A model-based test platform for rail signalling systems

Bouwman, Mark

Award date:
2019

Eindhoven University of Technology
Department of Mathematics and Computer Science
Formal System Analysis Group

A model-based test platform for rail signalling systems

Mark Bouwman

Master Thesis

Graduation supervisor:
Dr. Bas Luttik

Company supervisor:
Dr. Bob Janssen

February 6, 2019

Abstract

As technology progresses, newer, and more complex, solutions are employed to verify that signalling systems are safe. Formal methods, with their innate mathematical precision, provide ways to increase rigour in the verification process. This precision, accompanied by the ongoing increase of computational power of computers, also opens up ways to partially automate parts of the verification process.

This thesis presents an application of the formal modelling and model checking toolkit mCRL2 and the model-based testing tool JTorX in the signalling domain. The mCRL2 toolkit is used to formally model the behaviour of a system at the core of signalling solutions: the interlocking. The behaviour of the interlocking is validated through model checking, proving that relevant safety properties hold. Using JTorX, the formal model is turned into the benchmark in an automated testing platform for interlocking software. A working setup with actual interlocking software on a pre-existing testing platform (TeSys) is presented, though performance and stability remain an issue. The suitability of mCRL2 and JTorX in the signalling domain is evaluated and suggestions are given for improvement.

Keywords: automated testing, model checking, mCRL2, JTorX, interlocking, railway signalling, TeSys

Acknowledgments

After six months of researching, modelling, coding and a lot of writing the thesis before you is the final product. I would like to use this page of the document to thank the people who have helped and supported me during this project.

First of all I would like to thank my company supervisor, Bob Janssen, who has acted as somewhat of an oracle on any questions on signalling systems. Tapping into his knowledge has been crucial to finding my way in the signalling domain. His feedback has also helped make this document more readable for readers outside the formal methods expertise. My gratitude also goes to my graduation supervisor, Bas Luttik, who helped me find this project in the first place and has guided me along the way. Through our weekly Skype calls he gave guidance by asking critical questions and giving tips. With elaborate and quick feedback on my writings he has played a big role in shaping this document. I would also like to thank Rick Erkens, who attended many of the Skype calls and has helped me troubleshoot some of the issues I had with the formal model.

Working on a research project can be frustrating and stressful at times. Many times when I felt like my models and writing were rubbish, my loving wife Nienke was able to lift my spirits and give me new hope that the next day I would be able to solve the issue I was having. Of course, she was pretty much always right and her kind words have made my thesis much more enjoyable.

On a more philosophical note, I am grateful to live and breathe on this big blue planet floating through a vast universe. I would therefore like to thank the Creator for creating this wonderful world with so many wonderful human beings. I see it as gift that we, through cooperation, have the ability to create and reason about the most amazingly complex things ... such as mathematics and signalling systems.

Contents

Glossary

All abbreviations will be introduced in the main text but can also be found in the following table.

Abbreviation	**Meaning**
ATB	Automatische TreinBeïnvloeding
ATB-EG	ATB Eerste Generatie - First generation of ATB
ATB-NG	ATB Nieuwe Generatie - New generation of ATB
ATB-Vv	ATB Verbeterde versie - Improved version of ATB (used at train stations)
CLI	Command Line Interface
ERTMS	European Rail Traffic Management System
ETCS	European Train Control System
GESIM	Gesamtsimulation
GUIDO	Graphical User Interface with Dynamic Objects
IL/IXL	Interlocking
ioco	input output conformance
IUT/SUT	Implementation/System Under Test
LPS	Linear Process Specification
LTS	Labelled Transition System
MA	Movement Authority - a point a train has the authority to move to
mCRL2	micro Common Representation Language 2
PBES	Parameterized Boolean Equation System
TAK	Testautomatisierungskomponente
TeSys	Test Systemen

Chapter 1

Introduction

Train safety protocols and systems have a long history in which safety criteria have evolved alongside the available technology. Starting from simple schemes where track sections were reserved for trains based on a time schedule or tokens that allowed exclusive track access to a train, railway signalling has progressed. With the advent of the telegraph, messages could be sent ahead to signalmen to notify a train was approaching or that the previous train had cleared a section. These signalmen could then throw relevant points in the right position and set the signal to inform the train whether it was allowed to continue (and at what speed). Accidents spurred the use of extra safety features such as interlocking equipment that disallows conflicting train movements. Over time, signalmen were supported by train detection equipment such as track circuits and axle counters and centrally operated signals. Furthermore, systems were introduced to automatically stop a train when it ignored signals.

Nowadays, each country has its own set of guidelines, laws, safety protocols and equipment. In addition, the European Rail Traffic Management System (ERTMS) is slowly being rolled out to replace the legacy national safety systems. In the Netherlands, Automatische TreinBeïnvloeding (ATB) is the national system. ATB Eerste Generatie (ATB-EG) is the original version of ATB and is still being used on most tracks, a newer version, ATB Nieuwe Generatie (ATB-NG) is used on some less busy tracks, ATB Verbeterde versie (ATB-Vv) is used at yards and ERTMS is used on some tracks across the country. Eventually ERTMS will completely replace ATB, but this may still take decades.

Developing safety critical systems is a tough engineering challenge. From the design of higher level protocols to their electrical/mechanical implementation and to guidelines for train drivers: scrutiny is essential, a single flaw could potentially lead to a fatal accident. As the higher level protocols of railway signalling become more and more complex it also becomes increasingly difficult and costly to verify the correctness of these protocols.

1.1 Models in the signalling domain

A wider trend in systems and software engineering is the increased use of models. Models, depending on the type, can have the advantage of being more precise, analysable, adaptable and/or understandable than traditional knowledge representations such as text and pictures. An example of a model in the railway domain is the RailTopoModel [11], which models the topology and structural elements of railway signalling systems. The use of models in the rail signalling domain can improve the engineering process by easing communication, data exchange and allowing the use of digital tools to aid the engineering process.

Of course, different goals require different types of models; for example an effort to standardize track layouts will benefit from a different type of model than an effort to better estimate the costs of point maintenance. To analyse the safety of rail signalling systems behavioural models are needed. Formal methods aim to verify protocols, software and hardware through a range of mathematical techniques. These techniques include, among others, proof assistants, automatic theorem proving and model checking. There are, again, different types of formalisms for different goals. From here on we will use the term formal model to denote a mathematically founded specification of the behaviour of a system, a behavioural model. In the past formal methods has proven useful to improve the quality of (software) systems [6, 21], including within the railway domain[7]. Automatic theorem provers and model checkers have recently been used to find ambiguities in the initial specification of a new variant of ERTMS (hybrid level 3)[14, 1].

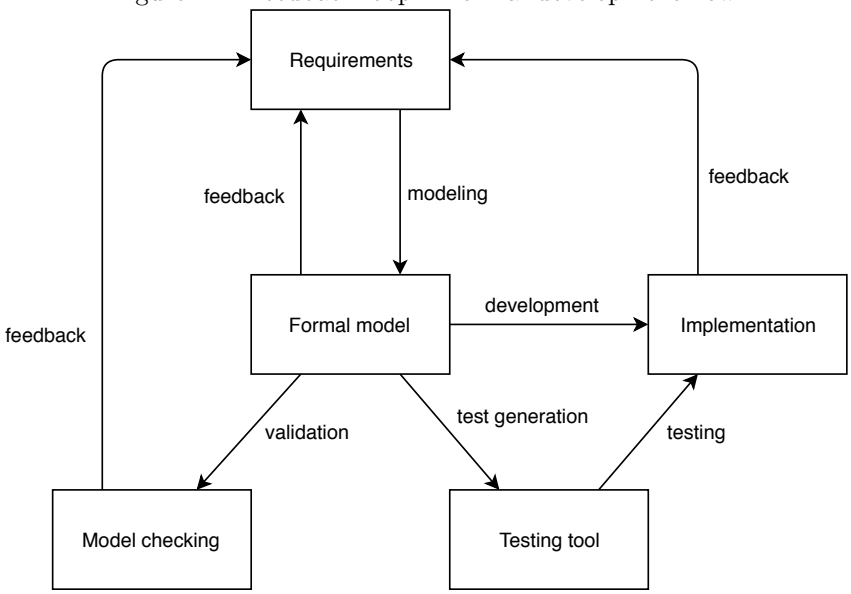
Formal models could also fulfil a role in the testing phase of system development. Given the high standards for safety and integrity (SIL 4) it is needed to test all equipment for compliance with the specification. For components that need to be configured for a specific track layout, test scenarios need to be designed specifically for that track layout and executed, which is time consuming. It is desirable to

automate this process of testing as much as possible.

1.2 Formal model as specification

Formal methods are mostly used in earlier stages of system development [21]. Without using formal methods, the main guidelines in system development are the requirements in natural language. Possibly, semi-formal specifications (such as UML diagrams) are used to model the architecture of the system to give extra guidelines. Formal models can be integrated early in the development process to analyse the correctness of the system design before it is implemented. This may reduce development costs significantly as it can detect design flaws that may otherwise be detected during the implementation or test phase. The process of creating a formal model itself already generates feedback on the requirements as ambiguous requirements and parts of the system that are underspecified will come to light. Model checking provides further feedback. Testing the implementation using model-based testing techniques creates feedback on the implementation. Figure 1.2 depicts these feedback flows in a diagram.

Figure 1.1: Feedback loop in formal development flow



1.3 Goals and results

Siemens wishes to continuously improve their production chain of signalling equipment by improving the rigour of verification whilst also making it more efficient. The signalling production chain, and with it the needed rigour, extends from obtaining a correct assessment of the current situation to testing each individual wire of the final system. The original assignment description put together by Siemens formulated their wish to investigate a production chain assisted by formal methods for the software of a component at the core of signalling systems: the interlocking. The mCRL2 toolkit, developed by the Mathematics and Computer Science department of the Technical University of Eindhoven is a formal methods toolkit that might fit their needs. This thesis investigates the suitability and performance of the mCRL2 toolkit in the signalling domain.

As Siemens already has a functional interlocking system, formal methods will be applied to verify the existing system, as opposed to the flow described in the previous section. We will discuss how formal methods can be integrated in earlier stages of the development cycle of signalling systems in Chapter 9. The focus of this thesis has zoomed in on using formal models for verification and testing. This thesis presents a case study where formal methods are used to model the behaviour of the interlocking. The formal model is analysed using the model checking capabilities of the mCRL2 toolkit to verify safety properties. We will demonstrate that desirable safety properties hold for the model, given some assumptions.

This thesis also presents a platform to both derive test cases for an interlocking and to execute them automatically. The goal of this platform is not to be a finished commercial product, but rather to show that, next to model checking, formal methods provide ways to perform test cases automatically on implemented signalling systems. The test derivation platform is based on a mCRL2 model and the

model-based testing tool JTorX. The tests are performed on an existing testing platform for interlocking software used by Siemens.

The contribution of this thesis is to demonstrate the applicability of formal methods, and in particular mCRL2, in the railway signalling domain as well as showing how formal methods can strengthen the production chain of interlocking equipment.

1.4 Organisation

This document is organised as follows. Chapters 2 and 3 introduce the reader to the preliminaries of railway signalling and the existing testing platform used by Siemens and Chapter 4 elaborates how core signalling concepts can be modelled. Chapters 5 and 6 present a formal model of the interlocking and the verification of the model. Subsequently, Chapters 7 and 8 present how the mCRL2 model can be used to create an automated testing platform with JTorX and the existing Siemens testing platform. In Chapter 9, we reflect on the results of model checking and testing and discuss the usefulness of formal methods in the railway signalling domain and how it could be further improved.

Chapter 2

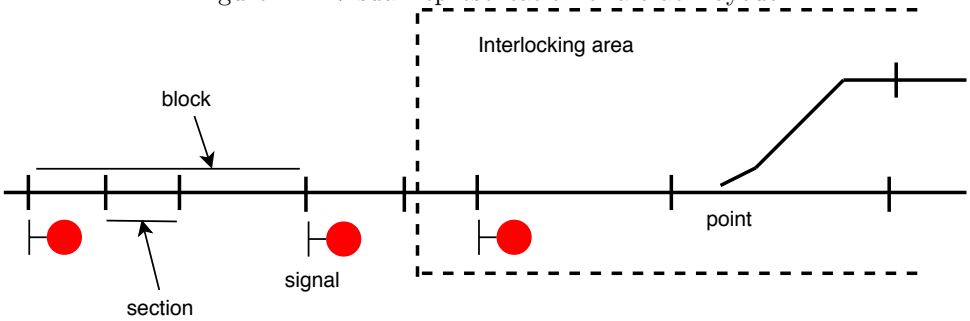
Introducing railway signalling systems

The rail network can be subdivided into two main categories: yards and open tracks connecting the yards. Yards contain points while open tracks do not contain points. Pieces of track (both open track and yards) are subdivided into sections. Each section has a track circuit or an axle counter as train detection equipment so that it can be determined whether there is (part of) a train on that section. This information is crucial to prevent conflicting routes.

The signalling setup for open tracks is quite simple. The open track can be divided into blocks, which may consists of multiple sections, with a signal at the end of each block. The component controlling the signal sees which sections of the two blocks ahead are occupied and sets the signal accordingly. In Dutch signalling the most common colour aspects a signal may show is either green to indicate that the train can continue to the next block at full speed, or yellow to indicate that the train should slow down as the next signal might be red, or red to indicate that the train is not allowed to enter the next block. In addition, more precise information regarding the maximum speed may be signalled to the train driver.

Adding points makes the task of signalling engineers more challenging. Point position depends on the train's route, points need to be moved in the right direction, conflicting routes need to be avoided and whether a signal can show proceed becomes more complex. The component that is at the heart of the safety systems of yards is called the interlocking (IL). The IL controls a set of sections, points and signals. It also has a connection to signalmen, who can request to route an incoming train from one signal in the yard to another signal. If the route is accepted, the interlocking will throw the points to the correct position and control the signals to safely guide the train to its destination.

Figure 2.1: Visual representation of a track layout



Relying on the driver to see and obey the signals is not enough; Automatic Train Protection (ATP) is necessary to prevent accidents in case the driver fails to obey for any reason. ATP systems consist of the combination of trackside (IL and signals) and trainside systems. The trainside system receives information on whether the train is allowed to proceed. In the cases that a train is moving faster than it is allowed to, the trainside system will intervene and start braking. Trains are considered to be fail-safe systems, meaning that when the safety systems break down the train will go to a safe state: standing still. Signals are also designed to be fail-safe; if they fail they will show a stop aspect.

This is a general summary of how most train signalling systems work. In practice there are many differences between various systems. ATB-EG can communicate the maximum speed continuously to the cabin whereas ATB-NG uses punctual transponders to communicate a movement authority (MA). A Movement Authority (MA) is a message to the train containing distance to run and possibly a speed

profile. ATP systems generally transform the semantics of a signal aspect into a MA. In ERTMS level 2 and 3 the movement authority is communicated over radio to the train, so physical trackside signals are not needed. Level 3 replaces fixed blocks by moving blocks that move along with the train rear and front. This requires accurate knowledge of the train's position and length. We will stick to the more traditional signalling setup with sections and signals, abstracting from national variations where possible.

Chapter 3

Existing testing platform

Siemens has an existing testing platform for their interlocking software called TeSys. This platform runs the real interlocking logic software in a testbed that simulates the railway environment. It will serve as the basis for the automated testing platform presented in Chapter 8. In this chapter we will explore TeSys and what testing features the platform offers.

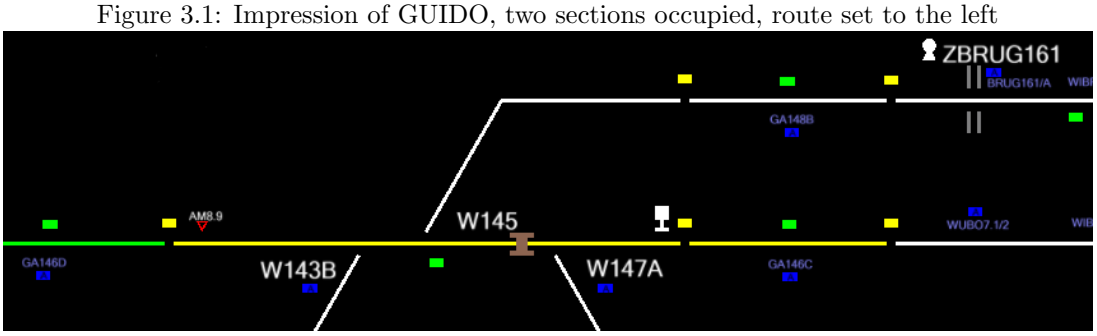
3.1 Introduction TeSys

The testing platform TeSys (short for Test Systemen) is a broad platform to simulate and test interlocking behaviour of the Siemens W interlocking. The Siemens W interlocking is an electronic interlocking which is the successor of the Siemens C interlocking. TeSys consists of a number of applications that run in parallel to simulate different parts of the system alongside a few applications to interact with the simulation and run tests. The actual interlocking software is run inside the simulation, all the field elements are simulated, the field element controllers are simulated and the internal buses are simulated. The simulation of all these components together is called the Gesamtsimulation (GeSim). There are two main ways to interact with the simulation, GUIDO and TAK.

3.2 GUIDO

GUIDO (short for Graphical User Interface with Dynamic Objects) provides, as the name suggests, a GUI to the user that shows the current state of the system and provides options to interact with the system. By clicking on sections the state can be toggled between free and occupied, a route can be requested by clicking on two signals and right clicking on points and signals gives extra options (to, for example, immobilize a point). The occupancy of sections, routes and other information is presented visually to the user. As an example: if a user requests a route, the route is first displayed white (if the route is free), the points start flashing to indicate that they are moving and once all points are in the right position the route becomes green and the entry signal is set to show proceed.

Figure 3.1: Impression of GUIDO, two sections occupied, route set to the left



3.3 TAK

TeSys also contains a Testautomatisierungskomponente (TAK), which is a testing platform that can also stimulate the simulation. TAK has its own scripting language to write test scripts. It consists of commands, preceded by a $ sign, and expected observations, preceded by a % or a | sign. The following

snippet states that TAK should send a command to the simulation that the section with identifier E10_HG_GA154A becomes occupied and that the interlocking should respond by setting the signal with identifier E10_SOM6_AS154 to stop:

```
$E10_HG_GA154A,BSZT
|E10_SOM6_AS154: "'Lampe_3_rot, ein'"
```

A recording function is present so it is possible to play out a scenario using GUIDO and record it with TAK. The scenario can then be replayed step by step or automatically. The user can also save the scenario as a .kat file for later use. There are several options for conformance checking: conformance checking can be turned off completely (so it will only play the scenario without checking how the interlocking reacts), it can be set to only check if the response specified in the test script is performed by the interlocking or it can be set to also check if every response by the interlocking is predicted in the test script. TAK does not check in which order the observations take place, every response specified in the test script should be done eventually but the order is ignored. According to the TAK manual, TAK is mainly intended to be used for regression testing.

Chapter 4

Modelling signalling systems

Before we present the formal model in Chapter 5 we will examine what the purpose of the model is and model some core concepts in a semi-formal manner. Some of the core concepts we will model in this chapter are the connections the interlocking has with other components, the track topology and the life-cycle of routes. Finally, we will examine what data is maintained internally by the interlocking. This will provide a stepping stone for the formal model.

4.1 Modelling goals

Considering the high complexity of railway interlocking systems it would not be desirable to try to model every aspect of the system in detail. It is beneficial from the point of view of analysability, to choose an appropriate level of abstraction. This a big strength of modelling: Through abstraction the system becomes easier to understand and analyse. We therefore abstract from lower level interfaces and message exchanges and model the higher level functional behaviour. Also on this higher level, abstractions help to get to the core of what we want to test. The rule of thumb is that abstractions should not be made on unfounded assumptions that could invalidate the results found, and may restrict which aspects of the system can be tested. An example is the absence of level crossings in the model, which restricts what can be tested but does not invalidate any results for tracks without level crossings.

The end goal of modelling is to evaluate the safety of real interlocking systems. By analysing whether safety properties hold for the formal model we wish to obtain a safe formal model for testing. By testing whether an implementation conforms to a formally verified model we wish to increase confidence that the implementation conforms to the safe model (or possibly detect errors). The two ways the model is used put different requirements on the model that can be contradictory. For model checking, a large state space (all the states the system can be in) may hamper verification. A smaller state space can partly be achieved by instantiating the model with a small yard but it also requires that the model does not cover too many features. For testing on the other hand, the size of the state space is less of an issue as the entire state space does not need to be considered or even generated. It is then desirable for testing to model more aspects of the system so that they may be tested.

ProRail has a requirements document (*Programma van Eisen*) for the interlocking that serves as a baseline specification for interlocking suppliers. It includes many functional requirements on what kind of features and interfaces need to be supported as well as a number of behavioural requirements. The behavioural requirements related to safety (such as the conditions when a signal can be set to show proceed) indirectly address three core safety concerns: train should not collide (apart from shunting movements in designated areas), trains should not derail by encountering a point that is not in the right or left position, and activated warning systems are a prerequisite for allowing a train to approach a level crossing. To limit the scope of the model we will restrict the required safety properties. We abstract from shunting movements and from what may happen outside the limits of the yard controlled by the interlocking as much as possible. The requirement for avoiding collisions is then reduced to the requirement that two trains should never occupy the same section. We abstract from level crossings completely. The end goal for the formal model can be summarized to be a model for the which the properties described above hold and it needs to be detailed enough for testing. From the model it should be possible to deduce concrete stimuli for testing, as well as being able to predict the messages coming back from the interlocking under test.

