Formal verification of a train traffic control system

van Meer, Jasper H.P.

Award date:
2020

Eindhoven University of technology
Dept. of Mathematics and Computer Science
Formal System Analysis Group

Formal Verification of a train traffic Control System

Master thesis

Jasper van Meer

University supervisors:
Bas Luttik
Mark Bouwman

Company supervisor:
Bert Bossink

Utrecht, June 30, 2020

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

Formal Verification of a Train Traffic Control System

Master Thesis

Jasper van Meer

University supervisors
Bas Luttik
Mark Bouwman

Company supervisors
Bert Bossink

Utrecht, June 30, 2020

Abstract

It is no secret that as technology evolves, software systems grow more complex. Difficulties arise with the specification of such complex systems. In the quest for higher quality and safer software the use of formal methods provides a solution to disambiguate and introduce precision and rigor through their mathematical foundation.

This thesis presents an application of the mCRL2 toolset on a Train Traffic Control System. The mCRL2 toolset is used for the formal specification and verification of a part of ProRail's 'Astris'. Astris is a system in the network of systems that control the train traffic of the Dutch railways. The use of formal specification and verification have proven to be able to have added value to ProRail's software development and maintenance process. The suitability of mCRL2 for train traffic control systems is evaluated. Suggestions are given for the use of formal methods in ProRail's current way of working.

Acknowledgements

The calligraphy of the front cover is made by me. It's the result of practicing since the start of this project. I still have a way to go but making this cover was fun. Thanks to Martin Renkema for his suggestion to do this.

As of writing this thesis, the world is in the midst of the Coronavirus pandemic. It is a strange experience to continue working from home while there is so much grief and sorrow outside. I therefore take the opportunity to extend my thanks and gratitude to the people that have supported me throughout the project in spite of these difficult times.

I would like to thank my company supervisor Bert Bossink, for all of his help. I have greatly appreciated his thorough feedback and our chit-chatting about hobbies and whatnot. Additionally, I would like to thank the fellow ProRailers Martin Renkema, Peter van de Werken and Berry Overkamp for their direct help in the project. And I also would like to extend my thanks to all colleagues from ProRail for their hospitality and taking an interest in the project.

My thanks also goes out to my university supervisor, Bas Luttik. I am grateful for the frequent and thorough feedback Bas has given me. I have greatly appreciated his guidance.

I was pleasantly surprised to hear that Mark Bouwman would be my second university supervisor since not too long ago we were both students successfully doing a project together. I extend my thanks to Mark for his feedback and very helpful contributions. I wish him all the best for his pursuit of a PhD.

Lastly, I would like to thank my parents for their endless support.

Contents

Glossary

Term or abbreviation	Meaning
ASTRIS	***A****ansturen en* ***ST****atusmelding* ***R****ail****I****nfra****S****tructuur*
BevNL	***Bev****eiliging* ***N****eder****l****and*
CSS	Control and Signalling System
ERTMS	European Rail Traffic Management System
Formal specification	A formal method; a mathematically rigorous specification
IL/IXL	Interlocking
Informal specification	The documentation provided by ProRail written in natural (Dutch) language
Model	The formal specification derived from the informal specification
mCRL2	micro Common Representation Language 2
PRL	***Pr****oces****l****eiding Rijwegen*
SOA	Service Oriented Architecture
TCS	Traffic Control System
TROTS	**Tr**ain **O**bservation and **T**racking **S**ystem

English	Dutch
Point	Wissel
Route	Rijweg
To set a route	Instellen van een rijweg
Track	Spoor
Signal	Sein
Cancel route	Herroepen

Chapter 1

Introduction

It is no secret that as technology evolves, software systems grow more complex. ProRail is a company that has to overcome the challenge of specifying such complex systems. In their quest for higher quality and safe software, ProRail seeks a solution. For this, ProRail investigates the use of formal methods.

1.1 Background

ProRail is a government task organisation that takes care of maintenance and extensions of the Dutch national railway network infrastructure (not the metro or tram), of allocating rail capacity, and of traffic control [20]. The traffic control systems are the focus of this thesis. Within ProRail, the department of ICT creates and maintains systems for the Dutch railways. This includes the traffic control systems. The subdivision for train traffic control (*treinbeheersing*) creates large custom applications where safety is highly valued. They do so with the help of software contractors.

The traffic control systems are responsible for the everyday operation of moving trains efficiently along the infrastructure in a safe manner. This is done almost fully automatically according to a pre-determined timetable. Therefore, human intervention is mostly only necessary in the event of e.g. accidents or rail maintenance.

The railway elements, e.g. points, signals or axle counters, are not controlled by the traffic control systems directly. In between are the 'control and signalling systems' that bear the responsibility for the safety of the infrastructure. The control and signalling systems prevent dangerous situations such as derailments and train collisions. Therefore, these systems are required to conform to the highest Safety Integrity Levels (SIL 3-4).

The traffic control systems provide an overarching network that ties all the independent control and signalling systems together such that the entire infrastructure can be controlled as a whole. While these traffic control systems do not bear a direct responsibility for safety, they should not rely on the control and signalling systems to resolve unsafe situations that the traffic control systems can prevent since this may cause delays. Prevention is better than cure after all. Thus, the traffic control systems attempt to prevent unsafe situations and increase availability as much as possible within their scope and, as a consequence of this responsibility, are required to conform to SIL 1.

A more detailed explanation of the Dutch infrastructure and the role of the traffic control systems can be found in Section 2.1.

1.2 The problem

ProRail's development and maintenance process for software applications follows the well-known waterfall model (also known or similar to the V-model or V-cycle) supported by the MIL-STD-498 [14] standard

documentation structure. When the concept and high level goals of a new application are clear, engineers and domain experts will meticulously write down all requirements to the point where almost a detailed design is dictated.

Having written down the requirements in a specification, this specification is handed over to (one or more) software contractors that will create the specified application. During this process ProRail regularly meets with the contractors to discuss progress and answer questions.

However, they find that the fact that the specifications are written in natural language gives freedom in interpretation by the ambiguity inherent to natural language. This leads to many questions, leading to discussions, leading to refinements of the specifications. And these refinements may itself be cause for more questions. This effect is strengthened by the notion that the software contractors are not necessarily specialised in the railway domain. Moreover, this problem is not limited to the scope of the transfer of knowledge between ProRail and the software contractors, but also internally within ProRail, e.g. the department responsible for testing or maintaining knowledge in the face of employee churn. Especially the latter can be important considering the aging workforce of ProRail.

Clearly, this is undesirable and ProRail seeks a solution to specify system requirements more precisely. As a potential solution, they are looking into the use of formal methods.

1.3 Formal methods in the railway domain

Formal methods is a collective name for methods or techniques like formal specification and formal verification, where 'formal' refers to the foundation in mathematics in the areas of mathematical logic and set theory. This allows the application of mathematical techniques on the models for the purpose of analysis, proving properties etc.

The use of formal methods is not new in the railway domain [2, 9]. They have been applied successfully many times over the last few decades in e.g. France [3], the Netherlands [10], Italy and the UK. The CENELEC EN 50128 [8] standard, which ProRail must conform to, mentions formal methods as a highly recommended practice for (Safety Integrity Level) SIL 3-4 applications. To that extent the transportation sector and specifically the railways gain a lot from formal methods and the potential for more is high.

The literature [19] is clear on the benefits of the use of formal specification. They may be used with requirements defined in natural language to clarify any areas of potential ambiguity. Since they are precise and unambiguous they remove doubt from the specification and avoid problems of language interpretation. These methods also enable automation in e.g. formal verification, test case generation, code generation and simulation. To this end formal methods seem to tackle exactly ProRail's problems described in Section 1.2.

The literature also mentions that formal methods can be cost-effective when used for core parts of critical systems where safety, reliability and security are particularly important. Not much is known about non- or less-critical systems that are under consideration in this thesis. However, that does not mean that it can't be. And thus, this thesis investigates this possibility.

Additionally, the use of formal methods can save time in the later phases of the development process (e.g. testing, integration, validation) at the expense of investing more time in the specification phase. Coincidentally, these later phases are where ProRail would like to save time.

Clearly, there is reason to believe that the use of formal methods is a suitable solution for ProRail's problem.

1.4 Goals

ProRail wishes to investigate the use of formal methods to determine its suitability in their development and maintenance process. Specifically, if and how formal methods solve the problem described in Section 1.2 and how these methods can be used together with the MIL-STD-498 standard documentation structure.

ProRail is mostly interested in what formal methods can do for them in the areas of specification, verification and testing. This thesis focuses on specification and verification.

For the end result, the goal is to demonstrate the state-of-the-art technologies for formal methods and their use to ProRail in the train traffic control domain. This thesis presents a case study where a part of an existing train traffic control application will be formally specified and verified for correctness and safety properties. For this, the mCRL2 toolset is used. The choice for this tool is motivated in Section 3.1 together with an introduction to mCRL2 in Section 3.2. The application in question is Astris, which will be discussed in more detail in Chapter 2. With the observations and feedback from ProRail we will be able to draw a conclusion on the suitability of formal methods in the software development and maintenance process of ProRail.

Moreover, a recommendation will be presented. Providing an additional point of view on the problem.

