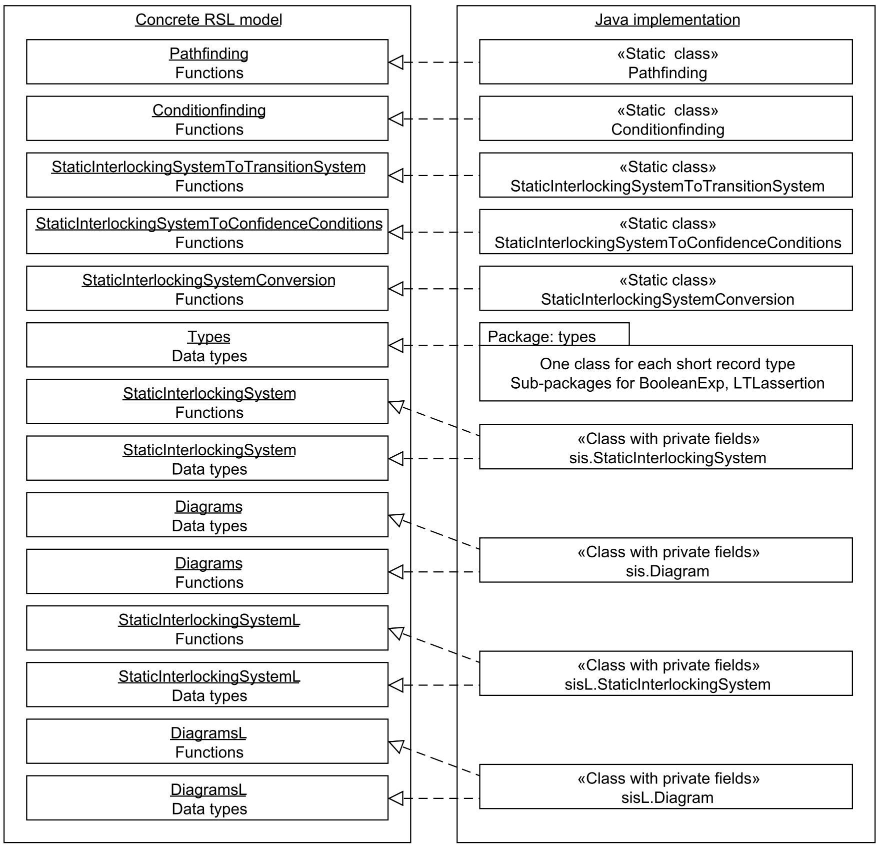

Concrete RSL model
Pathfinding
Functions
Conditionfinding
Functions
StaticInterlockingSystemToTransitionSystem
Functions
StaticInterlockingSystemToConfidenceConditions
Functions
StaticInterlockingSystemConversion
Functions
Types
Data types
StaticInterlockingSystem
Functions
StaticInterlockingSystem
Data types
Diagrams
Data types
Diagrams
Functions
StaticInterlockingSystemL
Functions
StaticInterlockingSystemL
Data types
DiagramsL
Functions
DiagramsL
Data types
Java implementation
«Static class»
Pathfinding
«Static class»
Conditionfinding
«Static class»
StaticInterlockingSystemToTransitionSystem
«Static class»
StaticInterlockingSystemToConfidenceConditions
«Static class»
StaticInterlockingSystemConversion
Package: types
One class for each short record type
Sub-packages for BooleanExp, LTLassertion
«Class with private fields»
sis.StaticInterlockingSystem
«Class with private fields»
sis.Diagram
«Class with private fields»
sisL.StaticInterlockingSystem
«Class with private fields»
sisL.Diagram

Figure 10.1: How the functions and types from the *RSL* schemes are implemented

process is initiated by the static class *XMLToRSLSAL*. By using an instance of an *XMLParser*, text in XML format is parsed from a stream. The *XMLParser* gives back a list-based *StaticInterlockingSystem*, called *SISL* in the figure. The process can only continue if *SISL* is well-formed. Otherwise, *XMLToRSLSAL* will abort the process.

After the well-formed check, *SISL* is given to the static class *StaticInterlockingSystemToValidationSystem* (in the figure called *SISLToValidationSystem*). *SISL* is then given to the static class *StaticInterlockingSystemConversion* (in the figure called *SISConversion*).

This class will convert the list-based *StaticInterlockingSystem*, *SISL*, and return a map-based *StaticInterlockingSystem*, *SIS*.

After the conversion, the static classes *StaticInterlockingSystemToTransitionSystem* and *StaticInterlockingSystemToConfidenceConditions* (in the figure called *SISToTransitionSystem* and *SISToConfidenceConditions*) are used for generating a *TransitionSystem* and its corresponding confidence conditions in terms of *LTLassertions*. These are stored in an instance of the class called *ValidationSystem* whose only purpose is to store a *TransitionSystem* and the corresponding *ConfidenceConditions*.

The *ValidationSystem* is returned to *XMLToRSLSAL*. *XMLToRSLSAL* will then ask an instance of the *RSLSALUnparser* to unparse the *ValidationSystem* and send it to a stream.

StaticInterlockingSystemToTransitionSystem and *StaticInterlockingSystemToConfidenceConditions* will behave as the corresponding schemes in the concrete model by invoking methods on the static class called *Conditionfinding*. This class implements every method of the corresponding *RSL* scheme and invokes methods on the static class called *Pathfinding*. Again, *Pathfinding* implements all the methods of its corresponding *RSL* scheme. The static structure of the classes used for making a *ValidationSystem* can be seen in figure 10.3.

10.1.3 Packages overview

This section will detail how the implementation is organised. The following packages are introduced:

conversion includes the needed classes for converting from a list-based *StaticInterlockingSystem* to a map-based *StaticInterlockingSystem* and from a map-based *StaticInterlockingSystem* to a *ValidationSystem*. This includes *Condi-*

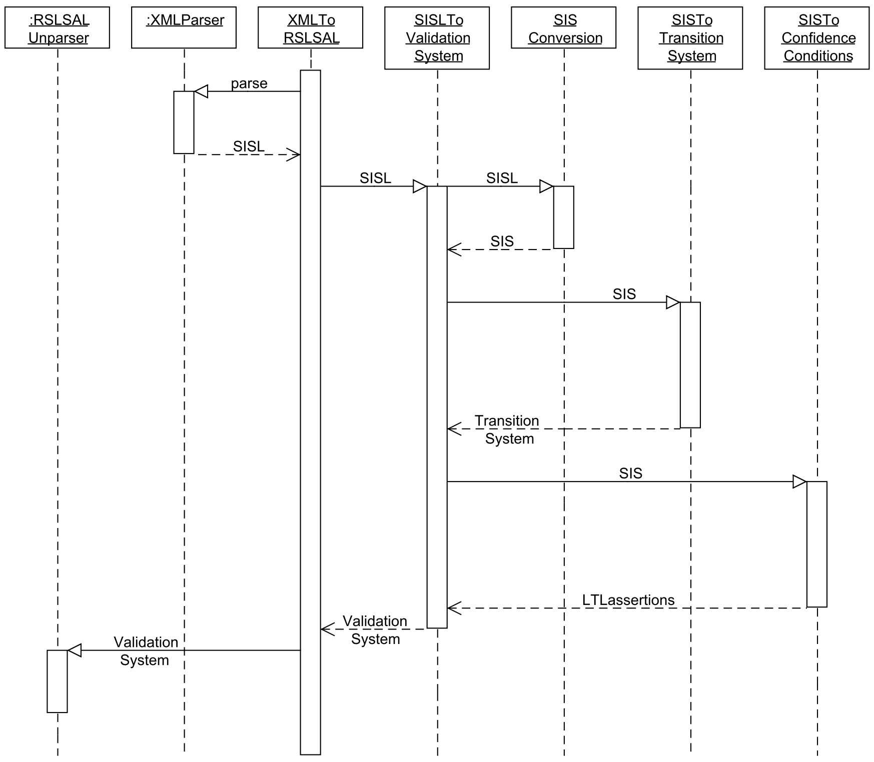


Figure 10.2: Sequence diagram: An overview of how an *RSL-SA*L transition system is made based on parsed data from a stream. *SIS* is used as a short for *StaticInterlockingSystem*.

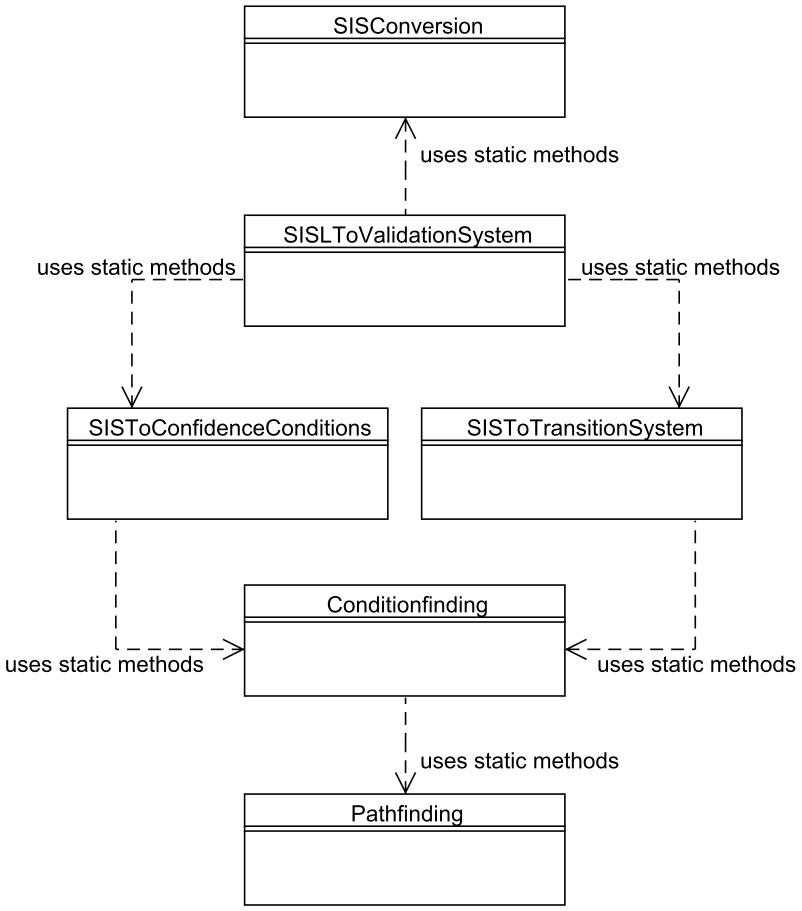


Figure 10.3: Class diagram: Structure of the static classes used for computing a *ValidationSystem* consisting of a *TransitionSystem* and its *ConfidenceConditions*. *SIS* is used as a short for *StaticInterlockingSystem*.

tionfinding and *Pathfinding.*

driver includes the class *Driver* that contains a main method. This method will be used for starting the conversion process from an XML file to a file containing concrete *RSL-SAL* syntax. The user will be asked to specify an input XML file, an output XML file, and the name of the generated *RSL-SAL* scheme.

exceptions includes the class *NotWellFormedException.* An instance of this exception is thrown if a parsed *StaticInterlockingSystem* is not well-formed.

parser includes everything related to parsing a *StaticInterlockingSystem* described using XML.

sis includes everything needed for representing a map-based *StaticInterlockingSystem.*

sisL includes everything needed for representing a list-based *StaticInterlockingSystem.*

transitionSystem includes everything needed for representing a *TransitionSystem* and a *ValidationSystem* except the common types representing *LTL* and Boolean expressions.

types includes Java implementations of most of the types given in the *Types* scheme of the concrete model.

types.bool includes the Java implementation of a *BooleanExp.*

types.common includes common functionality that is specific to the Java implementation.

types.ltl includes the Java implementation of an *LTLassertion.*

unparser includes everything related to unparsing a *ValidationSystem* to *RSL-SAL.*

10.2 Java implementation of the concrete *RSL* model

This section will focus on how the types of the concrete *RSL* model are converted to Java. Also, the section will explain how extra functionality is added to the Java implementation compared to the concrete model. However, the computation is still done in the same way as in the concrete model.

As Java does not support quantifiers, union types, short record types, and variant types, in some cases, it has been necessary to use some more Java-specific features when implementing the types.

10.2.1 Feedback on well-formed checks

The well-formed checks in *RSL* return true or false, but if something is not well-formed, they do not report why.

In order to increase the user-friendliness, it is decided to extend the well-formed checks in *RSL* with error-reporting features. This enables detailed feedback to the user in case something is not well-formed.

For making a standardised way of performing well-formed checks, an interface called *WellFormedCheckable* is introduced. The interface defines methods *isWellFormed()* of the type Boolean and *getErrorLog()* of the type String. Every class that defines a well-formed check must implement this interface. *isWellFormed()* is supposed to give true or false like a well-formed function in *RSL* does. If it detects an error, a log will be stored such that *getErrorLog()* can report the error. The well-formed checkable classes will therefore be on the following form:

```
public class A implements WellFormedCheckable{

  public boolean isWellFormed(){
    ...
  }

  public String getErrorLog(){
    ...
  }

  ...
}
```

Also, it has been decided to give diagrams a name. In that way, the error reporting functionality can give the name of a *Diagram* if it is not well-formed.

10.2.2 *Ids* and restriction on *Ids* and *Diagram* names

In the concrete *RSL* model, an *Id* has the type *Text*. Therefore, it has been decided to use the type String for an *Id*.

The *Ids* of the relays are also used for the names of the variables in the state. This means that they must be legal in an *RSL-SAL* context. For instance, spaces in variable names cannot be handled by *RSL-SAL*.

This leads to introducing a restriction on *Ids* such that each *Id* must match the following regular expression: $[a - zA - Z][a - zA - Z0 - 9]*$. An *Id* must therefore start with an upper-case or lower-case letter. The first letter can then be followed by an arbitrarily long and possibly empty sequence of upper-case letters, lower-case letters, and numbers.

Also, it has been decided to reserve some specific names. For instance, the *idle* variable must have a unique name. Therefore, the String "idle" cannot be used as an *Id* in a *StaticInterlockingSystem*.

In order to make a common way of checking an *Id*, the static class *IdRules* has been introduced with methods for checking whether an *Id* is legal and for giving text explanations of why an illegal *Id* is not legal.

10.2.3 Maps, sets, and lists

When implementing maps, sets, and lists, it is possible to take advantage of the Java collection framework. When using the collection framework, it is necessary to implement hash code methods and equals methods such that Java is capable of comparing elements. These methods will not be explained in this section, but the reader can assume that such methods are defined when needed.

The following generic classes have been used for representing maps, sets, and lists:

- *HashMap* for representing maps.
- *HashSet* and *LinkedHashSet* for representing sets. *LinkedHashSet* keeps the order in which the elements have been added.
- *ArrayList* for representing lists.

A static class called *SetOperations* is introduced in order to enable set operations that are not directly available in Java. For instance, a static method *hd* is introduced for representing the *RSL* operator *hd*:

```
public static <T> T hd(Collection<T> c){
  Iterator<T> it = c.iterator();
  if(!it.hasNext()){
    throw new RuntimeException("Cannot␣apply␣hd␣to␣an␣empty␣
        collection");
  }

  return it.next();
}
```

As the method is generic and works with Collections, it will both be applicable to sets and lists. Similar auxiliary methods are defined and the Java generics features are used when possible.

10.2.4 Representing components

The *Types* scheme of the concrete model (see appendix B.1, page 233) defines the components of a *Diagram* and a *StaticInterlockingSystem* in the following way:

```
type
   Component =
      Pole | Junction | RegularRelay | SteelRelay |
      Button | Contact
```

As Java does not support union types, it is necessary to use a more Java-specific feature for representing a *Component*. Also, as the components will have some basic functionality in common (e.g. an *id*), it is decided to introduce the abstract class *Component* that is specified in the following way:

```
public abstract class Component implements WellFormedCheckable{
  ...
}
```

And the components will be implemented in the following way:

```
public class AComponent extends Component{
  ...
}
```

The class diagram in figure 10.4 visualises how the components are defined in the Java implementation.

In order to enforce restrictions on the fields of a *Component*, e.g. enforcing that an *Id* is defined and is legal, it has been decided to let the *Component* class implement the *WellFormedCheckable* interface.

isWellFormed() checks the fields of the *Component* and gives true if they are acceptable. In case it gives false, *getErrorLog* can afterwards be used to obtain an explanation of the problem.

In *RSL*, types *Button*, *Junction*, and *Pole* only contain an *Id*. Therefore, these are almost pure extensions of *Component*.

A *Contact* also extends *Component*. Furthermore, *relayId* and *relayState* are added corresponding to the *Contact*-specific information of the concrete *RSL* model.

Both *RegularRelays* and *SteelRelays* will have an initial state. In order to avoid defining the same common functionality twice, the abstract class *InternalRelay* is defined as an extension of *Component*. It defines an initial state of a relay.

A *RegularRelay* is then an almost pure extension of *InternalRelay* while a *Steel-Relay* defines *Ids* of its 3 neighbours like in the concrete *RSL* model.

In principle, the *ExternalRelay* class could extend *InternalRelay* and thereby get the common relay information. However, that would make an *ExternalRelay* a *Component*. This would be inconsistent because an *ExternalRelay* is not supposed to occur in a *Diagram*. Therefore, an *ExternalRelay* must define an *Id* and an initial state.

The *Types* scheme of the concrete model (see appendix B.1, page 233) defined the type *State*:

```
type
  State == up | down
```

In order to represent such a *State*, the following enumeration is introduced:

```
public enum State {
  up, down
}
```

The implementation of the observer functions in the *Diagrams* and *StaticInterlockingSystems* schemes will then be capable of returning a *State*.

The *Poles* class is introduced to represent the type *Poles* defined in the *Types* scheme. As in the concrete *RSL* model, it defines fields for storing a positive *Pole* and a negative *Pole*.

Like *Component*, *Poles* and *ExternalRelay* implement *WellFormedCheckable* and must therefore be able to check the validity of their private fields.

10.2.5 Representing a list-based Static Interlocking System

As previously mentioned in section 10.1, it has been decided to merge the *RSL* scheme *StaticInterlockingSystemL* (as defined in appendix B.3, page 244) and its contained type for representing a list-based *StaticInterlockingSystem* into one class such that the data stored in private fields represent the data of *StaticInterlockingSystem* and the functions of *StaticInterlockingSystemL* can then be used on the private fields. The class *StaticInterlockingSystemL* is implemented in the following way:

```
package sisL;

public class StaticInterlockingSystem implements
    WellFormedCheckable {

  private ArrayList<Diagram> diagrams;

  private ArrayList<ExternalRelay> externalRelays;

  // functions from the StaticInterlockingSystemL scheme
  ...
}
```

The functionality will be the same as in the concrete *RSL* model. In the same way, the functionality from *DiagramsL* (see appendix B.2, page 236) and the list-based data type *Diagram* have been merged.

A class diagram representing the implementation of the list-based *StaticInterlockingSystem* can be seen in figure 10.5. A *Diagram* has fields for storing *ArrayLists* of the different *Component* types except *Pole*. As in the concrete *RSL* model, the poles are stored in an instance of the type *Poles*. Edges are stored in an *ArrayList* as instances of the *Edge* class corresponding to the *RSL* type *Edge*.