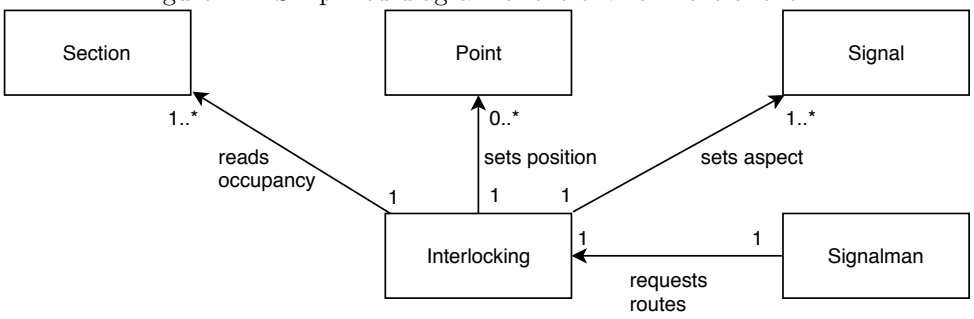
The necessary information to construct the model was provided mainly by Bob Janssen from Siemens, complemented by Daan van der Meij from ProRail and through a number of UML diagrams that ProRail supplied. Ambiguities that remained were cleared by testing how the interlocking responds to certain

conditions using a simulation of the interlocking software. Creating a model by observing how the implementation works is an uncommon practice in the application of formal methods, but a useful tactic in this project as the implementation was already there and complete specifications are unavailable.

4.2 Connections of the interlocking

The interlocking does not stand alone, it has connections with various other components with which it can communicate. It has a communication channel with the signalman and has connections to various field elements: sections, points and signals. Note that the interlocking does not have a connection with the train, it can only indirectly observe the train via occupancy detection. Figure 4.1 gives an overview of the connections the interlocking has with its environment.

Figure 4.1: Simplified diagram of the environment of the IL



The interlocking sees the occupancy of a section as a Boolean: occupied or not occupied. In its internal memory however, it has an additional state for sections: logically occupied. In the case that, in the perspective of the interlocking, a train moves non-sequentially over the sections (or even disappears) a section can become logically occupied. This can happen when a train is not properly detected (for example due to rust). In the Netherlands *volgordedwang* is applied: a section that has been detected to be occupied can only be considered free again if the train detection indicates that that section is not occupied and the next section is occupied.

Points can be thrown by the interlocking into the left or right position. Sensors give feedback to the interlocking on the current position of the point (proven left/right or not proven). In our model we abstract from these sensors and the time it takes to move a point: the position of a point is simply set by the interlocking. This reduces the requirement on avoiding derailment to the requirement that a point should not be moved while the section in which the point is located is occupied.

The interlocking sets the signal aspect of the signals and receives feedback on whether each lamp is on, off or failed. Signals can show a set of aspects, e.g. it may signal the maximum allowed speed, but we abstract from this. In our model, the interlocking only sets the colour aspect. The aspect a signal can show is restricted to red, green or yellow, whereas in reality other aspects are also possible in special circumstances. We also abstract from the possibility of lights failing.

The signalman can request a route from one signal to another and the interlocking responds by either accepting or rejecting the route.

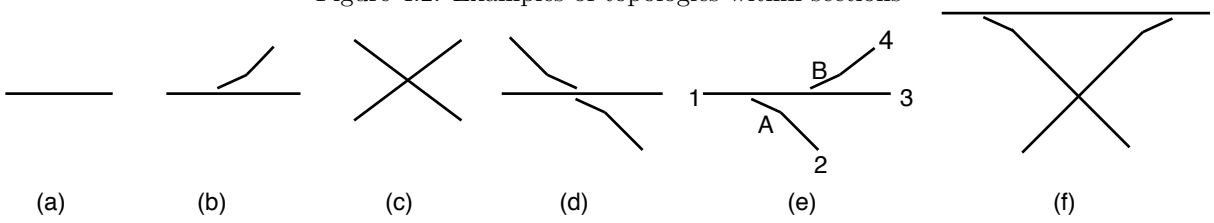
From	**To**	**Communication**	**Data passed**
Signalman	Interlocking	Request route	IDs of start signal and end signal
Interlocking	Signalman	Route decision	Decision: accepted or rejected
Interlocking	Point	Set position	The desired position (left or right)
Interlocking	Signal	Set aspect	Signal aspect: red, yellow or green
Section	Interlocking	Inform section occupancy	Boolean to indicate occupancy

Table 4.1: Overview of messages exchanged between the interlocking and its environment

4.3 Track topology

One of the core concepts in many railway models is the track topology. A way to model the topology is using graph concepts (used by the UIC [11] and ProRail [17]): straight pieces of track become the nodes and points become the edges, connecting nodes dependent on the point's position.

Figure 4.2: Examples of topologies within sections



The track layout can not just be divided into straight pieces of track and points however; sections with train detection also need to be taken into account. Each section may incorporate multiple points and pieces of straight track. For our purposes, the internal structure of sections is not that relevant, the main focus is on the sections: a train occupies a section and we wish to prevent that two trains can occupy the same section. We therefore do not wish to include the internal structure explicitly in the model: a section is indivisible and we model connections between sections more directly. A section can be considered to facilitate a route from its left-side neighbour to its right-side neighbour, depending on the positions of the points of the section. In terms of graph concepts, a section has a dual role, it is a node, an object that can be occupied, but it is also an edge connecting it's neighbouring sections. As an example: Figure 4.2e would facilitate the following connections:

left-side section	**right-side section**	**point positions**
1	2	A: right
1	3	A: left, B: right
1	4	A: left, B: left

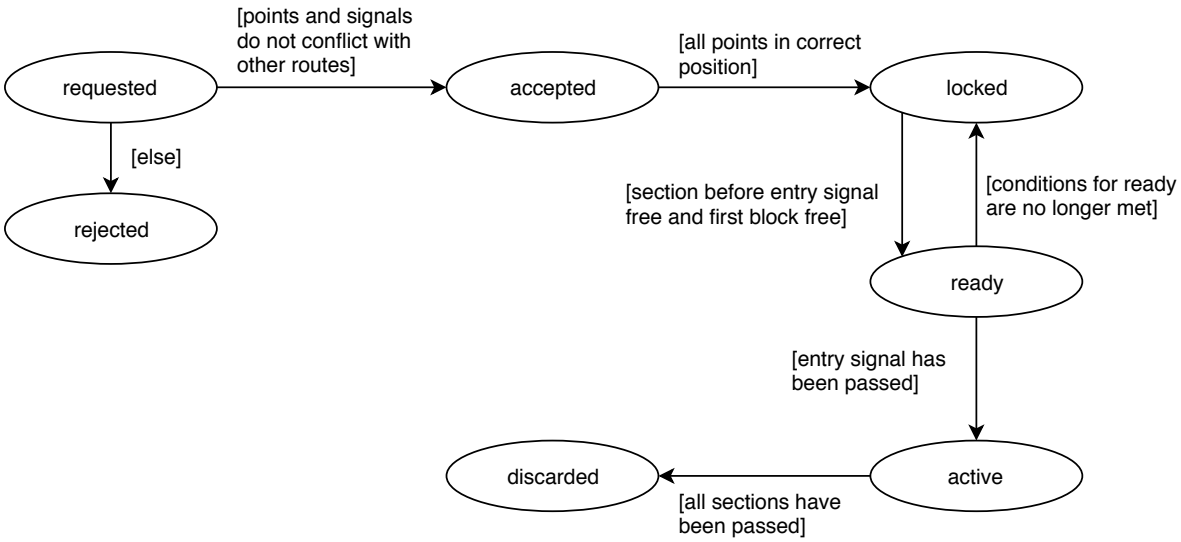
Abstracting from the internal structure of sections reduces the complexity of finding routes but remains very flexible. It can capture any combination of points and crossings as long as two neighbouring sections have a single track connecting them.

The topology also includes the position of signals. As a signal always stands at the border of a section, its position is easily pinpointed by specifying the sections on which border it stands. The direction the signal is facing is also part of its position.

4.4 Routes

A route is set from an entry signal to an exit signal. A route can not be set between every pair of signals; a list of possible entry/exit signal pairs is predefined. Typically, only routes between signals that are close to each other can be requested. Before opening the entry signal, the IL ensures that sections are vacant and points are in the required position. Virtual signals (signals to which a route can be set but which do not correspond to a physical signal) at the yard limits protect movement to and from open tracks. A route also requires flank protection (protection from other trains colliding into the side of the train). For each point and crossing of the route, the flanks need to be protected by either a signal or by some point that is not part of the route that diverts trains from the flanks. The life cycle of a route is shown in Figure 4.3.

Figure 4.3: UML state diagram of the life cycle of a route



4.5 Internal data structure

The behaviour of the interlocking is determined by it's data state. In this section we will analyze what data the interlocking must maintain in order to decide what outputs it can perform. From the interactions the interlocking can have (Section 4.2), the track topology (Section 4.3) it knows and the routes which it keeps track of (Section 4.4), we can derive what information is maintained internally by the interlocking. It knows the current state of each of the field elements, which it can address by some ID. As it knows the track topology, it knows which routes through a section are possible (the section connections of a section) and the position of each signal. The interlocking has a memory of which routes are set, the current status of each route and the field elements tied to that route. Moreover, for each route it must also remember how the flanks are protected.

Combining these pieces of information we can create a model of the internal data structure maintained by the interlocking. Figure 4.4 visualizes in a UML class diagram which information is maintained. Note that the class Point records the current position of a point whereas Section Connection, Route and Flank Protection use Point Position Requirement to record the required position of points, e.g. the position of certain points required to facilitate a route, which may differ from the current position of a point. Table 4.2 further details what the options are for the enumerations used in Figure 4.4. This model of the internal data structure is sufficient for our simplified view which focuses on the behaviour of the interlocking. More complex data models of the rail domain are being developed, such as the EULYNX data preparation model [5] and the ProRail ontology model [16]

Figure 4.4: UML class diagram of data structure

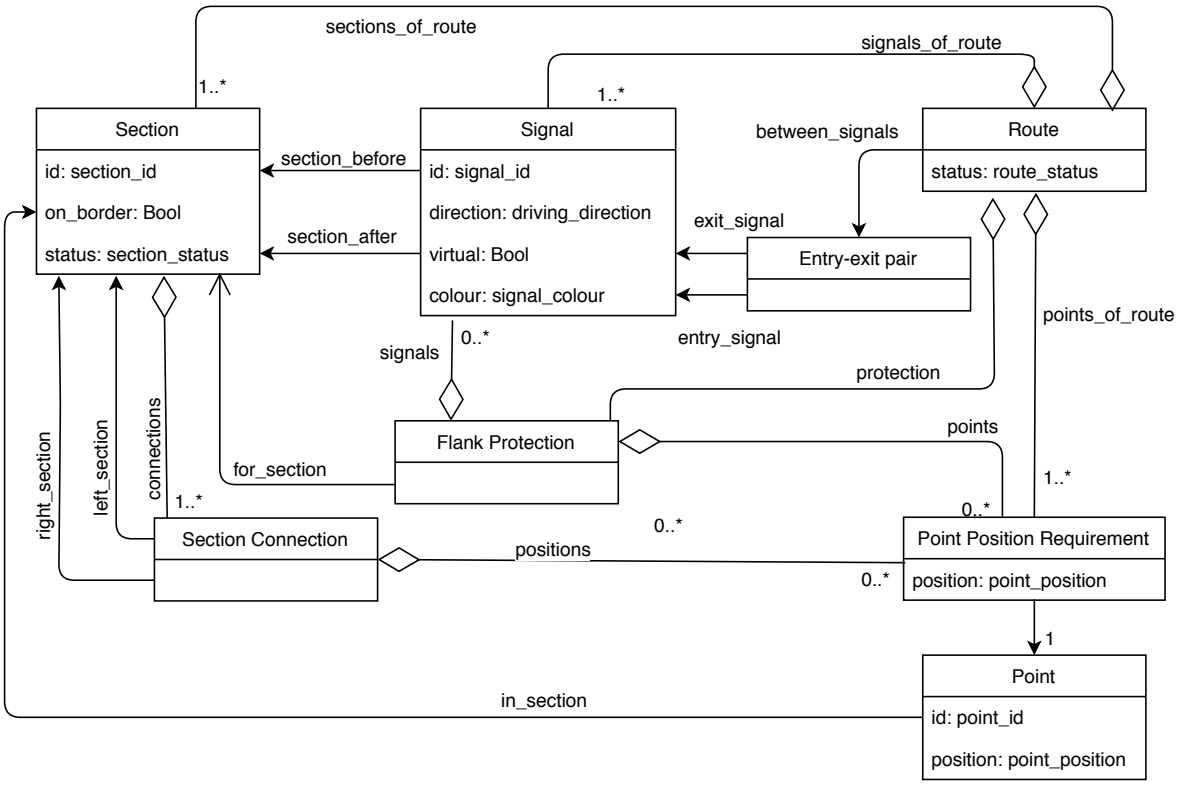


Table 4.2: Enumerations in data structure

Name	**Options**
section_status	free, occupied, logically occupied
driving_direction	left, right
signal_colour	red, yellow, green
point_position	left, right
route_status	requested, accepted, rejected, locked, ready, active, discarded

Chapter 5

Formally modelling an interlocking

In the previous chapter we sketched how an interlocking can be modelled: the interfaces it has, a model of the track topology and the life cycle of routes. In this chapter we will present how the constructs of the mCRL2 language are used to create an mCRL2 model of the interlocking. Before we dive into the formal model we will introduce the mCRL2 language and the mCRL2 toolkit.

5.1 Introduction mCRL2

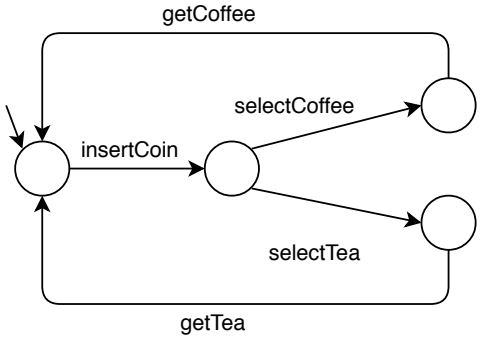
The mCRL2 toolkit is a collection of tools for creating formal models in the mCRL2 modelling language and subsequently analysing and proving properties of those models. An mCRL2 model is a behavioural model, it models the behaviour of a system. Even though an mCRL2 model may include a model of a data structure (such as a model of the track layout), it is instrumental in specifying the behaviour of a system (based on it's data state). The supported language for specifying properties is the modal μ-calculus. A precise description of the mCRL2 language and the modal μ-calculus can be found in the book by Groote and Mousavi [9]. The toolkit itself can be downloaded from the mCRL2 website [8], which also includes instructions on using the toolkit and a small tutorial on mCRL2 and the modal μ-calculus.

5.1.1 mCRL2 modelling language

The mCRL2 language includes a data language in which users can construct data types, and operate on them using mappings and equations defining the mappings. Common data types such as natural numbers and Booleans are built-in, as well as a few container types such as lists, sets and bags. Function types are also supported. The actual behaviour in mCRL2 comes from the processes, which can be specified using common process-algebraic constructions. A process specification starts with a declaration of actions, denoting the basic events relevant for the process to be specified. These actions can be parametrized with data terms. Actions can be composed into processes using sequential composition, non-deterministic choice and parallel composition. Data can influence the processes via conditional constructs. A powerful construct in mCRL2 is the ability to sum over a data domain, such as all the elements of a list. Processes can be defined via parametrized equations creating the possibility to specify infinite behaviour through recursion. Support for action renaming and hiding (renaming an action to τ which can be treated as unobservable) is in place. Communication between parallel processes can be enforced and multi-actions can be used.

The semantics of an mCRL2 model is a Labelled Transition System (LTS). Such an LTS consists of states, of which one is the initial state, and labelled transitions between states. A (compact) mCRl2 model may describe an enormous or even infinite LTS.

Figure 5.1: Example Labelled Transition System



5.1.2 Modal μ-calculus

The mCRL2 toolkit uses a first-order extension of the modal μ-calculus. It extends it by adding support for data and the specification of sequences of actions by regular expressions. The modal μ-calculus allows users to specify properties of an LTS, which can be verified for a specific LTS using the toolkit. For the example of Figure 5.1 it could for instance be proven that the behaviour of the coffee machine is deadlock free, i.e. it is always possible to do a next action.

5.1.3 General toolkit functionality

The toolkit has a wide selection of tools with many options, only a brief overview is given here. An editor is provided for the mCRL2 modelling language. An mCRL2 model can be converted to an LTS. An LTS can be visualized, which can give insights into the underlying process. The model can also be simulated. The simulation computes which transitions are possible from a state and allows the user to fire transitions. The state of the process and its data parameters can be inspected at each step. Simulation is a quick way to explore the model and to find mistakes. It can also be verified whether a modal μ-formula holds for the model. There is also the possibility to generate a counter example in the case that the formula evaluates to false. In the case of train safety properties this allows us to find a trace in the system that leads to an unsafe state.

5.2 mCRL2 model

The mCRL2 model formally captures the behaviour of the interlocking. Modelling only the interlocking however, would be insufficient to derive good test cases and would make it hard to prove certain properties. Modelling the environment constrains what kind of input can be sent to the interlocking for realistic scenarios. An example is that train processes can simulate following some route guiding what section occupations are sent to the interlocking. Modelling the environment also makes it easier to prove that the behaviour of the interlocking has some effect on its environment. Recall that we wish to prove that the model satisfies the property that trains can not collide. This becomes easier if we have separate section processes that can be set to be occupied by train processes. Therefore, the context in which the interlocking operates is also included in the formal model: processes for trains, sections, signals and points are included. Additionally, a generic model of an interlocking needs to be easily configurable for different track layouts so a configuration mechanism is included in the model. The complete mCRL2 model can be found in appendix A.

5.2.1 Data specification

As the interlocking's behaviour depends on its internal data state (section occupations, routes, etc.) a specification of the internal memory of the interlocking needs to be included in the formal model. The mCRL2 language provides ways to create data structures using structured types and function types and ways to manipulate the data structures using mappings and equations to specify rewrite rules.

Data structures

Each class in the UML model of Figure 4.4 corresponds to a structured type in the mCRL2 model. Enumerations such as the status of a section, the position of a point and the status of a route are also modelled using structured types. The attributes of the classes, as well as the association and aggregation

relations, become part of the structured type in the mCRL2 model. In the case that the multiplicity of a relation is not 1, the predefined container sort List is used. Note that in many cases, such as the set of connections of a section, the sort Set would be more natural as the order is not important. The benefit of lists, however, is that it is possible to iterate over them in the data operations, as opposed to sets. The mCRL2 code below shows an example of the use of structured types in the data specification.

```
sort
section_status = struct free | occupied | logically_occupied;

section_info = struct section_info(
  status: section_status,
  on_border: Bool,
  connections: List(section_connection)
);
```

Note that the ID is not included in the constructed sort *section_info*. Instead, a function type is used to map section IDs to section info objects, as shown below. This way, *sections* is a data structure similar to a key value mapping, mapping the ID to the associated object. Sections, signals and points are all modelled in this fashion. The benefit of using these function types is that structured types referencing sections, signals and/or points can store the ID, instead of duplicating the entire type. Moreover, the mCRL2 toolkit efficiently deals with natural numbers in function types.

```
sort
sections = section_id -> section_info;
```

Data operations

Based on the data structure, operations can be defined to update data, evaluate some condition based on the current state or do some other computation. Mappings and equations, which act as rewrite rules, can be defined to perform these computations. The equations are used by the processes and pass the parameters that the equation requires. They can generally be divided into three categories: simple get and set operations on the data structure, computations on the topology and computations on dynamic aspects (such as the current section occupations). The following mapping and equation define an operation that calculates whether the end of a given section has a signal:

```
map
    in_front_of_signal: section_id#section_id#signals -> Bool;
eqn
    in_front_of_signal(se,se2,sic) =
        exists signal:signal_id. legal_signal(signal)
        && section_before(sic(signal)) == se && section_after(sic(signal)) == se2
        && !virtual(sic(signal));
```

The code defines a mapping form two section IDs and the collection of signals to a Boolean. In natural language the equation states that a signal is ahead if and only if there exists a non-virtual signal between the given sections, facing the first given section. The second section parameter is necessary to pinpoint on what border of the first section the signal is located. The *legal_signal* predicate checks whether the given ID is a valid ID for which a *signal_info* object is defined. This is necessary as the ID is a natural number and thus part of an infinite set. The mCRL2 toolkit recognizes that the existential quantification is bounded by the legal_signal predicate, and thus that it is not necessary to consider all natural numbers when evaluating the existential quantification.

5.2.2 Process specification

A process is defined for each section, signal, point and train as well as for the interlocking. The processes for the field elements act as a kind of variables; their only behaviour is that they keep track of their state. It is always possible to change this variable; a train can not be prevented from occupying a section and the interlocking can not be prevented from setting the aspect of a signal or changing the position of a point. The train processes interact with the field elements by making sections occupied, obeying signal aspects and moving across the track in accordance with the position of points.

As can be expected, the process of the interlocking contains more complex behaviour. It reads the status of the sections, moves points, sets signals and receives route requests. To describe the behaviour as generally as possible, no assumption is made on the order of operations performed by the interlocking,

it can non-deterministically choose to perform any action for which the conditions are satisfied. The code below shows the main process of the interlocking and one of the sub-processes:

```
Interlocking(sec: sections, sic: signals, roc: Set(route_info),
            poc: points, pro:Set(route_info), rro: List(route_info)) =
  InterlockingUpdatingSignal()
  + InterlockingReadingSection()
  + InterlockingMovingPoint()
  + InterlockingReceivingRouteRequest()
  + InterlockingProcessingRoute()
  + InterlockingReadyRoute()
  + InterlockingNotReadyRoute()
  + InterlockingPermitTrainEntry();

InterlockingUpdatingSignal(sec: sections, sic: signals, roc: Set(route_info),
        poc: points, pro:Set(route_info), rro: List(route_info)) =
    sum result: signal_colour. sum si: signal_id. (legal_signal(si)
    && result == compute_signal(si,sec,sic,roc,poc)
    && !signal_get_virtual(si,sic))
    -> ((!(result == signal_get_colour(si,sic)))
    -> setSignalSend(si, result)
    .Interlocking(sic = signal_update_colour(si, result, sic)));
```

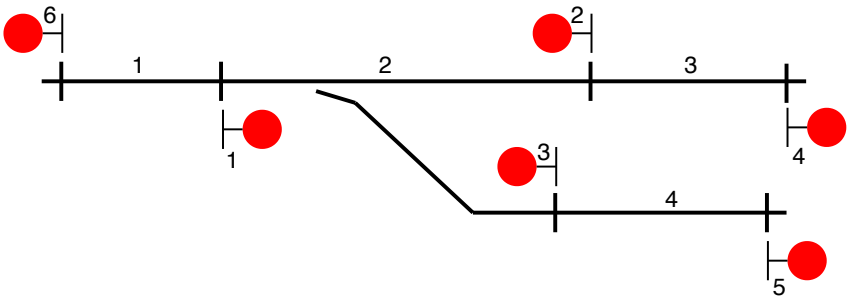
The process *InterlockingUpdatingSignal* sums over all the signals and all the colour aspects, requiring that the colour aspect matches the computation of the aspect for that signal. If the computation is different than the current aspect, the signal is updated. It then continues as the main process.