The contribution of this thesis is to demonstrate the applicability of formal methods, in particular using mCRL2, for train traffic control applications. Specifically for the use of specification and verification.

1.5 Results

This thesis describes the formal specification and verification of Astris using the mCRL2 toolset. We start our model by taking Astris' Element Component and its 'locking mechanism' under consideration in Chapter 4. We have verified a claim made by ProRail on this mechanism and thereby effectively introduced on a small scale what formal methods can do. We build upon this model in Chapter 5 and introduce interaction between a Route Component and the 'locking mechanism' of the Element Component. Here, through verification, we find that the Route Components' mechanism for dealing with conflicting locking of elements is not fail proof. Then, we expand the model even further by introducing the notion of 'coupled elements' in Chapter 6. Through verification we have proven ProRail's suspicion how deadlocks may occur with the introduction of these coupled elements. Finally, in Chapter 7 we go in-depth on the specification of a lengthy and complex requirement. Here we demonstrate concretely how the process of formal specification uncovers ambiguities, inconsistencies and unclarities in the informal specification. Moreover, the formal specification is more precise than the informal specification.

We have shown how the use of formal methods has added value and can thus be beneficial for ProRail in their software development and maintenance process.

Chapter 8 discusses the results in more detail.

1.6 Outline

The rest of the thesis is organised as follows. Chapter 2 gives a high level overview of the Dutch train traffic control system with special attention to the place and purpose of Astris, the system of interest in this thesis. Then, Chapter 3 explains the approach for formally specifying and verifying Astris. This includes a motivation and introduction of the mCRL2 toolset. In Chapters 4, 5 and 6 various mechanisms of Astris are formally specified and verified. Chapter 7 focuses on the formal specification of an elaborate requirement. Finally, in Chapter 8 we reflect on and discuss the results to finally come to a conclusion on the added value of formal methods for ProRail's software development and maintenance process. Additionally, a recommendation is given on the matter.

Chapter 2

Astris

The system of interest in this thesis is called Astris. We will first discuss train traffic control as a whole to gain an understanding of Astris's place and purpose. Then, the architecture of Astris is explained to go in-depth on some components of interest. Finally, we determine the direction and focus points for this thesis to guide ourselves to a starting point of the model.

2.1 Overview of traffic control

The railway infrastructure consists of a complex web of systems such as signals, points, sections, axle counters etc. that all work together to move trains along the infrastructure. However, these elements are not all controlled by a single entity. Figure 2.1 gives a visual overview of the components that work together to control the train traffic.

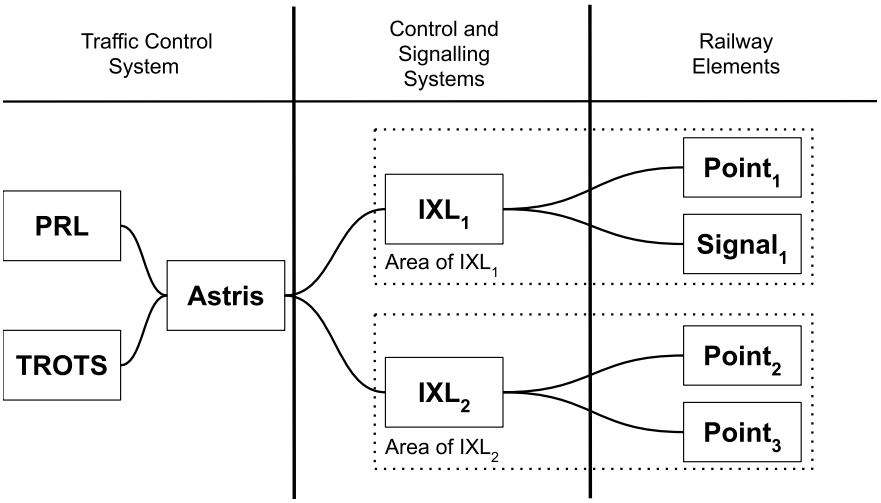


Figure 2.1: Overview of traffic control system

The rail network is partitioned into areas containing limited subsets of railway elements. Each one of such an area, and the railway elements in it, are directly controlled by a 'Control and Signalling System'. Often this is an interlocking system (IL/IXL). Depending on the scope, an interlocking may also be considered a railway element.

An interlocking system has the responsibility of guaranteeing the safety within its 'interlocking area'.

That is, it must prevent the occurrence of unsafe movements over the infrastructure in its area that may lead to train collisions or derailments. It achieves this by checking for and consequently disallowing the setting of conflicting routes that may lead to the aforementioned unsafe situations.

A 'Control and Signalling System' is able to receive commands to actuate points and signals for the purpose of e.g. setting routes. It can also be requested to share information on the position of trains on the infrastructure. The 'Control and Signalling Systems' receives such commands from the 'Traffic Control Systems'.

The Traffic Control Systems consist of a number of applications with different responsibilities. Let us consider two concrete examples from the figure:

- **PRL** is an abbreviation for '***Pr**ocesl**e**iding Rijwegen*' which roughly translates to 'process management of routes'. This application supports the *treindienstleiders* (train dispatchers) in their logistic and safety tasks for the process of controlling rail traffic.
- **TROTS** is an abbreviation for '**Tr**ain **O**bservation and **T**racking **S**ystem' where its function is self-explanatory from the name. Its purpose is to provide information on where trains are on the infrastructure.

However, these applications do not communicate directly with the 'Control and Signalling Systems'. The interface connecting the 'Traffic Control Systems' and the 'Control and Signalling Systems' is Astris. Astris is considered part of the 'Traffic Control Systems'. Note that there may be multiple instances of the applications of the 'Traffic Control Systems' running in parallel for distribution and redundancy purposes.

2.2 Purpose

The traffic control system under consideration in this thesis is called Astris. Astris is an acronym originating from "***A**ansturen en **ST**atusmelding **R**ail**I**nfra**S**tructuur*" which roughly translates to "**D**irecting and **ST**atus reporting **R**ail**I**nfra**S**tructure".

Obviously, as the rail infrastructure evolves, a lot of railway elements, e.g. points, interlockings, from different vendors are used. As a consequence, a number of different interfaces must be supported to interact with these elements. Astris aims to solve this problem by providing a uniform interface for applications that want to control and/or request information from these elements. In Figure 2.1 of te previous section we have seen such applications, i.e. **PRL** and **TROTS**.

2.3 Architecture

Astris is built on a Service Oriented Architecture (SOA) consisting of the following components: 'Area', 'Element', 'Route', 'Signalling', 'System' and 'Management'. This is reflected in Figure 2.2 below. The arrows in the figure indicate the dependencies between the components. Every component is at least connected to the Management component since it is responsible for managing the other components within Astris. The arrows denoting this relation are omitted in the figure.

The components reflect the general structure of the Dutch infrastructure. The country is logically partitioned into areas, which contain routes, which in turn contain elements. We can observe from the figure that these components heavily depend on the 'Signalling' component. This component is responsible for sending and receiving commands from the Control and Signalling Systems even though the name might not obviously imply such, hence, the dependency.

Within each instance of Astris, there can be multiple instances of every component except in the case of the Management component.

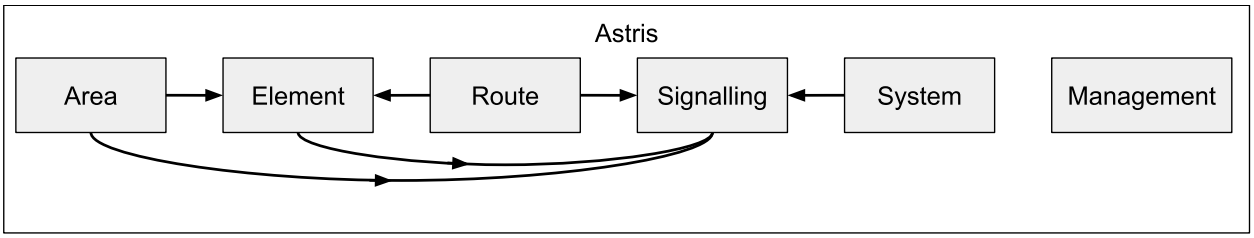


Figure 2.2: Astris components and dependencies

The components provide their services through a Request/Reply and Notification Message Exchange protocol. Concretely, each (instance of a) component is uniquely identifiable. As a result, service requests are always routed to the intended component. No matter if that is a component within the same Astris instance, or a component in an Astris instance on the other side of the country.

All components process service requests sequentially (not in parallel), i.e. one at a time. Thus, race conditions where conflicting service requests are processed at the same time by the same component cannot occur. Additionally, every component has its own message queue to fetch the service requests from. So, no problems can occur where multiple consumers fetch from the same queue. Moreover, obviously, as long as the queue does not overflow, no service request is lost.

2.4 Components under consideration

Since Astris is a very large software system we focus on a few components. The 'Element' and 'Route' Components have been selected since they have interesting logic and are fairly self-contained. Within these components the responsibilities are logically divided in three parts: a configuration part, the data management part and the core part responsible for offering functionality. The latter is most interesting for analysis. Below we elaborate on these components.