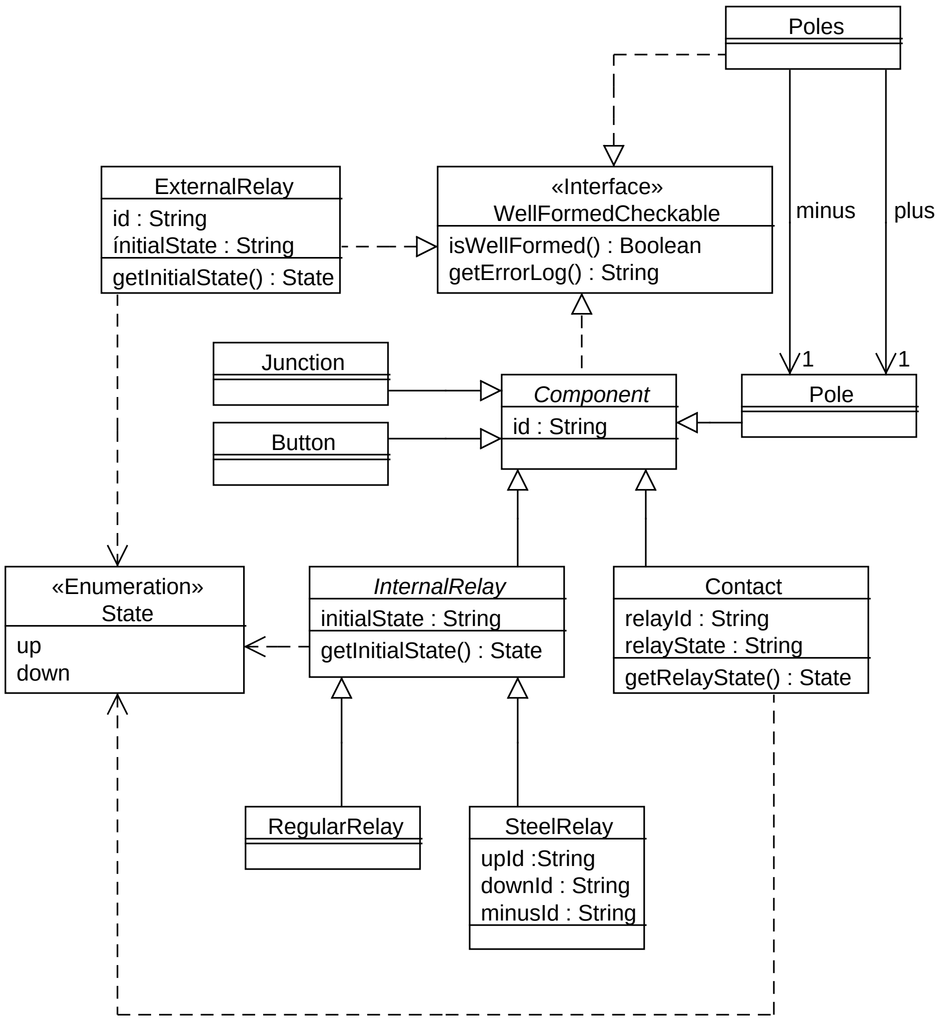


Figure 10.4: Class diagram: the *Components* of a *StaticInterlockingSystem*

In order to enable well-formed checks on a *Diagram*, the *Diagram* class implements the *WellFormedCheckable* interface. Furthermore, the necessary functions defined in *DiagramsL* of the concrete model have been implemented.

A *StaticInterlockingSystem* is also *WellFormedCheckable*. It contains *ArrayLists* of *Diagrams* and *ExternalRelays*. Again, the necessary functions in *StaticInterlockingSystemL* of the concrete model have been implemented.

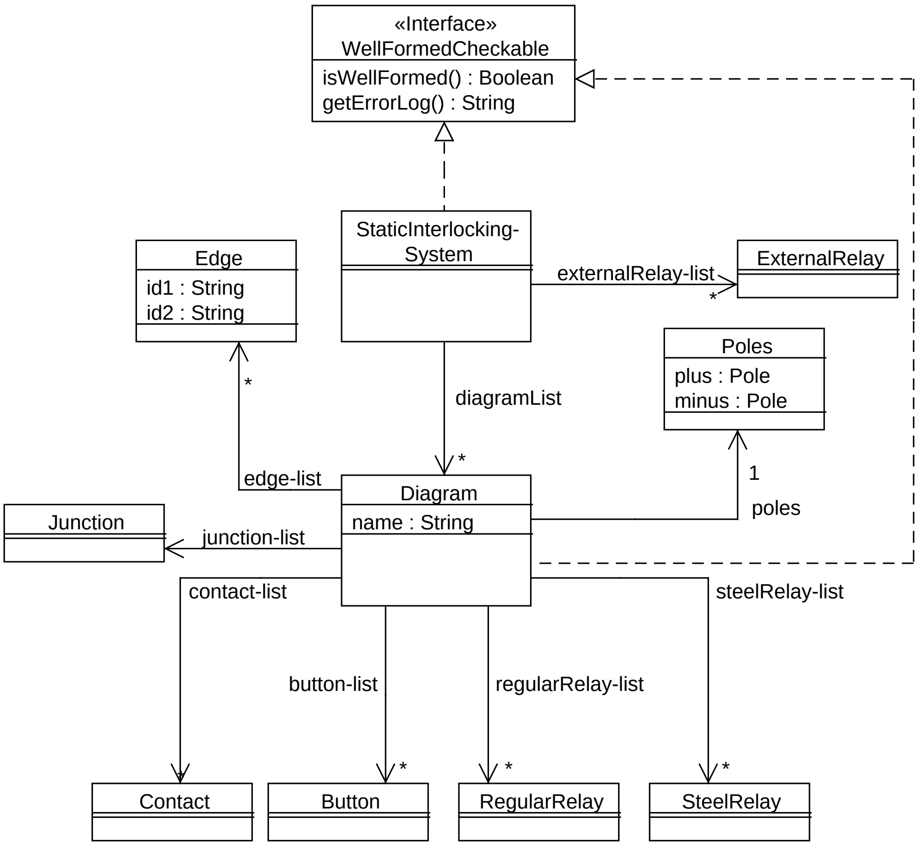


Figure 10.5: Class diagram: A list-based StaticInterlockingSystem

10.2.6 Representing a map-based Static Interlocking System

The functionality of the schemes of the map-based model, *StaticInterlockingSystem* (see appendix B.5, page 254) and *Diagram* (see appendix B.4, page 247), have, respectively, been merged with the map-based types for *StaticInterlockingSystem* and *Diagram*. The implementation of the class *StaticInterlockingSystem* is made in the following way:

```
package sis;

public class StaticInterlockingSystem implements
    WellFormedCheckable {

  private HashSet<Diagram> diagrams;

  private HashMap<String, ExternalRelay> externalRelays;

  // functions from the scheme StaticInterlockingSystem
  ...
}
```

Section 10.1 explained that well-formed checks are made on a list-based *StaticInterlockingSystem* before it can be converted to a map-based *StaticInterlockingSystem*. Therefore, the conversion will be applied only to well-formed, list-based *StaticInterlockingSystems*, implying that the conversion always will give a well-formed, map-based *StaticInterlockingSystem*. As the list-based *StaticInterlockingSystem* is well-formed, the map-based *StaticInterlockingSystem* resulting from the conversion will be well-formed. So there is no need for implementing the well-formed checks of a map-based *StaticInterlockingSystem*. Therefore, *WellFormedCheckable* is not implemented for a map-based *StaticInterlockingSystem*.

The *Diagram* class of the map-based *StaticInterlockingSystem* can be seen in figure 10.6. In the Java implementation, a map-based *Diagram* has a *HashSet* of *Edges*, a *HashMap* representing the *Components*, and a *HashMap* containing the *PoleTypes*. An enumeration has been introduced for representing the *RSL* variant type *PoleType*.

A *StaticInterlockingSystem* contains a *HashSet* of *Diagrams* and a *HashMap* for storing *ExternalRelays*.

All the necessary observer functions from the concrete *RSL* model have been implemented by *Diagram* and *StaticInterlockingSystem*.

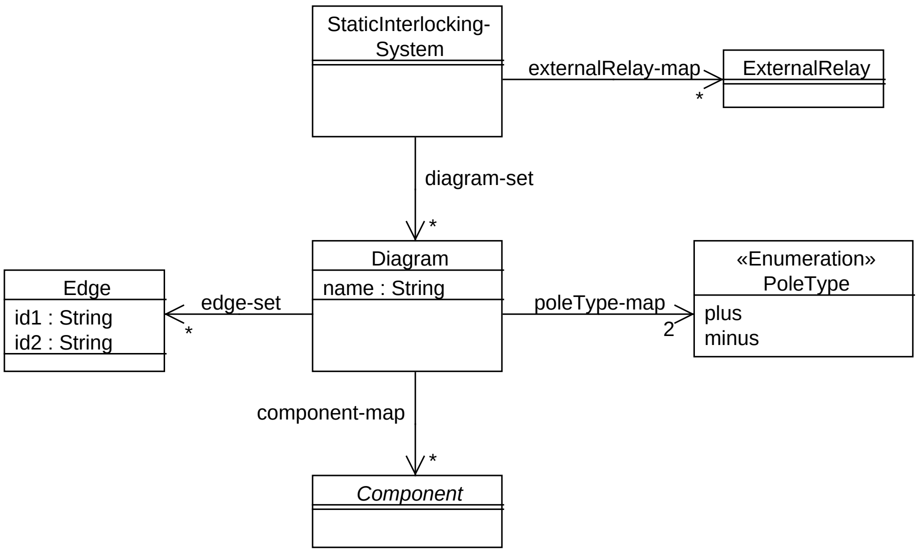


Figure 10.6: Class diagram: A map-based StaticInterlockingSystem

10.2.7 Representing Boolean expressions and *LTL* assertions

Variant types are not supported by Java. Therefore, when implementing *BooleanExp* and *LTLassertion* from the *Types* scheme of the concrete *RSL* model (see appendix B.1, 233), one needs to use Java features that have the same semantics as these *RSL* types.

In *RSL*, constructors can be used for creating a value of a variant type. These constructors are defined directly as part of the variant type.

Java will only allow constructors for classes. In order to make similar constructors in Java as the ones defined in the *RSL* versions of *BooleanExp* and *LTLassertion*, it has been decided to make one class for each constructor. Each class that corresponds to a constructor will then implement an interface that represents the type of which the class defines a constructor.

Suppose one has the following variant type in *RSL*:

type
  Variant == A(a : T1) | B(b : T2)

The following interface can then be used for defining the type:

```
public interface Variant {

}
```

And the classes for implementing the constructors can then be defined in the following way:

```
public class A implements Variant {
  public A(T1 a){
    ...
  }
  ...
}

public class B implements Variant {
  public B(T2 b){
    ...
  }
  ...
}
```

Figure 10.7 illustrates how the pattern has been applied for implementing the *RSL* type *BooleanExp*. An empty interface has been defined for representing a *BooleanExp*. The interface is implemented by the classes *And*, *Or*, *Neg*, and *Literal*.

Like in the concrete *RSL* model, *And* and *Or* contain sets. When computing the conditions based on paths, it will be helpful for a human reader if the conditions are presented in the order in which the *Buttons* and *Contacts* are present in the paths. In that way, it will be possible to understand a condition by manually following a path in a diagram. In order to enable this, *LinkedHashSet* are used as the set types of *And* and *Or*. A *LinkedHashSet* will remember the order of which the members where added, but will still avoid repetition of members. The order of a *LinkedHashSet* is not affected if an element is re-inserted.

Neg contains a single *BooleanExp* and *Literal* contains a String representing an *Id*.

In figure 10.7, one can also see that an interface is introduced for representing *LTLassertion*. Classes corresponding to the constructors *G*, *F*, *X*, *Imply*, and *B* have been introduced.

RSL-SAL allows for naming *LTL* assertions. In order to increase the user-friendliness, it is decided to name the confidence conditions after their type, the *Id* of the component for which a given confidence conditions states a property, and the name of the diagram containing the component. In order to enable this, the class *LTLassertionContainer* is introduced. It contains a name and an *LTLassertion.* The static class *StaticInterlockingSystemToConfidenceConditions* is then supposed to name assertions by including them in an *LTLassertionContainer.*

toString methods have been defined in order to enable an easy export of *LTL* assertions and Boolean expressions to *RSL-SAL* syntax.

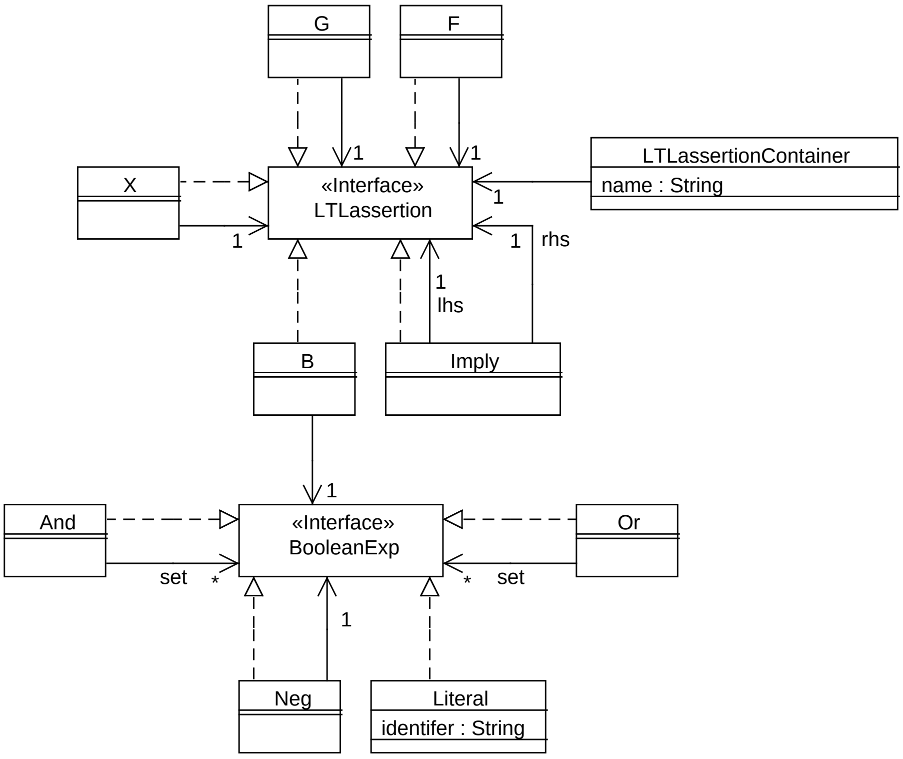


Figure 10.7: Class diagram: Representation of BooleanExp and LTLassertion

10.2.8 Simplification of formulas

Recall the *Conditionfinding* scheme of the concrete *RSL* model in section 9.5. A condition for having current through a given part of a circuit will be on the form $c_1 \vee ... \vee c_n$ where all the conditions $c_1, ..., c_n$ will be on the form $p_{i,1} \wedge ... \wedge p_{i,m}$.

It might be the case that one of the conditions c_1, ...,c_n, say c_x, contains a pair of complementary elements, e.g. a and $\sim a$. In that case, c_x is statically false and can be removed from the formula without changing the logical meaning of it. In other words, $c_1 \vee ... \vee c_x \vee ... \vee c_n$ can be reduced to $c_1 \vee ... \vee c_n$

In order to avoid evaluating condition parts that are always false, it it chosen to remove them when generating the *RSL-SAL* scheme. This is done without changing how the conditionfinding mechanism works. The *And* class seen in figure 10.7 is given a method for checking whether it represents something that is statically false. The *Or* will then have a method for removing every *And* instance inside of it that is statically false. After a disjunction of conjunctions is computed by *Conditionfinding*, it will be told to remove the parts of it that are statically false.

10.2.9 Presentation of validation- and transition systems

As previously mentioned, the Java class *ValidationSystem* is introduced for containing a transition system and its LTL assertions. Such a *ValidationSystem* can be seen in figure 10.8. It has references to an instance of *TransitionSystem* and an *ArrayList* of named *LTL* assertions represented by instances of *LTLassertionContainer*.

For representing the state, a *TransitionSystem* has an *ArrayList* of *Var* instances. As in the concrete *RSL* model, a *Var* has an *Id* and an initial value. For representing the *Boolean* variant type of the concrete specification, the enumeration *Boolean* is introduced. This enumeration is used for the initial value in a *Var*.

Furthermore, a *TransitionSystem* contains an *ArrayList* of *TransitionRules*. It has been decided to name each *TransitionRule* such that one can see which relay a given rule belongs to. As in the concrete *RSL* model, a *TransitionRule* has a guard of type *BooleanExp* and a *MultipleAssignment*. A *MultipleAssignment* contains a an *ArrayList* of *Assignment* instances. Like in the concrete *RSL* model, an *Assignment* has an *Id* and an assigned value of the type *BooleanExp*.

As for the map-based *StaticInterlockingSystem* and *Diagram*, it has been decided not to include the well-formed functions for *TransitionSystem*. If the input to the computation is well-formed, the result will be well-formed.

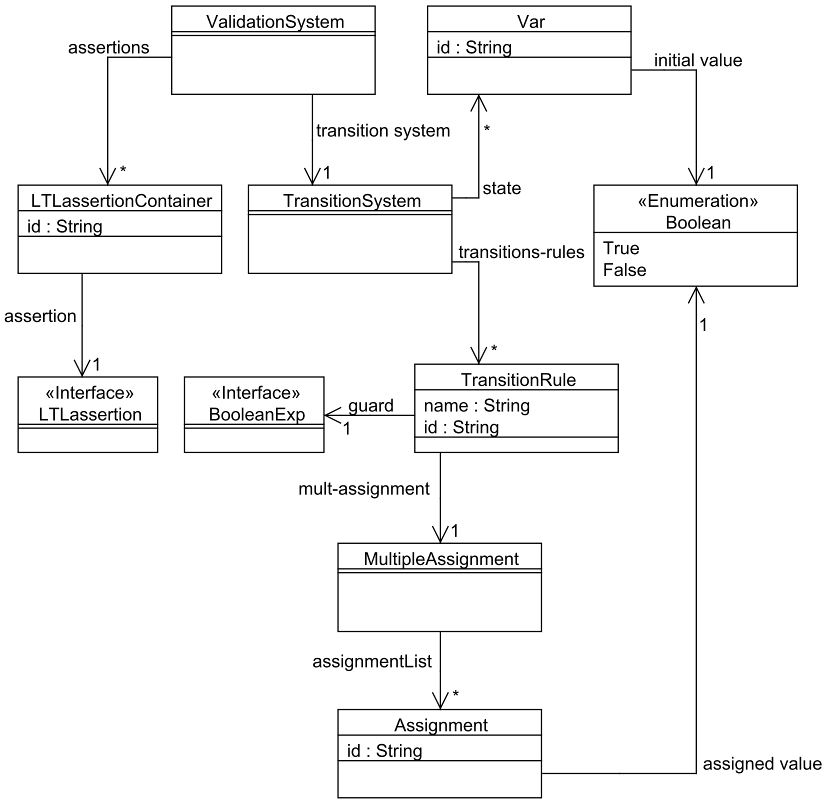


Figure 10.8: Class diagram: *ValidationSystem* and *TransitionSystem*

10.3 Input Format: XML

This section will describe the chosen input format that can be parsed by the Java program. The chosen format is XML. XML is selected because it is a well-known standard that is easy to parse for programs and is readable for a human user. Also, XML parsers are available for many programming languages and XML documents are extendible if one wishes to add further constructs in a *Diagram* or a *StaticInterlockingSystem*.

The purpose of the XML format is to make something as close to the list-based *StaticInterlockingSystem* as possible. In that way, a parser can easily make an instance of this model based on an XML document.

The chosen XML format declares a *StaticInterlockingSystem* on the following form:

```
<?DOCTYPE xml version="1.0" ?>
<StaticInterlockingSystem>
  <DiagramList>
    DL
  </DiagramList>

  <ExternalRelayList>
    ERL
  </ExternalRelayList>
</StaticInterlockingSystem>
```

ERL is a sequence of *ExternalRelays* on the following form:

```
<ExternalRelay id='extId1' initialState=X1/>
...
```

The attributes of each *ExternalRelay* correspond to the fields of an *ExternalRelay* in the concrete *RSL* mode. Definitions of *initialState* (in this case *X1*) must be equal to one of the two strings *'up'* and *'down'* .

DL is a sequence of *Diagrams*. A *Diagram* is on the following form:

```
<Diagram name = 'diagram1'>
  <Components>
    <Poles>
      <Plus> <Pole id = 'plusID'/> </Plus>
      <Minus> <Pole id = 'minusID'/> </Minus>
    </Poles>

    <ButtonList>
      BL
    </ButtonList>

    <ContactList>
      CL
    </ContactList>

    <JunctionList>
      JL
    </JunctionList>

    <SteelRelayList>
```

```
        SRL
      </SteelRelayList>

      <RegularRelayList>
        RRL
      </RegularRelayList>
    </Components>

    <EdgeList>
      EL
    </EdgeList>
</Diagram>
```

Poles corresponds to the type *Poles* of the concrete *RSL* model and it defines a positive and a negative pole.

BL defines a sequence of *Buttons* on the following form:

```
<Button id='buttonID1'/>
...
```

In that way, a *Button* defines the same field as in the concrete *RSL* model

CL defines a sequence of *Contacts* on the following form:

```
<Contact id='contactID1' conditionRelayId='relayId'
    relayState=X2/>
...
```

The attributes of a *Contact* corresponds to the fields of a *Contact* in the concrete *RSL* model. *conditionRelayId* defines the *Id* of the relay that rules the contact and *relayState* defines the required state of the relay to have the contact closed. Definitions of *relayState* (in this case *X2*) must be equal to one of the two strings *'up'* and *'down'* .