5.2.3 Initialization and configuration

The model needs to be configured for a specific track layout and the processes need to be initialized in accordance with this track layout and the initial state of the field elements. Multiple configurations can be included in a model, between which can be switched by changing a single variable. The processes are created and initialized based on the configuration. If, for example, the configurations specifies that the track layout contains 5 points, then 5 Point processes will be created each of which is initialized with their ID and their initial position. The configuration also specifies the number of trains and which section they will enter on. The code below shows configuration 3, visualized in Figure A.1.

```
sections_config(3)(1) = section_info(free, true, [section_connection([],efp,0,2)]);
sections_config(3)(2) = section_info(free, false,
    [section_connection([point_position_pair(1,left)],flank_protection(2,[],[3]),1,3),
    section_connection([point_position_pair(1,right)],flank_protection(2,[],[2]),1,4)]);
sections_config(3)(3) = section_info(free, true, [section_connection([],efp,2,0)]);
sections_config(3)(4) = section_info(free, true, [section_connection([],efp,2,0)]);
signals_config(3)(1) = signal_info(RD, R, false, 1,2);
signals_config(3)(2) = signal_info(RD, L, false, 3,2);
signals_config(3)(3) = signal_info(RD, L, false, 4,2);
signals_config(3)(4) = signal_info(GR, R, true, 3,0); %virtual signal always green
signals_config(3)(5) = signal_info(GR, R, true, 4,0); %virtual signal always green
signals_config(3)(6) = signal_info(GR, L, true, 1,0); %virtual signal always green
points_config(3)(1) = point_info(2, right);
trains_config(3)(1) = train_config(4,L,false);
trains_config(3)(2) = train_config(4,L,false);
routing_table_config(3) = [signal_pair(1,4), signal_pair(1,5), signal_pair(2,6),
    signal_pair(3,6)];
```

Figure 5.2: Example track layout



5.3 Variants internal actions

During the modelling process the question arises on how to model the internal behaviour of the interlocking that does not directly lead to observable actions. It is clear that when certain conditions are met, a visible action `setSignal` may occur, but how to model that, for example, a signal is marked as available again for a new route? It could be modelled by an (unobservable) action of the interlocking but it could also be made available instantly when it set to show stop after a train has passed the signal. In essence these choices influence what order(/interleavings) of internal decisions are possible. As no specification for these internal transition was available we decided to investigate what impact these choices have by creating several variants of the model.

Variant A

The first variant is highly non-deterministic; it allows any order of internal behaviour. There are four internal actions regarding the life cycle of the route (see Figure 4.3): `readyRoute, notReadyRoute, activateRoute, discardRoute`. As a train moves along a route the signals and sections part of that route become free for other routes. Removing sections and signals from a route is modelled using the internal actions `freeUpSignal` and `freeUpSection`.

Variant B

The second variant reduces the number of internal actions induced by the interlocking process. When a section becomes free the section can be removed from the route and if it was the last section of a route the route can be discarded. Variant A uses an internal action to free up a section and to discard a route, in variant B the routes are updated directly when a section becomes free using data equations.

```
Interlocking(sec = section_update_status(se,free,sec),
    roc = routes_handle_section_free(se,sic,poc,roc))
```

Similarly, when a section becomes occupied data functions are used to possibly activate a route and when a signal is set to show stop it is considered whether the signal can be freed up. It is more difficult to get rid of the actions `readyRoute` and `notReadyRoute` as they could be performed after a change of section occupancy or by setting a route. Especially when a route is set that connects an active/ready route to a locked route then the new route and the route after it both become ready. This happens in a specific order as first the intermediate route becomes ready and only then can the last route become ready. This is hard to do in data equations as the order in which the routes might become ready would need to be determined. For this reason the internal actions `readyRoute` and `notReadyRoute` remain in variant B.

Variant C

The third variant is similar to variant B, they share the concept of reducing internal actions by using data equations. The difference is that in variant C, the interlocking is always up to date concerning section occupations. This is achieved by including the interlocking in the communication to set the occupancy of a section, using the multi-action construction of mCRL2. This is realistic if the time between moving onto a section and the interlocking being informed is negligible. This should further reduce the state space induced by the model. Note that this does not prevent a train from moving to the next section as the interlocking can always perform the action `getStatusSectionRec`.

```
setStatusSectionSection|setStatusSectionTrain|getStatusSectionRec -> setStatusSection
```

Chapter 6

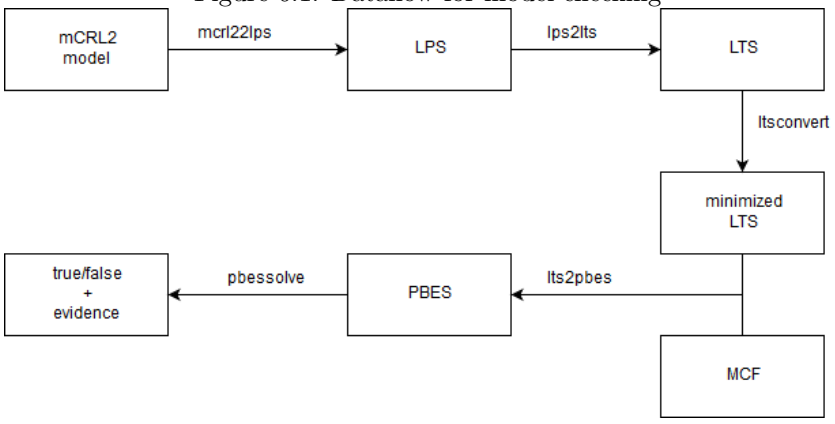
Model checking

The formal model, all variants of it, are analysed for safety. In this chapter the toolchain for model checking is presented and the properties to be verified are formalised. The results of model checking are presented as well.

6.1 Tool chain

The mCRL2 toolkit provides a collection of tools to analyse mCRL2 models. Let us take a closer look at the toolchain used for the verification of the model. The first step of the analysis of a model is to convert it to a Linear Process Specification (LPS). During the development of the model, lpsxsim is a useful tool to simulate the model to find mistakes and to verify whether a change in the model has the intended effect. To prove a property (a modal μ-formula) for a model, a Parameterized Boolean Equation System (PBES) can be generated and solved. The PBES can be generated directly from the LPS with lps2pbes but solving such a PBES is relatively slow compared to another strategy: first generating the LTS. With the tool lps2lts the LTS can be generated. The LTS can be made smaller by computing the smallest branching bisimilar LTS using ltsconvert. Using lts2pbes the PBES can be obtained, which can be solved with pbessolve, answering whether the given formula holds for the model. By selecting the counterexample option in lts2pbes a counterexample is generated by pbessolve in the case that the formula is false. This counterexample is an LTS file containing the trace that disproves the formula, which can be inspected using ltsgraph. Figure 6.1 shows the sequence of tools used to check a property for the model. To run all the tools in one go the bash script found in appendix B.1 can be used.

Figure 6.1: Dataflow for model checking



6.2 Modal μ-formulas

6.2.1 Safety

As mentioned in Chapter 4, the requirements on which we focus are collisions and derailments. As we only consider collisions within the yard and disallow shunting movements, the requirement for collisions reduces to: a section may never be occupied by two trains. If we formulate this in terms of actions in the model we would like to verify that it can never happen that $setStatusSection(section\_id, true)$ occurs

twice without $setStatusSection(section\_id, false)$ in between, for every section ID. This is captured by the following modal μ-formula:

```
forall section_id:Nat. (val(section_id <= last_section))
    => [true* . setStatusSection(section_id, true)
    . !setStatusSection(section_id, false)*
    . setStatusSection(section_id, true)] false.
```

Let us explain the formula. The universal quantifier has the effect that the formula after it is checked with every *section_id* below *last_section*. The construction $[]false$ states that the sequence of actions between the brackets should not be possible. The sequence between the brackets consists of several parts, separated by dots, which indicate that the subsequences should follow one-another. The formula $true*$ specifies a sequence of zero or more actions with no requirement on which actions this sequence can consist of. The formula $!setStatusSection(section\_id, false)*$ specifies a sequence of zero or more actions that does not contain the action $setStatusSection(section\_id, false)$.

As, in the model, points instantaneously change position, it suffices to verify that a point is not moved while the section in which it is located is occupied. To make verification easier, the section ID is included in the communication between a point and the interlocking. Formulated in terms of actions in the model we would like to verify that it can never happen that an occurrence of $setStatusSection(section\_id, true)$ is followed by $setPositionPoint(point\_id, left, section\_id)$ or $setPositionPoint(point\_id, right, section\_id)$ without $setStatusSection(section\_id, false)$ in between. The following modal μ-calculus formula captures this requirement:

```
forall section_id,point_id:Nat.(val(section_id <= last_section)
    && val(point_id <= last_point))
    => [true* . setStatusSection(section_id, true)
    . !setStatusSection(section_id, false)*]
    ([setPositionPoint(point_id,left,section_id)]false
    && [setPositionPoint(point_id,right,section_id)]false).
```

6.2.2 Liveness

An interlocking that would set all signals to show stop at all times would satisfy the safety requirements. It is therefore desirable to also prove some properties concerning liveness. A minimal requirement is that the interlocking can not end in a deadlock: a state in which it can no longer perform any action. The absence of deadlocks in the interlocking can be expressed as always being able to do a next action, captured in the following formula:

```
exists signal_id1,signal_id2:Nat.
  (val(signal_id1 <= last_signal) && val(signal_id2 <= last_signal))
  => [true*](<requestRoute(signal_id1,signal_id2)>true
    || <routeAccepted>true || <routeRejected>true).
```

The interlocking should always be able to do at least one of these actions.

A stronger liveness property in this context would be that every train that enters the yard reaches the other side. Unfortunately this is not always the case as two trains facing each other without options to pass each other can not reach the other side of the yard. For some track layouts, however, it is to be expected that all trains can cross the yard. Suppose that we have a configuration where all trains enter the yard on the right side and there is a single section on the left side denoted by Z connected to the open track. Also suppose that for each entry section on the right side the section on the left side is reachable. In this case it is expected that eventually all trains are able to cross the yard. The property can be formulated as a modal μ-formula as follows:

```
nu X(current:Nat = 0, total:Nat = last_train).
    ([!setStatusSection(Z,false)]X(current,total)
        && [setStatusSection(Z,false)]X(current+1,total)
    && (mu Y(current2: Nat = current, total2:Nat = total).
        (<!setStatusSection(Z,false)>Y(current2,total2)
            || <setStatusSection(Z,false)>Y(current2+1,total2)
        || val(current2 == total2)))).
```

The formula uses two fixpoint operators to express that after any trace, it should always be possible for all trains to cross the yard within finite steps. Note that, purposely, this formula does not express that all trains always cross the yard within finite steps. Such a formula would not hold as the behaviour of the interlocking contains loops of routes being requested and rejected.

If a train leaves behind a logical occupation this might prevent other trains from being able to cross the yard and reach the open track via section Z. In this case we would still require that the model contains at least a trace where all trains cross the yard, expressed in the following formula (assuming there are two trains):

```
<true*.setStatusSection(Z,false).true*.setStatusSection(Z,false)>true.
```

6.2.3 General checks

The property regarding collisions and derailments rely on how setStatusSection and setPositionPoint are used. More specifically, we assume that when a train wants to set the status of a section, no other process can prevent this. In the case of the safety property concerning moving points we assume that if the interlocking wants to set the position of a point, no other process can prevent this. We would like to verify that these underlying assumptions are indeed correct.

It is rather hard to verify these properties as it is hard to pinpoint when a train wants to communicate with a section to make it occupied. Similarly, it is hard to capture in a formula when the interlocking wants to communicate with the point to change the position. To make it easier to verify these properties we add actions wantSetStatusSection and wantSetPositionPoint in the model which trains and the interlocking use to announce when they want to engage in setting the occupancy of a section or the position of a point, respectively. With these actions the following two properties should hold:

```
forall section_id:Nat. (val(section_id <= last_section) && val(section_id >= first_section))
  => ([true*.wantSetStatusSection(section_id,true).(!setStatusSection(section_id,true))*]
    <setStatusSection(section_id,true)>true),

forall point_id:Nat. (val(point_id >= first_point) && val(point_id <= last_point))
  => (exists section_id:Nat. val(section_id <= last_section)
  && [true*.wantSetPositionPoint(point_id,left,section_id)
  .(!setPositionPoint(point_id,left,section_id))*]
    <setPositionPoint(point_id,left,section_id)>true
  && [true*.wantSetPositionPoint(point_id,right,section_id)
  .(!setPositionPoint(point_id,right,section_id))*]
    <setPositionPoint(point_id,right,section_id)>true).
```

It is also good practice to verify general properties, in order to 'debug' the model. A property we would like to verify is that eventually every route request is either accepted or rejected by the interlocking. The following formula expresses this property:

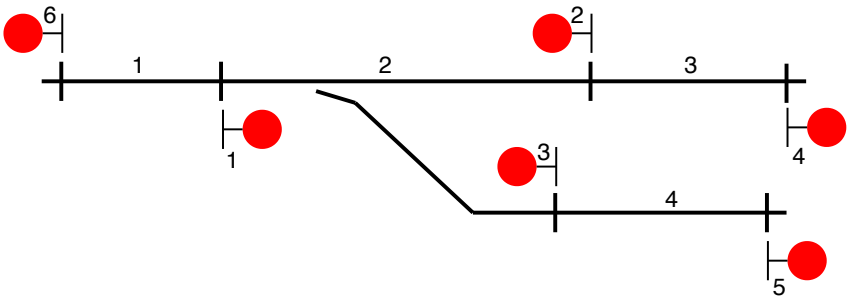
```
forall signal_id1,signal_id2:Nat.
  (val(signal_id1 <= last_signal) && val(signal_id2 <= last_signal))
    => ([true*.requestRoute(signal_id1,signal_id2)]
      (!(nu X. <!(routeAccepted || routeRejected)>X) && [true*]<true>true)).
```

It expresses that after requesting a route it is not possible to perform an infinite sequence of actions that does not include routeAccepted or routeRejected.

6.3 Verification

For the verification of these properties the toolchain described in Section 6.1 was used. The model needs to be instantiated with a track layout. This yard should contain a wide variety of possible scenarios as the safety properties will only be proven for this yard. On the other hand the state space of the model should be small enough to perform the verification so the track layout should not contain too many sections, possible routes and trains. The yard depicted in Figure 6.2 was used. Two scenarios were tested on this track layout: a scenario where one train may enter on section 1 and one trains may enter section 4 and one with two chasing trains both entering on section 4. To verify the stronger liveness property a different train configuration is used: two trains may enter on section 4 and only routes from signal 3 to signal 6 may be requested. The corresponding mCRL2 code for the configuration can be found in Section A.3. This layout was chosen as it contains a point, routes that may conflict head on or on flanks and it contains trains following each other on the same route.

Figure 6.2: Track layout used for verification



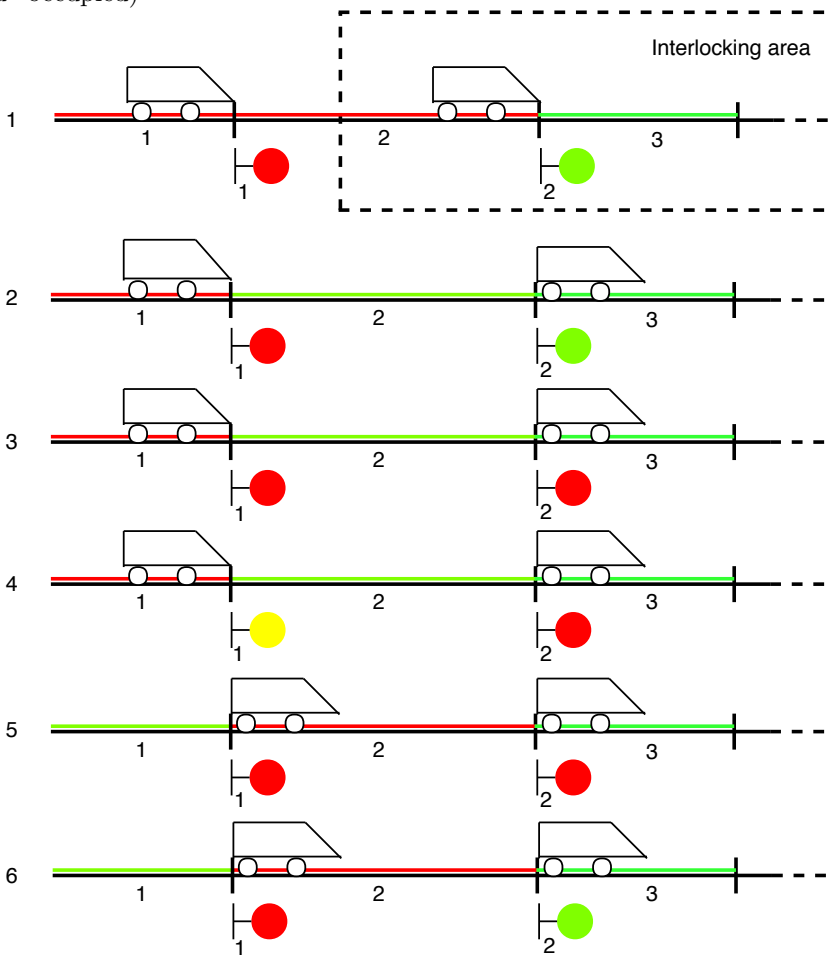
6.4 Results

For variant C the safety properties could be proven. In variant A and B however only the safety property concerning points could be proven to hold. Collisions in these variants can occur. More specifically, collisions can occur between chasing trains when trains disappear in the view of the interlocking. None of the variants contain deadlocks and it is possible for all trains to cross the yards. The stronger liveness property where it is required that all train always cross the yard only holds for variant C. This is the case as in the other variants it is possible for a train to leave behind logical occupations, preventing the chasing train from being able to cross the yard as well. For variant C the general properties were also investigated and were proven to hold.

6.4.1 Disappearing trains

The scenario depicted in Figure 6.3 shows how a dangerous situation can be reached (in variant A and B). The scenario begins with a train on the entry section (section 2) of the yard with a route set from signal 2 to some signal further ahead. The train passes the signal and section 2 is seen by the interlocking to be free but the section after the signal is not yet seen as occupied. This causes the entry signal to be set to show stop again in step 3. Signal 1, facing the chasing train, is set to show yellow and the chasing train enters the entry section of the interlocking area. As the section before the entry signal of the route is now occupied signal 2 is set to show green, creating a dangerous situation.

Figure 6.3: Scenario showing dangerous situation, colour depicts occupancy as seen by the interlocking (green=free, red=occupied)

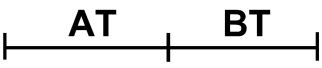


According to signalling experts, this scenario is not very realistic as by the time that the chasing train has started rolling and the interlocking has seen the entry section becoming occupied, it will also have seen the section after the entry signal becoming occupied. Still, it does lead us to ask what requirements there are on the timing of seeing section occupancy by the interlocking. These requirements can be translated to requirements on the length of sections and/or the maximum speed of the train.

Mitigation

The dangerous situation is dependent on the order in which section occupations are observed by the interlocking. The engineering firm Sweco has identified five faults that can occur due to timing issues regarding reading occupancy status [12]:

1. Section BT becomes free before AT becomes free
2. Section AT becomes free before BT becomes occupied
3. Section BT becomes occupied before AT becomes occupied
4. Section AT or BT is occupied too shortly to be seen by the interlocking
5. Section BT becomes free too shortly after AT becomes free



By making sections long enough and adding delays in some situations these five scenarios can all be excluded during normal operation. Excluding these scenarios in the model however, is rather difficult

as the order AT-BT is dependent on the direction the train is heading. It turns out that for variant B not all of these five faults need to be excluded for all sections, only scenario 2 needs to be excluded for entry sections and the section afterwards. So if AT is an entry section the interlocking can not read that section AT is free before it reads that section BT is occupied. It was proven that the safety properties hold for a model with this scenario excluded. It is, however, unclear if scenario 2 can occur in real life in rare events. We will refer to this model as variant B1.

Another way to exclude collisions is to make the entry section part of the route. The dangerous situation is possible because the entry section is not part of the route and can therefore not become logically occupied. Note that this deviates from how real interlocking systems operate. It is unclear why it was chosen to not include the entry section in routes. We will refer to this model as variant B2.

The following practical conclusions can be drawn from these results. An idealised version of the interlocking where the interlocking is always up to date regarding section occupations is safe. Letting go of this idealised version we either require that section lengths are chosen in such a way that some timing issues are excluded or that entry sections become part of a route. Regarding timing issues this also means that detection problems due to rust should be excluded. Furthermore, as the properties were not proven for variant A, we also require that after updating a signal or reading the occupancy status of a section, the interlocking simultaneously updates affected routes. If this is deemed to be unrealistic or too difficult, then further research is needed into making variant A safe.