2.4.1 Element Component

The Element Component, henceforth abbreviated to EC, provides the elementary 'Element' service within Astris, but also to other applications outside of Astris. This service supports high level service operations to, for example, operate or block the operation of individual elements. Internally, within the logical scope of Astris the EC provides a number of services for mutating and accessing information on railway elements. Additionally, there is the service shared with the 'Management' component for management purposes.

The EC has the task to receive service requests and execute them. Apart from the service requests themselves, the EC's objective is to safeguard the execution of these requests such that no conflicting use of any element can occur. That is, whenever two applications wish to use the same element simultaneously, the EC must determine if the simultaneous use is permissible or conflicting and prevent race conditions on these service requests.

2.4.2 Route Component

The Route Component, henceforth abbreviated to RC, provides the 'Route' services to i.a. reserve, prepare, cancel, revoke and set routes throughout the infrastructure. It does so by making use of the above described EC's services to call upon the railways elements.

The RC has the task to receive service requests and execute them. The RC's primary objective is to safeguard and alter the state of the routes. The state determines if a request can be accepted and/or executed or not.

2.5 Research direction

Having determined the components we take into consideration, we decide on a concrete starting point and direction for our analysis of Astris. We are looking for a small (relatively) self-contained and interesting part of a component to start our model and familiarise ourselves more with Astris. Moreover, we want to be able to verify interesting properties about this part to enthuse ProRail for what verification can do. Ideally, we can then build upon this model to be able to verify interactions between the EC and the RC. Together with ProRail, we find within the Element Component a mechanism meeting our criteria, namely the 'locking mechanism'.

Chapter 3

Approach

In this chapter, the choice of tooling will be motivated and said tool will be introduced. Then, the general approaches to formally specifying the system and formally verifying properties of the system are covered.

3.1 Tooling

In this thesis we make use of the mCRL2 toolkit. mCRL2 is a formal specification language with an associated toolset that can be used for modelling and verification of concurrent systems and protocols. mCRL2 is developed at Eindhoven University of Technology in collaboration with the University of Twente [13]. mCRL2 has been used successfully in a number of case studies. For example for CERN's Large Hadron Collider [12, 15]. But also in the railway domain for the modelling and analysis of ERTMS Hybrid level 3 [1] and of an interlocking system [6].

A number of formal methods tools have been considered and compared. The details of this comparison can be found in Section 3.3 of Appendix G. The selection consisted of the mCRL2 toolset and the tools recommended in CENELEC EN 50128 [8]. A tool is needed that meets the following criteria: formal specification, verification and possibly functionalities for testing. mCRL2 is one of the tools meeting these criteria. Additionally, not only are we already familiar with mCRL2, we also have support from the university. With other tools we do not have these advantages. Moreover, we can investigate the suitability of the mCRL2 toolkit in the train traffic control domain. Hence, the motivation for using mCRL2.

3.2 mCRL2

In this section we will introduce mCRL2. We will not, however, get into detail on how mCRL2 works. For this, we refer the reader to the book by Groote and Mousavi [11]. In the following subsections we briefly give an overview of mCRL2 to then discuss the specification and requirement languages. Finally, we discuss a subset of the available tools that are used in this thesis and how these come together in a 'tool chain' for generating and verifying our models.

3.2.1 Overview

The mCRL2 toolset is a collection of tools for creating, simulating, visualising, analysing and verifying formal models. Specifically, with mCRL2 we model the behaviour of a system, i.e. an mCRL2 model is a behavioral model. The language for specifying properties is the modal μ-calculus. Again, a more thorough and precise description of mCRL2 and the modal μ-calculus can be found in the book by Groote and Mousavi [11]. On the mCRL2 website [13] one can find i.a. downloads for the toolset itself, user documentation and tutorials. The paper [7] is also recommended as a good starting point to familiarise oneself with mCRL2.

3.2.2 mCRL2 specification

The mCRL2 language, extends the Algebra of Communicating Processes (ACP) process specification language with i.a. notions of time and data based on the theory of abstract data types.

The semantics of an mCRL2 model is a Labelled Transition System (LTS). An LTS consists of a set of states (where one state is the initial state) and a set of labelled transitions between the states of the LTS. There exist mCRL2 specifications that are small but have a very large, possibly infinite, LTS and vice versa.

3.2.3 Modal μ-calculus

We can verify an mCRL2 model for properties using mCRL2's requirement language which is a highly expressive extension of the modal μ-calculus. It is very suitable to express patterns of (dis)allowed behaviour of a system. A simple example is that a system does not deadlock unless an error occurs. Most of the tools in the mCRL2 toolset are dedicated to the verification of properties.

3.2.4 Tool chain

As mentioned before, there are a number of tools that help in the analysis and verification of mCRL2 models. Figure 3.1 gives a overview of the tools used for simulation and verification. Verification is no trivial matter and a number of steps must be taken to verify the model, i.e. to get a true or false answer to whether or not some property of interest is valid in the mCRL2 model. Initially, we have an mCRL2 specification which we want to analyse depicted in the figure on the top left. From here we transform an mCRL2 specification to an LPS using `mcrl22lps`. With an LPS we can simulate the model with `lpsxsim` or generate a PBES to verify a property using `lps2pbes` which takes both the LPS and a `mcf` file containing the μ-calculus formula. This PBES can then be solved using `pbessolve` which will give a true or false verdict on whether the property holds in the model or not. This includes, if desired, evidence supporting this verdict. For some specifications a potentially faster strategy is transforming the LPS to an LTS and then generate the PBES form this LTS and the `mcf` file using `lts2pbes`. But we can go beyond to minimizing the size of the obtained LTS by means of `ltsconvert`. The effectiveness of the reduction is dependent the model itself, but also on the equivalence requirement. In this thesis we consider the LTS and the minimized LTS to be strongly bisimilar. A bisimulation relation relates two transitions systems that are behaviourally equivalent to an observer watching from the outside. There are other relations that may be more lenient in the permissible behaviour and consequently provide better reduction. Strong bisimilarity is a stringent equivalence requirement.

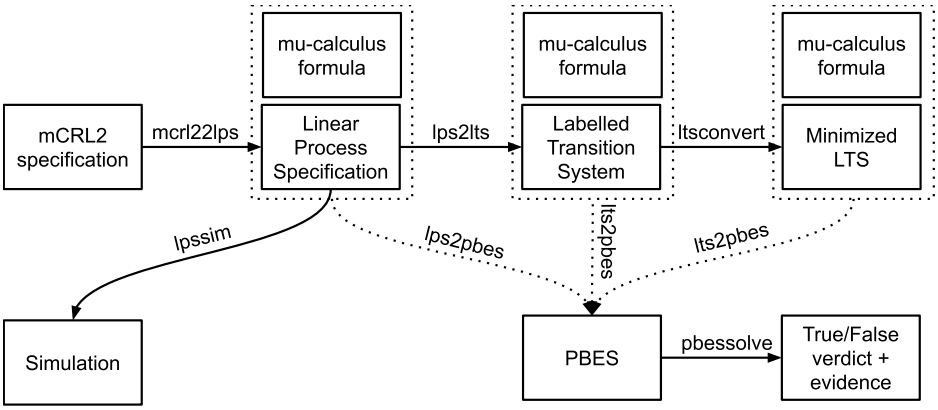


Figure 3.1: mCRL2 tool chain

For the development of the model in thesis, the `mcrl2ide` tool is used. This offers a Integrated Development Environment (IDE) where the user can edit the specification and launch visualisation and simulation

tools directly. Moreover, the IDE allows the editing and verification of properties. Currently, the tool lacks freedom in options and settings that are available when invoking the individual commands via the command line.

The performance results are gathered using an Apple Mac Pro 2009; 3.06 GHz 6-Core Intel Xeon, 256KB L2 per core, 12MB L3; 24 GB 1333 MHz DDR3. A script is used for the generation and reduction of the models and the verification of properties. This script, aiding in the gathering of (performance) results, is added in Appendix E.

3.3 Formal specification and verification

Like other methods, there are numerous ways of applying formal methods to get a formally specified and verified system. Let us, however, informally consider a few concrete steps that one can take.

The road to a formally specified and verified system starts with some informal documentation on the system of interest. Here we assume that the informal documentation contains sufficient information such that a formal specification can be produced from it. A different assumption is that there is a person bearing the knowledge that the informal documentation lacks. Then this person may act as an 'oracle' that answers all of our questions. Obviously, the system's purpose and responsibilities should be clear.

Using the informal documentation we identify the components that the systems interacts with and the interfaces that are used to do so. Then, we identify the architecture of the system itself, i.e. the components that the system is made up of, and determine the responsibilities for each component. Moreover, for each such component we determine its behaviour. This behaviour may be dependent on interactions from components outside of the system, interactions from components inside the system or it may not be dependent on any interactions at all.

A major benefit of formal specification in contrast to, e.g. programming, is that we can apply abstractions to the specification of the system and verify this abstracted notion of the system. Through abstraction we may be able to omit tedious or unnecessary details and ease the specification process.

Chapter 7 shows an in-depth example of how a formal specification is produced from the informal documentation.

Once we have a formal specification, the minimal requirement to make use of the tools that the mCRL2 toolset provides is met. As shown in Figure 3.1, the specification can now be transformed to an LPS, LTS or a minimized LTS. Moreover, the figure shows that with the specification we can simulate our model which allows us to 'step' through the behaviour of the model one action at a time. Another tool that is enabled by the specification, but is not shown in the figure, is the possibility to visualise the specification. The tools `ltsgraph` and `ltsview` show an LTS in 2D or 3D and have slightly different features and use cases.