JL defines a sequence of *Junctions* on the following form:

```
<Junction id='junctionID1'/>
...
```

SRL is a sequence of *SteelRelays* and is on the following form:

```
<SteelRelay id='steelRelayID1' initialState=X3 upId='cp_up' downId=
    'cp_down' minusId='cp_minus'/>
...
```

Again, definitions of *initialState* (in this case *X3*) must be equal to one of the two strings *'up'* and *'down'*. *downId* and *minusId* represent the up relation, the down relation, and the minus relation of a *SteelRelay* respectively.

RRL is almost equivalent to to *SRL*. It defines a sequence of *RegularRelay* in the following way:

```
<RegularRelay id='relayID1' initialState=X4/>
```

Definitions of *initialState* (in this case *X4*) must be equal to one of the two strings *'up'* and *'down'*.

In a Diagram, *EL* is a sequence of edges on the following form:

```
<Edge id1='componentId1' id2='componentId2'/>
...
```

The attributes of an *Edge* correspond to the fields of an *Edge* in the concrete *RSL* model.

10.4 Parsing and unparsing

This section will explain some of the principles for parsing an XML version of *StaticInterlockingSystem* and for unparsing a *ValidationSystem*.

10.4.1 Parser implementation

In our context, a parser is supposed to read text and give back a list-based *StaticInterlockingSystem*. As previously mentioned, XML will be used for describing the *StaticInterlockingSystems*, but one could have chosen another format. Therefore it is chosen to make the following parser interface that is to be implemented by every parser:

```
public interface Parser {
  public sisL.StaticInterlockingSystem parseStream(InputStream in)
      throws Exception;
}
```

The parser will then parse data from an *InputStream* and give back a list-based *StaticInterlockingSystem*. An *InputStream* allows for reading data from a file, from System.In etc.

When implementing the XML parser, one can take advantage of Java XML libraries instead of implementing every layer of the parsing. In this project, it has been chosen to use *Commons Digester 1.8* from the *Apache* project[1]. Its dependencies are *Commons Logging 1.1.x* and *Commons BeanUtils 1.7*. As explained in the *Apache* license[2], the parser library and its dependencies are free and can be redistributed.

The advantage of *Digester* is that it allows for setting up rules for when objects should be created and when methods should be called. For instance, the following will set up a rule for when a *StaticInterlockingSystem* instance should be created by an instance of *Digester* called *digester*:

```
digester.addObjectCreate("StaticInterlockingSystem", "sisL.
    StaticInterlockingSystem");
```

The rule specifies that when the root element $< StaticInterlockingSystem >$ is encountered, an instance of a list-based *StaticInterlockingSystem* should be created and pushed on a stack of objects.

"Setter methods" of *StaticInterlockingSystem* will then be called during the parsing and at the end, the parser will return the instance of *StaticInterlockingSystem* from the stack.

The advantage of *Digester* is that it allows for specifying high level rules for how to process XML. The disadvantage is that it cannot check that an XML file defines some specific tags and attributes. For instance, if an attribute of a tag is undefined, it will interpret it as *null*. Therefore, it will not check whether the required tags or attributes are defined. If other tags than the required are defined, *Digester* will ignore them.

After the parsing, it is therefore necessary to check every data structure for undefined attributes. It implies that the implementation must be defensive, which is an advantage if someone replaces the parser with another one.

10.4.2 Unparser implementation

As for the parser, one could replace the *RSL-SAL* unparser with another unparser. Therefore, the following interface has been defined for an unparser:

```
public interface Unparser {
  public void unparse(String name, ValidationSystem vs, PrintStream
       out);
}
```

[1]The parser and its dependencies can be found on *http://commons.apache.org/digester/*
[2]*http://commons.apache.org/license.html*

The *unparse* method takes a name that can be interpreted as an *RSL* scheme name, a *ValidationSystem* that is to be unparsed, and a *PrintStream* for giving the result of the unparsing. A *PrintStream* could be *System.out* or something that writes directly to a file.

The unparser takes advantage of the concept of delegation by using the *toString* methods of the Java classes. These methods are able to export the classes as concrete *RSL-SAL* syntax. For instance, the classes that define *LTL* assertions and Boolean expressions are capable of converting instances of them directly to concrete *RSL-SAL* syntax, allowing the unparser to abstract from such tasks.

In that way, the primary responsibility of the unparser is to ensure that everything is unparsed in the right order with a proper indentation. The following method is defined for sending a String to a *PrintStream* with a specific indentation:

```
private void println(String s, int indentation, PrintStream out){
  for(int i = 0; i < indentation ; i++){
    out.print(" ");
  }

  out.println(s);
}
```

When having this auxiliary method available, one can use the following method for unparsing the state of a *TransitionSystem*:

```
private void unparseState(ArrayList<Var> state,  PrintStream out){
  for(int i = 0; i < state.size(); i++){
    println(state.get(i) + (i < state.size() -1 ? ",":""), 8, out);
  }
}
```

Due to the principle of delegation, the method abstracts from converting *Var* instances to concrete *RSL-SAL* syntax. Similar methods are implemented for exporting lists that contain instances of *LTLassertion* and *TransitionRule*.

Chapter 11

Testing the Java Implementation

This chapter will explain how the Java implementation introduced in chapter 10 is tested. As explained in appendix C.6, the test files can be found on the attached CD together with a more detailed explanation.

Section 11.1 will explain the test strategy. Three categories of tests will be introduced and a strategy for implementing the tests is formulated.

Section 11.2 will present the results of the tests.

Section 11.3 will make a conclusion based on the results of the tests.

11.1 Test strategy

This section will introduce the test strategy. In order to focus each test on a specific aspect of the program, it is chosen to divide the tests into several categories.

If the input to the Java program is valid, the execution will perform the following steps:

Step 1: Parse the XML file.

Step 2: Check that the necessary attributes are well-defined (e.g that they are not null).

Step 3: Check that the parsed instance of the list-based *StaticInterlockingSystem* is well-formed.

Step 4: Transform the *StaticInterlockingSystem* to a *TransitionSystem.*

Step 5: Unparse the *TransitionSystem.*

We will now analyse each of the above steps in order to define test categories:

Step 1: The used XML parser is responsible for performing the parsing and it will throw exceptions if the XML cannot be parsed. This functionality is part of the *Digester* library class and we will assume that the used library is working as specified in its API. Therefore, it is not necessary to test this step.

Step 2: The rest of the execution relies on step 2 functioning as expected. Therefore, it will be relevant to test step 2 separately.

Step 3: The rest of the execution relies on step 3 functioning as expected. Therefore, it will be relevant to test step 3 separately.

Step 4: It is essential that the generated transition systems and their confidence conditions are correct with respect to the diagrams. Therefore, it is relevant to test the conversion of well-formed *StaticInterlockingSystems* to a *ValidationSystem.*

Step 5: The unparser can be tested separately, but it can also be tested together with step 4. Especially if the testing of step 4 is performed using functional testing, every test in this category will involve unparsing. As it will be explained later, step 4 will be tested using functional testing, making it unnecessary to make a direct test of step 4.

This leads to the following test categories:

1. Attribute-related tests (step 2).
2. Well-formed related tests (step 3).

3. Transition system and confidence condition generation tests (step 4).

A test strategy for each of these categories will be formulated in the next sections.

11.1.1 Attribute-related tests

The attribute-related tests are supposed to check that proper error messages are generated by the program if the defined attributes of the parsed XML file are unacceptable.

Consider the following XML declarations of Junctions. Neither of them are malformed XML, but they are not acceptable in our context.

Undefined ID:

```
<Junction />
```

Empty Id:

```
<Junction id=''/>
```

Illegal Id:

```
<Junction id='£a'/>
```

(As explained in section 10.2, the regular expression $[a-zA-Z][a-zA-Z0-9]*$ must be matched and the *Id* must not be a reserved (e.g. if it equals “idle”).)

Similar illegal declarations can be made for any type of component. Therefore, for each type of component, it must be tested that the expected error messages are shown and the computation is aborted if one of the fields is illegal.

For each type of component and the external relays, it is decided to make three tests for each of its fields:

1. A test where the field is not defined.
2. A test where the content of the field is empty.
3. A test where the content of the field is illegal. For *Ids*, this means that the regular expression $[a - zA - Z][a - zA - Z0 - 9]*$ is not matched or that the *Id* is reserved.

Making this test will ensure that the content of the lists with components is legal. However, one could also imagine the following inconsistencies:

1. Some required definitions are not present in the parsed XML.
2. Too many definitions are present in the parsed XML.

An example of inconsistency 2 is the following Diagram:

```
<Diagram name = 'diagram1'>
  <Components>
     ...
    <ButtonList>
       ...
    </ButtonList>
     ...
    <ButtonList>
       ...
    </ButtonList>
     ...
  </Components>
  ...
</Diagram>
```

It must be checked that proper error messages are shown to the user and that the execution is aborted if inconsistencies 1 or 2 occur. Such tests are made when necessary. For instance, it is tested that the external relays are not declared twice.

11.1.2 Well-formed related tests

That a *StaticInterlockingSystem* is well-formed is a precondition for converting it into a *TransitionSystem*. Therefore, it is essential that the computation is aborted in case a *StaticInterlockingSystem* is not well-formed. Also, it must be tested that proper error messages are shown to the user.

Therefore, it has been decided to make one or more tests related to each possible type of violation of the well-formed function of the list-based *StaticInterlockingSystem*. For instance, it must be checked that a *StaticInterlockingSystem* is not considered as being well-formed if a *RegularRelay* contained by it has the same *Id* as another *RegularRelay*, a positive *Pole*, a negative *Pole*, a *Contact*, a *Button*, a *Junction*, a *SteelRelay*, or an *ExternalRelay*.

Also, it is decided to focus the tests on borderline cases with respect to the well-formed constraints. For instance, if a given type of *Component* must have two neighbours, at least two tests will be made. One test with a *Diagram* containing such a component that has 1 neighbour. A second test with a *Diagram* containing such a component that has 3 neighbours.

11.1.3 Transition system and confidence condition generation tests

Transition system generation tests are introduced for testing that the generated transition system actually expresses the behavioural semantics of a given interlocking system. Also, it must be tested that the required confidence conditions are generated.

In general, the following things must be checked for the generated *RSL* scheme:

1. The state is initialised correctly:
   the correct number of variables are declared in the state. The buttons are released and the relays have the state specified in the *StaticInterlockingSystem*. The *idle* variable is set to false.
2. There are two transition rules for each relay, one for drawing it and one for dropping it.
3. The *idle* transition rule is declared.
4. No other transition rule is declared.
5. The guards of the transition rules are correct.
6. The *LTL* assertions *X(idle)* and *G(F(idle))* are declared.
7. Two *LTL* assertions are declared for each regular relay.
8. Three *LTL* assertions are declared for each steel core relay.
9. No other *LTL* assertions are generated.
10. The content of the *LTL* assertions is correct.
11. The generated *RSL-SAL* scheme is formulated using correct *RSL-SAL* syntax.

1-4, 6-9, and 11 can be checked statically without analysing the possible paths inside the *Diagrams*.

However, 5 and 10 require further analysis. When testing them, it is necessary to let the implemented tool generate transition systems and confidence conditions for interlocking systems of which the result is already known. In other words, for each of the interlocking systems used for the test, it must be possible to calculate the conditions for drawing and dropping relays manually. It will then be possible to compare the auto-generated results with the manually calculated conditions.

The implemented algorithm that calculates conditions for drawing and dropping relays relies on a pathfinding algorithm. When testing that the transition systems are calculated correctly, it is essential to test that the computed conditions correspond to the ones that can be obtained by considering the well-formed paths in a diagram. Therefore, it is chosen to make artificial interlocking systems that each test a specific property of the pathfinding algorithm. The tests will have the two following focuses:

- the algorithm works for parallel and series connections in general and every possible path is taken into consideration by the *Conditionfinding* algorithm.
- The computation does not consider any illegal path, e.g. a path where a rule related to steel core relay is not respected.

In other words, the strategy for testing the generation of transition systems and confidence conditions is to create a set of artificial interlocking systems that test the different rules of the computation of conditions and paths. These systems are made such that it is possible to compute all the possible paths by hand and use them when inspecting the auto-generated results. Each time one of these tests are made, 1-4, 6-9, and 11 will be tested too by inspecting the state, the transition rules, and the confidence conditions.

11.1.4 Strategy for test implementation

When implementing tests for each category, it will be possible to use unit testing and functional testing.

For Java, the unit test framework JUnit[1] is available. A benefit of using JUnit is that the tests can be repeated quickly and unattendedly in an automated manner. This is useful for projects with a long implementation phase or for projects where refactoring of the implementation is likely to happen.

[1]For further information, see http://www.JUnit.org

In our project, the Java implementation phase has been relatively short due to the fact that most of the functionality was already defined in the concrete model introduced in chapter 9. Also, refactoring of the implementation is not likely to happen because it would probably require refactoring of both the abstract and the concrete model. Therefore, we do not need these benefits of unit testing.

Furthermore, unit testing is useful when one wants to focus the tests on specific parts of a program instead of testing the complete program. However, with the chosen division of the test cases, it will be possible to formulate functional tests that only focus on specific parts of the program. Therefore, this specific benefit from unit testing can also be obtained when using functional testing.

As most benefits of unit testing are not needed and the other ones can be obtained by using functional testing, there is no specific reason for preferring unit testing instead of functional testing.

Making unit tests will require formulations of the expected results and inputs using their abstract syntaxes. This will be time consuming compared to making functional tests. For each functional test, it will be required to make an XML file containing a *StaticInterlockingSystem*. It will then be applied to the program and the result will be inspected.

As we do not gain anything from unit tests and the formulation of these will be more time-consuming than making the functional tests, it is chosen to use functional testing only.

11.2 Tests and results

As explained in the test strategy, the tests are divided into three categories:

1. **Attribute related tests**.
   For this category, 78 XML files have been formulated. Each of these files formulate a single test.
2. **Well-formed related tests**.
   For this category, 53 XML files have been formulated for testing the constraints defined by the well-formed function in the *sisL.Diagram* class. Furthermore, 27 XML files have been formulated for testing the rest of the constraints defined by the well-formed function in the *StaticInterlockingSystem* class. Each of these files formulate a single test.

3. **Transition system and confidence condition generation tests**. For this category, 5 XML files containing a *StaticInterlockingSystem* with a single diagram have been formulated. Each *Diagram* is used for testing specific aspects of the condition finding process. Furthermore, a file with a *StaticInterlockingSystem* containing all the *Diagrams* from the 5 files has been formulated.

As explained in appendix C.6, the test files can be found on the attached CD together with explanation of the tests.

Each of the 164 test files were applied to the program and the outputs were as expected, i.e. no bugs were detected during the testing phase. All the test were performed in a Windows XP SP2 environment using Java 1.6.0_05. Some of the tests have also been performed in a Ubuntu 8.04, Kernel Linux 2.6.24-19-generic environment using Java 1.6._06.

11.3 Conclusion

The formulated test program has a high coverage and it tests the important functionalities of the implementation. As all the tests gave the expected results, it is concluded that it is likely that the program functions as it is supposed to.

Chapter 12

Conclusion

The following conclusion consists of two parts: a project summary and suggestions for future work.

12.1 Project summary

The goal of this thesis was to develop a formal method for verifying that interlocking systems for railway stations guarantee the safety of trains.

In order to do so, we have studied and modelled relay interlocking systems for Danish railway stations: static and dynamic models have been specified on different levels of abstraction. Furthermore, a transformation from an instance of a static model to an instance of a dynamic model and its associated confidence conditions has been specified using RSL and afterwards implemented using Java. The Java program can transform an XML description of relay diagrams to an RSL-SAL scheme. To complete the scheme resulting from such a transformation, behaviour of the external world and safety properties must be added.

We have formulated patterns for the behaviour of external inputs to interlocking systems. The dynamic model must be extended with instances of these patterns such that it includes the interaction with the external world. Also, patterns for safety properties have been developed. Instances of these can be verified using the state-space based model checking tool SAL.

Altogether, the work presented in this thesis has successfully defined a method that uses ordinary station documentation as a starting point for verifying that interlocking systems guarantee the safety of trains at a station. It has been demonstrated that the method is fully applicable to a small Danish railway station. However, as the method has not been applied to larger stations, we cannot say anything about its scalability.

12.2 Suggestions for future work

Even though this thesis has successfully developed a method for verifying that interlocking systems guarantee the safety of trains, there is still room for future work.

First of all, implementation relations should be proven in order to ensure the correctness of the concrete RSL model and the Java implementation.

Also, one cannot know to which extent the size of the state space will constrain the verification of safety properties for larger stations. Therefore, it will be relevant to make a scalability study.

The user-friendliness of the developed method can be improved in several ways. A tool for auto-generation of external behaviour and safety properties can be implemented. Also, it will be useful to develop a graphical user interface that is able to transform a graphical representation of diagrams to the XML format specified by this thesis and visualise counter-examples generated by the SAL model checker.

One could also think of extending the Java implementation with an algorithm for simplifying the logical conditions for drawing and dropping relays. However, that might make the conditions less understandable for a human reader and it will not change the size of the state space.

Bibliography

[1] The rsltc repository. `http://www.iist.unu.edu/newrh/III/3/1/docs/rsltc/`.

[2] Symbolic analysis laboratory, official home page. `http://sal.csl.sri.com/`.

[3] Uppaal, the official home page. `http://www.uppaal.com/`.

[4] Banedanmark. *Functional safety requirements Version 1.1 Issue 1.3*. Internal Banedanmark publication, 2003.

[5] Henrik W. Carlson and Carsten S. Lundsten. *Introduktion til sikrings- og fjernstyringsanlæg*. Rambøll, 2006.

[6] Banetjenesten Danske Statsbaner. *DSB Baneanlæg*. Danske statsbaner, Teknisk Afdeling, 1989.

[7] Louise Elmose Eriksen and Boe Pedersen. *Simulation of Relay Interlocking Systems*. 2007. IMM-BSC-2007-52.

[8] Ana Garis. Raise tool user guide. `http://www.iist.unu.edu/newrh/III/3/1/docs/rsltc/user_guide/html/ug_62.html`.

[9] The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall Int., 1992.

[10] Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge University Press, 2004. ISBN 0 521 54310 X paperback.

[11] Niels E. Jensen and Benny Mølgaard Nielsen. *De danske jernbaners signaler og sikringssystemer gennem 150 år*. Banebøger, 1998.

[12] Joern Pachl. *Railway Operation and Control.* VTD Rwail Publishing, 3604 220th Pl. Sw, Mountlake Terrace WA 98043 USA, 2002. ISBN 0-9719915-1-0.

[13] Juan Ignacio Perna and Chris George. *Model Checking RAISE Applicative Specifications. In Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods.* IEEE Computer Society, 2007.

[14] Juan Ignacio Perna and Chris George. *Model checking RAISE specifications. Technical Report 331.* UNU-IIST, P.O.Box 3058, Macau, November 2006.

[15] Steven S. Skiena. *The Algorithm Design Manual*, chapter Combinatorial Search and Heuristic Methods. Springer-Verlag, 1997.

[16] DSB skolen. *Signaturer for sikringsplan Version 1.01.*

Appendix A

The abstract RSL model

This appendix contains the schemes introduced by chapters 5 and 6. The schemes correspond to the boxes marked with *A*, *B* and *E* in figure 4.2, page 48.