6.5 Comparing variants

As we have multiple variants of the model, we can ask ourselves: are the models behaviourally equivalent? It may be obvious that the variants that do not satisfy the safety requirements are not equivalent to the variants that do satisfy the safety requirements. The equivalence of the safe variants can be established by comparing their LTSs. By using the tool ltscompare to check for bisimulation relations it can be established what variants behave in the same way and what behaviour is possible in one variant and not in another. A bisimultaion relation relates two transition systems that are behaviourally equivalent to an observer.

It turns out that all variants are behaviourally different. The differences are caused by differences in which sections can become logically occupied.

6.6 Toolkit performance

We measured the performance for the scenario with two chasing trains. Even with the small track layout used, model checking is very demanding in terms of resources. Table 6.6 shows how many states and transitions the LTS each variant of the model consists of and how much time the various steps of the model checking toolchain take in seconds. For lts2pbes and pbessolve, which are executed for every formula, the average time is shown. These results are from the scenario with two chasing trains. The computations were done on a computer cluster of the Mathematics and Computer Science department of the University of Eindhoven. The cluster consists of Intel Xeon E5520 processors (2.26 GHz) and has 1TB of RAM. The tool lps2lts used the most memory, for variant A lps2lts used 41GB of memory. Using three trains in the configuration, one entering on section 1 and two entering on section 4, significantly increases the amount of time need to do the verification. Generating the state of variant A was not finished after 5 hours and was using 140GB of memory.

	Variant A	**Variant B**	**Variant B1**	**Variant B2**	**Variant C**
#states	151060	110229	7170	9837	1713
#transitions	741761	483428	26015	38929	4925
#states (minimized)	5506	4621	577	598	190
#transitions (minimized)	29299	23131	2322	2421	640
mcrl22lps	434	446	465	529	461
lps2lts	1637	1615	314	388	230
ltsconvert	8	6	1	1	1
lts2pbes (average)	5	4	1	1	1
pbessolve (average)	10	8	1	1	1

Chapter 7

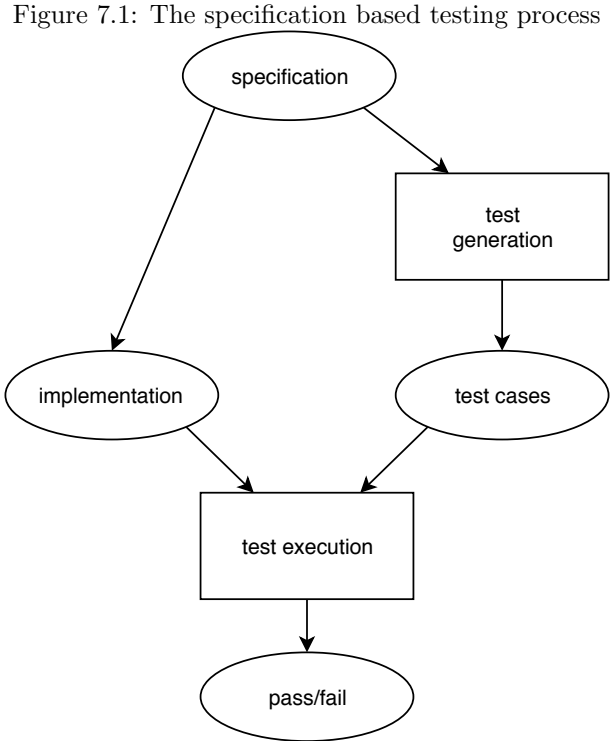
Exploring automated testing

Now that we have a model for which it is proven that safety properties hold for small yards we would like to test whether the interlocking software used by Siemens conforms to this model. Automated testing on the basis of the formal model is the goal. In this chapter the formal test theory is introduced as well as an automated testing tool compatible with the mCRL2 toolkit. The toolchain to create an automated testing platform using these existing tools is presented in Chapter 8.

7.1 Theory of model-based testing

Jan Tretmans first introduced a framework for formal testing in his PhD thesis [18]. Since then the theory has developed; the input output conformance (**ioco**) testing theory [19] underpins the tools used in our automated testing platform. What makes the **ioco** testing theory formal is that it has a strong mathematical foundation, the theory is grounded in Labelled Transition Systems (LTS). The principle elements of formal testing are a formal specification and an implementation of the specification: the system under test (SUT). The theory considers both the specification and the implementation as an LTS (while in practice the SUT is of course not an LTS). From the specification, test cases can be generated using an algorithm, which can be executed on the implementation, resulting in a pass (the implementation conforms to the specification) or a fail (the implementation deviates from the specification). The test derivation algorithm and when an implementation passes a test case are, again, formally defined using LTSs.

Figure 7.1: The specification based testing process



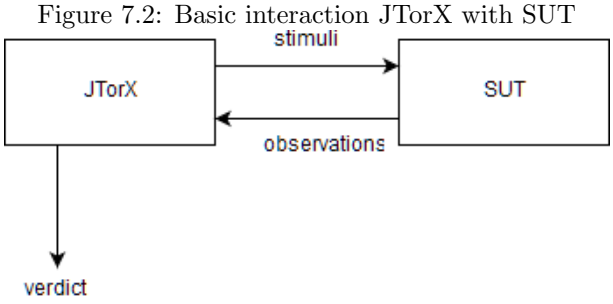
In the **ioco** testing theory the SUT is considered to be a black box and is actively stimulated during

testing. The specification is given by an input-output labelled transition system. The set of labels is partitioned into a set of input labels and a set of output labels, which are disjoint. As observing no output for a certain amount of time can also be considered an event, the notion of quiescence is used. A state is quiescent if it has no output action. The specification is enriched by adding a selfloop $p \xrightarrow{\delta} p$ for every quiescent state p. Traces from this transition system, which may consist of input actions, output actions and δ, are called suspension traces or Straces. The **ioco** relation between implementation and specification is said to hold when for any suspension trace in the specification, the enabled output actions of the implementation after performing that trace is a subset of the enabled output actions of the specification. In other words, the specification predicts what actions are enabled, possibly allowing for more behaviour than the implementation. Note that this is just an informal description of the formal theory. For the formal definitions of the **ioco** theory, the reader is referred to [19]. These definitions formally define when an implementation conforms to the specification, both given as an LTS. Recall that the formal semantics of an mCRL2 specification is an LTS. Thus, using the **ioco** testing theory we can formally define when an implementation conforms to an mCRL2 model.

7.2 JTorX

JTorX [2] is a model-based testing tool based on the **ioco** testing theory. It is a Java implementation with a graphical interface of its predecessor TorX [20]. JTorX derives from a given specification which stimuli it can send to the SUT and which observations it should expect.

Figure 7.2: Basic interaction JTorX with SUT



JTorX accepts Labelled Transition Systems in various formats such the Aldebaran (.aut) format, it supports Symbolic Transition Systems (.sax) and supports mCRL2 models that represent an infinite transition system via lps2torx. JTorX can be run from the command line or with a Graphical User Interface (GUI). The SUT can either be another model or an implementation. JTorX can communicate with the implementation over standard input/output, via TCP or via a user provided adapter. The adapter is then responsible for the communication with the SUT and, if necessary, should also initialize and close down the SUT. Testing is either done manually by clicking which stimulus should be sent or automatically by letting JTorX pick stimuli at random. As JTorX can also be run via the command line it is possible to create batch scripts to run many tests in one go. All tests can be logged, including the sequence of stimuli and observations and the final verdict. Logs can be played back on the SUT. More information on all features of JTorX, instructions on how to use the tool and downloads can be found on the web page [4] of JTorX. For more information on the theoretical foundation of JTorX we refer the reader to the PhD thesis of Axel Belifante [3], who developed JTorX.

Chapter 8

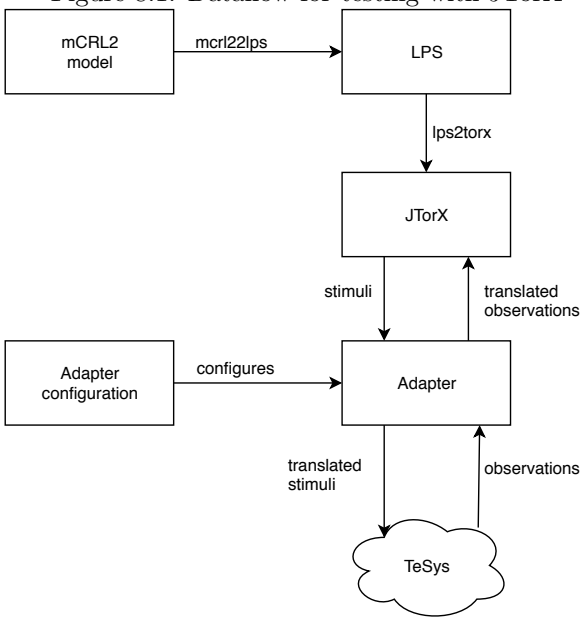
Testing platform

In this chapter the setup of the tool chain for testing with JTorX will be discussed, including interfaces, data-flows and configurations. It will be explained how JTorX needs to be configured, how TeSys is set up properly and documentation is given for the adapter that allows communication between JTorX and TeSys.

8.1 Adapter

As JTorX and TeSys do not share the same names for commands and elements, some translation is needed. Moreover, there is no Application Programming Interface (API) available for TeSys so enabling a connection at all between JTorX and TeSys is a task for the adapter. Unfortunately, no clear description of the inner workings of TeSys is available at Siemens Nederland nor could it be obtained from the development team in Germany. Some reverse engineering was thus needed to discover what options are there to establish a connection with TeSys. It turns out that communication between various subcomponents of TeSys is mostly done via named pipes and some TCP connections are set up between components, providing an opportunity to connect the adapter on one of these communication channels.

Figure 8.1: Dataflow for testing with JTorX



8.1.1 Connection to TeSys

As TAK (see Section 3.3) has similar communication with the rest of TeSys as our adapter (sending commands and receiving response messages from the interlocking), it was the perfect candidate to investigate its communication channels. The TAK tool turns out to consist of two processes: TAK.exe, which runs the GUI and taktakprocess.exe, which communicates with the rest of TeSys. Between these processes a number of named pipes are set up to exchange data and events. Two of these pipes are of

particular interest for our purposes: commandpipe and telegrambuffer. Commandpipe is used to send messages from TAK.exe to taktakprocess.exe to initiate the run of a TAK script and telegrambuffer is used by taktakprocess.exe to forward messages from the interlocking to the GUI of TAK.exe. These two pipes allow us to connect the adapter to taktakprocess.exe. By killing TAK.exe and letting the adapter connect to telegrambuffer and commandpipe, the adapter has the access it needs to TeSys.

The adapter is implemented in Java and Python. The main functionality is implemented in Java but as reading from telegrambuffer did not always go well (it could not always the determine when it received all the bytes of a message) a small Python script is used to read from telegrambuffer which forwards all the messages it receives to the Java program over a TCP connection.

To send commands using commandpipe, a small workaround is needed. Commandpipe is not used to directly send commands to taktakprocess.exe. Rather, it instructs it to open a TAK script and play it using certain settings. The solution to send a single command is then to write the command to a TAK script NextAction.kat and sending the following instructions to taktakprocess.exe over commandpipe:

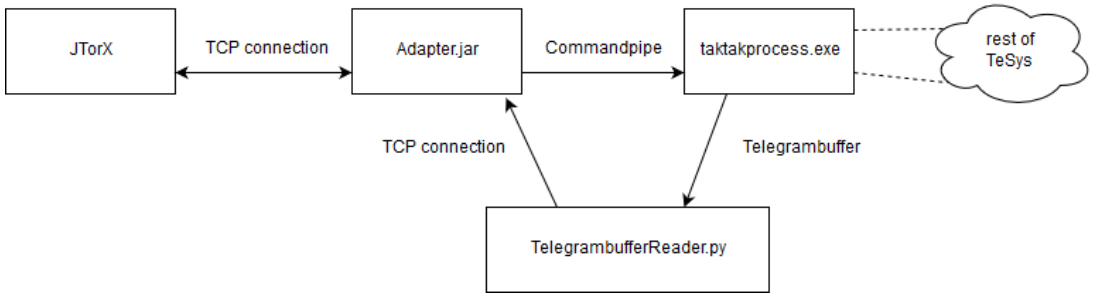
```
Close PATH_TO_NextAction.kat
Open PATH_TO_NextAction.kat
Mode_Single
SetOfflineMode Off
SetTimer 1000
Start 1
```

Processing the messages coming from the interlocking is relatively straightforward: it is a simple string in the language used by TAK with some bytes around the string of which it is unclear what the meaning is, possibly some counter, timestamp or information on the length of the message. The adapter inspects if the message contains information that needs to be sent to JTorX and if so translates the message to the format understood by JTorX.

8.1.2 Connection to JTorX

JTorX offers several ways to make a connection to the SUT: via standard input/output; via a TCP connection; or using a TorX adapter. By connecting the adapter to JTorX using a TCP connection, JTorX will simply view the adapter as the SUT. The benefit of using a TCP connection with the adapter is that the adapter does not need to be restarted every time JTorX starts a test. Instead, the adapter can be started once and JTorX can connect to the adapter once it wants to start a test. This prevents problems connecting to taktakprocess.exe which sometimes occur when the adapter is started over and over again.

Figure 8.2: Connections between the sub-modules of the adapter, JTorX and TeSys



8.1.3 Configuring the adapter

As the adapter needs to be able to translate section and signal names from the mCRL2 model to the corresponding names in TeSys, back and forth, there is the need for a configuration file for the specific track layout. This configuration file is passed to the adapter as an argument. It has the following simple format. The first line has the format "config: NUM_SECTIONS, NUM_SIGNALS, NUM_POINTS", where NUM_SECTIONS, NUM_SIGNALS and NUM_POINTS are natural numbers, specifying the largest ID of any section, signal and point, respectively. Subsequent lines specify the configuration for individual elements. For sections the format is "section: MCRL2_ID, TESYS_ID", where MCRL2_ID is a natural number and TESYS_ID is a string (without quotes). For signals the

format is "signal: MCRL2_ID, TESYS_SECTION_ID, TESYS_SIGNAL_ID", where MCRL2_ID is a natural number and TESYS_SECTION_ID and TESYS_SIGNAL_ID are strings (without quotes). The TESYS_SECTION_ID corresponds to the section on which the signal is positioned and is used to specify a route and TESYS_SIGNAL_ID corresponds to the actual ID of the signal, which is used to recognize that a signal aspect is set. Finally, for points the format is "point: MCRL2_ID, TESYS_ID", where MCRL2_ID is a natural number and TESYS_ID is a string (without quotes).

Example configuration file:

```
config: 2,1,1
section: 1, E10_HS_S80
section: 2, E10_HS_S82
signal: 1, E10_HS_S82, E10_SOM6_AS82
point: 1, E10_HW_W145
```

8.2 Running tests

The workflow to run tests using JTorX is quite involved as a number of steps need to be performed manually to set up the TeSys test environment and the adapter. In this section we will examine how to boot and configure all the relevant systems and tools.

TeSys consists of many different components that all need to be started and connected to each other. TeSys includes a script that starts all the subcomponents. Once it has started, the field elements need to be cleared and the user needs to log in as a traffic controller to be able to set routes. Table B.3 shows what steps need to be performed in what order. The JTorX adapter is added as a folder in the TeSys package. It includes Adapter.jar, Telegrambuffer.py, the configuration file of the adapter and a script that closes TAK.exe and starts the adapter. The track configuration file (see section 8.1.3) needs to be configured beforehand.

JTorX can be started by executing the jtorx.bat script in the installation folder. Table B.1 shows how JTorX is to be configured to properly connect it to the adapter. Note that it is convenient to uncheck all the boxes in the show field in the test tab as this prevents JTorX from opening a number of unnecessary windows. Prerequisite for using JTorX are that an LPS needs to be available, which can be obtained by instantiating the mCRL2 model with a track layout and running mcrl22lps. We used variant C to test the interlocking.

8.3 Results JTorX testing

By testing with JTorX, several inconsistencies between the model and the interlocking software of TeSys were found. None of the differences implied an error in the interlocking software, but rather showed flaws in the model. An example of an inconsistency is that the original model specified that after requesting a route, the next action of the interlocking is to accept or reject the route, no other action could come in between. It turns out that when a route is requested and some section occupation is changed shortly after, the interlocking might respond to the section occupation (by updating a signal aspect) before it accepts/rejects the requested route. These inconsistencies were fixed in the model, requiring that the model checking phase was repeated. In the context of this project where formal methods were applied after the system was built, testing with JTorX gave feedback on whether the model accurately captures the behaviour of an interlocking.

8.3.1 Test selection

As it is not possible to test the behaviour of the model exhaustively due to the size of the state space describing the behaviour of the interlocking, the question of test selection comes into play. What 'kind' of tests are important and does JTorX produce meaningful test cases from the model? For a large part the test cases that JTorX can derive are restricted by the model itself, more specifically, the train processes simulate a realistic order of section occupations. If these restrictions were not present in the model, JTorX, using random stimulus selection, would derive unrealistic test cases with trains appearing and disappearing all over the yard. Initially when we started testing the model, the model did not restrict route requests as much. JTorX thus derived test cases where it would request routes all over the yard (blocking other routes from being accepted) as opposed to more realistic scenarios where the signalman would request routes guiding a train from an entry section at the border to the other side of the yard. JTorX does not support ways to select these more realistic scenarios but the restriction could be placed

in the model by creating a signalman process that requests routes in a more realistic way. The downside of reducing what routes can be requested is that coverage is reduced. It is desirable to also test orders of route requests that are outside the range of normal operation. One of the strengths of automated test derivation is that it can derive 'weird' tests that an engineer might not think of. In this case the choice was made to restrict the route requests as it only rarely happened that a route was requested that could lead to an actual train movement.

8.3.2 Performance and stability

Performance of the tool chain with our model for testing was surprisingly bad. At first, a part consisting of 36 sections was chosen within the yard of the TeSys installation. After attempting to linearise the model for several hours the process was still not finished. After a lot of troubleshooting we were able to determine that the equations used to determine the signal aspect were causing the performance issue. Presumably, the rewriter needed many rewrite steps to rewrite the equation to true or false. The original equations iterated through the sections from a given signal until the next signal was reached, checking whether the sections in between are free. This was replaced with an equation that takes a signal as argument and checks whether there exists a route with status ready of which all the sections are free. This reduced the performance issue for linearisation and enabled us to generate the LPS of yard with 36 sections within half an hour. It does restrict what routes can be requested: only routes where there are no signals along the route are possible. The requirement that all sections of the route are free is stronger than the requirement that the sections until the next signal are free, unless the next signal is always the exit signal of the route.

While actually performing tests with JTorX more performance issues came to light. The first few testing steps are performed with reasonable performance but it quickly declines and memory usage increases with every step. For the yard of 36 sections it takes over half an hour per testing step and memory usage was at 8GB after 10-20 steps. The CPU load and memory usage are not from JTorX itself but from the lps2torx tool that provides access to the LTS to JTorX (without computing the entire LTS). Using a smaller yard with only 7 sections significantly sped up performance but was, considering the size of the yard, still not great. After 50-70 steps the system took approximately half a minute per test step with memory usage at 4GB. It is unknown what the exact cause of the performance issue is, but there are a few possibilities worth mentioning. Using lpsxsim to simulate the model is also rather slow and uses more memory with every step. It could be that the rewriter still needs many steps to rewrite certain equations. It was interesting to observe that both lps2torx and lpsxsim use an increasing amount of memory, even when it loops back to the same state (requesting routes that are rejected). An extra factor for lps2torx is that JTorX keeps a set of states the system might be in (as it can not observe internal actions). This might lead to more computations by lps2torx.

Another issue with the testing platform is stability. More specifically, the stability of the connection of the adapter with TeSys is an issue. The process taktakprocess.exe, which the adapter connects to, regularly crashes. As the connection, as described in 8.1.1, is somewhat of a hack, it is not surprising that the stability leaves something to be desired. As we could not obtain a proper description of the internal workings nor an API description, it is hard to troubleshoot the stability issues. The development team of TeSys/TAK should be able to resolve the issue.

Chapter 9

Discussion and conclusion

Recall that the initial goal was to explore the use of formal methods, and in particular the use of mCRL2, to verify interlocking design and automatically test interlocking software. In this chapter we will reflect on the results discussed in previous chapters and the suitability of the mCRL2 toolkit. We will also discuss more broadly how mCRL2 could be integrated in the interlocking design process and give recommendations on how the testing platform presented here could be improved.

9.1 Discussion of results

9.1.1 Model checking

The results discussed in Chapter 6 prove that the model instantiated with a specific track layout does not contain unsafe states. Possible threats for the validity of the results are that the safety properties may not hold for other track layouts or that the model does not describe the behaviour of the actual interlocking. As described in Section 6.3, the track layout used, was chosen such that it is small enough to analyse but does contain a variety of scenarios. To mitigate the risk of the model not properly describing the behaviour of actual interlockings, two tactics were used: combining different sources describing the internal behaviour and using the model to test the implementation. Using the model for testing contributed in this context where an implementation was already present, showing that the model can predict the behaviour of the existing interlocking.

The scope of model is restricted purposely, not all features related to safety are included in the model. Level crossings are not included and safety on the open track is not considered. The results therefore only apply to the two safety criteria and our model. Model checking is a powerful tool for detecting flaws in complex systems. If there would have been a flaw in the behaviour of the interlocking included in our model, model checking might have uncovered it. The fact that the safety properties of interest were proven to hold for the model increases confidence in the correctness of the system. Besides increasing confidence, the process of verifying the model also brought up interesting cases that give insight on what assumptions are crucial for safety. Formal verification directs the verification process and gives better insight into why the system is safe.

9.1.2 Model-based testing

Model-based black box testing, can never guarantee that there are no flaws in the implementation. Inputs or environmental factors that are not part of the model can not be tested for. If, for example, a system would only malfunction when it receives a message at precisely 6:23 AM and all tests are run during the day, the error would not be discovered. Moreover, exhaustive testing the behaviour of the model is not feasible for interlocking software as the state space is too large. We can therefore not prove that the implementation satisfies the safety properties proven in the model, the best thing possible is to demonstrate that the implementation passes a large number of diverse test cases. A risk related to automated model-based testing is that the model might be incorrect and accepts unsafe behaviour of the implementation. This is mitigated by model checking the model used for testing.