However, with the specification alone we cannot perform any verification. As mentioned before and shown in the figure, verifying a system for some property requires us to describe this property using the μ-calculus requirement language. Properties generally describe (dis)allowed behaviour. Examples will be shown in the following Chapters.

Having both a formal specification and a property enables us to verify this property against the formal specification. The previous section and Figure 3.1 describe a few strategies that, depending on the specification, may improve performance of the verification process. The toolset will provide us with a true or false answer on whether or not the property holds in the specification together with evidence. This evidence is in the form of an LTS that (dis)proves the property. This evidence can be visualised with the aforementioned visualisation tools.

Whenever a system of interest is of considerable size, formally specifying and verifying this system can be a daunting task. To cope with this, the formal specification and verification of a system may be an iterative process where we specify and verify (an isolated) part of the system and built upon this verified piece of specification in the next iteration.

3.3.1 Basic modelling of Astris

Since we consider a select few of Astris' components in this thesis, it is important to properly lay the foundations and pay attention to how we formally specify Astris' components and how these interact and communicate with one another.

From Section 2.3 we know that a component is a single threaded application that executes its received service requests sequentially. Therefore, a component can be modelled as a single process in contrast to a component consisting of multiple concurrent processes. Moreover, we should consider a special type of process, i.e. 'the environment', that may act on behalf of entities that do not (yet) exist in the model. Since a component's task is to receive service requests, a component's process is equal to the sum of its service requests.

Section 2.3 also tells us that components interact by placing a message in the message queue of other components. The purpose of a queue is to resolve a timing issue where we avoid losing messages while a component is processing a service request. In the model we abstract from the notion of time and only consider the order of the occurrence of actions. So, instead of modelling a queue, the components can synchronise on the actions of sending and receiving a service request. Vice versa for the responses of the service requests. By modelling the component's communication in this manner, the queue is abstracted away, while we do not lose behaviour. By omitting the queue, we do not have to model something we are not interested in and avoid additional complexity of the model.

Chapter 4

Locking mechanism

As we have determined in Section 2.5, a good starting point for the analysis of Astris is the analysis of the Element Component's 'locking mechanism'. Most of the functionality to meet that end resides in the Element Component (EC). Thus, this chapter considers the EC, specifically, the EC's 'locking' mechanism. This mechanism, which will be further explained in the next section, helps realising the EC's objective to prevent race conditions on the commands sent to the railway elements. And thus, this mechanism helps realising the goal of preventing conflicting use of railway elements. The following sections will discuss the formal specification and some observations from the modelling process. Furthermore, we verify a claim, made by ProRail, on this 'locking' mechanism. Then, note will be taken of the performance of the mCRL2 toolset for the model and formulas of this chapter. Finally, the findings, results and observations presented in this chapter are discussed.

4.1 The locking of elements

Whenever a controlling system or application wants to set a route through the infrastructure, it must send commands via Astris to the elements (via interlockings) in said route such that the train will be guided from some origin to the desired destination. However, there can be multiple routes that want to make use of the same railway elements. Let us consider a simple case of two trains approaching a junction controlled by a point as depicted in Figure 4.1. Both trains wish to move forward (to the right), and thus, two applications send a command to set a route, i.e. a request to switch the point to the left and the right respectively. The point cannot do both, however, and thus, a conflict has occurred. The solution is to let the trains pass the point one after another. For this, only one command at a time may be accepted. As a consequence, always one of the two trains has to wait for the other to pass the point. So, there must be a mechanism that makes sure only one command at a time is accepted.

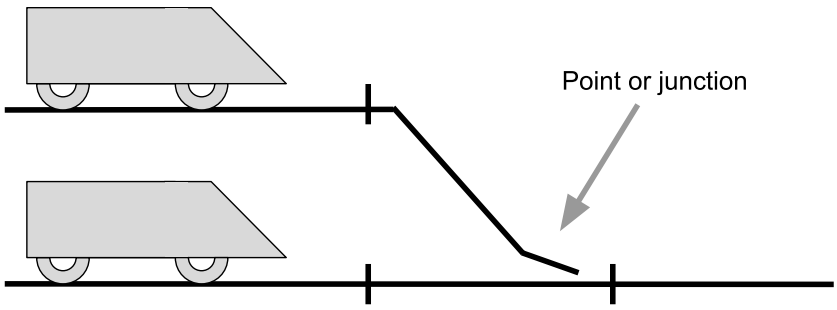


Figure 4.1: Two trains approaching a junction

The EC has a 'locking' or 'claiming' mechanism in place that allows applications to claim the individual elements in their route. Then, whenever other applications attempt to act upon the same elements, their requests get refused since the element has already been 'locked' by another application. To facilitate this mechanism, the EC can receive service requests to lock and unlock individual elements. It cannot, however, lock any arbitrary element. Each instance of an EC has a pre-determined finite set of elements it is responsible for. This set cannot be changed at runtime. These sets are known by the routing managers in the network such that messages pertaining to some railway element will be delivered to the correct EC that is responsible for this element. Each EC cannot (and will not) execute service requests for elements not in this set. Lastly, the sets of elements of all ECs are disjoint, an important fact that we could not find in the documentation of the EC.

ProRail has made a claim on this mechanism, which is as follows: "An element cannot be locked twice". So, we are going to verify that there does not exist a situation where whenever an element is locked, it is locked again without unlocking it first.

4.2 Specification

The specification of the EC makes use of the general approach to specifying Astris' components as discussed in Subsection 3.3.1.

The EC is modelled as a process that is able to accept either a locking or unlocking service request, represented in the model as `LockElement` and `UnlockElement` actions respectively. The informal specification lists quite a large number of service requests that the EC should support. However, since this chapter only considers the 'locking' mechanism, only the service requests pertaining to locking and unlocking elements are relevant and are thus the (for now) only modelled service requests of the EC.

In mCRL2, processes can carry data parameters. This allows a process to carry data associated with its functionality. In the case of the EC there are two relevant parameters: the set of elements the EC has currently locked and the set of elements the EC is responsible for. The latter is used by an EC instance to enable only the `LockElement` and `UnlockElement` actions that pertain to the elements in this set. Thus, an EC can only offer service requests for the elements it is responsible for. This realises the 'routing' functionality where a service request is always executed by the intended component.

We consider the parallel composition of a single EC and the environment. This is visualized in Figure 4.2. Since in this model the ECs do not communicate directly nor via the environment we need not consider more ECs in the model. Furthermore, the environment makes no assumptions on the order of how the EC(s) receive the service requests from the environment. The set of elements an EC is responsible for is a variable in the model. The amount should be kept low since it greatly influences the complexity of the model as will become evident when the performance of the toolset is discussed in Section 4.5.

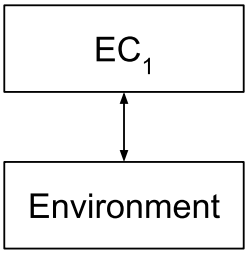


Figure 4.2: Locking model with EC environment and interactions

The informal specification[1] was not immediately clear on what the `LockElement` and `UnlockElement` service requests should do exactly.

Firstly, this is because the functionality is described in two different documents where the same idea is explained in different ways.

Secondly, the wording leaves room for interpretation. For example, for `LockElement` the description speaks of 'accepting' or 'the succeeding of' the service request. It is unclear if this refers to the process of locking an element or the execution the service request as a whole. Also, the vocabulary is not consistent. For example, 'service request' and 'service operation' are used interchangeably. Or, sometimes the specification speaks of 'placing a lock on an element' or 'locking an element'. The uninitiated do not know if this means the same or if it is actually something different.

Third, the specification is not as precise as it could be because it is not stated explicitly what the conditions should be for locking an element and when not locking an element. Though, it can be inferred from the text. Formally specifying this condition is one method for making this explicit. The same holds for what the response of the service requests should be. The `Respons` action will return with either `Accepted` or `NotAccepted`. The specification of `LockElement` says that if a lock is placed, the response should be `Accepted`. But it does not explicitly say that if no lock is placed the response should be `NotAccepted`.

By highlighting and discussing these points with ProRail, any indistinctness or ambiguities were made clear. The resulting formal specification of the service requests can be verified with ProRail such that we know it exactly reflects the desired behaviour. Obviously, the formal specification does not directly solve the above stated issues. Indirectly, it can help however, by using the formal specification to refine the informal specification. In the case that one finds the informal specification still insufficient, one can use the formal specification as a reference.

Inside the service requests, the `locked` and `unlocked` actions have been defined to denote the locking and unlocking of an element if the conditions for locking are met.

The mCRL2 specification can be found in Appendix A.