A.1 Types

```
scheme Types =
   class
      type
         /∗ identifier∗/
         Id,
         /∗ a boolean expression ∗/
         BooleanExp ==
            and(a : BooleanExp-set) |
            or(o : BooleanExp-set) |
            neg(n : BooleanExp) |
            literal(id : Id),
         /∗ a boolean value∗/
         Boolean == True | False,
         /∗ a state of a relay∗/
         State == up | down,

         /* type for LTL assertions */
         LTLassertion ==
            G(g : LTLassertion) |
            F(f : LTLassertion) |
            X(x : LTLassertion) |
            Imply(lhs : LTLassertion, rhs : LTLassertion) |
            B(b : BooleanExp)

      value
         /* auxiliary functions */

         /* gets all the ids that are inside a given
         boolean expression */
         idsInBoolExp : BooleanExp → Id-set
         idsInBoolExp(exp) ≡
            case exp of
               literal(l) → {l},
               neg(nexp) → idsInBoolExp(nexp),
               and(aset) → idsInBoolExpSet(aset),
               or(oset) → idsInBoolExpSet(oset)
            end,

         /* gets all the ids in a set of boolean expressions
         */
         idsInBoolExpSet : BooleanExp-set → Id-set
         idsInBoolExpSet(set) ≡
            if set = {} then {}
            else
               let head = hd set in
                  idsInBoolExp(head) ∪
                  idsInBoolExpSet(set \ {head})
               end
            end
   end
```

A.2 Diagrams

context: T

```
scheme Diagrams =
   class
      type Diagram

      value
         /∗ observer functions ∗/
         /∗ true if the ID is a positive pole in the diagram,
         otherwise false ∗/
         isPlus : T.Id × Diagram → Bool,

         /∗ true if the ID is a negative pole in the diagram,
         otherwise false ∗/
         isMinus : T.Id × Diagram → Bool,

         /∗ true if the ID is a regular relay in the diagram,
         otherwise false ∗/
         isRegularRelay : T.Id × Diagram → Bool,

         /∗ true if the ID is a steel relay in the diagram,
         otherwise false ∗/
         isSteelRelay : T.Id × Diagram → Bool,

         /∗ true if the ID is a contact in the diagram,
         otherwise false ∗/
         isContact : T.Id × Diagram → Bool,

         /∗ true if the ID is a button in the diagram,
         otherwise false ∗/
         isButton : T.Id × Diagram → Bool,

         /∗ true if the ID is a junction in the diagram,
         otherwise false ∗/
         isJunction : T.Id × Diagram → Bool,

         /∗ observer functions for steel relays ∗/

         /∗ if the steel relay is receiving current
         from the returned neighbour of this function,
         it can be drawn  ∗/
         upRelation : T.Id × Diagram →̃ T.Id,

         /∗ if the steel relay is receiving current
         from the returned neighbour of this function,
         it can be dropped  */
         downRelation : T.Id × Diagram $\overset{\sim}{\rightarrow}$ T.Id,

         /* gives a neighbour of a steel relay
         from which the steel relay cannot receive
         current */
         minusRelation : T.Id × Diagram $\overset{\sim}{\rightarrow}$ T.Id,

         /* observer function for regular relays and steel
         relays*/
         /* gives the initial state of a relay in a given
         diagram*/
         relayState : T.Id × Diagram $\overset{\sim}{\rightarrow}$ T.State,

         /* observer functions for contacts */

         /* for a contact in a given diagram, the function
         gives the relay that controls the contact */
         relayIdForContact : T.Id × Diagram $\overset{\sim}{\rightarrow}$ T.Id,

         /*  For a contact in a given diagram, the function
         gives the required state of the relay that controls
         the contact
         for the contact to be closed */
         relayStateForContact : T.Id × Diagram $\overset{\sim}{\rightarrow}$ T.State,

         /* true if the two ids are neighbours in the given
         diagram,
         otherwise false  */
         areNeighbours : T.Id × T.Id × Diagram → Bool,

         /* auxiliary functions */
         /* gives all the neighbours of a given id in
         a given diagram */
         neighboursOf : T.Id × Diagram → T.Id-set
         neighboursOf(id1, d) ≡
            {id2 |
             id2 : T.Id •
                id2 ∈ allIds(d) ∧ areNeighbours(id1, id2, d)},

         /* Gives all the ids of a given diagram */
         allIds : Diagram → T.Id-set
         allIds(d) as ids post
```

```
        ids =
          {id |
           id : T.Id •
              isPlus(id, d) ∨ isMinus(id, d) ∨
              isRegularRelay(id, d) ∨ isSteelRelay(id, d) ∨
              isContact(id, d) ∨ isButton(id, d) ∨
              isJunction(id, d)},

    /* well−formed functions for diagrams */

    /* true if a diagram is well−formed*/
    isWfDiagram : Diagram → Bool

axiom
    /* The definition of the well−formed function is
       underspecified such that further constraints
       can be added later. */
    ∀ d : Diagram • isWfDiagram(d) ⇒
        okNeighbourRelation(d) ∧ okNumberOfNeighbours(d) ∧
        twoPoles(d) ∧ noIdOverlaps(d) ∧
        okSteelRelayRelations(d)

value
    /* an id cannot be neighbour to it self
    and the areNeighbours function is symmetric
    */
    okNeighbourRelation : Diagram → Bool
    okNeighbourRelation(d) ≡
        (∀ id1, id2 : T.Id •
            id1 ∈ allIds(d) ∧ id2 ∈ allIds(d) ⇒
               ((areNeighbours(id1, id2, d) ⇒ id1 ≠ id2) ∧
                 (areNeighbours(id1, id2, d) ⇒
                     areNeighbours(id2, id1, d)))),

    /* checks that each id has the required number
    of neighbours */
    okNumberOfNeighbours : Diagram → Bool
    okNumberOfNeighbours(d) ≡
        (∀ id : T.Id •
            id ∈ allIds(d) ⇒
               (
               /* a positive pole has minimum 1 neighbour */
               (isPlus(id, d) ⇒ card neighboursOf(id, d) ≥ 1) ∧
               /* a negative pole has minimum 1 neighbour */
            (isMinus(id, d) ⇒
                  card neighboursOf(id, d) ≥ 1) ∧
            /∗ a regular relay has exactly 2 neighbours ∗/
            (isRegularRelay(id, d) ⇒
                  card neighboursOf(id, d) = 2) ∧
            /∗ a contact has exactly 2 neighbours ∗/
            (isContact(id, d) ⇒
                  card neighboursOf(id, d) = 2) ∧
            /∗ a button has exactly 2 neighbours ∗/
            (isButton(id, d) ⇒
                  card neighboursOf(id, d) = 2) ∧
            /∗ a steel relay has exactly 3 neighbours ∗/
            (isSteelRelay(id, d) ⇒
                  card neighboursOf(id, d) = 3) ∧
            /∗ a junction has exactly 3 neighbours ∗/
            (isJunction(id, d) ⇒
                  card neighboursOf(id, d) = 3))),

/∗ checks that there is exactly one positive pole,
one negative pole ∗/
twoPoles : Diagram → Bool
twoPoles(d) ≡
   let
      plus =
         {id |
          id : T.Id • id ∈ allIds(d) ∧ isPlus(id, d)},
      minus =
         {id |
          id : T.Id •
             id ∈ allIds(d) ∧ isMinus(id, d)}
   in
      card plus = 1 ∧ card minus = 1
   end,

/∗ checks that the ids are not overlapping,
e.g. an Id of a junction cannot be the ID of a Pole ∗/
noIdOverlaps : Diagram → Bool
noIdOverlaps(d) ≡
   let
      plus =
         {id |
          id : T.Id • id ∈ allIds(d) ∧ isPlus(id, d)},
      minus =
         {id |
              id : T.Id •
                 id ∈ allIds(d) ∧ isMinus(id, d)},
          regularRelays =
             {id |
              id : T.Id •
                 id ∈ allIds(d) ∧ isRegularRelay(id, d)},
          steelRelays =
             {id |
              id : T.Id •
                 id ∈ allIds(d) ∧ isSteelRelay(id, d)},
          contacts =
             {id |
              id : T.Id •
                 id ∈ allIds(d) ∧ isContact(id, d)},
          buttons =
             {id |
              id : T.Id •
                 id ∈ allIds(d) ∧ isButton(id, d)},
          junctions =
             {id |
              id : T.Id •
                 id ∈ allIds(d) ∧ isJunction(id, d)}
       in
          card plus + card minus + card regularRelays +
          card steelRelays + card contacts + card buttons +
          card junctions = card allIds(d)
       end,

    /∗ checks for each steel relay that the set of
     steel relay relations equals the set of neighbours ∗/
    okSteelRelayRelations : Diagram → Bool
    okSteelRelayRelations(d) ≡
       (∀ id : T.Id •
           id ∈ allIds(d) ⇒
              (isSteelRelay(id, d) ⇒
                    {upRelation(id, d), downRelation(id, d),
                     minusRelation(id, d)} = neighboursOf(id, d))
       )
end
```

A.3 StaticInterlockingSystem

```
context: T, D
scheme StaticInterlockingSystem =
  class
    type StaticInterlockingSystem

    value
      /* observer functions */
      /* gives the diagrams of a static interlocking
      system*/
      diagrams : StaticInterlockingSystem → D.Diagram-set,
      /* gives the external relay ids of a static interlocking
      system */
      externalRelayIds :
        StaticInterlockingSystem → T.Id-set,

      /* gives the initial state of an external relay in a given
      static interlocking system */
      externalRelayState :
        T.Id × StaticInterlockingSystem ~→ T.State,

      /* auxiliary functions */
      /* gives all the internal relay ids of a
      static interlocking system */
      internalRelayIds :
        StaticInterlockingSystem → T.Id-set
      internalRelayIds(sis) ≡
        {id |
         id : T.Id • id ∈ allRelayIds(sis) ∧ (
           (∃ d : D.Diagram •
             d ∈ diagrams(sis) ∧
             (D.isRegularRelay(id, d) ∨
               D.isSteelRelay(id, d))))},


      /* gives all the relay ids of a static interlocking
      system*/
      allRelayIds : StaticInterlockingSystem ~→ T.Id-set
      allRelayIds(sis) ≡
        internalRelayIds(sis) ∪ externalRelayIds(sis),
      /∗ well−formed functions∗/

      /∗ true if a StaticInterlockingSystem is well−formed ∗/
      isWfStaticInterlockingSystem :
        StaticInterlockingSystem → Bool

    axiom
      /∗ The definition of the well−formed function is
         underspecified such that further constraints
         can be added later. ∗/
      ∀ sis : StaticInterlockingSystem •
      isWfStaticInterlockingSystem(sis) ⇒
        (∀ d : D.Diagram •
           d ∈ diagrams(sis) ⇒ D.isWfDiagram(d)) ∧
        uniqueIds(sis) ∧ contactsHaveRelays(sis)

    value
      /∗ checks that the ids are unique∗/
      uniqueIds : StaticInterlockingSystem → Bool
      uniqueIds(sis) ≡
        /∗ the ids of two different diagrams cannot overlap
        ∗/
        (∀ d1, d2 : D.Diagram •
           d1 ∈ diagrams(sis) ∧ d2 ∈ diagrams(sis) ⇒
           (d1 ≠ d2 ⇒ D.allIds(d1) ∩ D.allIds(d2) = {})) ∧
        /∗ the ids of a diagram and the external ids cannot overlap
        ∗/
        (∀ d : D.Diagram •
           d ∈ diagrams(sis) ⇒
             D.allIds(d) ∩ externalRelayIds(sis) = {}),

      /∗ for each contact in a diagram, the relay
      that controls the contact must be defined ∗/
      contactsHaveRelays : StaticInterlockingSystem → Bool
      contactsHaveRelays(sis) ≡
        (∀ d : D.Diagram, id : T.Id •
           d ∈ diagrams(sis) ∧ id ∈ D.allIds(d) ∧
           D.isContact(id, d) ⇒
             D.relayIdForContact(id, d) ∈
               allRelayIds(sis))
  end
```

A.4 Pathfinding

```
context: T, D
scheme Pathfinding =
   class
      type Path = T.Id*

      value
       /* well formed functions */
         /* checks if a path is well formed*/
         isWfPath : Path × D.Diagram $\overset{\sim}{\rightarrow}$ Bool
         isWfPath(p, d) ≡
            /* a path does not repeat ids */
            noDuplicates(p) ∧
            /* minimum 2 ids in a path*/
            len p ≥ 2 ∧
            /* a path starts with a positive pole
            and ends with a negative pole */
            D.isPlus(p(1), d) ∧ D.isMinus(p(len p), d) ∧
            /* (p(1),p(2)),...,(p(n−1),p(n)) are neighbours
            */
            (∀ n : Nat •
                n ∈ (inds p \ {len p}) ∧
                D.areNeighbours(p(n), p(n + 1), d)) ∧
            /* the up and down path of a steel relay
            cannot be in the same path at the same time
            */
            noSteelRelayProblem(p, d) pre D.isWfDiagram(d),

         /* checks that a path has no duplicate elements */
         noDuplicates : Path → Bool
         noDuplicates(p) ≡ card elems p = len p,

         /*  checks that the up and down part of a steel relay
            are not in the same path at the same time */
         noSteelRelayProblem : Path × D.Diagram $\overset{\sim}{\rightarrow}$ Bool
         noSteelRelayProblem(p, d) ≡
            (∀ id : T.Id •
                id ∈ elems p ∧ D.isSteelRelay(id, d) ⇒
                   ∼ (D.upRelation(id, d) ∈ elems p ∧
                         D.downRelation(id, d) ∈ elems p))
         pre
            D.isWfDiagram(d) ∧
```

```
        (∀ id : T.Id •
            id ∈ elems p ⇒ id ∈ D.allIds(d)),

    /∗ for path computation ∗/
    /∗ true if a set of ids is contained by a path
    ∗/
    isPathFor : Path × T.Id-set → Bool
    isPathFor(p, ids) ≡ ids ⊆ elems p,

    /∗ gives all the paths through a given set of
    ids
    in a give diagram ∗/
    allPathsFor : T.Id-set × D.Diagram ~→ Path-set
    allPathsFor(ids, d) ≡
        {p | p : Path • isWfPath(p, d) ∧ isPathFor(p, ids)}
    pre D.isWfDiagram(d) ∧ ids ⊆ D.allIds(d)
end
```

A.5 Conditionfinding

```
context: T, D, PF
scheme Conditionfinding =
  class
    value
      /∗ gives the boolean expression that is true iff
      there can be
      current throuh a give button or a given contact
      ∗/
      isConducting : T.Id × D.Diagram ~→ T.BooleanExp
      isConducting(id, d) ≡
        if D.isButton(id, d) then T.literal(id)
        else
          case D.relayStateForContact(id, d) of
            T.up → T.literal(D.relayIdForContact(id, d)),
            T.down →
              T.neg(T.literal(D.relayIdForContact(id, d)))
          end
        end
      pre
        D.isWfDiagram(d) ∧
        (D.isButton(id, d) ∨ D.isContact(id, d)),

      /∗ generation of boolean expressions ∗/
      /∗ gives the expression that is true iff
      the path is conductive
      (note: poles and relays are always conductive,
      but buttons and contacts must be closed in order
      to obtain conductivity)∗/
      isConducting : PF.Path × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
      isConducting(p, d) ≡
        T.and(
            {isConducting(id, d) |
            id : T.Id •
              id ∈ **elems** p ∧
              (D.isButton(id, d) ∨ D.isContact(id, d))})
      **pre** D.isWfDiagram(d) ∧ PF.isWfPath(p, d),

      /∗ gives the boolean expression that is true iff
      there is current through the given regular relay
      ∗/
      currentThroughRegularRelay :
        T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
      currentThroughRegularRelay(id, d) ≡
        T.or(
            {isConducting(p, d) |
            p : PF.Path • p ∈ PF.allPathsFor({id}, d)})
      **pre** D.isWfDiagram(d) ∧ D.isRegularRelay(id, d),

      /∗ gives the condition for not having current
      through a given
      regular relay∗/
      noCurrentThroughRegularRelay :
        T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
      noCurrentThroughRegularRelay(id, d) ≡
        T.neg(currentThroughRegularRelay(id, d))
      **pre** D.isWfDiagram(d) ∧ D.isRegularRelay(id, d),

      /∗ gives the boolean expression that is true iff
      there is no current through the given regular
      relay ∗/
      canDrawRegularRelay :
        T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
      canDrawRegularRelay(id, d) ≡
```

```
   T.and(
       {T.neg(T.literal(id)),
         currentThroughRegularRelay(id, d)})
pre D.isWfDiagram(d) ∧ D.isRegularRelay(id, d),

/∗ gives the boolean expression that is true iff
a given regular relay
can be dropped ∗/
canDropRegularRelay :
   T.Id × D.Diagram $\overset{\sim}{\to}$ T.BooleanExp
canDropRegularRelay(id, d) ≡
   T.and(
       {T.literal(id),
         noCurrentThroughRegularRelay(id, d)})
pre D.isWfDiagram(d) ∧ D.isRegularRelay(id, d),

/∗ gives the boolean expression that is true iff
there is current through the given regular relay
that makes it draw
∗/
drawingCurrentThroughSteelRelay :
   T.Id × D.Diagram $\overset{\sim}{\to}$ T.BooleanExp
drawingCurrentThroughSteelRelay(id, d) ≡
   T.or(
       {isConducting(p, d) |
         p : PF.Path •
            p ∈
               PF.allPathsFor(
                   {D.upRelation(id, d), id,
                     D.minusRelation(id, d)}, d)})
pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d),

/∗ gives the boolean expression that is true iff
there is current through the given regular relay
that makes it drop
∗/
droppingCurrentThroughSteelRelay :
   T.Id × D.Diagram $\overset{\sim}{\to}$ T.BooleanExp
droppingCurrentThroughSteelRelay(id, d) ≡
   T.or(
       {isConducting(p, d) |
         p : PF.Path •
            p ∈
```

```
                    PF.allPathsFor(
                        {D.downRelation(id, d), id,
                          D.minusRelation(id, d)}, d)})
    pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d),

    /∗ gives the boolean expression that is true iff
    a given steel relay
    can be drawn∗/
    canDrawSteelRelay :
      T.Id × D.Diagram ~→ T.BooleanExp
    canDrawSteelRelay(id, d) ≡
      T.and(
          {T.neg(T.literal(id)),
            drawingCurrentThroughSteelRelay(id, d)})
    pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d),

    /∗ gives the boolean expression that is true iff
    a given steel relay
    can be dropped ∗/
    canDropSteelRelay :
      T.Id × D.Diagram ~→ T.BooleanExp
    canDropSteelRelay(id, d) ≡
      T.and(
          {T.literal(id),
            droppingCurrentThroughSteelRelay(id, d)})
    pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d)
  end
```

A.6 TransitionSystem

```
context: T
scheme TransitionSystem =
  class
    type
      /∗ a variable in the state of a transition system
      ∗/
      Var :: id : T.Id   val : T.Boolean,

      /∗ an assignment in a transition rule.
      the Var with the given id in the current state
      will be assigned the new value */
      Assignment :: id : T.Id   assign : T.Boolean,

      /* all the assignments in a transition rule */
      MultipleAssignment = Assignment*,

      /* a transition rule has a guard and
      a multiple assignment */
      TransitionRule ::
         guard : T.BooleanExp
         assignments : MultipleAssignment,

      /* a transition system has an initial state
      and some transition rules */
      TransitionSystem ::
         state : Var*
         transitionRules : TransitionRule*

    value

      /* well formed functions */
      /* checks that a transition system is well formed */
      isWfTransitionSystem : TransitionSystem → Bool
      isWfTransitionSystem(ts) ≡
         isWfState(state(ts)) ∧
         areWfTransitionRules(state(ts), transitionRules(ts)),

      /* the variables in the state must have unique ids
         and duplicates are not allowed */
      isWfState : Var* → Bool
      isWfState(state) ≡
         (∀ v1, v2 : Var •
             v1 ∈ state ∧ v2 ∈ state ∧ id(v1) = id(v2) ⇒
                v1 = v2)
          ∧
          card elems state = len state,

      /* checks that all the transition rules are well formed */
      areWfTransitionRules :
         Var* × TransitionRule* → Bool
      areWfTransitionRules(state, transitionRules) ≡
         let
            /* all the ids that are defined in the state */
            ids = {id(var) | var : Var • var ∈ elems state}
        in
          (∀ tr : TransitionRule •
             tr ∈ elems transitionRules ⇒
               /∗ The ids in a guard must be defined in the state ∗/
               T.idsInBoolExp(guard(tr)) ⊆ ids ∧
               /∗ There must minimum be one assignment ∗/
               len assignments(tr) > 0 ∧
               /∗ the id in an assignment must be defined in the state ∗/
               (∀ a : Assignment •
                   a ∈ assignments(tr) ⇒ id(a) ∈ ids) ∧
               /∗ a variable cannot be assigned several values in
               the same multiple assignment ∗/
               (∀ a1, a2 : Assignment •
                   a1 ∈ elems assignments(tr) ∧
                   a2 ∈ elems assignments(tr) ∧
                   id(a1) = id(a2) ⇒ a1 = a2) ∧
               card elems assignments(tr) = len assignments(tr))
        end

  end
```