Model checking and model-based testing complement each other. Model checking increases confidence in the model used to test the implementation. Testing, in turn, helps to verify that the mCRL2 model describes the behaviour of an interlocking accurately. In this way, different formal methods can strengthen each other in the development and verification of (signalling) systems.

9.2 Test coverage

Coverage is an important aspect of testing software systems in general but even more so within the signalling domain where safety criteria are high. Let us take a closer look at the coverage requirements currently used for manual testing and what level of coverage can be achieved and documented using an automated testing platform.

Currently, test cases for a specific yard are derived from a standard set of more abstract test specifications. This standard set of tests has been put together by signalling experts. An example of one of these test specifications is: request every route that is in the routing table and simulate a train following the route. Such a test case is accompanied with a list of requirements on the behaviour of the interlocking during the test case. To validate a configured interlocking, all test cases are performed and the results documented to supply proof that the interlocking has been thoroughly tested.

At the moment our automated testing platform does not guarantee that such a set of standard tests are performed as the tests are generated randomly. It also does not provide proper documentation of which parts of the system have been tested. There are two ways to alleviate these issues. One way is to build-in a documentation tool that analyses and records which parts of the system have been tested. This is not an ideal solution as the tests that have not been covered would still need to be done manually. Another solution is to adapt the model to purposely work through a set of tests. The train and signalman processes in the model could be replaced with a test manager that performs all tests. It should, for example, be quite easy to construct a test manager that sequentially requests a route and simulates a train moving over the sections. In such a setup it should also be straightforward to prove and document that all tests have been performed. Another advantage of this approach would be performance as the behaviour described by the model is much more restricted: the test manager would ensure that there is a single order of route requests and section occupations. This might increase performance of lps2torx and it might even be possible to generate the state space and provide JTorX with an LTS, which we expect would increase performance of JTorX even more. The possibility of generating the state space for larger yards is speculative as the interleaving between the test manager and the interlocking might still lead to a state space explosion. Besides these directed tests, random testing is still useful as it can perform tests outside of the range of standard tests, after all, the standard tests cover only part of the state space. Random testing complements the human signalling expert.

9.3 Performance mCRL2 toolkit

Performance is often an issue with formal methods as state spaces tend to be very large for non-trivial models. By choosing an appropriate level of abstraction in our model we were able to prove the relevant properties for a small track layout. Model-based testing was possible with decent performance using a track layout consisting of 7 sections. Testing with a larger track layout suffers from performance issues.

If automated testing were to (almost) completely replace the current testing procedure with TeSys performance could become more of an issue. Automatically testing all relevant aspects of the behaviour of the interlocking would require that the model describes more details of the behaviour such as the behaviour when a light fails, level crossings, shunting movements, etc. Automatic testing might still be possible, at least for smaller yards, but verifying properties of the model used for testing would become harder. This reduces confidence in the model itself, making tests based on the model less reliable as the model might not fail the implementation even when a collision occurs. A way to mitigate this risk would be to not only rely on the conformance to the model but to actively check for certain properties during testing. The adapter could for instance be adapted to keep track of section occupations and log an error when it observes a collision or a point being moved in an occupied section.

9.4 Recommendations formal methods in the signalling domain

9.4.1 Integration with TeSys

Currently, configuring the track layout in both the mCRL2 model and the adapter is inconvenient. It is time consuming to manually fill in all the information and error prone as it is easy to get an ID wrong. It would be better to generate the configurations for the mCRL2 model and the adapter from some existing digital format used within Siemens. The method to configure the TeSys package with a track layout might also be used to configure the testing platform presented here. Ideally, both the track layout configuration of the TeSys package and the mCRL2 configuration should be based on some standardized format. The EULYNX data prep standard of which the main goal is to standardize the data transfer

format between infrastructure manager and signalling manufacturer would be an ideal candidate. It should be straightforward to generate the mCRL2 configuration from a EULYNX data prep structure.

Siemens could even further develop the testing platform by integrating it into the TeSys environment. It could then serve to complement/extend TAK. The automated testing platform could then also be properly connected to the other TeSys components, without needing to hack into the pipes used by TAK. This thesis demonstrates Siemens, and other signalling system manufacturers, how formal methods may be employed in the analysis and testing of their systems.

Apart from testing the software in TeSys it is also possible to perform tests on a an actual interlocking installation. The mCRL2 model does not need to be modified for this purpose but a different adapter is needed. Using relay control boxes, section occupations can be simulated and by registering the adapter as a signalman to the interlocking, routes can be set. By reading the telegrams sent by the interlocking, the adapter can determine which outputs the interlocking performs. An initial exploration showed that it is feasible to connect to the various interfaces to perform automated tests on a live interlocking system.

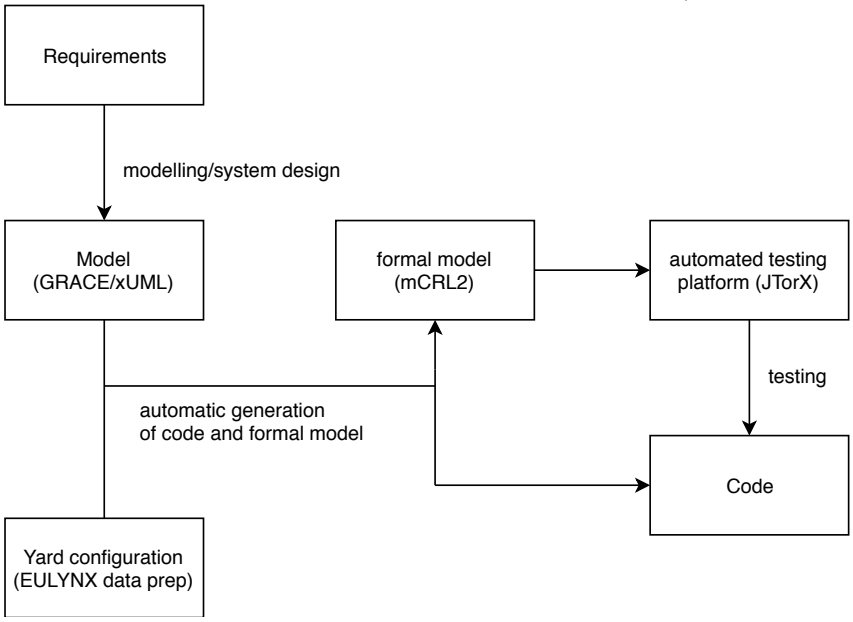
Further research for development signalling systems

As mentioned in the introduction, the main question at the start of this project was to explore how to improve the production chain of signalling systems from creation to configuration to testing. In the end, the focus of this thesis has been mainly on automatically testing an existing interlocking system. The presented testing platform is created in such a way that it is configurable for different track layouts.

The workflow depicted in Figure 1.2 could be applied in the signalling domain. Many players in the domain are already using other modelling languages than mCRL2 in their workflow. It would be beneficial for them if they could profit from the advantages of formal methods without needing to convert all their models or radically changing their current workflow. Siemens currently uses GRACE, a graphical modelling language, to design the interlocking logic. From GRACE the interlocking software is generated. Note that we did not have access to a precise description of GRACE. The EULYNX initiative uses UML models to standardize certain aspects of the behaviour of interlockings.

If it would be possible to generate a formal model and the implementation code from the models already being used, the flow depicted in Figure 9.4.1 could be achieved. Some research has been done on generating mCRL2 code from xUML (a subset of UML) [10]. Moreover, code generators for UML models exist [15][13]. As stated, implementation code can be generated from GRACE. Siemens has not (yet) researched generating a formal model from GRACE. Note that, depending on the goal of the UML model, code generation and/or formal models may or may not be of interest.

Figure 9.1: Further integration formal methods in signalling system/standard development



Signalling engineers generally do not have the expertise to use formal modelling languages and toolkits. Getting the expertise in formal methods requires a significant time investment. Using an intermediate modelling language would allow signalling experts to use more intuitive modelling languages while formal methods experts could manage the analysis of the formal models.

9.4.2 New signalling paradigms

As the signalling paradigm that was captured in our model has proven itself through extensive use over decades it was not expected that any flaws would be uncovered. New signalling paradigms however, such as ERTMS hybrid level 3 or systems that are yet to be developed, do not share this track record. For new systems especially, (formal) modelling can help to analyse systems for which there is no experience in verification. In the current signalling paradigm, signalling engineers know what kind of boundary cases should be tested for, but for innovative new signalling solutions identifying these cases is hard and formal methods can help find them. Another benefit of using formal methods in the development of new systems is that flaws can be found early on, reducing the cost to repair them.

9.5 Recommendations mCRL2 and JTorX

9.5.1 Configurability

An issue that is prevalent in the signalling domain is how to deal with different configurations. Track layout configurations are different for each interlocking but also the behaviour of the interlocking is dependent on several factors. For example, the behaviour depends on the country the interlocking is in and the Automatic Train Protection system used in that area. The mCRL2 toolkit currently does not have features to handle these configuration issues. An mCRL2 model is a single file with a single interpretation. Some configuration is possible by building a configuration mechanism into the model, like we did to be able to select between track configurations, but the solution is not ideal as the configurations are explicitly present in a model that represents the behaviour of the interlocking for any track layout. Moreover, for the variants that were tested in Chapter 6, there was no way to properly separate the common data equations from the different process definitions.

A solution to these configuration issues in the mCRL2 toolkit should not be specific for the railway domain as the toolkit is not domain specific. A solution should therefore be as generic and flexible as possible. Furthermore, it would be preferable that the syntax and especially the semantics of the mCRL2 language would remain (almost) the same. An elegant solution might then be to introduce a simple include mechanism that allows the user to reference another mCRL2 file. This would not alter the semantics as it would be based on the mCRL2 code where all include references are resolved. It would also be backwards compatible with existing models as they would still be syntactically correct. Another benefit is that it would be easier for teams to work on a model as each member can work on different (sub)components in different files.

9.5.2 Troubleshooting

An issue with the mCRL2 toolkit we came across that is not related to the signalling domain is that it is very hard to troubleshoot issues with the rewriting system. The mCRL2 toolkit contains a rewriter that rewrites data terms, a predicate on some data terms can for instance be rewritten to true or false using the data equations as rewrite rules. Heuristics are used to determine which term is rewritten next. The sum operator in process definitions and forall/exists statements in data equations, may cause the rewriter to crash if it tries to evaluate an infinite domain, such as all natural numbers. This can be prevented by restricting the infinite domain to a finite domain of interest. We might for instance restrict a summation over the domain of route_info objects to all the route_info objects that are in the list of current routes maintained by the interlocking.

The issue is that when a mistake is present in the model, it is hard to figure out which statement is causing the rewriter to keep rewriting indefinitely. The symptom is usually that the rewriter starts using more and more memory, until the program or the operating system crashes. It would be helpful if the toolkit would provide some feedback to the user. The instantiator might be improved to warn the user when an exists/forall/sum statement causes it to enumerate all natural numbers. Another possibility would be to provide the user with the option to display a log of the rewriter when a certain amount of memory has been used.

If the rewriter does not evaluate an infinite domain but nevertheless does need a large number of rewrite steps to rewrite certain terms it can have a significant impact on the performance of the toolkit. State-space generation, simulation and linearising can all be significantly slowed down if the rewriter needs many steps. Again, finding out what part of the model is causing the issue is difficult. It would be helpful if mcrl22lps and/or lpsxsim would contain an option to use a profiler on the rewriter, showing how many rewrite steps were needed for which data terms.

9.6 Conclusion

This thesis shows the applicability of mCRL2 in the signalling domain. The mCRL2 toolkit, in combination with JTorX, has shown to be adequate to perform meaningful verification and to create a novel automated testing platform for interlocking software. Performance during automated testing though, should be improved for it to be used in practice. The ability to have formally analysable models and to perform automated testing could benefit Siemens and other players within the signalling domain.

To increase the practicality and maintainability of the testing platform, Siemens could integrate it more in the existing TeSys environment. This should include creating a tool to configure the mCRL2 model using an existing digital format for track layouts. The model could also be adapted to perform the standard test cases for interlocking software. The mCRL2 toolkit could be improved by adding features to more easily configure a model, for instance by adding an include mechanism. Better troubleshooting options in the toolkit would reduce the effort needed to create (complex) models. The JTorX toolset and the lps2torx tool should be actively maintained if they wish to be more widely used. Further research is needed into test selection and coverage for interlocking software, especially in the context of certification. The usefulness of formal methods in the signalling domain would also benefit from automatic conversion tools that can translate models in a modelling languages already being used in the industry (such as UML and GRACE) to a formal modelling language.

Bibliography

[1] M. Bartholomeus, B. Luttik, and T. Willemse. Modelling and analysing ertms hybrid level 3 with the mcrl2 toolset. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 98–114. Springer, 2018.

[2] A. Belinfante. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. pages 266–270. Springer, Berlin, Heidelberg, 2010.

[3] A. Belinfante. *JTorX: exploring model-based testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 2014.

[4] A. Belinfante. JTorX webpage. `https://fmttools.ewi.utwente.nl/redmine//projects/jtorx/wiki/`, Accessed: 2018-09-20.

[5] M. Blazic. EULYNX Data preparation requirements. Technical Report Eu.Doc.51, Baseline 1.0 (1.A), Baseline set 2, EULYNX initiative, 2017.

[6] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. In *Z User Workshop, London 1992*, pages 1–5. Springer, 1993.

[7] A. Fantechi. Twenty-five years of formal methods and railways: what next? In *International Conference on Software Engineering and Formal Methods*, pages 167–183. Springer, 2013.

[8] J. F. Groote. mCRL2 webpage. `https://www.mcrl2.org/`, Accessed: 2018-09-20.

[9] J. F. Groote and M. R. Mousavi. *Modeling and analysis of communicating systems*. MIT press, 2014.

[10] H. Hvid Hansen, J. Ketema, B. Luttik, M. Mousavi, J. van de Pol, and O. M. dos Santos. Automated Verification of Executable UML Models. pages 225–250. Springer, Berlin, Heidelberg, 11 2011.

[11] RailTopoModel, IRC 30100. Standard, International Union of Railways.

[12] H. Kamsma. Timinganalyse systeemcombinatie EBS/Az-LM en GRS. Technical Report SWNL0232227, Sweco, 2018.

[13] A. Knapp and S. Merz. Model checking and code generation for uml state machines and collaborations. *Proc. 5th Wsh. Tools for System Design and Verification*, pages 59–64, 2002.

[14] A. Mammar, M. Frappier, S. J. Tueno Fotso, and R. Laleau. An event-b model of the hybrid ertms/etcs level 3 standard. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 353–366. Springer International Publishing, 2018.

[15] I. A. Niaz and J. Tanaka. Code generation from uml statecharts. In *Proc. 7 th IASTED International Conf. on Software Engineering and Application (SEA 2003), Marina Del Rey*, pages 315–321, 2003.

[16] ProRail. Object Type Library (OTL) webpage. `https://otl.prorail.nl`, Accessed: 2018-12-03.

[17] ProRail. Object Type Library (OTL) webpage, page concerning points. `https://otl.prorail.nl/vigerend/Wissel`, Accessed: 2018-12-03.

[18] J. Tretmans. A Formal Approach to Conformance Testing, 1992.

[19] J. Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.

[20] J. Tretmans and E. Brinksma. TorX : Automated Model Based Testing ote de Resyste. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.

[21] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):19, 2009.

Appendices

Appendix A

mCRL2 model

The entire mCRL2 model is included here as an appendix. Section A.1 contains variant C in its entirety. Section A.2 lists all the differences between the variants.