4.3 Deadlock freedom

A general property to check is if the model is free of undesired deadlocks. Here, a deadlock is defined as a state of the model with no outgoing transitions or actions. Note that this is different from other definitions where a system in deadlock "makes no 'meaningful' progression". This definition is actually stronger, but not relevant in this model since this would require to check that the model always makes progression to some

[1] [16], Section 3.6.12.53, p179; Section 3.6.12.8, p122; [5], Appendix D, Section 2.2 and 2.1

end-goal. However, in this model there is no such end-goal. And thus, it is not necessary to strengthen our notion of deadlock here. The notion that always an action can be taken is described in the following formula 4.1:

```
[true*]<true>true
```
(4.1)

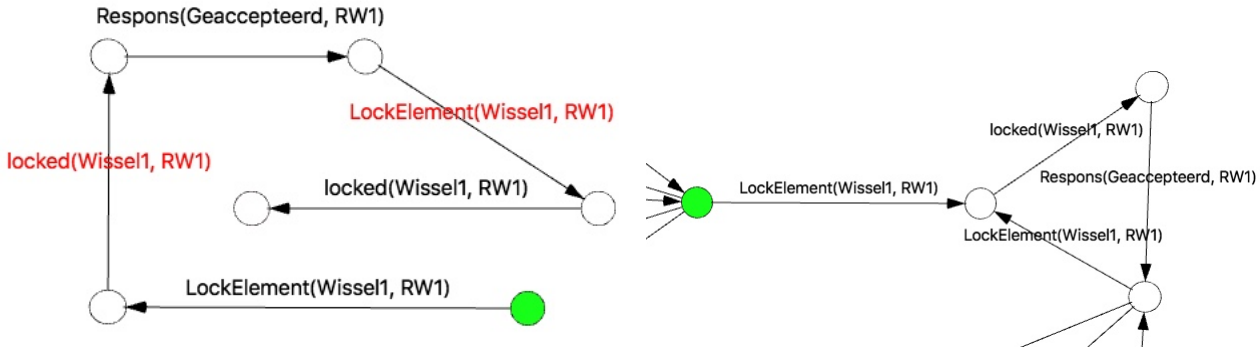
We verify this formula against the model and expect the mechanism to not contain any deadlocks. Thus, we expect the formula to evaluate to `true`. Application of the tools confirm this expectation, i.e. the formula evaluates to `true` in the model. Thus, we may conclude that the model is deadlock free.

4.4 Verification of the claim

The claim made by ProRail, as described in the above Section 4.1, asks the question if there is a situation where whenever an element is locked, it is locked again without unlocking it first. This is captured in the following formula 4.2:

```
forall e:ElementID . forall m,o,p:OpdrachtID .
!(<true*.locked(e,m).(!unlocked(e,o))*.locked(e,p)>true)
```
(4.2)

The formula evaluates to `false`. The generated counterexample, Figure 4.3(a), shows a path of two locking actions on the same element (highlighted in red) without that element being unlocked in between. Furthermore, by visual inspection of the state-space, Figure 4.3(b), we find this is actually a cycle. So, this behaviour can occur indefinitely. where the `OpdrachtID` is the same.



(a) Counterexample

(b) Inspection state-space

Figure 4.3: Analysis counterexample Formula 4.2

These findings are consistent with the informal specification which states that a lock may be placed if no lock yet exists on the element or if a lock already exists on the element and the associated `OpdrachtID` is equal to the one given in the service request. The described purpose of this is to handle situations where a lock is requested on an element before and after a crash. Then, no application can be refused to place a lock by a lock it had placed prior to crashing. So, the counterexample is spurious, and we should refine the formula to reflect more precision in the claim.

One could argue and raise the concern that in the case that `OpdrachtID` is a finite and reused identification, two applications can accidentally have the same `OpdrachtID` possibly leading to undesired behaviour. However, since `OpdrachtID` contains the date and time of when the command was issued, the identification is always unique and thus refutes the concern.

So, we can refine the claim to take the above findings into consideration. That is, if there exists a path where an element is locked twice without being unlocked in between. Then, the `OpdrachtID` associated with the `locked` actions must be the same. This is captured in the following formula 4.3:

```
forall e:ElementID . forall n,m:OpdrachtID .
((<true*.locked(e,n).  ((forall o:OpdrachtID . !unlocked(e,o)))*.
locked(e,m)>true) => val(n==m))
```
(4.3)

The formula evaluates to `true`. Therefore, we may conclude that the claim "An element cannot be locked twice unless the `OpdrachtID` is the same" holds.

4.5 Tool performance

In Table 4.1 the number of states and transitions for different configurations of the model are presented. The different configurations are determined by the amount of elements the EC is responsible for. We consider only a few configurations to get an impression of the effects of the parameters on the state-space. Additionally, we consider the reduced state-space under strong bisimulation. This means that the non-reduced state-space and the reduced state-space must be strongly bisimilar. Strong bisimulation is a strict notion of behavioral equivalence.

As the amount of elements increases, the size of the model increases as well. It is remarkable to see that despite the strictness of strong bisimulation, the reduction is considerable. So, many states that differ in some data parameter exhibit the same behaviour and can thus be merged. By visual inspection of the reduced and the non-reduced state-space we can see that this is mainly due to the transitions and associated states for the `UnlockElement` service requests. Since the unlocking of elements can happen irrespective of the `OpdrachtID`, states that are normally distinct because of the differing `OpdrachtID` are thus merged. Thus, we could argue that in the case of the `UnlockElement` service request, the `OpdrachtID` has no use in this model and can therefore be omitted. However, this parameter is relevant for the response of the service request. The response must be received by the correct callee. In the models of the next chapters, this is realised by reusing the `OpdrachtID`. Else a response can be received by the wrong component that is coincidentally awaiting a response at that time.

	Not reduced		Reduced	
#Elements EC	**#states**	**#transitions**	**#states**	**#transitions**
1	25	34	15	24
2	141	204	69	132
3	621	918	263	560
4	2457	3672	933	2148

Table 4.1: States and transitions of locking model

Since the model is fairly small it takes about a second for the toolset to generate the state-space of the largest model of Table 4.1. There is no noticeable difference in execution time between generating the state-space with or without reduction.

Table 4.2 contains the execution times of the toolset to verify the formulas against the different configurations of the model. Again, we consider only a few configurations to get an impression of the effects of the parameters on the execution time. The shown execution time is the sum of the execution times of generating the Parameterised Boolean Equation System (PBES) and the subsequent solving of this system.

#Elements per EC	4.1	4.2	4.3
1	80	100	84
2	94	146	114
3	145	238	239
4	361	539	738

Table 4.2: Execution times (in milliseconds) of verifying locking model

We can observe that the execution time scales similarly to the amount of states in the model as we have seen in Table 4.1.

4.6 Discussion

The result is that thanks to the model and verification we were able to conclude that the claim was not precise enough. Moreover, the issue was clearly identified and visualised by the counter example produced by the toolset. As a consequence the claim could be refined and the refined claim was proven correct in the model.

The response to sharing these results with ProRail was predominantly positive. It is a perfect example of showing what formal verification can do. The small context and simple claim made it easy to follow and understand. Even for the colleagues that are not necessarily familiar with the exact inner workings of Astris. It is made clear how imprecisions can be discovered by asking the formal specification a question through formal verification. But also how a specification can be made more precise by formal specification. It is understood how this can be generalised to more complicated questions.

We raised a concern however, that if the lock was implemented like a semaphore, that the element can potentially be locked indefinitely in the event that strictly more locks than unlocks occur. In response, ProRail pointed to the data structure for elements and the attribute holding the lock[2]. A lock is an attribute holding the `OpdrachtID` and is thus not implemented like a semaphore. As such, the concern can be disregarded.

Apart from the exact content of the service requests, the documentation was not very clear on what 'locking' meant. From the text anyone familiar with concurrency in software would think about the notion of mutual exclusion realised by e.g. a semaphore or mutex. This imprecision in the text led to ambiguity in the meaning and purpose. A better description of this mechanism would be to 'claim' an element.

[2][16], Section 3.5.2.2.1

Chapter 5

Route conflict resolution

This chapter adds to the model and results of the previous Chapter 4. That is, in this chapter we are going to take the Route Component, introduced in Section 2.4.2, into consideration. Specifically, we consider the part of the RC that interacts with an EC and makes use of the service requests for the locking and unlocking of elements. This part, pertaining to the setting of routes throughout the infrastructure, also contains a method for resolving conflicts where two RCs attempt to lock the same element as will be explained in the next section. The following sections will discuss the formal specification and the verification of the conflict resolution method. Then, the performance of the mCRL2 toolset for the model and formulas of this chapter are noted. Finally, the findings, results and observations in this chapter are discussed.

5.1 Setting a route

In the Dutch railways, a route is a logical part of the infrastructure starting from some 'begin signal' up to the 'end signal', i.e. the begin signal of some other route. Though, the 'end signal' can also be fictional to, for example, represent a buffer stop in case of the track ending. From one begin signal, there may be multiple routes leading to different end signals. Figure 5.1 shows an example where there are two possible routes from the begin signal to signal A and signal B. A Route Component is then responsible for all routes associated with one begin signal. An RC then controls the elements in these routes visualised in the route area.