A.7 StaticInterlockingSystemToTransitionSystem

```
context: T, TS, D, CF, SIS
scheme StaticInterlockingSystemToTransitionSystem =
  class
    value
      /∗ the identifier of the idle variable ∗/
      idleId : T.Id,
      /∗ the idle variable ∗/
      idle : TS.Var = TS.mk_Var(idleId, T.False),

      /∗ calculates the behavioural semantics
      (a transition system) of
      a static interlocking system ∗/
      makeBehaviouralSemantics : SIS.StaticInterlockingSystem ~→ TS.TransitionSystem
      makeBehaviouralSemantics(sis) as ts post
        TS.isWfTransitionSystem(ts) ∧ stateRel(sis, ts) ∧
        transitionRel(sis, ts)
      pre SIS.isWfStaticInterlockingSystem(sis),
      /* observer functions */
      /* gives all the ids in a static interlocking
      system
      that
      should be defined in the state of the final transition
      system,
      i.e. the ids of the buttons and the relays */
      stateIds : SIS.StaticInterlockingSystem $\overset{\sim}{\to}$ T.Id-set
      stateIds(sis) $\equiv$
        {id |
         id : T.Id •
            id $\in$ SIS.externalRelayIds(sis) $\vee$
            ($\exists$ d : D.Diagram •
                   d $\in$ SIS.diagrams(sis) $\wedge$
                   (D.isRegularRelay(id, d) $\vee$
                    D.isSteelRelay(id, d) $\vee$ D.isButton(id, d)))},

      /* true if a multiple assignment assigns a given
      value
      to a given id, otherwise false */
      assignsValue :
        TS.MultipleAssignment $\times$ T.Id $\times$ T.Boolean $\to$ Bool
      assignsValue(ma, id, b) $\equiv$
        TS.mk_Assignment(id, b) $\in$ elems ma,

      /* true if a multiple assignment assigns a given
      value
      to a given id and makes no other assignments,
      otherwise false */
      assignsOnly :
        TS.MultipleAssignment $\times$ T.Id $\times$ T.Boolean $\to$ Bool
      assignsOnly(ma, id, b) $\equiv$
        ma = $\langle$TS.mk_Assignment(id, b)$\rangle$,

      /* true if a state matches a given boolean expression.
      true should be equivalent to up,
      false should be equivalent to down */
      stateMatchesBoolean : T.State $\times$ T.Boolean $\to$ Bool
      stateMatchesBoolean(s, b) $\equiv$
        case s of
            T.up $\to$ b = T.True,
            T.down $\to$ b = T.False
        end,

      /* check the state of the end result */
      stateRel :
        SIS.StaticInterlockingSystem × TS.TransitionSystem $\overset{\sim}{\rightarrow}$
          Bool
      stateRel(sis, ts) ≡
        /* the external relays are defined in the
        initial state of the transition system.
        the initial value in the initial state of the
        transition
        system must match the state of the relay. */
        (∀ id : T.Id •
            id ∈ SIS.externalRelayIds(sis) ⇒
              (∃ var : TS.Var •
                  var ∈ elems TS.state(ts) ∧
                  TS.id(var) = id ∧
                  stateMatchesBoolean(
                    SIS.externalRelayState(id, sis),
                    TS.val(var)))) ∧
        /* relays from the diagrams are defined in the
        initial state of the transition system.
        the initial value in the initial state of the
        transition
        system must match the initial state of the relay.
        */
        (∀ id : T.Id, d : D.Diagram •
            d ∈ SIS.diagrams(sis) ∧
            (D.isRegularRelay(id, d) ∨ D.isSteelRelay(id, d)) ⇒
              (∃ var : TS.Var •
                  var ∈ elems TS.state(ts) ∧
                  TS.id(var) = id ∧
                  stateMatchesBoolean(
                    D.relayState(id, d), TS.val(var)))) ∧
        /* buttons from the diagram are defined in the
        initial state of the transition system.
        the values in the inital state are false. */
        (∀ id : T.Id, d : D.Diagram •
            d ∈ SIS.diagrams(sis) ∧ D.isButton(id, d) ⇒
              (∃! var : TS.Var •
                  var ∈ elems TS.state(ts) ∧
                  TS.id(var) = id ∧ TS.val(var) = T.False)) ∧
        /* every variable in the state of the transition
        system
        has a corresponding button or relay id in the
        the static
        interlocking system  or is the idle variable
        */
        (∀ var : TS.Var •
            var ∈ elems TS.state(ts) ⇒
              TS.id(var) ∈ stateIds(sis) ∨ var = idle) ∧
        /* the idle variable is defined in the state
        */
        idle ∈ elems TS.state(ts)
      pre
        SIS.isWfStaticInterlockingSystem(sis) ∧
        TS.isWfTransitionSystem(ts),

      /* checks the transitions of the end result */
      transitionRel :
        SIS.StaticInterlockingSystem × TS.TransitionSystem $\overset{\sim}{\to}$
          Bool
      transitionRel(sis, ts) ≡
        /* Regular relays have transition rules */
        (∀ id : T.Id, d : D.Diagram •
            d ∈ SIS.diagrams(sis) ∧
            D.isRegularRelay(id, d) ⇒
              /* a specific rule for drawing is defined*/
              ((∃! tr : TS.TransitionRule •
                      tr ∈ elems TS.transitionRules(ts) ∧
                      TS.guard(tr) =
                    CF.canDrawRegularRelay(id, d) ∧
                      assignsOnly(TS.assignments(tr), id, T.True)) ∧
                /* a specific rule for dropping is defined */
                (∃! tr : TS.TransitionRule •
                      tr ∈ elems TS.transitionRules(ts) ∧
                      TS.guard(tr) =
                    CF.canDropRegularRelay(id, d) ∧
                      assignsOnly(TS.assignments(tr), id, T.False)
                ) ∧
                /* exactly one transition rule can draw the regular
                relay*/
                (∃! tr : TS.TransitionRule •
                      tr ∈ elems TS.transitionRules(ts) ∧
                      assignsValue(TS.assignments(tr), id, T.True)
                ) ∧
              /∗ exactly one transition rule can drop the regular
              relay∗/
              (∃! tr : TS.TransitionRule •
                  tr ∈ elems TS.transitionRules(ts) ∧
                  assignsValue(
                      TS.assignments(tr), id, T.False)))) ∧
      /∗ Steel relays have transition rules ∗/
      (∀ id : T.Id, d : D.Diagram •
        d ∈ SIS.diagrams(sis) ∧ D.isSteelRelay(id, d) ⇒
            /∗ a specific rule for drawing is defined∗/
            ((∃! tr : TS.TransitionRule •
                  tr ∈ elems TS.transitionRules(ts) ∧
                  TS.guard(tr) = CF.canDrawSteelRelay(id, d) ∧
                  assignsOnly(TS.assignments(tr), id, T.True)) ∧
              /∗ a specific rule for dropping is defined ∗/
              (∃! tr : TS.TransitionRule •
                  tr ∈ elems TS.transitionRules(ts) ∧
                  TS.guard(tr) = CF.canDropSteelRelay(id, d) ∧
                  assignsOnly(TS.assignments(tr), id, T.False)
              ) ∧
              /∗ exactly one transition rule can draw the steel
              relay∗/
              (∃! tr : TS.TransitionRule •
                  tr ∈ elems TS.transitionRules(ts) ∧
                  assignsValue(TS.assignments(tr), id, T.True)
              ) ∧
              /∗ exactly one transition rule can drop the steel
              relay∗/
              (∃! tr : TS.TransitionRule •
                  tr ∈ elems TS.transitionRules(ts) ∧
                  assignsValue(
                      TS.assignments(tr), id, T.False)))) ∧
      /∗ every transition rule assigns a value to a
      corresponding internal
      relay that is obtained from a diagram  or is
      the idle
      transitionRule∗/
      (∀ tr : TS.TransitionRule •
        tr ∈ elems TS.transitionRules(ts) ⇒
            len TS.assignments(tr) = 1 ∧
            (TS.id(TS.assignments(tr)(1)) ∈
              SIS.internalRelayIds(sis)) ∨
            tr = makeIdleTransitionRule(sis)) ∧
      /∗ the idle transition rule is defined in the
```

```
    transition rules
    */
    makeIdleTransitionRule(sis) $\in$
        **elems** TS.transitionRules(ts)
**pre**
    SIS.isWfStaticInterlockingSystem(sis) $\wedge$
    TS.isWfTransitionSystem(ts),

/* makes the idle transition rule from a list
of transition rules */
makeIdleTransitionRule :
    SIS.StaticInterlockingSystem $\xrightarrow{\sim}$ TS.TransitionRule
makeIdleTransitionRule(sis) $\equiv$
    **let**
        regularUpConditions =
            {CF.canDrawRegularRelay(rr, d) |
             rr : T.Id, d : D.Diagram •
                rr $\in$ SIS.internalRelayIds(sis) $\wedge$
                d $\in$ SIS.diagrams(sis) $\wedge$
                D.isRegularRelay(rr, d)},
        regularDownConditions =
            {CF.canDropRegularRelay(rr, d) |
             rr : T.Id, d : D.Diagram •
                rr $\in$ SIS.internalRelayIds(sis) $\wedge$
                d $\in$ SIS.diagrams(sis) $\wedge$
                D.isRegularRelay(rr, d)},
        steelUpConditions =
            {CF.canDrawSteelRelay(rr, d) |
             rr : T.Id, d : D.Diagram •
                rr $\in$ SIS.internalRelayIds(sis) $\wedge$
                d $\in$ SIS.diagrams(sis) $\wedge$
                D.isSteelRelay(rr, d)},
        steelDownConditions =
            {CF.canDropSteelRelay(rr, d) |
             rr : T.Id, d : D.Diagram •
                rr $\in$ SIS.internalRelayIds(sis) $\wedge$
                d $\in$ SIS.diagrams(sis) $\wedge$
                D.isSteelRelay(rr, d)},
        conditions =
            regularUpConditions $\cup$
            regularDownConditions $\cup$
            steelUpConditions $\cup$ steelDownConditions
    **in**
```

```
          TS.mk_TransitionRule(
              T.and(
                  {T.neg(T.literal(TS.id(idle))),
                   T.neg(T.or(conditions))}),
              ⟨TS.mk_Assignment(TS.id(idle), T.True)⟩)
      end
    pre SIS.isWfStaticInterlockingSystem(sis)
end
```

A.8 StaticInterlockingSystemToConfidenceConditions

```
context: T, TS, D, CF, SIS, SIStoTS
scheme StaticInterlockingSystemToConfidenceConditions =
  class
    value
      /* calculates the confidence conditions of the
      behavioural semantics
      (a transition system) of
      a static interlocking system */
      makeConfidenceConditions :
        SIS.StaticInterlockingSystem $\overset{\sim}{\rightarrow}$ T.LTLassertion$^*$
      makeConfidenceConditions(sis) as cc post
        T.G(T.F(T.B(T.literal(TS.id(SIStoTS.idle))))) ∈
          elems cc ∧
        T.X(T.B(T.literal(TS.id(SIStoTS.idle)))) ∈
          elems cc ∧
        (∀ d : D.Diagram, id : T.Id •
           d ∈ SIS.diagrams(sis) ∧
           id ∈ SIS.internalRelayIds(sis) ∧
           D.isRegularRelay(id, d) ⇒
             let
               canDraw = CF.canDrawRegularRelay(id, d),
               canDrop = CF.canDropRegularRelay(id, d)
             in
               elems trueUntilChangeLTL(id, canDraw, canDrop) ⊆
                 elems cc
             end) ∧
        (∀ d : D.Diagram, id : T.Id •
           d ∈ SIS.diagrams(sis) ∧
           id ∈ SIS.internalRelayIds(sis) ∧
           D.isSteelRelay(id, d) ⇒
             let
               canDraw = CF.canDrawSteelRelay(id, d),
               canDrop = CF.canDropSteelRelay(id, d),
               drawingCurrent =
                 CF.drawingCurrentThroughSteelRelay(id, d),
               droppingCurrent =
                 CF.droppingCurrentThroughSteelRelay(id, d)
             in
               elems trueUntilChangeLTL(id, canDraw, canDrop) ⊆
                 elems cc ∧
               mutualExclusionLTL(
                   droppingCurrent, drawingCurrent) ∈
                 elems cc
             end)
      pre SIS.isWfStaticInterlockingSystem(sis),

      /*defines the 2 true concurrency ltl assertions
      for an id*/
      trueUntilChangeLTL :
        T.Id × T.BooleanExp × T.BooleanExp →
          T.LTLassertion*
      trueUntilChangeLTL(id, upGuard, downGuard) ≡
        ⟨T.G(
             T.Imply(
                   T.B(upGuard),
                   T.X(
                       T.Imply(
                             T.B(T.neg(upGuard)),
                             T.B(T.literal(id)))))),
          T.G(
             T.Imply(
                   T.B(downGuard),
                   T.X(
                       T.Imply(
                             T.B(T.neg(downGuard)),
                             T.B(T.neg(T.literal(id)))))))⟩,

      /*defines the mutual exclusion rule : the two
      BooleanExp cannot be true at the same time*/
      mutualExclusionLTL :
        T.BooleanExp × T.BooleanExp → T.LTLassertion
      mutualExclusionLTL(b1, b2) ≡
        T.G(T.B(T.neg(T.and({b1, b2}))))
```

end

A.9 The objects

context: Types
object T : Types

context: Diagrams
object D : Diagrams

context: StaticInterlockingSystem
object SIS : StaticInterlockingSystem

context: Pathfinding
object PF : Pathfinding

context: Conditionfinding
object CF : Conditionfinding

context: TransitionSystem
object TS : TransitionSystem

context: StaticInterlockingSystemToTransitionSystem
object SIStoTS : StaticInterlockingSystemToTransitionSystem

Appendix B

The concrete RSL model

This appendix contains the schemes introduced by chapter 9. The schemes correspond to the boxes marked with *C*, *D* and *E* in figure 4.2, page 48.

B.1 Types

```
scheme Types =
  class
    type
      /∗ identifier∗/
      Id = Text,
      /∗ a boolean expression ∗/
      BooleanExp ==
        and(a : BooleanExp-set) |
        or(o : BooleanExp-set) |
        neg(n : BooleanExp) |
        literal(id : Id),
      /∗ a boolean value∗/
      Boolean == True | False,
      /∗ a state of a relay∗/
      State == up | down,
      /∗ Added types ∗/
      /∗ Edge for connection the components in a diagram
      ∗/
      Edge = Id × Id,
      /∗ There must be two poles in a diagram ∗/
      Pole :: getId : Id,
      Poles :: plus : Pole   minus : Pole,
      /∗ For the introduction of a branch in the diagram
      ∗/
      Junction :: getId : Id,
      Contact ::
        /∗ The Id of the contact∗/
        getId : Id
        /∗ The Id of the relay that is controlling the
        contact∗/
        getRelayId : Id
        /∗ The required state of the controlling relay
        for having the
        contact closed ∗/
        getRelayState : State,
      Button :: getId : Id,
      /∗ A regular relay in a graph ∗/
      RegularRelay ::
        /∗ The Id of the relay ∗/
        getId : Id
        /∗ The initial state of the relay ∗/
        getInitState : State,
      /∗ A Steel Core Relay ∗/
      SteelRelay ::
        /∗ The Id of the relay ∗/
        getId : Id
        /∗ The initial state of the relay∗/
        getInitState : State
        /∗ A steel relay will be connected by edges to
        3 other components .
        The following Ids are Ids of these components.
        If there is current through the relay
        from up to minus, the relay will be drawn. If
        there is current from down to minus, the
        relay will be dropped. ∗/
        getUp : Id   getDown : Id   getMinus : Id,
      /∗ components in a diagram ∗/
      Component =
        Pole | Junction | RegularRelay | SteelRelay |
        Button | Contact,
    /* external relay in a StaticInterlockingSystem
    */
    ExternalRelay :: getId : Id   getInitState : State,
    /* type for LTL assertions */
    LTLassertion ==
        G(g : LTLassertion) |
        F(f : LTLassertion) |
        X(x : LTLassertion) |
        Imply(lhs : LTLassertion, rhs : LTLassertion) |
        B(b : BooleanExp)

value
    /* auxiliary functions */
    /* gets all the ids that are inside a given
    boolean expression */
    idsInBoolExp : BooleanExp → Id-set
    idsInBoolExp(exp) ≡
        case exp of
            literal(l) → {l},
            neg(nexp) → idsInBoolExp(nexp),
            and(aset) → idsInBoolExpSet(aset),
            or(oset) → idsInBoolExpSet(oset)
        end,

    /* gets all the ids in a set of boolean expressions
    */
    idsInBoolExpSet : BooleanExp-set → Id-set
    idsInBoolExpSet(set) ≡
        if set = {} then {}
        else
            let head = hd set in
                idsInBoolExp(head) ∪
                idsInBoolExpSet(set \ {head})
            end
        end,

    /* gets the ID of a component */
    idFromComponent : Component → Id
    idFromComponent(c) ≡
        case c of
            mk_Pole(_) → getId(Component_to_Pole(c)),
            mk_Junction(_) → getId(Component_to_Junction(c)),
            mk_RegularRelay(_) →
            getId(Component_to_RegularRelay(c)),
          mk_SteelRelay(_) →
            getId(Component_to_SteelRelay(c)),
          mk_Button(_) → getId(Component_to_Button(c)),
          mk_Contact(_) → getId(Component_to_Contact(c))
        end
  end
```