A.1 Main model

```
sort
  signal_colour = struct RD | GL | GR; %red, yellow, green
  train_general_position = struct before_track | on_track | after_track;
  point_position = struct left | right;
  %as the track can be viewed as an acyclic graph,
  % it can be said to have a left side and right side
  driving_direction = struct L | R;
  route_status = struct requested | accepted | locked | ready | active | rejected;
  %section status in view of the interlocking
  section_status = struct free | occupied | logically_occupied;

  section_id = Nat;
  signal_id = Nat;
  train_id = Nat;
  point_id = Nat;

  section_info =
     struct section_info(
      status: section_status,
      on_border: Bool, %border/edge of the yard, connection to open track
      connections: List(section_connection) %connections to other sections
     );
  signal_info =
    struct signal_info(
      colour: signal_colour,
      direction: driving_direction,
      virtual: Bool, %virtual signals can be used to set a route but are not a physical object
      section_before: section_id,
      section_after: section_id
    );
  route_info =
    struct route_info(
      entry_signal: signal_id,
      exit_signal: signal_id,
      direction: driving_direction,
      sections_of_route: List(section_id),
      signals_of_route: List(signal_id),
      points_of_route: List(point_position_pair),
      protection: List(flank_protection),
      status: route_status
```

```
    );
  flank_protection =
    struct flank_protection(
      for_section: section_id,
      flank_protection_points: List(point_position_pair),
      flank_protection_signals: List(signal_id)
    );
  point_position_pair =
    struct point_position_pair(
      point: point_id,
      position: point_position
    );
  point_info =
    struct point_info(
      in_section: section_id,
      position: point_position
    );
  section_connection = struct section_connection(
    positions: List(point_position_pair),
    protection: flank_protection,
    left_section: section_id,
    right_section: section_id
  );
  signal_pair = struct signal_pair(
    entry_signal: signal_id,
    exit_signal: signal_id
  );

  signals = signal_id -> signal_info;
    sections = section_id -> section_info;
  points = point_id -> point_info;
  routing_table = List(signal_pair);

%declarations and static positional calculations
map
  efp: flank_protection;
  get_points_of_section: section_id#points -> List(point_id);
  get_points_of_section_iter: section_id#point_id#List(point_id)#points -> List(point_id);
  enumerate_point_position_pairs: List(point_id) -> List(List(point_position_pair));
  enumerate_point_position_pairs_aux: List(point_id)#List(List(point_position_pair))
    -> List(List(point_position_pair));
  enumerate_point_position_pairs_aux_aux: point_id#List(List(point_position_pair))
    #List(List(point_position_pair)) -> List(List(point_position_pair));
  legal_section: section_id -> Bool;
  legal_signal: signal_id -> Bool;
  legal_point: point_id -> Bool;
  in_front_of_signal: section_id#section_id#signals -> Bool;
  in_front_of_any_signal: section_id#section_id#signals -> Bool;
  get_signal_ahead: section_id#section_id#signals -> signal_id;
  get_signal_ahead_iter: section_id#section_id#signals#signal_id -> signal_id;
  expand_route: route_info#driving_direction#List(point_position_pair)
    #sections#signals#points -> List(route_info);
  expand_route_point_combinations: route_info#driving_direction#sections#signals#points
    -> List(route_info);
  expand_route_point_combinations_aux: route_info#driving_direction#sections#signals#points
    #List(List(point_position_pair))#List(route_info) -> List(route_info);
  find_route: signal_id#signal_id#sections#signals#points -> route_info;
  find_route_expand_options: signal_id#driving_direction#List(route_info)
    #List(route_info)#sections#signals#points -> route_info;
  find_route_check_done: signal_id#driving_direction#List(route_info)
    #List(route_info)#sections#signals#points -> route_info;
```

```
  get_right_section_connections: section_id#Set(point_position_pair)#Bool
    #List(section_connection) -> section_id;
  get_left_section_connections: section_id#Set(point_position_pair)#Bool
    #List(section_connection) -> section_id;
  check_equality_set: Set(point_position_pair)#Set(point_position_pair)#Bool -> Bool;
  get_next_section_static: section_id#section_id#driving_direction
    #List(point_position_pair)#Bool#sections -> section_id;
  get_flank_protection: section_id#section_id#List(point_position_pair)#sections
    -> flank_protection;
  get_flank_protection_aux: section_id#Set(point_position_pair)#List(section_connection)
    -> flank_protection;
var
  se,se2,se3: section_id;
  si,si2: signal_id;
  p: point_id;
  pos: point_position;
  sc: section_connection;
  di: driving_direction;
  strict: Bool;
  sic: signals;
  sec: sections;
  poc: points;
  ro: route_info;
  flp: flank_protection;
  plist: List(point_id);
  ppp: point_position_pair;
  point_poss_todo: List(List(point_position_pair));
  point_poss_done: List(List(point_position_pair));
  point_poss: List(point_position_pair);
  point_poss_set, point_poss_set2: Set(point_position_pair);
  conns: List(section_connection);
  roo_todo: List(route_info);
  roo_done: List(route_info);
eqn
  legal_point(p) = p >= first_point && p <= last_point;
  legal_section(se) = se >= first_section && se <= last_section;
  legal_signal(si) = si >= first_signal && si <= last_signal;
  efp = flank_protection(0,[],[]);
  get_points_of_section(se,poc) = if(last_point == 0, [],
    get_points_of_section_iter(se,first_point,[],poc)); %catch case without points
  get_points_of_section_iter(se,p,plist,poc) =
    %iterate over all points and add point to the list if it is in the given section
    if(p > last_point, plist, if(in_section(poc(p)) == se,
      get_points_of_section_iter(se,p+1,plist <| p,poc),
      get_points_of_section_iter(se,p+1,plist,poc)));

  %calculates all possible combinations of point positions pairs of a given list of points
  enumerate_point_position_pairs(plist) = enumerate_point_position_pairs_aux(plist,[]);
  enumerate_point_position_pairs_aux([],point_poss_done) = point_poss_done;
  enumerate_point_position_pairs_aux(p |> plist,point_poss_done) =
    enumerate_point_position_pairs_aux(plist,
    enumerate_point_position_pairs_aux_aux(p,point_poss_done,[]));
  enumerate_point_position_pairs_aux_aux(p,[], point_poss_done) =
    ((point_poss_done <| [point_position_pair(p,left)])
    <| [point_position_pair(p,right)]) <| [];
  enumerate_point_position_pairs_aux_aux(p,point_poss |> point_poss_todo, point_poss_done) =
    enumerate_point_position_pairs_aux_aux(p,point_poss_todo,
      ((point_poss_done <| (point_poss <| point_position_pair(p,left)))
      <| (point_poss <| point_position_pair(p,right))) <| point_poss);

  in_front_of_signal(se,se2,sic) = %virtual signals excluded
```

```
  exists signal:signal_id. legal_signal(signal)
  && section_before(sic(signal)) == se && se2 == section_after(sic(signal))
  && !virtual(sic(signal));
in_front_of_any_signal(se,se2,sic) = %virtual signals included
  exists signal:signal_id. legal_signal(signal)
  && section_before(sic(signal)) == se && se2 == section_after(sic(signal));
get_signal_ahead(se,se2,sic) = get_signal_ahead_iter(se,se2,sic,first_signal);
get_signal_ahead_iter(se,se2,sic,si) =
  %iterate over the signals until the correct one is found
  if(si > last_signal, 0,
    if(legal_signal(si) && se == section_before(sic(si)) && se2 == section_after(sic(si)),
    si, get_signal_ahead_iter(se,se2,sic,si+1)));

%finds a route between two signals. The functions ensure that from the starting signal
%the route options are expanded. Expanding means that it 'drives' one section further
%down the route. When a point is encountered the search may continue along
%two paths: a path with the point in the left position and a path with the point
%in the right position. The search follows a breadth first strategy.
%After each expansion it is evaluated whether the destination is reached.
find_route(si,si2,sec,sic,poc) =
  find_route_check_done(si2,direction(sic(si)),
    [route_info(si,si2,direction(sic(si)),[section_before(sic(si)),
    section_after(sic(si))],[si],[],[],requested)]
    ,[],sec,sic,poc);

%expand route options
find_route_expand_options(si,di,roo_todo,roo_done,sec,sic,poc) =
  if(#roo_todo == 0, find_route_check_done(si,di,roo_done,[],sec,sic,poc),
    find_route_expand_options(si,di,tail(roo_todo),roo_done ++
    expand_route_point_combinations(head(roo_todo),di,sec,sic,poc),sec,sic,poc));
expand_route(ro,di,point_poss,sec,sic,poc) =
   %expands route given a specific point position
  if(!legal_section(next_section),
    [], %ditch route as we have come outside the track
    [route_info(
      entry_signal(ro),
      exit_signal(ro),
      direction(ro),
      sections_of_route(ro) <| next_section,
      if(in_front_of_signal(rhead(sections_of_route(ro)),next_section,sic),
        signals_of_route(ro) <| get_signal_ahead(rhead(sections_of_route(ro)),
          next_section,sic),
        signals_of_route(ro)),
      points_of_route(ro) ++ point_poss,
      protection(ro) ++ if(get_flank_protection(rhead(rtail(sections_of_route(ro))),
        rhead(sections_of_route(ro)),point_poss,sec) == efp
        , [],
        [get_flank_protection(rhead(rtail(sections_of_route(ro))),
        rhead(sections_of_route(ro)),point_poss,sec)]),
        status(ro)
    )]) whr next_section = get_next_section_static(rhead(rtail(sections_of_route(ro))),
      rhead(sections_of_route(ro)),di,point_poss,true,sec) end;
%goes through every point combination of the current section and expands the route options
expand_route_point_combinations(ro,di,sec,sic,poc) =
  expand_route_point_combinations_aux(ro,di,sec,sic,poc,
  enumerate_point_position_pairs(get_points_of_section(rhead(sections_of_route(ro)),poc)),[]);
expand_route_point_combinations_aux(ro,di,sec,sic,poc,[],roo_done) =
  roo_done ++ expand_route(ro,di,[],sec,sic,poc);
expand_route_point_combinations_aux(ro,di,sec,sic,poc,point_poss |> point_poss_todo,roo_done) =
  expand_route_point_combinations_aux(ro,di,sec,sic,poc,point_poss_todo,roo_done ++
  expand_route(ro,di,point_poss,sec,sic,poc));
```

```
%check if destination is reached on one of the paths
find_route_check_done(si,di,roo_todo,roo_done,sec,sic,poc) =
  if(#roo_todo == 0 && #roo_done == 0, %check if no more viable routes are available
    route_info(0,0,L,[],[],[],[],rejected), %if so, return rejected route_info
    if(#roo_todo == 0, %check if more to check
       %all checked so continue expanding
      find_route_expand_options(si,di,roo_done,[],sec,sic,poc),
      if(rhead(sections_of_route(head(roo_todo))) == section_before(sic(si))
        && di == direction(sic(si)), %check if destination signal is reached
        %found route to destination so done, remove first section
        %as entry section was initially included
        route_free_first_section(head(roo_todo),sic,poc),
        %check next
        find_route_check_done(si,di,tail(roo_todo),roo_done <| head(roo_todo),sec,sic,poc))));

%functions to determine the next section given the previous section
%and the point positions of the current section.
%parameter strict makes a difference in the case that there are multiple points in a section.
%If strict the given point positions need to exactly be the points needed to go from A to B
%(no points included that are not on the route).
%This is useful for the find_route function to not find routes with points
%that are not actually part of the route
get_next_section_static(se,se2,R,point_poss,strict,sec) =
  get_right_section_connections(se,{ppp: point_position_pair | ppp in point_poss},
  strict,connections(sec(se2)));
get_next_section_static(se,se2,L,point_poss,strict,sec) =
  get_left_section_connections(se,{ppp: point_position_pair | ppp in point_poss},
  strict,connections(sec(se2)));
get_right_section_connections(se,point_poss_set,strict,[]) = 0;
get_right_section_connections(se,point_poss_set,strict, sc |> conns) =
  if(left_section(sc) == se
  && check_equality_set({ppp: point_position_pair | ppp in positions(sc)},
  point_poss_set,strict),
    right_section(sc),
    get_right_section_connections(se,point_poss_set,strict,conns));
get_left_section_connections(se,point_poss_set,strict,[]) = 0;
get_left_section_connections(se,point_poss_set,strict,sc |> conns) =
  if(right_section(sc) == se
  && check_equality_set({ppp: point_position_pair | ppp in positions(sc)},
  point_poss_set,strict),
    left_section(sc),
    get_left_section_connections(se,point_poss_set,strict,conns));

%based on point position pairs of a section and a neighbouring section,
%find the section connection and return the accompanying flank protection
get_flank_protection(se,se2,point_poss,sec) =
  get_flank_protection_aux(se,{ppp: point_position_pair | ppp in point_poss},
  connections(sec(se2)));
get_flank_protection_aux(se,point_poss_set,[]) = efp;
get_flank_protection_aux(se,point_poss_set,sc |> conns) =
  if((left_section(sc) == se || right_section(sc) == se)
    && check_equality_set({ppp: point_position_pair | ppp in positions(sc)},
    point_poss_set,true),
    protection(sc),
    get_flank_protection_aux(se,point_poss_set,conns));

check_equality_set(point_poss_set,point_poss_set2,true) =
  point_poss_set == point_poss_set2;
check_equality_set(point_poss_set,point_poss_set2,false) =
  point_poss_set * point_poss_set2 == point_poss_set; %set1 is subset of set2
```

```
%get and set operations
map
  section_get_occupance: section_id#sections -> Bool;
  section_get_status: section_id#sections -> section_status;
    section_update_status: section_id#section_status#sections -> sections;
  section_on_border: section_id#sections -> Bool;
  signal_get_colour: signal_id#signals -> signal_colour;
  signal_get_virtual: signal_id#signals -> Bool;
  signal_update_colour: signal_id#signal_colour#signals -> signals;
  route_update_status: route_info#route_status -> route_info;
  route_update_sections: route_info#List(section_id) -> route_info;
  route_update_signals: route_info#List(signal_id) -> route_info;
  route_update_points: route_info#List(point_position_pair) -> route_info;
  route_update_protection: route_info#List(flank_protection) -> route_info;
  route_remove_points_by_section: route_info#section_id#List(point_position_pair)
    #List(point_position_pair)#points -> route_info;
  route_remove_protection_by_section: route_info#section_id
    #List(flank_protection)#List(flank_protection) -> route_info;
  route_collection_update_route: route_info#route_info#List(route_info) -> List(route_info);
  route_collection_update_route_aux: route_info#route_info#List(route_info)#List(route_info)
    -> List(route_info);
  route_collection_discard_route: route_info#List(route_info) -> List(route_info);
  route_collection_discard_route_aux: route_info#List(route_info)#List(route_info)
    -> List(route_info);
  get_second_section_of_route: route_info -> section_id;
  route_free_first_signal: route_info -> route_info;
  route_free_first_section: route_info#signals#points -> route_info;
  section_before_signal: signal_id#signals -> section_id;
  section_after_signal: signal_id#signals -> section_id;
  point_update_position: point_id#point_position#points -> points;
  routes_handle_section_occupied: section_id#signals#List(route_info) -> List(route_info);
  routes_handle_section_occupied_aux: section_id#signals#List(route_info)
    #List(route_info)#List(route_info) -> List(route_info);
  routes_handle_section_free: section_id#signals#points#List(route_info) -> List(route_info);
  routes_handle_section_free_aux: section_id#signals#points#List(route_info)
    #List(route_info)#List(route_info) -> List(route_info);
  routes_handle_update_signal: signal_id#signal_colour#sections#signals#List(route_info)
    -> List(route_info);
  routes_handle_update_signal_aux: signal_id#signal_colour#sections#signals
    #List(route_info)#List(route_info) -> List(route_info);
  possibly_discard_route: route_info -> List(route_info);
  possibly_free_section: section_id#route_info#signals#points#List(route_info) -> route_info;
  possibly_activate_route: section_id#route_info#signals#List(route_info) -> route_info;
  possibly_free_signal:signal_id#signal_colour#sections#signals#route_info -> route_info;
var
  se,se2: section_id; %se short for section
  si,si2: signal_id; %si short for signal
  po:point_id;
  pos: point_position;
  ss: section_status;
  co: signal_colour; %co short for colour
  sec: sections; %sec short for section collection
  sic: signals; %sic short for signal collection
  poc: points;
  ro,ro2,newro: route_info; %ro short for route
  roc,roc2,roc3: List(route_info); %roc short for route collection
  ppp: point_position_pair;
  ppp_list, ppp_list2: List(point_position_pair);
  flp: flank_protection;
  flp_list, flp_list2: List(flank_protection);
```

```
  ros: route_status; %ros short for route_status
  se_list: List(section_id); %se_list short for section list
  si_list: List(signal_id); %si_list short for signal list
eqn
  section_get_occupance(se,sec) =
    !(status(sec(se)) == free); %true if occupied or logically occupied
  section_get_status(se,sec) = status(sec(se));
  section_update_status(se,ss,sec) =
    sec[se -> section_info(ss,on_border(sec(se)),connections(sec(se)))];
  section_on_border(se,sec) = on_border(sec(se));
  signal_get_colour(si,sic) = colour(sic(si));
  signal_get_virtual(si, sic) = virtual(sic(si));
  signal_update_colour(si,co,sic) =
    sic[si -> signal_info(co,direction(sic(si)),virtual(sic(si)),
    section_before(sic(si)),section_after(sic(si)))];
  route_update_status(ro,ros) =
    route_info(entry_signal(ro), exit_signal(ro), direction(ro), sections_of_route(ro),
    signals_of_route(ro),points_of_route(ro),protection(ro), ros);
  route_update_sections(ro,se_list) =
    route_info(entry_signal(ro), exit_signal(ro), direction(ro), se_list,
    signals_of_route(ro),points_of_route(ro),protection(ro), status(ro));
  route_update_signals(ro,si_list) =
    route_info(entry_signal(ro), exit_signal(ro), direction(ro), sections_of_route(ro),
    si_list,points_of_route(ro),protection(ro), status(ro));
  route_update_points(ro,ppp_list) =
    route_info(entry_signal(ro), exit_signal(ro), direction(ro), sections_of_route(ro),
    signals_of_route(ro),ppp_list,protection(ro), status(ro));
  route_update_protection(ro,flp_list) =
    route_info(entry_signal(ro),exit_signal(ro), direction(ro),sections_of_route(ro),
    signals_of_route(ro),points_of_route(ro),flp_list,status(ro));
  get_second_section_of_route(ro) =
    if(#sections_of_route(ro) < 2, 0,head(tail(sections_of_route(ro))));
  route_free_first_signal(ro) = route_update_signals(ro, tail(signals_of_route(ro)));
  section_before_signal(si,sic) = section_before(sic(si));
  section_after_signal(si,sic) = section_after(sic(si));
  point_update_position(po,pos,poc) = poc[po -> point_info(in_section(poc(po)), pos)];

  route_collection_update_route(newro,ro,roc) =
    route_collection_update_route_aux(newro,ro,roc,[]);
  route_collection_update_route_aux(newro,ro,[],roc2) = roc2;
  route_collection_update_route_aux(newro,ro,ro2 |> roc,roc2) =
    route_collection_update_route_aux(newro,ro,roc,roc2 <| if(ro==ro2, newro, ro2));

  route_collection_discard_route(ro,roc) = route_collection_discard_route_aux(ro,roc,[]);
  route_collection_discard_route_aux(ro,[],roc2) = roc2;
  route_collection_discard_route_aux(ro,ro2 |> roc,roc2) =
    route_collection_discard_route_aux(ro,roc,roc2 ++ if(ro==ro2, [], [ro2]));

  %frees up the first section of the route and calls functions
  %to also free any flank protection and points of the section
  route_free_first_section(ro,sic,poc) =
      route_remove_protection_by_section(
        route_remove_points_by_section(
          route_update_sections(ro, tail(sections_of_route(ro)))
          ,head(sections_of_route(ro)),points_of_route(ro),[],poc)
        ,head(sections_of_route(ro)),protection(ro),[]);

  %goes through the entire list of points of the route
  %and removes the ones that are in the given section
  route_remove_points_by_section(ro,se,[],ppp_list2,poc) =
    route_update_points(ro,ppp_list2);
```

```
  route_remove_points_by_section(ro,se,ppp |> ppp_list,ppp_list2,poc) =
    if(se == in_section(poc(point(ppp))),
      route_remove_points_by_section(ro,se,ppp_list,ppp_list2,poc),
      route_remove_points_by_section(ro,se,ppp_list,ppp_list2 <| ppp,poc));

  %goes through the entire list of flank protections of the route
  %and removes the ones that are of the given section
  route_remove_protection_by_section(ro,se,[],flp_list2) =
    route_update_protection(ro,flp_list2);
  route_remove_protection_by_section(ro,se,flp |> flp_list,flp_list2) =
    if(se == for_section(flp),
      route_remove_protection_by_section(ro,se,flp_list,flp_list2),
      route_remove_protection_by_section(ro,se,flp_list,flp_list2 <| flp));

  %update routes based on a section being marked as free
  routes_handle_section_free(se,sic,poc,roc) =
    routes_handle_section_free_aux(se,sic,poc,roc,roc,[]);
  routes_handle_section_free_aux(se,sic,poc,roc,[],roc3) = roc3;
  routes_handle_section_free_aux(se,sic,poc,roc,ro |> roc2,roc3) =
    routes_handle_section_free_aux(se,sic,poc,roc,roc2,roc3 ++
    possibly_discard_route(possibly_free_section(se,ro,sic,poc,roc)));
  possibly_discard_route(ro) = if(#sections_of_route(ro) == 0, [], [ro]);
  possibly_free_section(se,ro,sic,poc,roc) = if(#sections_of_route(ro) != 0
    && status(ro) == active && section_first_of_route(se,ro),
    route_free_first_section(ro,sic,poc), ro);

  %update routes based on a section being marked as occupied
  routes_handle_section_occupied(se,sic,roc) =
    routes_handle_section_occupied_aux(se,sic,roc,roc,[]);
  routes_handle_section_occupied_aux(se,sic,roc,[],roc3) = roc3;
  routes_handle_section_occupied_aux(se,sic,roc,ro |> roc2,roc3) =
    routes_handle_section_occupied_aux(se,sic,roc,roc2,roc3
    <| possibly_activate_route(se,ro,sic,roc));
  possibly_activate_route(se,ro,sic,roc) =
    if(status(ro) == ready && se == section_after(sic(entry_signal(ro))),
    route_update_status(ro,active),ro);

  %update routes based on a signal being set to a new colour
  %(a signal might be removed from a route)
  routes_handle_update_signal(si,co,sec,sic,roc) =
    routes_handle_update_signal_aux(si,co,sec,sic,roc,[]);
  routes_handle_update_signal_aux(si,co,sec,sic,[],roc2) = roc2;
  routes_handle_update_signal_aux(si,co,sec,sic,ro |> roc,roc2) =
    routes_handle_update_signal_aux(si,co,sec,sic,roc,roc2
    <| possibly_free_signal(si,co,sec,sic,ro));
  possibly_free_signal(si,co,sec,sic,ro) =
    if(co == RD && signal_first_of_route(si,ro) &&
    section_after_signal_occupied(si,sec,sic),route_free_first_signal(ro),ro);


%dynamic calculations (dependent on current section occupations
%, point positions and routes)
map
  all_points_locked: route_info#sections#points -> Bool;
  border_unclaimed: section_id#List(route_info) -> Bool;
  get_points_to_be_locked: route_info#sections#points -> Set(point_position_pair);
  route_conflict_free_points: route_info#sections#List(route_info)#points -> Bool;
  route_conflict_free_flank_signals: route_info#sections#List(route_info) -> Bool;
  route_conflict_free_head_on: route_info#List(route_info) -> Bool;
  compute_signal: signal_id#sections#signals#List(route_info)#points -> signal_colour;
  section_first_of_route: section_id#route_info -> Bool;
```

```
  section_part_of_route: section_id#List(route_info) -> Bool;
  section_first_of_active_route: section_id#List(route_info) -> Bool;
  section_last_of_active_route: section_id#List(route_info) -> Bool;
  second_section_of_route_occupied: section_id#sections#List(route_info) -> Bool;
  sections_free: List(section_id)#sections -> Bool;
  signal_part_of_ready_route: signal_id#List(route_info) -> Bool;
  signal_first_of_route: signal_id#route_info -> Bool;
  section_before_signal_part_of_active_route: signal_id#signals#List(route_info)
    -> Bool;
  section_before_signal_occupied: signal_id#sections#signals -> Bool;
  section_after_signal_occupied: signal_id#sections#signals -> Bool;
  signal_free: signal_id#List(route_info) -> Bool;
  sections_of_route_free: route_info#sections -> Bool;
  get_next_section: section_id#section_id#driving_direction#sections#points
    -> section_id;
  get_point_positions: List(point_id)#points#List(point_position_pair)
     -> List(point_position_pair);
  get_possible_routes: routing_table#sections#signals#points#Set(route_info)
    -> Set(route_info);
var
  se,se2: section_id;
  si,si2: signal_id;