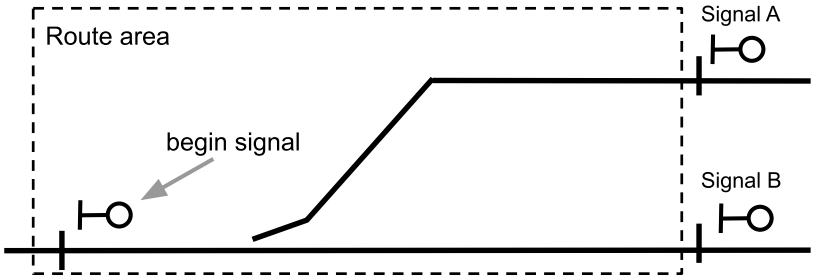


Figure 5.1: Example of a route

The above figure shows an infrastructure configuration where the route area does not overlap with other route areas. An example of a configuration where route areas would overlap is shown in Figure 5.2. In this figure we can observe two begin signals, signal A and signal C. From signal A there is a route to signal B and from signal C there is a route to signal D. These routes share a point. However, the routes fall under

the responsibility of different RCs. Thus, problems could arise when both RCs want to set the same point simultaneously to the left or right.

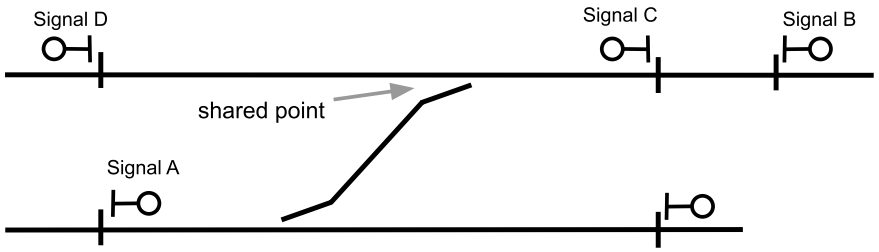


Figure 5.2: Example of two routes sharing a point

Whenever an RC wants to set a route, it must reserve all of the individual elements in the route before sending commands to the infrastructure. These 'usage markings' prevent that two RCs send conflicting commands to the infrastructure for the same element. If two routes sharing elements reserve elements that the other route requires, then both RCs will be unable to reserve all of the elements in the route and thus both RCs will fail in setting their route.

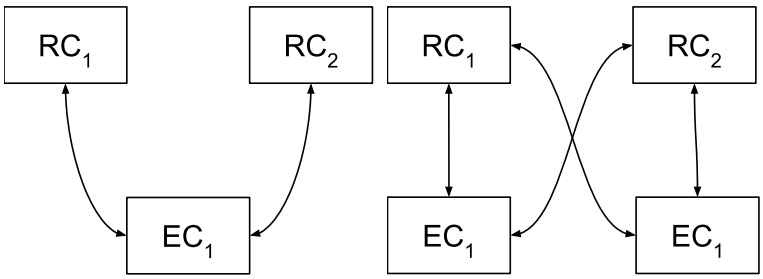
This is where the locking mechanism shows its purpose. An RC has to lock all the elements in the route before an RC can continue to actually reserve the elements in the route and send commands to the infrastructure. In such a case, the RC is successful in setting the route. If an RC is unable to lock any one element, then it must unlock all previously locked elements and retry locking all the elements in the route after a set amount of time. The idea is that an element is always temporarily locked and it could be unlocked when an RC tries to lock a element in the next attempt. An RC can retry a finite amount of times. If an RC has exhausted all of its attempts, it reports back that it has failed.

A functionality of the EC that was omitted in the previous chapter that makes sure an element is not locked indefinitely. In the case that an RC crashes, the elements locked by that RC might stay locked forever since the crashed RC might not unlock it. The EC will automatically unlock an element after a set amount of time. This is known as the `removeLockTimeout`.

5.2 Specification

For the specification of the Route Component we take a similar approach as for the Element Component described in Section 4.2. That is, the RC is modelled as a single process that is able to receive and execute service requests. In the RC, the locking of elements in the route is a sub-routine that occurs at the start of every service request. Since this is the only part of the RC we are modelling, the process of the RC consists solely, and is therefore equal to, this sub-routine.

Since we want to verify the correctness of the conflict resolution method we require a model where circumstances for such conflicts may occur. In the previous section we observed how such situations can occur with at least two routes. In our model we will therefore consider the minimal case of two RCs. A few variations are considered in the model. That is, with one EC or two ECs in parallel composition with the two RCs, the amount of elements in the route of the RCs, the amount of attempts the RCs have and possibly having the `removeLockTimeout` enabled or not. This is visualised in Figure 5.3. Having these configurations allows us to start simple and add complexity in a step-wise manner such that we can easily limit the scope to help interpret results. This helps with the modelling and verification process.



(a) One EC (b) Two ECs

Figure 5.3: Route conflict resolution model with EC(s) and interactions

The documentation on the Route Components is clear on what this sub-routine of locking elements in the route should do. The required information is present in only one document and in only one section. Initially, there were no inconsistencies or ambiguities in the requirements describing the behaviour. However, during the modelling process one small issue came up. The documentation did not impose any order in which the elements in the route should be locked. ProRail said that it is in the order of first the 'begin' signal and then the rest of the elements in order of the direction of travel of the route. While the documentation on the infrastructural configuration data structures[1] does mention this, it is nowhere stated that this order should be respected when locking elements in the route. Luckily, this is of no consequence for the model since the verification will not depend on the order.

The 'locking of elements in the route' is modelled as a process accepting an `OpdrachtID` to distinguish the two RCs. So, every RC in the model has a different `OpdrachtID`. The rest of the process attempts to lock an arbitrary list of elements (denoting the elements in the route) one after another. To be able to verify this mechanism we make a distinction between whether or not the mechanism was successful in locking all the elements in the route or not. We describe this as `success` and `fail` actions in the model. Moreover, whenever both RCs have ended in either success or failure, the model will end with (and deadlocks after) a `sync` action. Thus, deadlocks may only occur after a `sync` action has happened.

The notion of `removeLockTimeout` is modelled in the EC as a non-deterministic step that can be taken when the EC is not processing a service request. It can be interpreted as a service request of an EC to itself. This action will unlock one of the locked elements. This approach is somewhat naive since a non-deterministic step is not constrained by time as is the case in reality. Thus, `removeLockTimeout` can happen directly after locking an element or it may never happen at all. This is an important notion to keep in mind when interpreting results of the model and relating this to reality.

The mCRL2 specification can be found in Appendix B.

5.3 Freedom of deadlock

First, we want to verify that the model does not contain any deadlocks. As we have explained in the previous section, the model ends in a `sync` action. And thus, we want to verify that as long as the `sync` action has not yet occurred, the model should not deadlock. This is captured in the formula 5.1:

```
[!sync*]<true>true
```
(5.1)

The formula evaluates to true. So, we may conclude that there does not exist a situation in which the model, and therefore the mechanism, can end in deadlock.

[1] [4], Section "Alle elementen in en bij de rijweg", accompanying text for the `ElementInRijwegIdentificatie` and `ElementBijRijwegIdentificatie` attributes

5.4 Verification of the route conflict resolution method

A general property we want to verify is that each RC ends in either `success` or `failure` once. So, in our model with two RCs, we want to verify that the number of occurrences of `success` or `fail` is always two. Moreover, every one occurrence of either `success` or `failure` must be related to only one RC. That is, it may not be the case that one RC produces two `success` and/or `fail` actions and the other does not. This is captured in formula 5.2. Here, we pay special attention to the `success`, `fail` and `sync` actions. In the third line we cover all except the previously mentioned actions. This is not immediately obvious because of the `exists` which expands the disjunction of actions over the domain of `OpdrachtID`.

```
mu X(s:Nat = 0, f:Nat = 0, os:OpdrachtIDList = []) .
((forall o:OpdrachtID . [success(o)]X(s+1, f, os++[o]) && [fail(o)]X(s,f+1, os++[o])) &&
[!(exists o:OpdrachtID . success(o) || fail(o) || sync)]X(s,f, os) &&
<true>true && [sync]val(s+f == 2 && #os == 2 && os.0 != os.1))
```

(5.2)

The toolset evaluates the formula to `true`. Therefore, we may conclude that every RC in the model always terminates in either `success` or `fail` once.

Next, we can verify the overall correctness of the route conflict resolution method. Ideally the conflict resolution method should make sure that:

- If the routes of both RCs do not share any elements, then there is no conflict and both RCs must be able to lock all the elements in the route. Thus, both RCs end in success.
- If the routes of both RCs share at least one element, then there is a conflict and only one of the two RCs must be able to lock all the elements in the route and the other not. Thus, one RC ends in success and the other in failure.

The above is captured in the Formula 5.3. Similarly to formula 5.2 we keep track of the occurrences of `success` and `fail`. In this formula we pay attention to the `LockElement` actions to determine the list of elements in the route of both RCs identified by `o1` and `o2`. In the sixth and seventh lines we cover all except the previously mentioned actions. Then, after the `sync` action occurs we verify that either the routes do not share any elements and both RCs end in `success` or the routes share one or more elements and one RC has ended in `success` and the other in `fail`.

```
forall o1, o2 :  OpdrachtID . val(o1 != o2) =>
mu X(s:Nat = 0, f:Nat = 0, a:ElementList = [], b:ElementList = []) .
(forall e :  ElementID . ([LockElement(e, o1)]X(s, f, a ++ [e], b) &&
[LockElement(e, o2)]X(s, f, a, b ++ [e]) &&
(forall o :  OpdrachtID . [success(o)]X(s+1, f, a, b) && [fail(o)]X(s,f+1, a, b)) &&
[!(exists o :  OpdrachtID . exists ep :  ElementID . LockElement(ep,o) ||
success(o) || fail(o) || sync)]X(s, f, a, b) && <true>true &&
[sync]val( (s==2 && f==0 && listIntersectSize(a,b)==0) ||
(s==1 && f==1 && listIntersectSize(a,b)>=1))))
```

(5.3)

If we assume the above described ideal behaviour of the route conflict resolution method to be correct, then we expect the formula to evaluate to `true`. And it does in the case where we consider routes consisting of a single element. However, if the routes have more than one element, then the formula evaluates to `false`.