B.2 DiagramsL

```
context: T
scheme DiagramsL =
  class
    type
      /∗ the components of a diagram ∗/
      Components ::
        getPoles : T.Poles
        getContacts : T.Contact*
        getButtons : T.Button*
        getRegularRelays : T.RegularRelay*
        getSteelRelays : T.SteelRelay*
        getJunctions : T.Junction*,
      /∗ A diagram corresponding to one circuit ∗/
      Diagram ::
        /∗ All the components in the diagram ∗/
        getComponents : Components
        /∗ All the edges between the components in the
        diagram∗/
        getEdges : T.Edge*

    value
      /∗ observer functions ∗/
      /∗ true if the ID is a positive pole in the diagram,
      otherwise false ∗/
      isPlus : T.Id × Diagram → Bool
      isPlus(id, d) ≡
        T.getId(T.plus(getPoles(getComponents(d)))) = id,

      /∗ true if the ID is a negative pole in the diagram,
      otherwise false ∗/
```

```
isMinus : T.Id × Diagram → Bool
isMinus(id, d) ≡
   T.getId(T.minus(getPoles(getComponents(d)))) = id,

/∗ true if the ID is a regular relay in the diagram,
otherwise false ∗/
isRegularRelay : T.Id × Diagram → Bool
isRegularRelay(id, d) ≡
   (∃ rr : T.RegularRelay •
      rr ∈ getRegularRelays(getComponents(d)) ∧
      T.getId(rr) = id),

/∗ true if the ID is a steel relay in the diagram,
otherwise false ∗/
isSteelRelay : T.Id × Diagram → Bool
isSteelRelay(id, d) ≡
   (∃ sr : T.SteelRelay •
      sr ∈ getSteelRelays(getComponents(d)) ∧
      T.getId(sr) = id),

/∗ true if the ID is a contact in the diagram,
otherwise false ∗/
isContact : T.Id × Diagram → Bool
isContact(id, d) ≡
   (∃ c : T.Contact •
      c ∈ getContacts(getComponents(d)) ∧
      T.getId(c) = id),

/∗ true if the ID is a button in the diagram,
otherwise false ∗/
isButton : T.Id × Diagram → Bool
isButton(id, d) ≡
   (∃ b : T.Button •
      b ∈ getButtons(getComponents(d)) ∧
      T.getId(b) = id),

/∗ true if the ID is a junction in the diagram,
otherwise false ∗/
isJunction : T.Id × Diagram → Bool
isJunction(id, d) ≡
   (∃ j : T.Junction •
      j ∈ getJunctions(getComponents(d)) ∧
      T.getId(j) = id),
/* observer functions for steel relays */
/* if the steel relay is receiving current
from the returned neighbour of this function,
it can be drawn  */
upRelation : T.Id × Diagram $\xrightarrow{\sim}$ T.Id
upRelation(id, d) ≡
   T.getUp(
       **hd** {sr |
              sr : T.SteelRelay •
                 sr ∈
                    **elems** getSteelRelays(getComponents(d))})
**pre** isSteelRelay(id, d),

/* if the steel relay is receiving current
from the returned neighbour of this function,
it can be dropped  */
downRelation : T.Id × Diagram $\xrightarrow{\sim}$ T.Id
downRelation(id, d) ≡
   T.getDown(
       **hd** {sr |
              sr : T.SteelRelay •
                 sr ∈
                    **elems** getSteelRelays(getComponents(d))})
**pre** isSteelRelay(id, d),

/* gives a neighbour of a steel relay
from which the steel relay cannot receive
current */
minusRelation : T.Id × Diagram $\xrightarrow{\sim}$ T.Id
minusRelation(id, d) ≡
   T.getMinus(
       **hd** {sr |
              sr : T.SteelRelay •
                 sr ∈
                    **elems** getSteelRelays(getComponents(d))})
**pre** isSteelRelay(id, d),

/* observer function for regular relays and steel
relays*/
/* gives the initial state of a relay in a given
diagram*/
relayState : T.Id × Diagram $\xrightarrow{\sim}$ T.State
relayState(id, d) ≡
```

```
  case
    (hd {c |
          c : T.Component •
            c ∈
              elems getSteelRelays(getComponents(d)) ∪
              elems getRegularRelays(getComponents(d)) ∧
            T.idFromComponent(c) = id})
  of
    T.mk_RegularRelay(_, state) → state,
    T.mk_SteelRelay(_, state, _, _, _) → state
  end
pre isSteelRelay(id, d) ∨ isRegularRelay(id, d),

/∗ observer functions for contacts ∗/
/∗ for a contact in a given diagram, the function
gives the relay that controls the contact ∗/
relayIdForContact : T.Id × Diagram ∼→ T.Id
relayIdForContact(id, d) ≡
  T.getRelayId(
     hd {ct |
          ct : T.Contact •
            ct ∈ elems getContacts(getComponents(d))}
  )
pre isContact(id, d),

/∗  For a contact in a given diagram, the function
gives the required state of the relay that controls
the contact
for the contact to be closed ∗/
relayStateForContact : T.Id × Diagram ∼→ T.State
relayStateForContact(id, d) ≡
  T.getRelayState(
     hd {ct |
          ct : T.Contact •
            ct ∈ elems getContacts(getComponents(d))}
  )
pre isContact(id, d),

/∗ true if the two ids are neighbours in the given
diagram,
otherwise false  ∗/
areNeighbours : T.Id × T.Id × Diagram → Bool
areNeighbours(id1, id2, d) ≡
```

```
    (id1, id2) ∈ elems getEdges(d) ∨
    (id2, id1) ∈ elems getEdges(d),

/∗ auxiliary functions ∗/
/∗ gives all the neighbours of a given id in
a given diagram ∗/
neighboursOf : T.Id × Diagram → T.Id-set
neighboursOf(id1, d) ≡
    {id2 |
     id2 : T.Id •
        id2 ∈ allIds(d) ∧ areNeighbours(id1, id2, d)},

/∗ Gives all the ids of a given diagram ∗/
allIds : Diagram → T.Id-set
allIds(d) ≡
    let
        componentSet =
            {T.plus(getPoles(getComponents(d)))} ∪
            {T.minus(getPoles(getComponents(d)))} ∪
            (elems getRegularRelays(getComponents(d))) ∪
            (elems getSteelRelays(getComponents(d))) ∪
            (elems getContacts(getComponents(d))) ∪
            (elems getButtons(getComponents(d))) ∪
            (elems getJunctions(getComponents(d)))
    in
        {T.idFromComponent(c) |
         c : T.Component • c ∈ componentSet}
    end,

/∗ well formed functions for diagrams ∗/
/∗ true if a diagram is well formed∗/
isWfDiagram : Diagram → Bool
isWfDiagram(d) ≡
    okNeighbourRelation(d) ∧ okNumberOfNeighbours(d) ∧
    twoPoles(d) ∧ noIdOverlaps(d) ∧
    okSteelRelayRelations(d) ∧
    /∗ extra checks for DiagramL ∗/
    isWfEdges(getEdges(d)) ∧
    isWfComponents(getComponents(d)),

/∗ an id cannot be neighbour to it self
and the areNeighbours function is symmetric
∗/
okNeighbourRelation : Diagram → Bool
```

```
okNeighbourRelation(d) ≡
   (∀ id1, id2 : T.Id •
       id1 ∈ allIds(d) ∧ id2 ∈ allIds(d) ⇒
          ((areNeighbours(id1, id2, d) ⇒ id1 ≠ id2) ∧
            (areNeighbours(id1, id2, d) ⇒
                areNeighbours(id2, id1, d)))),

/∗ checks that each id has the required number
of neighbours ∗/
okNumberOfNeighbours : Diagram → **Bool**
okNumberOfNeighbours(d) ≡
   (∀ id : T.Id •
       id ∈ allIds(d) ⇒
          (
          /∗ a positive pole has minimum 1 neighbour ∗/
          (isPlus(id, d) ⇒ card neighboursOf(id, d) ≥ 1) ∧
          /∗ a negative pole has minimum 1 neighbour ∗/
          (isMinus(id, d) ⇒
                card neighboursOf(id, d) ≥ 1) ∧
          /∗ a regular relay has exactly 2 neighbours ∗/
          (isRegularRelay(id, d) ⇒
                card neighboursOf(id, d) = 2) ∧
          /∗ a contact has exactly 2 neighbours ∗/
          (isContact(id, d) ⇒
                card neighboursOf(id, d) = 2) ∧
          /∗ a button has exactly 2 neighbours ∗/
          (isButton(id, d) ⇒
                card neighboursOf(id, d) = 2) ∧
          /∗ a steel relay has exactly 3 neighbours ∗/
          (isSteelRelay(id, d) ⇒
                card neighboursOf(id, d) = 3) ∧
          /∗ a junction has exactly 3 neighbours ∗/
          (isJunction(id, d) ⇒
                card neighboursOf(id, d) = 3))),

/∗ checks that there is exactly one positive pole,
one negative pole ∗/
twoPoles : Diagram → Bool
twoPoles(d) ≡
   let
      plus =
         {id |
          id : T.Id • id ∈ allIds(d) ∧ isPlus(id, d)},
      minus =
            {id |
             id : T.Id •
                id ∈ allIds(d) ∧ isMinus(id, d)}
    in
        card plus = 1 ∧ card minus = 1
    end,

/* checks that the ids are not overlapping,
e.g. a junction cannot be a pole, a pole cannot
be a contact etc.*/
noIdOverlaps : Diagram → Bool
noIdOverlaps(d) ≡
    let
        plus =
            {id |
             id : T.Id • id ∈ allIds(d) ∧ isPlus(id, d)},
        minus =
            {id |
             id : T.Id •
                id ∈ allIds(d) ∧ isMinus(id, d)},
        regularRelays =
            {id |
             id : T.Id •
                id ∈ allIds(d) ∧ isRegularRelay(id, d)},
        steelRelays =
            {id |
             id : T.Id •
                id ∈ allIds(d) ∧ isSteelRelay(id, d)},
        contacts =
            {id |
             id : T.Id •
                id ∈ allIds(d) ∧ isContact(id, d)},
        buttons =
            {id |
             id : T.Id •
                id ∈ allIds(d) ∧ isButton(id, d)},
        junctions =
            {id |
             id : T.Id •
                id ∈ allIds(d) ∧ isJunction(id, d)}
    in
        card plus + card minus + card regularRelays +
        card steelRelays + card contacts + card buttons +
        card junctions = card allIds(d)
    end,

/∗ checks for each steel relay that the set of
steel relay relations equals the set of neighbours
∗/
okSteelRelayRelations : Diagram → Bool
okSteelRelayRelations(d) ≡
    (∀ id : T.Id •
        id ∈ allIds(d) ⇒
            (isSteelRelay(id, d) ⇒
                {upRelation(id, d), downRelation(id, d),
                 minusRelation(id, d)} = neighboursOf(id, d))
    ),

/∗ Checks that the edges do not contain duplicates
and that
(id2,id1) is not an edge if (id1,id2) is an edge
∗/
isWfEdges : T.Edge* → Bool
isWfEdges(el) ≡
    (∀ (id1, id2) : T.Edge •
        (id1, id2) ∈ elems el ⇒
            (id2, id1) ∉ elems el) ∧
    len el = card elems el,

/∗ checks that the components do not
contain duplicates ∗/
isWfComponents : Components → Bool
isWfComponents(cs) ≡
    len getContacts(cs) = card elems getContacts(cs) ∧
    (∀ ct1, ct2 : T.Contact •
        ct1 ∈ elems getContacts(cs) ∧
        ct2 ∈ elems getContacts(cs) ∧
        T.getId(ct1) = T.getId(ct2) ⇒ ct1 = ct2) ∧
    len getButtons(cs) = card elems getButtons(cs) ∧
    (∀ b1, b2 : T.Button •
        b1 ∈ elems getButtons(cs) ∧
        b2 ∈ elems getButtons(cs) ∧
        T.getId(b1) = T.getId(b2) ⇒ b1 = b2) ∧
    len getRegularRelays(cs) =
        card elems getRegularRelays(cs) ∧
    (∀ rr1, rr2 : T.RegularRelay •
        rr1 ∈ elems getRegularRelays(cs) ∧
        rr2 ∈ elems getRegularRelays(cs) ∧
```

```
            T.getId(rr1) = T.getId(rr2) ⇒ rr1 = rr2) ∧
        len getSteelRelays(cs) =
           card elems getSteelRelays(cs) ∧
        (∀ sr1, sr2 : T.SteelRelay •
            sr1 ∈ elems getSteelRelays(cs) ∧
            sr2 ∈ elems getSteelRelays(cs) ∧
            T.getId(sr1) = T.getId(sr2) ⇒ sr1 = sr2)
end
```

B.3 StaticInterlockingSystemL

```
context: T, DL
scheme StaticInterlockingSystemL =
   class
      type
         StaticInterlockingSystem ::
            getDiagrams : DL.Diagram*
            getExternal : T.ExternalRelay*

      value
         /* observer functions */
         /* gives the diagrams of a static interlocking
         system*/
         diagrams : StaticInterlockingSystem → DL.Diagram-set
         diagrams(sis) ≡ elems getDiagrams(sis),

         /* gives the external relay ids of a static interlocking
         system */
         externalRelayIds :
            StaticInterlockingSystem → T.Id-set
         externalRelayIds(sis) ≡
            {T.getId(er) |
             er : T.ExternalRelay •
               er ∈ elems getExternal(sis)},

         /* gives the initial state of an external relay
         in a given
         static interlocking system */
         externalRelayState :
            T.Id × StaticInterlockingSystem ~→ T.State
```

```
externalRelayState(id, sis) ≡
   T.getInitState(
       hd {er |
             er : T.ExternalRelay •
                er ∈ elems getExternal(sis) ∧
                T.getId(er) = id})
pre id ∈ externalRelayIds(sis),

/∗ auxiliary functions ∗/
/∗ gives all the internal relay ids of a
static interlocking system ∗/
internalRelayIds :
   StaticInterlockingSystem → T.Id-set
internalRelayIds(sis) ≡
   {id |
    id : T.Id •
       (∃ d : DL.Diagram •
             d ∈ diagrams(sis) ∧
             (DL.isRegularRelay(id, d) ∨
               DL.isSteelRelay(id, d)))},

/∗ gives all the relay ids of a static interlocking
system∗/
allRelayIds : StaticInterlockingSystem → T.Id-set
allRelayIds(sis) ≡
   internalRelayIds(sis) ∪ externalRelayIds(sis),

/∗ well formed functions∗/
/∗ checks that a static interlocking system
is well formed ∗/
isWfStaticInterlockingSystem :
   StaticInterlockingSystem → Bool
isWfStaticInterlockingSystem(sis) ≡
   (∀ d : DL.Diagram •
       d ∈ diagrams(sis) ⇒ DL.isWfDiagram(d)) ∧
   uniqueIds(sis) ∧ contactsHaveRelays(sis) ∧
   /∗ Extra checks for StaticInterlockingSystemL
   ∗/
   noDuplicateDiagrams(getDiagrams(sis)) ∧
   noDuplicateExternal(getExternal(sis)),

/∗ checks that the ids are unique∗/
uniqueIds : StaticInterlockingSystem → Bool
uniqueIds(sis) ≡
```

```
          /* the ids of two different diagrams cannot overlap
          */
          (∀ d1, d2 : DL.Diagram •
               d1 ∈ diagrams(sis) ∧ d2 ∈ diagrams(sis) ⇒
                  (d1 ≠ d2 ⇒
                        DL.allIds(d1) ∩ DL.allIds(d2) = {})) ∧
          /* the ids of a diagram and the external ids cannot
          overlap
          */
          (∀ d : DL.Diagram •
               d ∈ diagrams(sis) ⇒
                  DL.allIds(d) ∩ externalRelayIds(sis) = {}),

      /* for each contact in a diagram, the relay
      that controls the contact must be defined */
      contactsHaveRelays : StaticInterlockingSystem → Bool
      contactsHaveRelays(sis) ≡
          (∀ d : DL.Diagram, id : T.Id •
               d ∈ diagrams(sis) ∧ id ∈ DL.allIds(d) ∧
               DL.isContact(id, d) ⇒
                  DL.relayIdForContact(id, d) ∈
                     allRelayIds(sis)),

      /* The diagram list does not contain duplicates
      */
      noDuplicateDiagrams : DL.Diagram* → Bool
      noDuplicateDiagrams(dl) ≡ card elems dl = len dl,

      /* The external relay list does not contain duplicates
      */
      noDuplicateExternal : T.ExternalRelay* → Bool
      noDuplicateExternal(erl) ≡
          card elems erl = len erl ∧
          (∀ er1, er2 : T.ExternalRelay •
               er1 ∈ elems erl ∧ er2 ∈ elems erl ∧
               T.getId(er1) = T.getId(er2) ⇒ er1 = er2)
end
```

B.4 Diagrams

```
context: T
scheme Diagrams =
  class
    type
      /∗ poles can be plus or minus ∗/
      PoleType == plus | minus,
      Diagram ::
        getComponentMap : T.Id $\overrightarrow{m}$ T.Component
        getEdges : T.Edge-set
        /∗ Information on the type of the poles ∗/
        getPoleType : T.Id $\overrightarrow{m}$ PoleType

    value
      /∗ observer functions ∗/
      /∗ true if the ID is a positive pole in the diagram,
      otherwise false ∗/
      isPlus : T.Id × Diagram → Bool
      isPlus(id, d) ≡
        id ∈ dom getPoleType(d) ∧
        getPoleType(d)(id) = plus,

      /∗ true if the ID is a negative pole in the diagram,
      otherwise false ∗/
      isMinus : T.Id × Diagram → Bool
      isMinus(id, d) ≡
        id ∈ dom getPoleType(d) ∧
        getPoleType(d)(id) = minus,

      /∗ true if the ID is a regular relay in the diagram,
      otherwise false ∗/
      isRegularRelay : T.Id × Diagram → Bool
      isRegularRelay(id, d) ≡
        id ∈ dom getComponentMap(d) ∧
        case getComponentMap(d)(id) of
          T.mk_RegularRelay(_) → true,
          _ → false
        end,

      /∗ true if the ID is a steel relay in the diagram,
      otherwise false ∗/
      isSteelRelay : T.Id × Diagram → Bool
```

```
isSteelRelay(id, d) ≡
   id ∈ dom getComponentMap(d) ∧
   case getComponentMap(d)(id) of
      T.mk_SteelRelay(_) → true,
      _ → false
   end,

/∗ true if the ID is a contact in the diagram,
otherwise false ∗/
isContact : T.Id × Diagram → Bool
isContact(id, d) ≡
   id ∈ dom getComponentMap(d) ∧
   case getComponentMap(d)(id) of
      T.mk_Contact(_) → true,
      _ → false
   end,

/∗ true if the ID is a button in the diagram,
otherwise false ∗/
isButton : T.Id × Diagram → Bool
isButton(id, d) ≡
   id ∈ dom getComponentMap(d) ∧
   case getComponentMap(d)(id) of
      T.mk_Button(_) → true,
      _ → false
   end,

/∗ true if the ID is a junction in the diagram,
otherwise false ∗/
isJunction : T.Id × Diagram → Bool
isJunction(id, d) ≡
   id ∈ dom getComponentMap(d) ∧
   case getComponentMap(d)(id) of
      T.mk_Junction(_) → true,
      _ → false
   end,

/∗ observer functions for steel relays ∗/
/∗ if the steel relay is receiving current
from the returned neighbour of this function,
it can be drawn  ∗/
upRelation : T.Id × Diagram ~→ T.Id
upRelation(id, d) ≡
```

```
   T.getUp(
       T.Component_to_SteelRelay(getComponentMap(d)(id)))
pre isSteelRelay(id, d),

/* if the steel relay is receiving current
from the returned neighbour of this function,
it can be dropped  */
downRelation : T.Id × Diagram $\overset{\sim}{\rightarrow}$ T.Id
downRelation(id, d) ≡
   T.getDown(
       T.Component_to_SteelRelay(getComponentMap(d)(id)))
pre isSteelRelay(id, d),

/* gives a neighbour of a steel relay
from which the steel relay cannot receive
current */
minusRelation : T.Id × Diagram $\overset{\sim}{\rightarrow}$ T.Id
minusRelation(id, d) ≡
   T.getMinus(
       T.Component_to_SteelRelay(getComponentMap(d)(id)))
pre isSteelRelay(id, d),

/* observer function for regular relays and steel
relays*/
/* gives the initial state of a relay in a given
diagram*/
relayState : T.Id × Diagram $\overset{\sim}{\rightarrow}$ T.State
relayState(id, d) ≡
   case getComponentMap(d)(id) of
      T.mk_RegularRelay(_) →
         T.getInitState(
             T.Component_to_RegularRelay(
                 getComponentMap(d)(id))),
      T.mk_SteelRelay(_) →
         T.getInitState(
             T.Component_to_SteelRelay(
                 getComponentMap(d)(id)))
   end
pre isSteelRelay(id, d) ∨ isRegularRelay(id, d),

/* observer functions for contacts */
/* for a contact in a given diagram, the function
gives the relay that controls the contact */
```

```
relayIdForContact : T.Id × Diagram $\xrightarrow{\sim}$ T.Id
relayIdForContact(id, d) ≡
   **case** getComponentMap(d)(id) **of**
      T.mk_Contact(_) →
         T.getRelayId(
             T.Component_to_Contact(getComponentMap(d)(id))
         )
   **end**
**pre** isContact(id, d),

/∗  For a contact in a given diagram, the function
gives the required state of the relay that controls
the contact
for the contact to be closed ∗/
relayStateForContact : T.Id × Diagram $\xrightarrow{\sim}$ T.State
relayStateForContact(id, d) ≡
   **case** getComponentMap(d)(id) **of**
      T.mk_Contact(_) →
         T.getRelayState(
             T.Component_to_Contact(getComponentMap(d)(id))
         )
   **end**
**pre** isContact(id, d),

/∗ true if the two ids are neighbours in the given
diagram,
otherwise false  ∗/
areNeighbours : T.Id × T.Id × Diagram → **Bool**
areNeighbours(id1, id2, d) ≡
   (id1, id2) ∈ getEdges(d) ∨
   (id2, id1) ∈ getEdges(d),

/∗ auxiliary functions ∗/
/∗ gives all the neighbours of a given id in
a given diagram ∗/
neighboursOf : T.Id × Diagram → T.Id-**set**
neighboursOf(id1, d) ≡
   {id2 |
    id2 : T.Id •
       id2 ∈ allIds(d) ∧ areNeighbours(id1, id2, d)},

/∗ Gives all the ids of a given diagram ∗/
allIds : Diagram → T.Id-**set**
```

```
allIds(d) ≡ dom getComponentMap(d),

/∗ well formed functions for diagrams ∗/
/∗ true if a diagram is well formed∗/
isWfDiagram : Diagram → Bool
isWfDiagram(d) ≡
   okNeighbourRelation(d) ∧ okNumberOfNeighbours(d) ∧
   /∗ the following check is an extra check for the
   concrete Diagram ∗/
   polesHaveType(d) ∧ twoPoles(d) ∧
   noIdOverlaps(d) ∧ okSteelRelayRelations(d) ∧
   /∗ Extra checks for concrete Diagram ∗/
   idsMatchComponents(getComponentMap(d)) ∧
   isWfEdges(getEdges(d)),

/∗ an id cannot be neighbour to it self
and the areNeighbours function is symmetric
∗/
okNeighbourRelation : Diagram → Bool
okNeighbourRelation(d) ≡
   (∀ id1, id2 : T.Id •
       id1 ∈ allIds(d) ∧ id2 ∈ allIds(d) ⇒
           ((areNeighbours(id1, id2, d) ⇒ id1 ≠ id2) ∧
             (areNeighbours(id1, id2, d) ⇒
                 areNeighbours(id2, id1, d)))),

/∗ checks that each id has the required number
of neighbours ∗/
okNumberOfNeighbours : Diagram → Bool
okNumberOfNeighbours(d) ≡
   (∀ id : T.Id •
       id ∈ allIds(d) ⇒
           (
           /∗ a positive pole has minimum 1 neighbour ∗/
           (isPlus(id, d) ⇒ card neighboursOf(id, d) ≥ 1) ∧
           /∗ a negative pole has minimum 1 neighbour ∗/
           (isMinus(id, d) ⇒
                 card neighboursOf(id, d) ≥ 1) ∧
           /∗ a regular relay has exactly 2 neighbours ∗/
           (isRegularRelay(id, d) ⇒
                 card neighboursOf(id, d) = 2) ∧
           /∗ a contact has exactly 2 neighbours ∗/
           (isContact(id, d) ⇒
                 card neighboursOf(id, d) = 2) ∧
```

```
            /∗ a button has exactly 2 neighbours ∗/
            (isButton(id, d) ⇒
                  card neighboursOf(id, d) = 2) ∧
            /∗ a steel relay has exactly 3 neighbours ∗/
            (isSteelRelay(id, d) ⇒
                  card neighboursOf(id, d) = 3) ∧
            /∗ a junction has exactly 3 neighbours ∗/
            (isJunction(id, d) ⇒
                  card neighboursOf(id, d) = 3))),

/∗ checks that there is exactly one positive pole,
one negative pole ∗/
twoPoles : Diagram → Bool
twoPoles(d) ≡
   let
      plus =
         {id |
          id : T.Id • id ∈ allIds(d) ∧ isPlus(id, d)},
      minus =
         {id |
          id : T.Id •
             id ∈ allIds(d) ∧ isMinus(id, d)}
   in
      card plus = 1 ∧ card minus = 1
   end,

/∗ checks that the ids are not overlapping,
e.g. a junction cannot be a pole, a pole cannot
be a contact etc.∗/
noIdOverlaps : Diagram → Bool
noIdOverlaps(d) ≡
   let
      plus =
         {id |
          id : T.Id • id ∈ allIds(d) ∧ isPlus(id, d)},
      minus =
         {id |
          id : T.Id •
             id ∈ allIds(d) ∧ isMinus(id, d)},
      regularRelays =
         {id |
          id : T.Id •
             id ∈ allIds(d) ∧ isRegularRelay(id, d)},
      steelRelays =
         {id |
          id : T.Id •
             id ∈ allIds(d) ∧ isSteelRelay(id, d)},
      contacts =
         {id |
          id : T.Id •
             id ∈ allIds(d) ∧ isContact(id, d)},
      buttons =
         {id |
          id : T.Id •
             id ∈ allIds(d) ∧ isButton(id, d)},
      junctions =
         {id |
          id : T.Id •
             id ∈ allIds(d) ∧ isJunction(id, d)}
   in
      card plus + card minus + card regularRelays +
      card steelRelays + card contacts + card buttons +
      card junctions = card allIds(d)
   end,

/* checks for each steel relay that the set of
steel relay relations equals the set of neighbours
*/
okSteelRelayRelations : Diagram → Bool
okSteelRelayRelations(d) ≡
   (∀ id : T.Id •
       id ∈ allIds(d) ⇒
          (isSteelRelay(id, d) ⇒
                  {upRelation(id, d), downRelation(id, d),
                   minusRelation(id, d)} = neighboursOf(id, d))
   ),

isWfEdges : T.Edge-set → Bool
isWfEdges(es) ≡
   (∀ (id1, id2) : T.Edge •
       (id1, id2) ∈ es ⇒ (id2, id1) ∉ es),

/* checks that each id in the map is mapped to
a component with the same id*/
idsMatchComponents : (T.Id →m T.Component) → Bool
idsMatchComponents(componentMap) ≡
   (∀ id : T.Id •
       id ∈ componentMap ⇒
```

```
                id = T.idFromComponent(componentMap(id))),

        /* checks that each pole has a type */
        polesHaveType : Diagram → Bool
        polesHaveType(d) ≡
            (∀ c : T.Component •
                c ∈ rng getComponentMap(d) ⇒
                    case c of
                        T.mk_Pole(id) → id ∈ dom getPoleType(d),
                        _ → true
                    end)
    end
```