  max_sec_to_go: Nat;
  o: Bool; %o short for occupance
  p: point_id;
  co: signal_colour;
  sec: sections; %sec short for section collection
  sic: signals; %sic short for signal collection
  poc: points; %poc short for point collection
  pos: point_position;
  di: driving_direction;
  si_togo: Nat;
  ro,ro2: route_info; %ro short for route
  sp: signal_pair;
  plist: List(point_id);
  ppplist: List(point_position_pair);
  rot: routing_table;
  roc,roc2,roc3: List(route_info); %roc short for route collection
  rocset: Set(route_info);
  ros: route_status;
  se_list: List(section_id);
  si_list: List(signal_id);
eqn
  %goes through all the allowed signal combinations for which
  %a route can be requested and calculates the route between the signals using find_route
  %this function is used to precalculate all the routes
  %during linearisation to improve efficiency
  get_possible_routes([],sec,sic,poc,rocset) = rocset;
  get_possible_routes(sp |> rot,sec,sic,poc,rocset) =
    get_possible_routes(rot,sec,sic,poc,rocset+
    {find_route(entry_signal(sp),exit_signal(sp),sec,sic,poc)});

  %verify that no route is set to a given border section
  border_unclaimed(se,roc) = forall ro:route_info. (ro in roc)
  => (#sections_of_route(ro) == 0 || rhead(sections_of_route(ro)) != se);

  %functions to check/get the points of a given route
  %that are not yet in the right position
  all_points_locked(ro,sec,poc) = get_points_to_be_locked(ro,sec,poc) == {};
  get_points_to_be_locked(ro,sec,poc) =
    {ppp: point_position_pair| ppp in points_of_route(ro)
```

```
  && position(poc(point(ppp))) != position(ppp)}
  + {ppp: point_position_pair| exists fp:flank_protection. fp in protection(ro)
  && ppp in flank_protection_points(fp) && position(poc(point(ppp))) != position(ppp)};

%checks whether the points of a given route conflict with the points
%of a route that has already been accepted,
%also takes into account flank protection points
route_conflict_free_points(ro,sec,roc,poc) =
  forall p:point_id. (legal_point(p)
  && exists ppp:point_position_pair. ppp in get_points_to_be_locked(ro,sec,poc)
  && p == point(ppp))  =>
  (forall other_route:route_info. (other_route in roc) =>
  (forall other_route_p:point_id. (legal_point(other_route_p)
    && exists ppp2:point_position_pair. ppp2 in points_of_route(other_route)
    && other_route_p == point(ppp2)) => (p != other_route_p)));

%checks whether the flank protection signals of a given route conflict
%with the signals of a route that has already been accepted
route_conflict_free_flank_signals(ro,sec,roc) =
   forall si:signal_id.  (legal_signal(si)
   && exists fp:flank_protection. fp in protection(ro)
   && si in flank_protection_signals(fp))
  => (forall other_route:route_info. other_route in roc && other_route != ro
  => (forall si2:signal_id. (legal_signal(si2) && si2 in signals_of_route(other_route))
  => (si != si2)));

route_conflict_free_head_on(ro,roc) =
  forall other_route:route_info. (other_route in roc)
    => (direction(ro) == direction(other_route)
    || (forall se:section_id. (se in sections_of_route(ro))
    => !(se in sections_of_route(other_route))));

%gets the next section based on the previous section and the current point positions
get_next_section(se,se2,di,sec,poc) =
  get_next_section_static(se,se2,di,
  get_point_positions(get_points_of_section(se2,poc),poc,[]),false,sec);
get_point_positions([],poc,ppplist) = ppplist;
get_point_positions(p |> plist,poc,ppplist) =
get_point_positions(plist,poc,ppplist <| point_position_pair(p, position(poc(p))));

%remaining equations are self explanatory
section_before_signal_part_of_active_route(si,sic,roc) =
  legal_signal(si) && exists route:route_info.
  (route in roc && (status(route) == active || status(route) == ready))
  && (exists route_section: section_id. legal_section(route_section)
  && (route_section in sections_of_route(route))
  && (route_section == section_before_signal(si,sic)));
signal_first_of_route(si,ro) =
  si == head(signals_of_route(ro)) && #signals_of_route(ro) > 0;
section_first_of_route(se,ro) =
  se == head(sections_of_route(ro)) && #sections_of_route(ro) > 0;
section_part_of_route(se,roc) =
  exists route:route_info. exists section:section_id . route in roc
  && section in sections_of_route(route);
section_first_of_active_route(se,roc) =
  exists route:route_info. route in roc
  && status(route) == active && #sections_of_route(route) > 0
  && se == head(sections_of_route(route));
section_last_of_active_route(se,roc) =
  exists route:route_info. route in roc
  && status(route) == active && #sections_of_route(route) == 1
```

```
    && se == head(sections_of_route(route));
  second_section_of_route_occupied(se,sec,roc) =
    exists route:route_info. route in roc && #sections_of_route(route) > 1
    && se == head(sections_of_route(route))
    && section_get_occupance(head(tail(sections_of_route(route))),sec);
  signal_part_of_ready_route(si,roc) =
    exists route:route_info. route in roc && status(route) == ready
    && si in signals_of_route(route);
  sections_of_route_free(ro,sec) =
    forall se:section_id. se in sections_of_route(ro) => !section_get_occupance(se,sec);
  sections_free(se_list,sec) =
    forall se:section_id. se in se_list => !section_get_occupance(se,sec);
  section_before_signal_occupied(si,sec,sic) =
    section_get_occupance(section_before_signal(si,sic), sec);
  section_after_signal_occupied(si,sec,sic) =
    section_get_occupance(section_after_signal(si,sic),sec);
  signal_free(si,roc) =
    forall other_route:route_info. (other_route in roc) =>
      (forall other_route_signal: signal_id.
      (other_route_signal in signals_of_route(other_route))
        => other_route_signal != si);

  %contains the simple signal logic: default is red, if signal is the entry signal
  %of a ready route and all sections of route are free: at least yellow.
  %if the exit signal is not red than this signal can be set to green.
  compute_signal(si,sec,sic,roc,poc) =
    if(!(exists ro:route_info. ro in roc && entry_signal(ro) == si && status(ro) == ready
    && sections_of_route_free(ro,sec)),
      RD,
      if(exists ro:route_info. ro in roc && entry_signal(ro) == si && status(ro) == ready
      && sections_of_route_free(ro,sec)
          && signal_get_colour(exit_signal(ro),sic) != RD
          && !signal_get_virtual(exit_signal(ro),sic),
        GR,
        GL));

%track layout configuration, several instances can be configured
%and the instance to be used can be specified below
map instance: Nat;
eqn instance = 1;
sort
  constants_sort =
    struct c(
      last_section: section_id,
      last_signal: signal_id,
      last_point: point_id,
      last_train: Nat
    );

  train_config =
    struct train_config(
      entry_section: section_id,
      direction: driving_direction,
      start_over: Bool
    );
  trains = Nat -> train_config;
map
  first_section, last_section: section_id;
  first_signal, last_signal: signal_id;
  first_point, last_point: point_id;
  last_train: Nat;
```

```
  constants: Nat -> constants_sort;
  sections_config : Nat -> sections;
  signals_config : Nat -> signals;
  points_config: Nat -> points;
  trains_config: Nat -> trains;
  routing_table_config: Nat -> routing_table;
eqn
  %configurations go here

  first_section = 1;
  last_section = last_section(constants(instance));
  first_signal = 1;
  last_signal = last_signal(constants(instance));
  first_point = 1;
  last_point = last_point(constants(instance));

act
  getStatusSection, getStatusSectionSend, getStatusSectionRec:
    Nat # Bool; %parameters: section id, occupied
  seeSignal, seeSignalSend, seeSignalRec: Nat # signal_colour; %first parameter: signal id
  setStatusSection,setStatusSectionSection,setStatusSectionTrain:
    Nat # Bool; %parameters: section id, occupied
  permissionTrain, permissionInterlocking, permission: section_id # Bool;
  setSignalSend, setSignalRec, setSignal: Nat # signal_colour; %first parameter: signal id
  getPositionPoint, getPositionPointSend, getPositionPointRec: point_id#point_position;
  setPositionPoint, setPositionPointSend, setPositionPointRec, setPositionPointRecTrain:
    point_id#point_position#section_id;
  requestRoute: signal_id # signal_id; %parameters: entry signal, exit signal
  routeAccepted;
  routeRejected;
  activateRoute;
  readyRoute;
  notReadyRoute;
  freeUpSection: section_id;
  discardRoute;
  freeUpSignal: signal_id;
  mergeRoutes;
  skip;

proc
  %sections act as a sort of variable that can be set by the train and read by the interlocking
  %Abbreviations: cs->current status
  Section(id: section_id, cs: Bool) =
    (sum b:Bool. setStatusSectionSection(id, b) . Section(cs = b))
    + getStatusSectionSend(id, cs). Section();

  %Signals can be set by the interlocking and read by a train
  %Abbreviations: cc->current colour, nc->new colour
  Signal(id: signal_id, cc: signal_colour) =
    (sum nc: signal_colour . setSignalRec(id, nc) . Signal(id, nc))
    + seeSignalSend(id, cc). Signal();

  %Points are set by the interlocking
  Point(id: point_id, pos: point_position, se:section_id) =
    (sum npos: point_position. setPositionPointRec(id, npos,se).Point(pos = npos))
    + getPositionPointSend(id, pos). Point();

  %Trains go through the lifecycle before_track->on_track->after_track.
  %To enter the track permission from the interlocking is neccesary.
  %A train can occuppy at most two sections. If it occupies only one,
  %the next section can become occupied. Before this, it needs to see that
```

```
%the signal is yellow or green (if there is a signal) If it occupies two sections,
%the section occupied by the back of the train can become unoccupied.
%Once the train has reached the other side of the yard it can leave the yard.
%If the boolean so is set to true, it can start over and simulate a new train.
%Abbreviations: gp->general position, es->entry section, os->occupied sections,
%so->start over, prevs->previous section,
%sec->section collection, sic->signal collection, poc->point collection
Train(gp: train_general_position, can_proceed: Bool, es: section_id,
  os: List(section_id), prevs:section_id,
  di: driving_direction, so: Bool, sec: sections, sic:signals, poc: points) =
  ((gp == before_track) -> %entering first section only if permission by interlocking
      permissionTrain(es, true).Train(gp = on_track, os = [es]))
  + ((gp == on_track) -> sum next_section:section_id.
    (next_section == get_next_section(prevs,head(os),di,sec,poc)) -> (
      (#os == 2) ->
          setStatusSectionTrain(head(os),false).Train(prevs = head(os), os = tail(os))
      +(#os == 1 && legal_section(next_section) && !can_proceed) -> (
        (in_front_of_signal(rhead(os),next_section, sic))
          -> (seeSignalRec(get_signal_ahead(rhead(os),next_section, sic),GR)
           + seeSignalRec(get_signal_ahead(rhead(os),next_section, sic),GL))
        +(!in_front_of_signal(rhead(os),next_section, sic)) -> skip
      ).Train(can_proceed = true)
      +(#os == 1 && legal_section(next_section) && can_proceed)
          -> setStatusSectionTrain(next_section,true)
        .Train(os = os <| next_section, can_proceed = false)
      +(#os == 1 && !legal_section(next_section)) ->
        setStatusSectionTrain(head(os),false).Train(os = [], gp = after_track, prevs = 0)
    )
  )
  + ((gp == after_track && so) -> skip.Train(gp = before_track))
  + (sum npos: point_position. sum p:point_id. sum se:section_id.
      (legal_section(se) && legal_point(p))
      -> setPositionPointRecTrain(p, npos, se).Train(poc = point_update_position(p,npos,poc)));

%Interlocking can non deterministically choose to do any action that is enabled
%Abbreviations: sec->section collection, sic->signal collection, roc->route collection,
poc->point collection, pro->precalculated routes, rro->requested routes
Interlocking(sec: sections, sic: signals, roc:List(route_info), poc: points,
  pro:Set(route_info), rro: List(route_info)) =
  InterlockingUpdatingSignal()
  + InterlockingReadingSection()
  + InterlockingMovingPoint()
  + InterlockingReceivingRouteRequest()
  + InterlockingProcessingRoute()
  + InterlockingReadyRoute()
  + InterlockingNotReadyRoute()
  + InterlockingPermitTrainEntry();

%To enter the yard the interlocking needs to give permission. Permission is
%given when the entry section is free, the signal on that section is showing
%stop and the border is not claimed (a route set to the open track via that
%section)
InterlockingPermitTrainEntry(sec: sections, sic: signals, roc:List(route_info),
  poc: points, pro:Set(route_info), rro: List(route_info)) =
  sum se:section_id. sum next_section:section_id.
    ((next_section == get_next_section(0,se,L,sec,poc)
    || next_section == get_next_section(0,se,R,sec,poc))
    && (legal_section(se) && !section_get_occupance(se,sec)
    && border_unclaimed(se,roc)
    && in_front_of_signal(se,next_section,sic)
    && signal_get_colour(get_signal_ahead(se,next_section,sic),sic) == RD))
```

```
     -> permissionInterlocking(se,true)
     .Interlocking(sec = section_update_status(se,occupied,sec));

%Read the status of a section. Reading a section is occupied always results
%in the interlocking seeing the section as occupied. When a section becomes
%unoccupied the interlocking can see it as truely free or logically occupied.
%It can become logically occupied when it is the first section of a route and
%the next section has not (yet) become occupied (the train disappeared in view
%of the interlocking). Once a section has become free or occupied the effect
%on the routes is also calculated immediately.
InterlockingReadingSection(sec: sections, sic: signals, roc:List(route_info),
  poc: points, pro:Set(route_info), rro: List(route_info)) =
  sum s: Bool. sum se: section_id.legal_section(se) -> getStatusSectionRec(se, s)
      .((!s && !section_on_border(se,sec)
        && !second_section_of_route_occupied(se,sec,roc)
        && section_first_of_active_route(se,roc)) ->
        Interlocking(sec = section_update_status(se,logically_occupied,sec))
      +  (!s && (section_on_border(se,sec) || second_section_of_route_occupied(se,sec,roc)
        || !section_first_of_active_route(se,roc))) ->
          Interlocking(sec = section_update_status(se,free,sec),
          roc = routes_handle_section_free(se,sic,poc,roc))
      + (s) -> Interlocking(sec = section_update_status(se,occupied,sec), roc =
      routes_handle_section_occupied(se,sic,roc)));

%receive a route request that is in the precalculated route_info set pro
%If a previous route request has not been answered yet a new
%route request is not possible.
InterlockingReceivingRouteRequest(sec: sections, sic: signals, roc:List(route_info),
  poc: points, pro:Set(route_info), rro: List(route_info)) =
  sum ro: route_info. (ro in pro && #rro == 0)
  -> requestRoute(entry_signal(ro), exit_signal(ro)).Interlocking(rro = [ro]);

%evaluate whether a requested route can be accepted or not
%accepted when there is no conflict with another route already accepted
InterlockingProcessingRoute(sec: sections, sic: signals, roc:List(route_info),
  poc: points, pro:Set(route_info), rro: List(route_info)) =
  (#rro > 0) -> ((signal_free(entry_signal(head(rro)),roc)
    && route_conflict_free_flank_signals(head(rro),sec,roc)
    && route_conflict_free_points(head(rro),sec,roc,poc)
    && route_conflict_free_head_on(head(rro),roc)) ->
    routeAccepted.Interlocking(roc = roc <| if(all_points_locked(head(rro),sec,poc),
    route_update_status(head(rro),locked),
      route_update_status(head(rro),accepted)), rro = [])
  + (!(signal_free(entry_signal(head(rro)),roc)
    && route_conflict_free_flank_signals(head(rro),sec,roc)
    && route_conflict_free_points(head(rro),sec,roc,poc)
    && route_conflict_free_head_on(head(rro),roc))) ->
    routeRejected.Interlocking(rro = []));

%if a route is accepted but not all points have been moved to the correct position yet,
%this process can move one of those points
InterlockingMovingPoint(sec: sections, sic: signals, roc:List(route_info),
  poc: points, pro:Set(route_info), rro: List(route_info)) =
  sum ro:route_info. sum ppp:point_position_pair. (ppp in get_points_to_be_locked(ro,sec,poc)
  && ro in roc && status(ro) == accepted) ->
    (setPositionPointSend(point(ppp), position(ppp), in_section(poc(point(ppp))))
    .Interlocking(poc = point_update_position(point(ppp),position(ppp),poc),
      roc = if(all_points_locked(ro,sec,point_update_position(point(ppp),position(ppp),poc)),
      route_collection_update_route(route_update_status(ro,locked),ro,roc),roc)));

%if the compute signal function returns a different signal aspect
```

```
  %than the current aspect, it is updated
  InterlockingUpdatingSignal(sec: sections, sic: signals, roc:List(route_info),
    poc: points, pro:Set(route_info), rro: List(route_info)) =
    sum result: signal_colour. sum si: signal_id. (legal_signal(si)
      && result == compute_signal(si,sec,sic,roc,poc)
      && !signal_get_virtual(si,sic)) ->
      ((!(result == signal_get_colour(si,sic)))
      -> setSignalSend(si, result).Interlocking(sic = signal_update_colour(si, result, sic),
        roc = routes_handle_update_signal(si,result,sec,sic,roc)));

  %a route that is locked and of which the section before the entry signal is occupied,
  %becomes ready (the signal can show proceed)
  InterlockingReadyRoute(sec: sections, sic: signals, roc:List(route_info),
    poc: points, pro:Set(route_info), rro: List(route_info)) =
    sum ro:route_info. (ro in roc && status(ro) == locked) ->
      (section_before_signal_occupied(entry_signal(ro),sec,sic)
      || section_before_signal_part_of_active_route(entry_signal(ro),sic,roc)) ->
      (legal_signal(entry_signal(ro)))->
      (sections_of_route_free(ro,sec)) ->
      readyRoute
      .Interlocking(roc = route_collection_update_route(route_update_status(ro,ready),ro,roc));

  %a route can go from ready backed to locked if the conditions for ready no longer apply
  InterlockingNotReadyRoute(sec: sections, sic: signals, roc:List(route_info),
    poc: points, pro:Set(route_info), rro: List(route_info)) =
    sum ro:route_info. (ro in roc && status(ro) == ready &&
      !(section_before_signal_occupied(entry_signal(ro),sec,sic)
      || section_after_signal_occupied(entry_signal(ro),sec,sic)
      || section_before_signal_part_of_active_route(entry_signal(ro),sic,roc))) ->
      notReadyRoute.
      Interlocking(roc = route_collection_update_route(route_update_status(ro,locked),ro,roc));

% Initialization
init
  %hide internal communication
  hide({skip},
  allow( % Allow the following actions to happen
    {setSignal, getStatusSection, setStatusSection,seeSignal, setPositionPoint,
    requestRoute,routeAccepted,routeRejected,activateRoute,readyRoute,notReadyRoute,
    freeUpSection,discardRoute,freeUpSignal,skip},
    comm(
      {
        seeSignalSend|seeSignalRec -> seeSignal,
        setSignalSend|setSignalRec -> setSignal,
        setStatusSectionSection|setStatusSectionTrain|getStatusSectionRec  -> setStatusSection,
        setPositionPointSend|setPositionPointRec|setPositionPointRecTrain
          |setPositionPointRecTrain -> setPositionPoint
      },comm({permissionTrain|permissionInterlocking|setStatusSectionSection -> setStatusSection},
      %Initial situation, based on the configuration the right number of processes is created.
      %Some processes are also passed parameters based on the configuration.
      ((1 <= last_section) -> Section(1, false))
      || ((2 <= last_section) -> Section(2, false))
      || ((3 <= last_section) -> Section(3, false))
      || ((4 <= last_section) -> Section(4, false))
      || ((5 <= last_section) -> Section(5, false))
      || ((6 <= last_section) -> Section(6, false))
      || ((7 <= last_section) -> Section(7, false))
      || ((8 <= last_section) -> Section(8, false))
      || ((9 <= last_section) -> Section(9, false))
      || ((10 <= last_section) -> Section(10, false))
      || ((11 <= last_section) -> Section(11, false))
```

```
|| ((12 <= last_section) -> Section(12, false))
|| ((13 <= last_section) -> Section(13, false))
|| ((14 <= last_section) -> Section(14, false))
|| ((15 <= last_section) -> Section(15, false))
|| ((16 <= last_section) -> Section(16, false))
|| ((17 <= last_section) -> Section(17, false))
|| ((18 <= last_section) -> Section(18, false))
|| ((19 <= last_section) -> Section(19, false))
|| ((20 <= last_section) -> Section(20, false))
|| ((21 <= last_section) -> Section(21, false))
|| ((22 <= last_section) -> Section(22, false))
|| ((23 <= last_section) -> Section(23, false))
|| ((24 <= last_section) -> Section(24, false))
|| ((25 <= last_section) -> Section(25, false))
|| ((26 <= last_section) -> Section(26, false))
|| ((27 <= last_section) -> Section(27, false))
|| ((28 <= last_section) -> Section(28, false))
|| ((29 <= last_section) -> Section(29, false))
|| ((30 <= last_section) -> Section(30, false))
|| ((31 <= last_section) -> Section(31, false))
|| ((32 <= last_section) -> Section(32, false))
|| ((33 <= last_section) -> Section(33, false))
|| ((34 <= last_section) -> Section(34, false))
|| ((35 <= last_section) -> Section(35, false))
|| ((36 <= last_section) -> Section(36, false))
|| ((1 <= last_signal && !virtual(signals_config(instance)(1))) -> Signal(1, RD))
|| ((2 <= last_signal && !virtual(signals_config(instance)(2))) -> Signal(2, RD))
|| ((3 <= last_signal && !virtual(signals_config(instance)(3))) -> Signal(3, RD))
|| ((4 <= last_signal && !virtual(signals_config(instance)(4))) -> Signal(4, RD))
|| ((5 <= last_signal && !virtual(signals_config(instance)(5))) -> Signal(5, RD))
|| ((6 <= last_signal && !virtual(signals_config(instance)(6))) -> Signal(6, RD))
|| ((7 <= last_signal && !virtual(signals_config(instance)(7))) -> Signal(7, RD))
|| ((8 <= last_signal && !virtual(signals_config(instance)(8))) -> Signal(8, RD))
|| ((9 <= last_signal && !virtual(signals_config(instance)(9))) -> Signal(9, RD))
|| ((10 <= last_signal && !virtual(signals_config(instance)(10))) -> Signal(10, RD))
|| ((11 <= last_signal && !virtual(signals_config(instance)(11))) -> Signal(11, RD))
|| ((12 <= last_signal && !virtual(signals_config(instance)(12))) -> Signal(12, RD))
|| ((13 <= last_signal && !virtual(signals_config(instance)(13))) -> Signal(13, RD))
|| ((14 <= last_signal && !virtual(signals_config(instance)(14))) -> Signal(14, RD))
|| ((15 <= last_signal && !virtual(signals_config(instance)(15))) -> Signal(15, RD))
|| ((16 <= last_signal && !virtual(signals_config(instance)(16))) -> Signal(16, RD))
|| ((17 <= last_signal && !virtual(signals_config(instance)(17))) -> Signal(17, RD))
|| ((18 <= last_signal && !virtual(signals_config(instance)(18))) -> Signal(18, RD))
|| ((1 <= last_point) -> Point(1, position(points_config(instance)(1)),
  in_section(points_config(instance)(1))))
|| ((2 <= last_point) -> Point(2, position(points_config(instance)(2)),
  in_section(points_config(instance)(2))))
|| ((3 <= last_point) -> Point(3, position(points_config(instance)(3)),
  in_section(points_config(instance)(3))))
|| ((4 <= last_point) -> Point(4, position(points_config(instance)(4)),
  in_section(points_config(instance)(4))))
|| ((5 <= last_point) -> Point(5, position(points_config(instance)(5)),
  in_section(points_config(instance)(5))))
|| ((6 <= last_point) -> Point(6, position(points_config(instance)(6)),
  in_section(points_config(instance)(6))))
|| Train(before_track, false, entry_section(trains_config(instance)(1)), [], 0,
  direction(trains_config(instance)(1)),
  start_over(trains_config(instance)(1)),
  sections_config(instance), signals_config(instance), points_config(instance))
|| Train(before_track, false, entry_section(trains_config(instance)(2)), [], 0,
  direction(trains_config(instance)(2)),
```

```
        start_over(trains_config(instance)(2)),
        sections_config(instance), signals_config(instance), points_config(instance))
      %|| Train(before_track, false, entry_section(trains_config(instance)(3)), [], 0,
      %  direction(trains_config(instance)(3)),
      %  start_over(trains_config(instance)(3)),
      %  sections_config(instance), signals_config(instance), points_config(instance))
      %|| Train(before_track, false, entry_section(trains_config(instance)(4)), [], 0,
      %  direction(trains_config(instance)(4)),
      %  start_over(trains_config(instance)(4)),
      %  sections_config(instance), signals_config(instance), points_config(instance))
      || Interlocking(sections_config(instance), signals_config(instance), [],
        points_config(instance),
        get_possible_routes(routing_table_config(instance),
        sections_config(instance),signals_config(instance),
          points_config(instance),{}),[])
))));
```