Upon inspection of the counterexample generated by the toolset we can observe an example of a situation where both RCs end in failure, i.e. both RCs end with the `fail` action. Let us take a closer look at the trace of this counterexample. Since the generated counter example is quite a bit larger than in Figure 4.3(a) and there are more components involved, the counterexample is visualised in a more comprehensive fashion in Figure 5.4. Additionally, some comments on each step are provided.

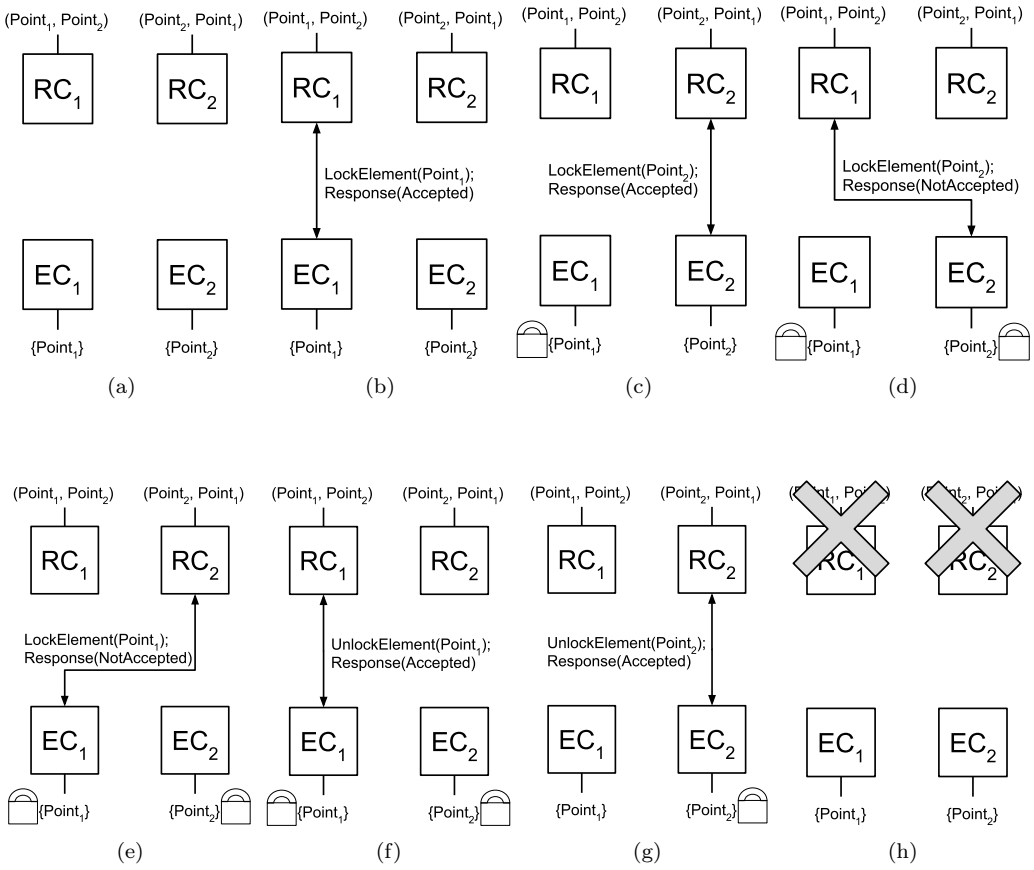


Figure 5.4: Visualisation of the counterexample showing both RCs failing

(a) The initial situation of two RCs and two ECs. EC_1 is responsible for the element $Point_1$. Likewise for EC_2 and $Point_2$. The route of RC_1 consist of a list of elements ($Point_1$, $Point_2$). Likewise for RC_2 and ($Point_2$, $Point_1$). Note how the routes of the RC share two (both) elements and the lists are in opposite order. Both routes have only one attempt

(b) RC_1 sends a `LockElement` service request to EC_1 to lock element $Point_1$. The response is `Accepted`. Thus, $Point_1$ is locked by RC_1

(c) The lock on $Point_1$ is visualised. RC_2 sends a `LockElement` service request to EC_2 to lock element $Point_2$. The response is `Accepted`. Thus, $Point_2$ is locked by RC_2

(d) The lock on $Point_2$ is visualised. RC_1 sends a `LockElement` service request to EC_2 to lock element $Point_2$. The response is `NotAccepted`. Thus, $Point_2$ is *not* locked by RC_1

(e) RC_2 sends a `LockElement` service request to EC_1 to lock element $Point_1$. The response is `NotAccepted`. Thus, $Point_1$ is *not* locked by RC_2

(f) Since RC_1 is unable to lock all the elements in the route, it will unlock the previously locked elements, i.e. RC_1 sends a `UnlockElement` service request to EC_1 to unlock element $Point_1$. The response is `Accepted`. Thus, element $Point_1$ is henceforth no longer locked

(g) The absence of a lock on $Point_1$ is visualised. Since RC_2 is unable to lock all the elements in the route, it will unlock the previously locked elements, i.e. RC_2 sends a `UnlockElement` service request to EC_2 to unlock element $Point_2$. The response is `Accepted`. Thus, element $Point_2$ is henceforth no longer locked

(h) The absence of a lock on $Point_1$ is visualised. Both RCs have exhausted all of their one attempt. And thus, both end in failure

A first observation is that this is only possible if the two routes share at least two elements. Moreover, it allows us to derive a infrastructural configuration from the counterexample where these types of problems may occur. Figure 5.5 shows such an abstract representation of a configuration. The two arrows represent two routes that intersect at two places. The direction of the arrow determines the order in which the route locks the elements in the route. At the intersections are the two elements that the routes share.

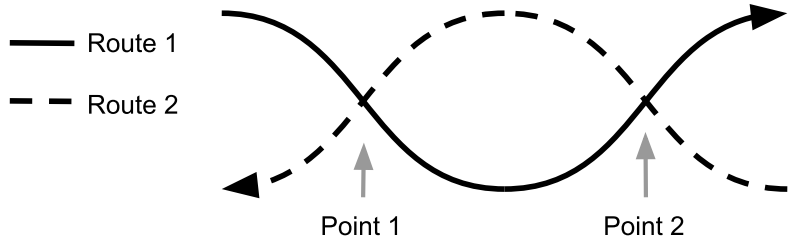


Figure 5.5: Diamond pattern configuration of two routes sharing two elements

The visualisation of Figure 5.4 clearly shows that the failure of both RCs depends on the order of the steps, i.e. first steps (b) and (c), then (d) and (e), finally (f), (g) and (h). Within this ordering of steps, no matter the order of the actions, it yields the same result.

Now, one might recall that an RC may have multiple attempts. Perhaps the solution to the above lies in having more than one attempt. Verifying the model where we allow the RCs to have two attempts yields the same result. From the counterexample we can observe that it is possible to 'bypass' an attempt and then in the next attempt go through the same scenario as discussed before that will result in the failure of both RCs. Moreover, since we now know it is possible to bypass one attempt, we know that the scenario can be repeated for any number of attempts. And therefore, we do not need to verify the model with more attempts. One can argue though, that with every extra attempt, the chances of this exact sequence of actions happening is further reduced. However, the possibility remains. By reusing the steps of Figure 5.4 the steps of the counterexample can be demonstrated by the following trace: (a)(b)(c)(d)(f)(b)(d)(f)(e)(g)(c)(e)(g)(h).

The consequence of both RCs failing to lock the elements in their route is that both RCs will report back that they failed. Let us assume that the system that the RCs report back to issues a new command to the RCs such that this time not both RCs fail. Then, the above described entire process where both RCs both end in failure has taken up time because of the timeouts and routing of messages. As a consequence, the desired end-result of moving trains over the infrastructure may become delayed. This is undesirable. Delays will be discussed more elaborately in Subsection 5.6.2.