B.5 StaticInterlockingSystem

```
context: T, D
scheme StaticInterlockingSystem =
    class
        type
            StaticInterlockingSystem ::
                getDiagrams : D.Diagram-set
                getExternal : T.Id →m T.ExternalRelay

        value
            /* observer functions */
            /* gives the diagrams of a static interlocking
            system*/
            diagrams : StaticInterlockingSystem → D.Diagram-set
            diagrams(sis) ≡ getDiagrams(sis),

            /* gives the external relay ids of a static interlocking
            system */
            externalRelayIds :
                StaticInterlockingSystem → T.Id-set
            externalRelayIds(sis) ≡ dom getExternal(sis),

            /* gives the initial state of an external relay
            in a given
            static interlocking system */
            externalRelayState :
```

```
    T.Id × StaticInterlockingSystem $\xrightarrow{\sim}$ T.State
externalRelayState(id, sis) ≡
    T.getInitState(getExternal(sis)(id))
**pre** id ∈ externalRelayIds(sis),

/∗ auxiliary functions ∗/
/∗ gives all the internal relay ids of a
static interlocking system ∗/
internalRelayIds :
    StaticInterlockingSystem → T.Id-**set**
internalRelayIds(sis) ≡
    {id |
     id : T.Id •
        (∃ d : D.Diagram •
                d ∈ diagrams(sis) ∧
                (D.isRegularRelay(id, d) ∨
                   D.isSteelRelay(id, d)))},

/∗ gives all the relay ids of a static interlocking
system∗/
allRelayIds : StaticInterlockingSystem → T.Id-**set**
allRelayIds(sis) ≡
    internalRelayIds(sis) ∪ externalRelayIds(sis),

/∗ well formed functions∗/
/∗ checks that a static interlocking system
is well formed ∗/
isWfStaticInterlockingSystem :
    StaticInterlockingSystem → **Bool**
isWfStaticInterlockingSystem(sis) ≡
    (∀ d : D.Diagram •
        d ∈ diagrams(sis) ⇒ D.isWfDiagram(d)) ∧
    uniqueIds(sis) ∧ contactsHaveRelays(sis) ∧
    /∗ Extra constraints added ∗/
    isWfExternalRelays(sis),

/∗ checks that the ids are unique∗/
uniqueIds : StaticInterlockingSystem → **Bool**
uniqueIds(sis) ≡
    /∗ the ids of two different diagrams cannot overlap
    ∗/
    (∀ d1, d2 : D.Diagram •
        d1 ∈ diagrams(sis) ∧ d2 ∈ diagrams(sis) ⇒
           (d1 ≠ d2 ⇒
                D.allIds(d1) ∩ D.allIds(d2) = {})) ∧
        /∗ the ids of a diagram and the external ids cannot
        overlap
        ∗/
        (∀ d : D.Diagram •
            d ∈ diagrams(sis) ⇒
                D.allIds(d) ∩ externalRelayIds(sis) = {}),

    /∗ for each contact in a diagram, the relay
    that controls the contact must be defined ∗/
    contactsHaveRelays : StaticInterlockingSystem → Bool
    contactsHaveRelays(sis) ≡
        (∀ d : D.Diagram, id : T.Id •
            d ∈ diagrams(sis) ∧ id ∈ D.allIds(d) ∧
            D.isContact(id, d) ⇒
                D.relayIdForContact(id, d) ∈
                    allRelayIds(sis)),

    /∗ Checks that the Ids in the domain of the externa
    relay map
    matches the components in the range of the map
    ∗/
    isWfExternalRelays : StaticInterlockingSystem → Bool
    isWfExternalRelays(sis) ≡
        (∀ id : T.Id •
            id ∈ dom getExternal(sis) ⇒
                id = T.getId(getExternal(sis)(id)))
end
```

B.6 StaticInterlockingSystemConversion

```
context: SIS, SISL, D, DL
scheme StaticInterlockingSystemConversion =
  class
    value
      /∗ converts a DL.Diagram to a D.Diagram ∗/
      convertDiagram : DL.Diagram $\xrightarrow{\sim}$ D.Diagram
      convertDiagram(dl) ≡
        let
          componentSet =
              {T.plus(DL.getPoles(DL.getComponents(dl)))} ∪
              {T.minus(DL.getPoles(DL.getComponents(dl)))} ∪
              (elems DL.getRegularRelays(DL.getComponents(dl))) ∪
              (elems DL.getSteelRelays(DL.getComponents(dl))) ∪
              (elems DL.getContacts(DL.getComponents(dl))) ∪
              (elems DL.getButtons(DL.getComponents(dl))) ∪
              (elems DL.getJunctions(DL.getComponents(dl))),
          componentMap =
              [ T.idFromComponent(c) ↦ c |
               c : T.Component • c ∈ componentSet ],
          edges = elems DL.getEdges(dl),
          plus = T.plus(DL.getPoles(DL.getComponents(dl))),
          minus = T.minus(DL.getPoles(DL.getComponents(dl))),
          poleTypeMap =
              [ T.getId(plus) ↦ D.plus,
               T.getId(minus) ↦ D.minus ]
        in
          D.mk_Diagram(componentMap, edges, poleTypeMap)
        end
      pre DL.isWfDiagram(dl),

      /∗ converts a SISL.StaticInterlockingSystem to
      a SIS.StaticInterlockingSystem∗/
      convertStaticInterlockingSystem :
        SISL.StaticInterlockingSystem $\xrightarrow{\sim}$
          SIS.StaticInterlockingSystem
      convertStaticInterlockingSystem(sisl) ≡
        let
          diagrams =
              {convertDiagram(dl) |
               dl : DL.Diagram •
                   dl ∈ SISL.getDiagrams(sisl)},
          external =
              [ T.getId(er) ↦ er |
               er : T.ExternalRelay •
                   er ∈ elems SISL.getExternal(sisl) ]
        in
          SIS.mk_StaticInterlockingSystem(diagrams, external)
        end
      pre SISL.isWfStaticInterlockingSystem(sisl)
  end
```

B.7 Pathfinding

```
context: T, D
scheme Pathfinding =
  class
    type
      Path = T.Id*,
      PoleIds :: getPlus : T.Id   getMinus : T.Id

    value
      /* well formed functions */
      /* checks if a path is well formed*/
      isWfPath : Path × D.Diagram $\overset{\sim}{\rightarrow}$ Bool
      isWfPath(p, d) ≡
        /* a path does not repeat ids */
        noDuplicates(p) ∧
        /* minimum 2 ids in a path*/
        len p ≥ 2 ∧
        /* a path starts with a positive pole
        and ends with a negative pole */
        D.isPlus(p(1), d) ∧ D.isMinus(p(len p), d) ∧
        /* (p(1),p(2)),...,(p(n−1),p(n)) are neighbours
        */
        (∀ n : Nat •
            n ∈ (inds p \ {len p}) ∧
            D.areNeighbours(p(n), p(n + 1), d)) ∧
        /* the up and down path of a steel relay
        cannot be in the same path at the same time
        */
        noSteelRelayProblem(p, d) pre D.isWfDiagram(d),

      /* checks that a path has no duplicate elements
      */
      noDuplicates : Path → Bool
      noDuplicates(p) ≡ card elems p = len p,

      /*  checks that the up and down part of a steel
      relay
      are not in the same path at the same time */
      noSteelRelayProblem : Path × D.Diagram $\overset{\sim}{\rightarrow}$ Bool
      noSteelRelayProblem(p, d) ≡
        (∀ id : T.Id •
            id ∈ elems p ∧ D.isSteelRelay(id, d) ⇒
            ~ (D.upRelation(id, d) ∈ elems p ∧
                D.downRelation(id, d) ∈ elems p))
pre
    D.isWfDiagram(d) ∧
    (∀ id : T.Id •
        id ∈ elems p ⇒ id ∈ D.allIds(d)),

/* for path computation */
/* true if a set of ids is contained by a path
*/
isPathFor : Path × T.Id-set → Bool
isPathFor(p, ids) ≡ ids ⊆ elems p,

/* gives all the paths through a given set of
ids
in a give diagram */
allPathsFor : T.Id-set × D.Diagram ~→ Path-set
allPathsFor(ids, d) ≡
    {p |
     p : Path •
        p ∈ makePathsBetweenPoles(d) ∧
        isPathFor(p, ids)}
pre D.isWfDiagram(d) ∧ ids ⊆ D.allIds(d),

/* Added functions */
makePoles : D.Diagram ~→ PoleIds
makePoles(d) ≡
    let
        plusSet =
            {id |
             id : T.Id •
                id ∈ D.allIds(d) ∧ D.isPlus(id, d)},
        minusSet =
            {id |
             id : T.Id •
                id ∈ D.allIds(d) ∧ D.isMinus(id, d)}
    in
        mk_PoleIds(hd plusSet, hd minusSet)
    end
pre D.isWfDiagram(d),

/* checks that a new possible path does not contain
duplicates or a steel relay problem */
```

```
legalPathExtension : Path × D.Diagram $\xrightarrow{\sim}$ Bool
legalPathExtension(p, d) ≡
   noDuplicates(p) ∧ noSteelRelayProblem(p, d)
pre D.isWfDiagram(d),

/∗ Gives the set of well formed paths in a diagram
∗/
makePathsBetweenPoles : D.Diagram $\xrightarrow{\sim}$ Path-set
makePathsBetweenPoles(d) ≡
   let poles = makePoles(d) in
      makePathsBetweenComponents(
          getMinus(poles), ⟨getPlus(poles)⟩, d)
   end
pre D.isWfDiagram(d),

/∗ tries to extend a path with each id in neighboursToBeVis
ted.
each legal extension will be returned ∗/
extendPath :
   T.Id × T.Id-set × Path × D.Diagram $\xrightarrow{\sim}$ Path-set
extendPath(
    endComponent, neighboursToBeVisited, currentPath, d) ≡
   if neighboursToBeVisited = {} then {}
   else
      let
         nextComponent = hd neighboursToBeVisited,
         nextPath = currentPath ^ ⟨nextComponent⟩
      in
         if legalPathExtension(nextPath, d)
         then
            makePathsBetweenComponents(
                endComponent, nextPath, d)
         else {}
         end ∪
         extendPath(
             endComponent,
             neighboursToBeVisited \ {nextComponent},
             currentPath, d)
      end
   end
pre
   D.isWfDiagram(d) ∧ endComponent ∈ D.allIds(d) ∧
   neighboursToBeVisited ⊆ D.allIds(d) ∧
```

```
        elems currentPath ⊆ D.allIds(d),

      /∗ makes all possible extensions of currentPath
      that are legal
      and has endComponent as the end of the path
      ∗/
      makePathsBetweenComponents :
        T.Id × Path × D.Diagram $\xrightarrow{\sim}$ Path-set
      makePathsBetweenComponents(
         endComponent, currentPath, d) ≡
        let
          currentComponent = currentPath(len currentPath)
        in
          if currentComponent = endComponent
          then {currentPath}
          else
            extendPath(
               endComponent,
               D.neighboursOf(currentComponent, d),
               currentPath, d)
          end
        end
      pre
        len currentPath > 0 ∧ D.isWfDiagram(d) ∧
        endComponent ∈ D.allIds(d) ∧
        elems currentPath ⊆ D.allIds(d)
  end
```

B.8 Conditionfinding

```
context: T, D, PF
scheme Conditionfinding =
  class
    value
      /∗ gives the boolean expression that is true iff
      there can be
      current throuh a give button or a given contact
      ∗/
      isConducting : T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
      isConducting(id, d) ≡
    **if** D.isButton(id, d) **then** T.literal(id)
    **else**
        **case** D.relayStateForContact(id, d) **of**
            T.up $\rightarrow$ T.literal(D.relayIdForContact(id, d)),
            T.down $\rightarrow$
                T.neg(T.literal(D.relayIdForContact(id, d)))
        **end**
    **end**
**pre**
    D.isWfDiagram(d) $\wedge$
    (D.isButton(id, d) $\vee$ D.isContact(id, d)),

/$*$ generation of boolean expressions $*$/
/$*$ gives the expression that is true iff
the path is conductive
(note: poles and relays are always conductive,
but buttons and contacts must be closed in order
to obtain conductivity)$*$/
isConducting : PF.Path $\times$ D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
isConducting(p, d) $\equiv$
    T.and(
        {isConducting(id, d) |
          id : T.Id •
              id $\in$ **elems** p $\wedge$
              (D.isButton(id, d) $\vee$ D.isContact(id, d))})
**pre** D.isWfDiagram(d) $\wedge$ PF.isWfPath(p, d),

/$*$ gives the boolean expression that is true iff
there is current through the given regular relay
$*$/
currentThroughRegularRelay :
    T.Id $\times$ D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
currentThroughRegularRelay(id, d) $\equiv$
    T.or(
        {isConducting(p, d) |
          p : PF.Path • p $\in$ PF.allPathsFor({id}, d)})
**pre** D.isWfDiagram(d) $\wedge$ D.isRegularRelay(id, d),

/$*$ gives the condition for not having current
through a given
regular relay$*$/
noCurrentThroughRegularRelay :
    T.Id $\times$ D.Diagram $\overset{\sim}{\rightarrow}$ T.BooleanExp
```

```
noCurrentThroughRegularRelay(id, d) ≡
   T.neg(currentThroughRegularRelay(id, d))
pre D.isWfDiagram(d) ∧ D.isRegularRelay(id, d),

/∗ gives the boolean expression that is true iff
there is no current through the given regular
relay ∗/
canDrawRegularRelay :
   T.Id × D.Diagram ~→ T.BooleanExp
canDrawRegularRelay(id, d) ≡
   T.and(
       {T.neg(T.literal(id)),
         currentThroughRegularRelay(id, d)})
pre D.isWfDiagram(d) ∧ D.isRegularRelay(id, d),

/∗ gives the boolean expression that is true iff
a given regular relay
can be dropped ∗/
canDropRegularRelay :
   T.Id × D.Diagram ~→ T.BooleanExp
canDropRegularRelay(id, d) ≡
   T.and(
       {T.literal(id),
         noCurrentThroughRegularRelay(id, d)})
pre D.isWfDiagram(d) ∧ D.isRegularRelay(id, d),

/∗ gives the boolean expression that is true iff
there is current through the given regular relay
that makes it draw
∗/
drawingCurrentThroughSteelRelay :
   T.Id × D.Diagram ~→ T.BooleanExp
drawingCurrentThroughSteelRelay(id, d) ≡
   T.or(
       {isConducting(p, d) |
         p : PF.Path •
            p ∈
               PF.allPathsFor(
                  {D.upRelation(id, d), id,
                    D.minusRelation(id, d)}, d)})
pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d),

/∗ gives the boolean expression that is true iff
```

```
      there is current through the given regular relay
      that makes it drop
      */
      droppingCurrentThroughSteelRelay :
         T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
      droppingCurrentThroughSteelRelay(id, d) ≡
         T.or(
            {isConducting(p, d) |
              p : PF.Path •
                 p ∈
                    PF.allPathsFor(
                       {D.downRelation(id, d), id,
                         D.minusRelation(id, d)}, d)})
      pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d),

      /* gives the boolean expression that is true iff
      a given steel relay
      can be drawn*/
      canDrawSteelRelay :
         T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
      canDrawSteelRelay(id, d) ≡
         T.and(
            {T.neg(T.literal(id)),
              drawingCurrentThroughSteelRelay(id, d)})
      pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d),

      /* gives the boolean expression that is true iff
      a given steel relay
      can be dropped */
      canDropSteelRelay :
         T.Id × D.Diagram $\xrightarrow{\sim}$ T.BooleanExp
      canDropSteelRelay(id, d) ≡
         T.and(
            {T.literal(id),
              droppingCurrentThroughSteelRelay(id, d)})
      pre D.isWfDiagram(d) ∧ D.isSteelRelay(id, d)
   end
```