A.2 Variants

As the majority of the model is the same for all variants we will simply list the differences between the variants.

A.2.1 Difference C-B

Compared to C, variant B has slightly different communications regarding setting/reading the occupancy of a section. In variant B, setting and reading are done separately, resulting in the following communication operator:

```
comm(
{
seeSignalSend|seeSignalRec -> seeSignal,
setSignalSend|setSignalRec -> setSignal,
setStatusSectionSection|setStatusSectionTrain -> setStatusSection,
getStatusSectionSend|getStatusSectionRec -> getStatusSection,
setPositionPointSend|setPositionPointRec|setPositionPointRecTrain
|setPositionPointRecTrain -> setPositionPoint
},comm({permissionTrain|permissionInterlocking -> setStatusSectionTrain}.
```

To prevent that variant B continuously receives the same status from a section it will only read a section occupation when it is different than in the memory. The restriction is in process InterlockingReadingSection:

```
sum s: Bool. sum se: section_id.
  (legal_section(se) && section_get_occupance(se,sec)!=s) -> getStatusSectionRec(se, s).
```

A.2.2 Difference B-B1

The only difference between variant B and B1 is that B1 excludes a specific timing issue. The restriction is in process InterlockingReadingSection:

```
sum s: Bool. sum se: section_id.(legal_section(se)
  && section_get_occupance(se,sec)!=s) ->
  (!(section_on_border(se,sec)
  && !section_last_of_active_route(se,roc)
  && ((legal_section(get_next_section(0,se,L,sec,poc))
  && !section_get_occupance(get_next_section(0,se,L,sec,poc),sec))
  || (legal_section(get_next_section(0,se,R,sec,poc))
  && !section_get_occupance(get_next_section(0,se,R,sec,poc),sec))) && !s))
  ->getStatusSectionRec(se, s).
```

A.2.3 Difference B-B2

The difference between variant B and B2 is that in variant B2, the section before the entry signal is part of the route. To make it part of the route, the equation find_route_check_done is modified to not use free_first_section on the end result, which normally removes the section before the entry signal. A few adaptations need to made in other places

as well. The equation section_before_signal_part_of_active_route is modified to take a route as parameter and excludes it in the check.

```
map
  section_before_signal_part_of_active_route: signal_id#route_info#signals#List(route_info)
    -> Bool;
eqn
  section_before_signal_part_of_active_route(si,ro,sic,roc) =
    legal_signal(si) && exists route:route_info.
    (route in roc && route != ro && (status(route) == active || status(route) == ready))
    && (exists route_section: section_id. legal_section(route_section)
    && (route_section in sections_of_route(route))
    && (route_section == section_before_signal(si,sic)));
```

The equation section_first_of_active_route is modified to also include ready routes.

```
section_first_of_active_route(se,roc) =
  exists route:route_info. route in roc
    && (status(route) == active || status(route) == ready)
    && #sections_of_route(route) > 0
    && se == head(sections_of_route(route));
```

The compute_signal equation is modified to only check that sections after the entry signal are free.

```
compute_signal(si,sec,sic,roc,poc) =
  if(!(exists ro:route_info. ro in roc && entry_signal(ro) == si
    && status(ro) == ready && sections_free(tail(sections_of_route(ro)),sec)),
    RD,
      if(exists ro:route_info. ro in roc && entry_signal(ro) == si
        && status(ro) == ready && sections_free(tail(sections_of_route(ro)),sec)
        && signal_get_colour(exit_signal(ro),sic) != RD
        && !signal_get_virtual(exit_signal(ro),sic),
        GR,
      GL));
```

To prevent that when a train leaves the yard a logical occupation is left behind all variants have a check to exclude logical occupations of sections on the border. The equation section_last_of_active_route in variant B2 is used to limit this restriction to only prevent that the last section of a route that is also on the border can become logically occupied (the entry section can become logically occupied). The process InterlockingReadingSection is modified as follows:

```
.((!s && !(section_on_border(se,sec)
  && section_last_of_active_route(se,roc))
  && !second_section_of_route_occupied(se,sec,roc)
  && section_first_of_active_route(se,roc))
  -> Interlocking(sec = section_update_status(se,logically_occupied,sec))
+ (!s && ((section_on_border(se,sec) && section_last_of_active_route(se,roc))
  || second_section_of_route_occupied(se,sec,roc)
  || !section_first_of_active_route(se,roc)))
  -> Interlocking(sec = section_update_status(se,free,sec),
    roc = routes_handle_section_free(se,sic,poc,roc))
```

The equation section_before_signal_part_of_active_route, which was modified, is used by the processes InterlockingReadyRoute and InterlockingNotReadyRoute; they are modified to pass the route_info they sum over to the equation. The process InterlockingReadyRoute is also modified to use `sections_free(tail(sections_of_route(ro)),sec)` instead of `sections_of_route_free(ro,sec)`.

A.2.4 Differences B-A

Variant A, compared to variant B, performs some transitions in the life cycle of a route through actions instead of via data equations. The processes InterlockingReadingSection and InterlockingUpdatingSignal therefore do not use equations to update routes when recursing back to the main process Interlocking. The process Interlocking also has the choice to perform these extra actions:

```
+ InterlockingActivateRoute()
+ InterlockingFreeUpSection()
+ InterlockingFreeUpSignal()
+ InterlockingDiscardObsoleteRoute().
```

The processes are defined below:

```
%once the train has moved passed the entry signal the route becomes active,
%the sections can now be cleared as they become free
InterlockingActivateRoute(sec: sections, sic: signals, roc:List(route_info),
  poc: points, pro:Set(route_info), rro: List(route_info)) =
  sum ro:route_info. (ro in roc && status(ro) == ready
    && section_after_signal_occupied(entry_signal(ro),sec,sic))
    -> activateRoute
    .Interlocking(roc = route_collection_update_route(route_update_status(ro,active),ro,roc));

%sections that are part of a route that have already been passed
%by the train can be freed up for other routes/trains
InterlockingFreeUpSection(sec: sections, sic: signals, roc:List(route_info),
  poc: points, pro:Set(route_info), rro: List(route_info)) =
  sum se: section_id. sum ro:route_info. (legal_section(se)
    && ro in roc && #sections_of_route(ro) != 0
    && !section_get_occupance(se,sec)
    && (!(section_on_border(se,sec)) => section_get_occupance(get_second_section_of_route(ro),sec))
    && status(ro) == active && section_first_of_route(se,ro))
      -> freeUpSection(se)
      .Interlocking(roc = route_collection_update_route(route_free_first_section(ro,sic,poc),ro,roc));

%signals that have been passed by the train can be freed up for other routes/trains
InterlockingFreeUpSignal(sec: sections, sic: signals, roc:List(route_info),
  poc: points, pro:Set(route_info), rro: List(route_info)) =
  sum si: signal_id. sum ro:route_info.
    (legal_signal(si) && ro in roc && #signals_of_route(ro) != 0
    && signal_get_colour(si,sic) == RD
    && status(ro) == active && signal_first_of_route(si,ro))
      -> freeUpSignal(si)
      .Interlocking(roc = route_collection_update_route(route_free_first_signal(ro),ro,roc));

%once all sections of a route have been freed up, the route can be discarded
InterlockingDiscardObsoleteRoute(sec: sections, sic: signals, roc:List(route_info),
poc: points, pro:Set(route_info), rro: List(route_info)) =
  sum ro:route_info. (ro in roc && sections_of_route(ro) == []) -> discardRoute
  .Interlocking(roc = route_collection_discard_route(ro,roc));
```

A.2.5 Differences C-GESIM

To restrict what routes can be requested during testing a Signalman process was added. The process is given below.

```
Signalman(sec: sections, sic: signals, rot:routing_table, exit_signals:Set(signal_id)) =
  sum sip: signal_pair. (sip in rot &&
  (on_border(sec(section_before(sic(entry_signal(sip))))) || entry_signal(sip) in exit_signals))
  -> requestRouteSend(entry_signal(sip), exit_signal(sip))
  . (routeAcceptedRec.Signalman(exit_signals = exit_signals + {exit_signal(sip)})
  + routeRejectedRec.Signalman());
```

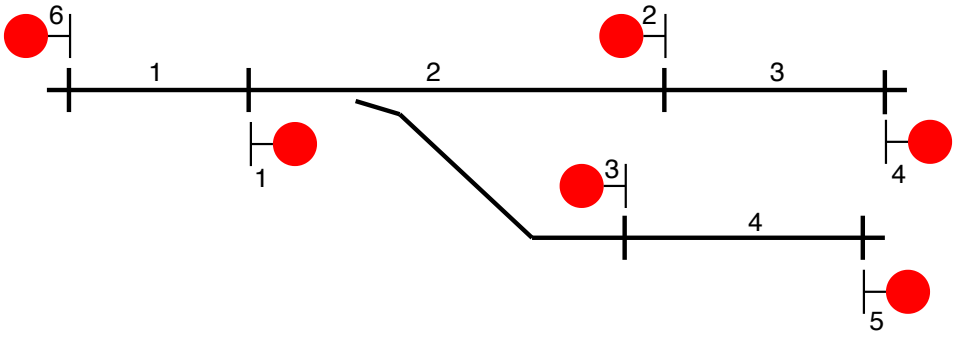
To enforce communication the requestRoute, routeAccepted and routeRejected are split into a receive and send action, which are forced to communicate. The interlocking is also adapted to use the send/receive actions.

```
requestRoute, requestRouteRec, requestRouteSend: signal_id # signal_id;
routeAccepted, routeAcceptedRec, routeAcceptedSend;
routeRejected, routeRejectedRec, routeRejectedSend;

routeAcceptedSend|routeAcceptedRec -> routeAccepted,
routeRejectedSend|routeRejectedRec -> routeRejected,
requestRouteSend|requestRouteRec -> requestRoute,
```

A.3 Track layouts for model checking

The following track layout was used for model checking. Three configurations were used for model checking: configuration 1 and 2 specify different train scenarios and are used for checking most properties. Configuration 3 was used to check the stronger liveness property that all trains always cross the yard (only one route can be requested in this configuration).

Figure A.1: Track layout verification

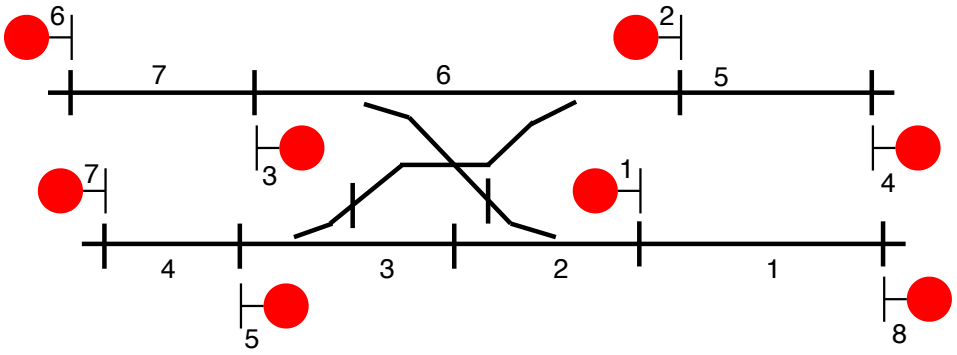



```
constants(1) = constants(3);
sections_config(1) = sections_config(3);
signals_config(1) = signals_config(3);
points_config(1) = points_config(3);
trains_config(1)(1) = train_config(4,L,false);
trains_config(1)(2) = train_config(4,L,false);
routing_table_config(1) = [signal_pair(1,4), signal_pair(1,5),
  signal_pair(2,6),signal_pair(3,6)];

constants(2) = constants(3);
sections_config(2) = sections_config(3);
signals_config(2) = signals_config(3);
points_config(2) = points_config(3);
trains_config(2)(1) = train_config(4,L,false);
trains_config(2)(2) = train_config(1,R,false);
routing_table_config(2) = routing_table_config(1);

constants(3) = c(4,6,1,3);
sections_config(3)(1) = section_info(free, true, [section_connection([],efp,0,2)]);
sections_config(3)(2) = section_info(free, false,
    [section_connection([point_position_pair(1,left)],flank_protection(2,[],[3]),1,3),
    section_connection([point_position_pair(1,right)],flank_protection(2,[],[2]),1,4)]);
sections_config(3)(3) = section_info(free, true, [section_connection([],efp,2,0)]);
sections_config(3)(4) = section_info(free, true, [section_connection([],efp,2,0)]);
signals_config(3)(1) = signal_info(RD, R, false, 1,2);
signals_config(3)(2) = signal_info(RD, L, false, 3,2);
signals_config(3)(3) = signal_info(RD, L, false, 4,2);
signals_config(3)(4) = signal_info(GR, R, true, 3,0); %virtual signal always green
signals_config(3)(5) = signal_info(GR, R, true, 4,0); %virtual signal always green
signals_config(3)(6) = signal_info(GR, L, true, 1,0); %virtual signal always green
points_config(3)(1) = point_info(2, right);
trains_config(3)(1) = train_config(4,L,false);
trains_config(3)(2) = train_config(4,L,false);
routing_table_config(3) = [signal_pair(3,6)];
```

A.4 Track layout testing GESIM

The following track layout was used for testing with JTorX.

Figure A.2: Visual representation track layout




```
constants(4) = c(7,8,4,4);
sections_config(4)(1) = section_info(free, true, [section_connection([],efp,2,0)]);
sections_config(4)(2) = section_info(free, false,
    [section_connection([point_position_pair(4,left)],
        flank_protection(2,[point_position_pair(3,left)],[]),3,1),
section_connection([point_position_pair(4,right)],flank_protection(2,[],[5]),6,1)]);
sections_config(4)(3) = section_info(free, false,
    [section_connection([point_position_pair(1,left)],flank_protection(3,[],[1]),4,6),
    section_connection([point_position_pair(1,right)],
        flank_protection(3,[point_position_pair(2,right)],[]),4,2)]);
sections_config(4)(4) = section_info(free, true, [section_connection([],efp,0,3)]);
sections_config(4)(5) = section_info(free, true, [section_connection([],efp,6,0)]);
sections_config(4)(6) = section_info(free, false,
    [section_connection([point_position_pair(3,left),point_position_pair(2,right)],
        flank_protection(6,[point_position_pair(4,left),
        point_position_pair(1,right)],[]),7,5),
    section_connection([point_position_pair(3,right)],
        flank_protection(6,[point_position_pair(1,right),
        point_position_pair(2,right)],[5,2]),7,2),
    section_connection([point_position_pair(2,left)],
        flank_protection(6,[point_position_pair(3,left),
        point_position_pair(4,left)],[3,1]),3,5)]);
sections_config(4)(7) = section_info(free, true, [section_connection([],efp,0,6)]);
signals_config(4)(1) = signal_info(RD, L, false, 1,2);
signals_config(4)(2) = signal_info(RD, L, false, 5,6);
signals_config(4)(3) = signal_info(RD, R, false, 7,6);
signals_config(4)(4) = signal_info(GR, R, true, 5,0); %virtual signal always green
signals_config(4)(5) = signal_info(RD, R, false, 4,3);
signals_config(4)(6) = signal_info(GR, L, true, 7,0); %virtual signal always green
signals_config(4)(7) = signal_info(GR, L, true, 4,0); %virtual signal always green
signals_config(4)(8) = signal_info(GR, R, true, 1,0);%virtual signal always green
points_config(4)(1) = point_info(3, right);
points_config(4)(2) = point_info(6, right);
points_config(4)(3) = point_info(6, right);
points_config(4)(4) = point_info(2, right);
trains_config(4)(1) = train_config(1,L,false);
trains_config(4)(2) = train_config(5,L,false);
trains_config(4)(3) = train_config(4,R,false);
trains_config(4)(4) = train_config(7,R,false);
routing_table_config(4) = [signal_pair(1,6), signal_pair(1,7),signal_pair(2,6),
    signal_pair(2,7), signal_pair(3,4), signal_pair(3,8), signal_pair(5,4),
    signal_pair(5,8)];
```

Appendix B

Scripts and settings

B.1 Bash script to generate state space and check modal μ-formulas

The following script assumes that the mCRL2 model is located in the same folder as the script, named modelrail.mcrl2. Furthermore, a folder called "Formulas" should be present in the parent folder, containing all the modal μ-calculus formulas.

```
time mcrl22lps --verbose --timings=time.txt modelrail.mcrl2 modelrail.lps
time lps2lts --verbose --timings=time.txt -rjittyc --cached modelrail.lps modelrail.lts
time ltsconvert modelrail.lts modelrail.min.lts --timings=time.txt
    --equivalence=branching-bisim --out=modelrail.min.lts
    --tau="seeSignal,getStatusSection,activateRoute,readyRoute,notReadyRoute,freeUpSection,
    discardRoute,freeUpSignal"
    --verbose

echo "Checking for collisions"
time lts2pbes -v modelrail.min.lts --timings=time.txt modelrail_no_collision.pbes
--counter-example --formula=../Formulas/noCollision.mcf --lps=modelrail.lps
time pbessolve -rjitty --verbose --timings=time.txt modelrail_no_collision.pbes --in=pbes
--search=breadth-first --strategy=2

echo "Checking points moved while occupied"
time lts2pbes -v modelrail.min.lts --timings=time.txt modelrail_no_move_point.pbes
--counter-example --formula=../Formulas/noMovePointWhileOccupied.mcf --lps=modelrail.lps
time pbessolve -rjitty --verbose  --timings=time.txt modelrail_no_move_point.pbes --in=pbes
--search=breadth-first --strategy=2

echo "Checking for deadlock"
time lts2pbes -v modelrail.min.lts --timings=time.txt modelrail_no_deadlock.pbes
--counter-example --formula=../Formulas/nodeadlock.mcf --lps=modelrail.lps
time pbessolve -rjitty --verbose  --timings=time.txt modelrail_no_deadlock.pbes --in=pbes
--search=breadth-first --strategy=2

echo "Checking if all trains always cross the yard"
time lts2pbes -v modelrail.min.lts --timings=time.txt modelrail_always_cross.pbes
--counter-example --formula=../Formulas/allTrainsAlwaysCross.mcf --lps=modelrail.lps
time pbessolve -rjitty --verbose  --timings=time.txt modelrail_always_cross.pbes --in=pbes
--search=breadth-first --strategy=2

echo "Checking if all trains can cross the yard"
time lts2pbes -v modelrail.min.lts --timings=time.txt modelrail_can_cross.pbes
--counter-example --formula=../Formulas/allTrainsCanCross.mcf --lps=modelrail.lps
time pbessolve -rjitty --verbose  --timings=time.txt modelrail_can_cross.pbes --in=pbes
--search=breadth-first --strategy=2

echo "Checking if all sections can become occupied at any time"
time lts2pbes -v modelrail.min.lts --timings=time.txt modelrail_can_occupy_section.pbes
```

```
--counter-example --formula=../Formulas/alwaysCanMakeSectionOccupied.mcf --lps=modelrail.lps
time pbessolve -rjitty --verbose  --timings=time.txt modelrail_can_occupy_section.pbes --in=pbes
--search=breadth-first --strategy=2

echo "Checking if all points can be moved at any time"
time lts2pbes -v modelrail.min.lts --timings=time.txt modelrail_can_move_point.pbes
--counter-example --formula=../Formulas/alwaysCanMovePoint.mcf --lps=modelrail.lps
time pbessolve -rjitty --verbose  --timings=time.txt modelrail_can_move_point.pbes --in=pbes
--search=breadth-first --strategy=2

echo "Checking if all route requests are eventually answered"
time lts2pbes -v modelrail.min.lts --timings=time.txt modelrail_request_answer.pbes
--counter-example --formula=../Formulas/routeRequestAlwaysAnswered.mcf --lps=modelrail.lps
time pbessolve -rjitty --verbose  --timings=time.txt modelrail_request_answer.pbes --in=pbes
--search=breadth-first --strategy=2

read -p "Press [Enter] key to close window"
```

B.2 Settings JTorX

Field	**Value**
Model	Path to lps2torx script (torxspec.tx)
Implementation	real program, comm. labels via tcp, JTorX is client
Hostname!Portnr	IP!9999
Timeout	5 seconds
Interpretation	action names below
Trace kind	Straces
Input actions	requestRoute, setStatusSection
Output actions	routeAccepted, routeRejected, setSignal, setPositionPoint

Table B.1: JTorX configuration, where IP differs per session

The file torxspec is a simpe bash script that starts lps2torx. The contents of the file are:

```
#!/bin/sh
exec lps2torx -rjittyc /path/to/modelrail.lps
```

B.3 Steps to boot TeSys and prepare it for testing

Step	Action	Notes
1	Run _GESIM_START_COMPLETE_VICOS_PIA.cmd	Boots the TeSys components
2	In GUIDO, log in as Techniker mit bes. Sicherheitsverantwordung	The role authorized to free up the track elements.
3	Click “Aufruesten” and wait until all red labels have turned green	
4	In TAK, load the script init.tak from the TAK folder and run it	Frees up all track elements
5	In GUIDO, log off and log in as Fahrdienstleiter	The role authorized to set routes
6	Click on BEDIENBEREICH_1 in the lower left corner and select BG_AAN	Assigns the left part of the yard to the Fahrdienstleiter
7	Click on BEDIENBEREICH_2 in the lower left corner and select BG_AAN	Assigns the right part of the yard to the Fahrdienstleiter
8	Run START_ADAPTER.cmd in the JTorXAdapter folder	Closes the TAK GUI and starts the adapter
9	In the command line interface (CLI) of the adapter type “setStatusSection!0!true” and verify in GUIDO that a section has become occupied	Tests the connection between the adapter and TeSys
10	In the CLI of the adapter type “setStatusSection!0!false”	To remove the occupation before testing
11	In the CLI of the adapter type “switch” and note what the IP of the server is	Switches the adapter from standard input/output mode to TCP mode
12	Start JTorX with jtorx.bat, enter the details as described in Table B.1 and click start in the tab “test”	Establishes a connection with the adapter
13	Enter a number of steps and click “auto”	Automatically performs the specified number of testing steps