B.9 TransitionSystem

```
context: T
scheme TransitionSystem =
   class
      type
         /∗ a variable in the state of a transition system
         ∗/
         Var :: id : T.Id   val : T.Boolean,

         /∗ an assignment in a transition rule.
         the Var with the given id in the current state
         will be assigned the new value ∗/
         Assignment :: id : T.Id   assign : T.Boolean,

         /∗ all the assignments in a transition rule ∗/
         MultipleAssignment = Assignment*,

         /∗ a transition rule has a guard and
         a multiple assignment ∗/
         TransitionRule ::
            guard : T.BooleanExp
            assignments : MultipleAssignment,

         /∗ a transition system has an initial state
         and some transition rules ∗/
         TransitionSystem ::
            state : Var*
            transitionRules : TransitionRule*

      value

         /∗ well formed functions ∗/
         /∗ checks that a transition system is well formed ∗/
         isWfTransitionSystem : TransitionSystem → Bool
         isWfTransitionSystem(ts) ≡
            isWfState(state(ts)) ∧
            areWfTransitionRules(state(ts), transitionRules(ts)),

         /∗ the variables in the state must have unique ids
            and duplicates are not allowed ∗/
         isWfState : Var* → Bool
         isWfState(state) ≡
```

```
        (∀ v1, v2 : Var •
            v1 ∈ state ∧ v2 ∈ state ∧ id(v1) = id(v2) ⇒
                v1 = v2)
         ∧
         card elems state = len state,

    /∗ checks that all the transition rules are well formed ∗/
    areWfTransitionRules :
        Var* × TransitionRule* → Bool
    areWfTransitionRules(state, transitionRules) ≡
        let
            /∗ all the ids that are defined in the state ∗/
            ids = {id(var) | var : Var • var ∈ elems state}
        in
            (∀ tr : TransitionRule •
                tr ∈ elems transitionRules ⇒
                    /∗ The ids in a guard must be defined in the state ∗/
                    T.idsInBoolExp(guard(tr)) ⊆ ids ∧
                    /∗ There must minimum be one assignment ∗/
                    len assignments(tr) > 0 ∧
                    /∗ the id in an assignment must be defined in the state ∗/
                    (∀ a : Assignment •
                          a ∈ assignments(tr) ⇒ id(a) ∈ ids) ∧
                    /∗ a variable cannot be assigned several values in
                    the same multiple assignment ∗/
                    (∀ a1, a2 : Assignment •
                          a1 ∈ elems assignments(tr) ∧
                          a2 ∈ elems assignments(tr) ∧
                          id(a1) = id(a2) ⇒ a1 = a2) ∧
                    card elems assignments(tr) = len assignments(tr))
        end

end
```

B.10 StaticInterlockingSystemToTransitionSystem

```
context: T, TS, D, CF, SIS
scheme StaticInterlockingSystemToTransitionSystem =
    class
        value
```

```
/∗ the identifier of the idle variable ∗/
idleId : T.Id = "idle",
/∗ the idle variable ∗/
idle : TS.Var = TS.mk_Var(idleId, T.False),

/∗ calculates the behavioural semantics
(a transition system) of
a static interlocking system ∗/
makeBehaviouralSemantics :
   SIS.StaticInterlockingSystem $\xrightarrow{\sim}$ TS.TransitionSystem
makeBehaviouralSemantics(sis) ≡
   TS.mk_TransitionSystem(
       makeState(sis), makeTransitionRules(sis))
pre SIS.isWfStaticInterlockingSystem(sis),

/∗ makes a list of Var that contains all ids combined
with b∗/
makeIdVars : T.Id-set × T.Boolean → TS.Var$^*$
makeIdVars(ids, b) ≡
   if ids = {} then ⟨⟩
   else
      let id = hd ids in
         ⟨TS.mk_Var(id, b)⟩ ⌢ makeIdVars(ids \ {id}, b)
      end
   end,

/∗ makes the vars of all the buttons in a diagram.
all the vars are false because buttons are not
pushed in the normal state ∗/
makeButtonVars : D.Diagram-set → TS.Var$^*$
makeButtonVars(ds) ≡
   if ds = {} then ⟨⟩
   else
      let
         d = hd ds,
         buttonIds =
            {id |
             id : T.Id •
                id ∈ D.allIds(d) ∧ D.isButton(id, d)}
      in
         makeIdVars(buttonIds, T.False) ⌢
         makeButtonVars(ds \ {d})
      end
```

```
    **end**
**pre**
    (∀ d : D.Diagram • d ∈ ds ⇒ D.isWfDiagram(d)),

/∗ Makes the vars of all the external relays
∗/
makeExternalRelayVars :
    T.Id-**set** × SIS.StaticInterlockingSystem $\xrightarrow{\sim}$
        TS.Var$^*$
makeExternalRelayVars(ids, sis) ≡
    **if** ids = {} **then** ⟨⟩
    **else**
        **let**
            id = **hd** ids,
            initState =
                **case** SIS.externalRelayState(id, sis) **of**
                    T.up → T.True,
                    T.down → T.False
                **end**
        **in**
            ⟨TS.mk_Var(id, initState)⟩ ⁀
            makeExternalRelayVars(ids \ {id}, sis)
        **end**
    **end**
**pre**
    SIS.isWfStaticInterlockingSystem(sis) ∧
    ids ⊆ SIS.externalRelayIds(sis),

/∗ makes the vars for all the buttons and all
the external relays in a static interlocking
system ∗/
makeExternalVars :
    SIS.StaticInterlockingSystem $\xrightarrow{\sim}$ TS.Var$^*$
makeExternalVars(sis) ≡
    makeButtonVars(SIS.diagrams(sis)) ⁀
    makeExternalRelayVars(SIS.externalRelayIds(sis), sis)
**pre** SIS.isWfStaticInterlockingSystem(sis),

/∗ makes the relay vars in a diagram ∗/
makeRelayVars : T.Id-**set** × D.Diagram $\xrightarrow{\sim}$ TS.Var$^*$
makeRelayVars(ids, d) ≡
    **if** ids = {} **then** ⟨⟩
    **else**
```

```
        let
          id = hd ids,
          initState =
            case D.relayState(id, d) of
              T.up → T.True,
              T.down → T.False
            end
        in
          ⟨TS.mk_Var(id, initState)⟩ ⌢
          makeRelayVars(ids \ {id}, d)
        end
      end
  pre
    D.isWfDiagram(d) ∧
    (∀ id : T.Id •
       id ∈ ids ⇒
         (D.isSteelRelay(id, d) ∨
           D.isRegularRelay(id, d))),

  /∗ makes the relay vars of all the diagrams in
  a diagram set∗/
  makeInternalRelayVars : D.Diagram-set ~→ TS.Var*
  makeInternalRelayVars(ds) ≡
    if ds = {} then ⟨⟩
    else
      let
        d = hd ds,
        relays =
          {id |
           id : T.Id •
             id ∈ D.allIds(d) ∧
             (D.isRegularRelay(id, d) ∨
               D.isSteelRelay(id, d))}
      in
        makeRelayVars(relays, d) ⌢
        makeInternalRelayVars(ds \ {d})
      end
    end
  pre
    (∀ d : D.Diagram • d ∈ ds ⇒ D.isWfDiagram(d)),

  /∗ makes the initial state of a transition system
  ∗/
```

```
makeState :
   SIS.StaticInterlockingSystem $\xrightarrow{\sim}$ TS.Var$^*$
makeState(sis) $\equiv$
   makeInternalRelayVars(SIS.diagrams(sis)) $^\frown$
   makeExternalVars(sis) $^\frown$ $\langle$idle$\rangle$
pre SIS.isWfStaticInterlockingSystem(sis),

/* gives all the transition rules from the
relays in a diagram that are also defined in
the set of ids */
rulesInSet :
   T.Id-set $\times$ D.Diagram $\xrightarrow{\sim}$ TS.TransitionRule$^*$
rulesInSet(ids, d) $\equiv$
   if ids = {} then $\langle\rangle$
   else
      let id = hd ids in
         if D.isRegularRelay(id, d)
         then
            $\langle$TS.mk_TransitionRule(
                  CF.canDrawRegularRelay(id, d),
                  $\langle$TS.mk_Assignment(id, T.True)$\rangle$),
               TS.mk_TransitionRule(
                  CF.canDropRegularRelay(id, d),
                  $\langle$TS.mk_Assignment(id, T.False)$\rangle$)$\rangle$
         else
            $\langle$TS.mk_TransitionRule(
                  CF.canDrawSteelRelay(id, d),
                  $\langle$TS.mk_Assignment(id, T.True)$\rangle$),
               TS.mk_TransitionRule(
                  CF.canDropSteelRelay(id, d),
                  $\langle$TS.mk_Assignment(id, T.False)$\rangle$)$\rangle$
         end $^\frown$ rulesInSet(ids \ {id}, d)
      end
   end
pre
   D.isWfDiagram(d) $\wedge$
   ($\forall$ id : T.Id •
      id $\in$ ids $\Rightarrow$
         (D.isSteelRelay(id, d) $\vee$
            D.isRegularRelay(id, d))),

/* makes all the internal transition rules from
the diagrams in a diagram set */
```

```
makeInternalTransitionRules :
   D.Diagram-set $\stackrel{\sim}{\rightarrow}$ TS.TransitionRule$^*$
makeInternalTransitionRules(ds) $\equiv$
   if ds = {} then $\langle\rangle$
   else
      let
         d = hd ds,
         relayIds =
            {id |
             id : T.Id •
                id $\in$ D.allIds(d) $\wedge$
                (D.isRegularRelay(id, d) $\vee$
                  D.isSteelRelay(id, d))}
      in
         rulesInSet(relayIds, d) $\frown$
         makeInternalTransitionRules(ds \ {d})
      end
   end
pre
   ($\forall$ d : D.Diagram • d $\in$ ds $\Rightarrow$ D.isWfDiagram(d)),

/* makes the idle transition rule from a list
of transition rules */
makeIdleTransitionRule :
   SIS.StaticInterlockingSystem $\rightarrow$ TS.TransitionRule
makeIdleTransitionRule(sis) $\equiv$
   let
      regularUpConditions =
         {CF.canDrawRegularRelay(rr, d) |
          rr : T.Id, d : D.Diagram •
             rr $\in$ SIS.internalRelayIds(sis) $\wedge$
             d $\in$ SIS.diagrams(sis) $\wedge$
             D.isRegularRelay(rr, d)},
      regularDownConditions =
         {CF.canDropRegularRelay(rr, d) |
          rr : T.Id, d : D.Diagram •
             rr $\in$ SIS.internalRelayIds(sis) $\wedge$
             d $\in$ SIS.diagrams(sis) $\wedge$
             D.isRegularRelay(rr, d)},
      steelUpConditions =
         {CF.canDrawSteelRelay(rr, d) |
          rr : T.Id, d : D.Diagram •
             rr $\in$ SIS.internalRelayIds(sis) $\wedge$
```

```
                d ∈ SIS.diagrams(sis) ∧
                D.isSteelRelay(rr, d)},
          steelDownConditions =
             {CF.canDropSteelRelay(rr, d) |
              rr : T.Id, d : D.Diagram •
                rr ∈ SIS.internalRelayIds(sis) ∧
                d ∈ SIS.diagrams(sis) ∧
                D.isSteelRelay(rr, d)},
          conditions =
             regularUpConditions ∪
             regularDownConditions ∪
             steelUpConditions ∪ steelDownConditions
       in
          TS.mk_TransitionRule(
              T.and(
                    {T.neg(T.literal(TS.id(idle))),
                     T.neg(T.or(conditions))}),
              ⟨TS.mk_Assignment(TS.id(idle), T.True)⟩)
       end,

    /* makes all the transition rules from a static
    interlocking system */
    makeTransitionRules :
       SIS.StaticInterlockingSystem →
          TS.TransitionRule*
    makeTransitionRules(sis) ≡
       let
          internalTransitions =
             makeInternalTransitionRules(SIS.diagrams(sis))
       in
          internalTransitions ^
          ⟨makeIdleTransitionRule(sis)⟩
       end
    pre SIS.isWfStaticInterlockingSystem(sis)
 end
```

B.11 StaticInterlockingSystemToConfidenceConditions

context: T, TS, D, CF, SIS, SIStoTS
scheme StaticInterlockingSystemToConfidenceConditions =

```
class
  value
    /∗ calculates the confidence conditions of the
    behavioural semantics
    (a transition system) of
    a static interlocking system ∗/
    makeConfidenceConditions :
      SIS.StaticInterlockingSystem $\xrightarrow{\sim}$ T.LTLassertion$^*$
    makeConfidenceConditions(sis) ≡
      makeConfidenceConditionsFromDiagramSet(
          SIS.diagrams(sis)) $^\frown$
      ⟨T.G(T.F(T.B(T.literal(TS.id(SIStoTS.idle))))),
         T.X(T.B(T.literal(TS.id(SIStoTS.idle))))⟩
    pre SIS.isWfStaticInterlockingSystem(sis),

    /∗ makes the confidence conditions for the relays
    defined in ids and d
    ∗/
    assertionsInSet :
      T.Id-set × D.Diagram $\xrightarrow{\sim}$ T.LTLassertion$^*$
    assertionsInSet(ids, d) ≡
      if ids = {} then ⟨⟩
      else
        let id = hd ids in
          if D.isRegularRelay(id, d)
          then
            trueUntilChangeLTL(
                id, CF.canDrawRegularRelay(id, d),
                CF.canDropRegularRelay(id, d))
          else
            let
              canDraw = CF.canDrawSteelRelay(id, d),
              canDrop = CF.canDropSteelRelay(id, d),
              drawingCurrent =
                CF.drawingCurrentThroughSteelRelay(id, d),
              droppingCurrent =
                CF.droppingCurrentThroughSteelRelay(id, d)
            in
              trueUntilChangeLTL(id, canDraw, canDrop) $^\frown$
              ⟨mutualExclusionLTL(
                    drawingCurrent, droppingCurrent)⟩
            end
          end $^\frown$ assertionsInSet(ids \ {id}, d)
        end
    end
pre
    D.isWfDiagram(d) $\wedge$
    ($\forall$ id : T.Id $\bullet$
         id $\in$ ids $\Rightarrow$
             (D.isSteelRelay(id, d) $\vee$
                D.isRegularRelay(id, d))),

/* makes the confidence conditions from the diagrams
in a diagram set */
makeConfidenceConditionsFromDiagramSet :
    D.Diagram-**set** $\xrightarrow{\sim}$ T.LTLassertion$^*$
makeConfidenceConditionsFromDiagramSet(ds) $\equiv$
    **if** ds = {} **then** $\langle\rangle$
    **else**
        **let** d = **hd** ds **in**
            assertionsInSet(D.allIds(d), d) $\widehat{}$
            makeConfidenceConditionsFromDiagramSet(ds \ {d})
        **end**
    **end**
**pre**
    ($\forall$ d : D.Diagram $\bullet$ d $\in$ ds $\Rightarrow$ D.isWfDiagram(d)),

/*defines the 2 true concurrency ltl assertions
for an id*/
trueUntilChangeLTL :
    T.Id $\times$ T.BooleanExp $\times$ T.BooleanExp $\to$
        T.LTLassertion$^*$
trueUntilChangeLTL(id, upGuard, downGuard) $\equiv$
    $\langle$T.G(
            T.Imply(
                    T.B(upGuard),
                    T.X(
                        T.Imply(
                                T.B(T.neg(upGuard)),
                                T.B(T.literal(id)))))),
        T.G(
            T.Imply(
                    T.B(downGuard),
                    T.X(
                        T.Imply(
                                T.B(T.neg(downGuard)),
                    T.B(T.neg(T.literal(id)))))))⟩,

      /*defines the mutual exclusion rule : the two
      BooleanExp cannot be true at the same time*/
      mutualExclusionLTL :
         T.BooleanExp × T.BooleanExp → T.LTLassertion
      mutualExclusionLTL(b1, b2) ≡
         T.G(T.B(T.neg(T.and({b1, b2}))))
  end
```

B.12 The objects

context: Types
object T : Types

context: DiagramsL
object DL : DiagramsL

context: StaticInterlockingSystemL
object SISL : StaticInterlockingSystemL

context: Diagrams
object D : Diagrams

context: StaticInterlockingSystem
object SIS : StaticInterlockingSystem

context: Pathfinding
object PF : Pathfinding

context: Conditionfinding
object CF : Conditionfinding

context: TransitionSystem
object TS : TransitionSystem

context: StaticInterlockingSystemToTransitionSystem
object SIStoTS : StaticInterlockingSystemToTransitionSystem

Appendix C

CD Overview

This appendix details the content of the CD attached to this thesis.

C.1 Report

Two electronic versions of the report can be found on the CD in the folder *Report*. *IMM-MSc-2008-68-Print.pdf* is the print-version and *IMM-MSc-2008-68-Net.pdf* is the net-version.

C.2 Example of interlocking system behaviour

Folder *Example* on the CD contains *behaviourExample.pdf*. This file presents an example of interlocking system behaviour during the locking of a train route at Stenstrup Station.

C.3 RSL specifications

The sub-folders of folder *RSL* on the CD contain the *.rsl* files of the specifications presented in appendices A and B.

C.3.1 Abstract model

Folder *RSL/Abstract* on the CD contains the *.rsl* files of the abstract model presented in appendix A.

C.3.2 Concrete model

Folder *RSL/Concrete* on the CD contains the *.rsl* files of the concrete model presented in appendix B.

C.4 Application to Stenstrup Station

The sub-folders of folder *Stenstrup* on the CD contain material related to the application of the method developed in this thesis to Stenstrup Station.

C.4.1 Station documentation

Folder *Stenstrup/StationDocumentation* on the CD contains the station documentation of Stenstrup Station:

- Layout of the station: *stenstrupLayout.jpg*
- Operator's panel: *stenstrupOperatorPanel.jpg*
- Train route table: *stenstrupTrainRouteTable.jpg*
- Diagrams: The diagrams of Stenstrup Station are located in folder *Stenstrup/StationDocumentation/Diagrams*, along with file *DiagramModifications.pdf* that explains how they are modified before the translation to XML.

- XML representation of the diagrams, used for generating the internal behaviour: *stenstrup.xml*

C.4.2 External behaviours

Folder *Stenstrup/ExternalBehaviours* on the CD contains the external behaviours for Stenstrup Station:

- *externalBehaviourButtons.rsl* contains instantiations of the patterns for button behaviour.
- *externalBehaviourPoints.rsl* contains instantiations of the patterns for point behaviour.
- *externalBehaviourTrackOrdered.rsl* contains instantiations of the patterns for *ordered track relay behaviour.*
- *externalBehaviourTrackRandom.rsl* contains instantiations of the patterns for *random track relay behaviour.*

The general patterns for external behaviour are described in section 7.1.

C.4.3 Safety properties

Folder *Stenstrup/SafetyAssertions* on the CD contains the safety properties specified for Stenstrup Station:

- *collision_derailing_assertions.rsl* contains instantiations of the patterns expressing the basic safety goals.
- *train_route_table_assertions.rsl* contains instantiations of the patterns expressing the safety properties derived from the train route table.

The general patterns for safety properties are described in section 7.2.

C.4.4 TransitionSystems

Folder *Stenstrup/TransitionSystems* on the CD contains the *RSL-SAL* schemes that have been used for verification of safety properties:

This sub-folder contains several *RSL-SAL* schemes:

- *internalTransitionSystem.rsl* contains the auto-generated transition system and its confidence conditions, i.e. the internal behaviour of the interlocking system at Stenstrup Station.
- *randomStenstrup_CC.rsl* contains both internal and external behaviour, assuming random track relay behaviour. The only assertions specified in this scheme are the confidence conditions.
- *randomStenstrup_TrainRoute.rsl* contains both internal and external behaviour, assuming random track relay behaviour. The only assertions specified in this scheme are the ones that are extracted from the train route table.
- *orderedStenstrup_CC.rsl* contains both internal and external behaviour, assuming ordered track relay behaviour. The only assertions specified in this scheme are the confidence conditions.
- *orderedStenstrup_TrainRoute.rsl* contains both internal and external behaviour, assuming ordered track relay behaviour. The only assertions specified in this scheme are the ones that are extracted from the train route table.
- *orderedStenstrup_Basic.rsl* contains both internal and external behaviour, assuming ordered track relay behaviour. The only assertions specified in this scheme are the ones used for expressing the basic safety goals.

C.5 Java implementation

The sub-folders of folder *Java* on the CD contain everything related to the Java implementation presented in chapter 10.

C.5.1 Source

The sub-folders of *Java/Source* on the CD contain the packages and the source files of the Java implementation.

C.5.2 API

Folder *Java/API* on the CD contains the API of the Java implementation, generated by the Java JDK 6 (also referred to as 1.6.X) *javadoc* tool.

C.5.3 Executable

Folder *Java/Executable* on the CD contains an executable version of the Java implementation, *InternalBehaviourGeneration.jar*. If one wants to execute the program, Java RE 6 (also referred to as 1.6.X) is required. Information on how to execute the program can be found in file *readme.txt*.

The dependencies of the Java implementation are located in folder *Java/Executable/lib* together with their license.

C.6 Testing

Folder *TestFiles* and its sub-folders on the CD contain the performed tests on the Java implementation. File *test.pdf* explains the performed tests.

C.6.1 Input XML files

Folder *TestFiles/Functional* on the CD contains all the XML files that have been used as input to the program when performing *attribute-related tests* and *well-formed related tests.*

C.6.2 Generated transition systems

Folder *TestFiles/Result* on the CD contains 6 sub-folders, one for each of the well-formed static interlocking systems that have been used as part of the testing. Each folder contain a picture of the used diagrams for the test, the input XML file to the tested program, and the generated transition system. Comments on the results can be found in *TestFiles/test.pdf.*