Having found a problem with the route conflict resolution method, we still want to know if there are any more problems. So, we adapt Formula 5.3 to ignore and allow the above described issue. Concretely, in

the case where the routes of the RCs share two or more elements, it is allowed that both RCs fail. This is described in Formula 5.4. This formula is equal to Formula 5.3 but with the adaption of the second last line where we no longer consider the sharing of one or more elements, but exactly one element. And the addition of the last line which states that that if the routes share two or more elements, then either both RCs fail or only one fails and the other succeeds.

```
forall o1, o2 :  OpdrachtID . val(o1 != o2) =>
mu X(s:Nat = 0, f:Nat = 0, a:ElementList = [], b:ElementList = []) .
(forall e :  ElementID . ([LockElement(e, o1)]X(s, f, a ++ [e], b) &&
[LockElement(e, o2)]X(s, f, a, b ++ [e]) &&
(forall o :  OpdrachtID . [success(o)]X(s+1, f, a, b) && [fail(o)]X(s,f+1, a, b)) &&
[!(exists o :  OpdrachtID . exists ep :  ElementID . LockElement(ep,o) ||
success(o) || fail(o) || sync)]X(s, f, a, b) && <true>true &&
[sync]val( (s==2 && f==0 && listIntersectSize(a,b)==0) ||
(s==1 && f==1 && listIntersectSize(a,b)==1) ||
(((s==1 && f==1) || (s==0 && f==2)) && listIntersectSize(a,b)>=2))))
```

(5.4)

This formula evaluates to `true` in the absence of the notion of `removeLockTimeout` in the model. With `removeLockTimeout` Formula 5.4 evaluates to `false`.

Inspection of the counterexample shows an example of a situation where both RCs are able to lock all the elements in their route, i.e. both RCs end with the `success` action. Let us take a closer look at the, by the toolset generated, counterexample visualised in Figure 5.6.

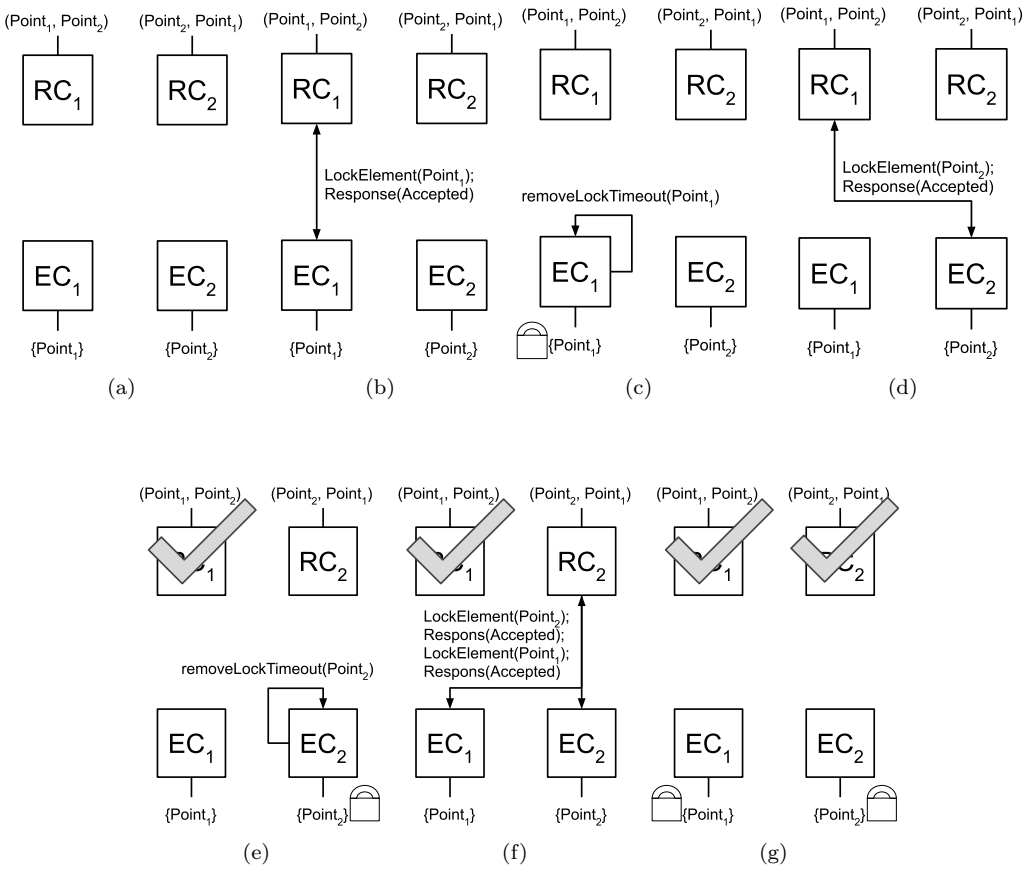


Figure 5.6: Visualisation of the counterexample showing both RCs yielding success

In this visualisation we can observe how steps (a) and (b) are the same as in Figure 5.4. However, in the subsequent steps we can observe how `removeLockTimeout` unlocks the elements that where previously locked by RC_1. The problem is that RC_1 does not have any knowledge of the occurrence of this event. It does not, for example, get notified. So, RC_1 continues under the assumption it has locked all the elements in the route while this is evidently not the case. And thus, RC_1 wrongfully concludes `success`. Afterwards, RC_2 locks the by `removeLockTimeout` unlocked elements in (f) and rightfully claims `success` in (g).

Since both RCs think they are successful in obtaining all the locks, they will both attempt to reserve the elements in the route. However, this will fail since the reservation safeguards the sending of conflicting commands to the infrastructure as we had established in Section 5.1. However, this may have delays as a consequence which is undesirable.

We adapt our formula once again such that it ignores and allows the above discussed issue as shown in

Formula 5.5 which is equal to Formula 5.4 except for a refinement of the last two lines.

```
forall o1, o2:OpdrachtID . val(o1 != o2) =>
mu X(s:Nat = 0, f:Nat = 0, a:ElementList = [], b:ElementList = []) .
(forall e :  ElementID . ([LockElement(e, o1)]X(s, f, a ++ [e], b) &&
[LockElement(e, o2)]X(s, f, a, b ++ [e]) &&
(forall o :  OpdrachtID . [success(o)]X(s+1, f, a, b) && [fail(o)]X(s,f+1, a, b)) &&
[!(exists o :  OpdrachtID . exists ep :  ElementID . LockElement(ep,o) ||
success(o) || fail(o) || sync)]X(s, f, a, b) && <true>true &&
[sync]val( (s==2 && f==0 && listIntersectSize(a,b)==0) ||
(((s==1 && f==1) || (s==2&&f==0)) && listIntersectSize(a,b)==1) ||
(((s==1 && f==1) || (s==0 && f==2) || (s==2 && f==0)) && listIntersectSize(a,b)>=2))))
```
(5.5)

This formula evaluates to `true` for all configurations of the model.

Table 5.1 summarises the verification results of this chapter. The table shows only the parameters of the configurations that lead to changes in the result. The number of ECs and number of attempts did not influence the result and are therefore omitted from the table.

#Elements in the route	`removeLockTimeout` **enabled**	5.1	5.2	5.3	5.4	5.5
1		`true`	`true`	`true`	`true`	`true`
1	✓	`true`	`true`	`false`	`false`	`true`
2		`true`	`true`	`false`	`true`	`true`
2	✓	`true`	`true`	`false`	`false`	`true`

Table 5.1: Summary of verification results Chapter 5

5.5 Tool performance

In Table 5.2 the number of states and transitions for different configurations of the model are presented. Since this model has a lot of configurations we limit the amount by setting the number of ECs to two and the amount of attempts for the RCs to one. Specifically these since they do not affect the results. Finally, again, we consider the reduced state-space under strong bisimulation.

Since the execution time of generating the state-space is no longer always within a single second, this has been added in the table as well as the time it takes to reduce this generated state-space. These are described in the **Gen.** and **Red.** columns respectively.

#Elements in the route	`removeLockTimeout` **enabled**	Not reduced **#states**	**#trans.**	**Gen.**	Reduced **#states**	**#trans.**	**Red.**
1		79	128	.24	49	89	.004
1	✓	120	264	.25	101	216	.004
1-2		387	694	.25	241	453	.05
1-2	✓	960	2222	.28	566	1304	.07
1-3		63685	124102	2.5	19777	41659	.85
1-3	✓	444864	1093376	21.2	106008	281826	8.17

Table 5.2: Generation and reduction times (in seconds) and number of states and transitions of route conflict resolution model

A first observation is that it is clear that the size of the model and the execution times make a tremendous jump when we start considering one to three elements in the routes of the RCs. This is because, in the model, the amount of possible routes the model considers grows considerably.

A second observation is that the non-determinism of the enabling of the `removeLockTimeout` makes the size of the model increase.

A last observation is that the reduction under strong bisimulation is still very effective. By visual inspection of the state-spaces, we find this is largely due to the following reasons:

- The `Respons` action in the model can be ambiguously used for many service requests. This allows states to be merged.
- In addition, in the last actions of the process are always somewhat the same, i.e. one of the RCs received the last response and determines either failure or success. A number of these states can be merged.

Table 5.3 contains the execution times of the toolset to verify the formulas against the above considered configuration of the model. The shown execution time is the sum of the execution times of generating the Parameterised Boolean Equation System (PBES) and the subsequent solving of this system.

#**Elements in the route**	`removeLockTimeout` **enabled**	5.1	5.2	5.3	5.4	5.5
1		.065	.071	.083	.082	.083
1	✓	.079	.089	.113	.112	.113
1-2		.111	.125	.185	.185	.189
1-2	✓	.216	.266	.408	.410	.505
1-3		8.52	9.45	21.45	21.74	21.60
1-3	✓	61.30	67.33	114.42	97.846	264.53

Table 5.3: Execution times (in seconds) of verifying locking model

We can see that usually the performance for 5.3, 5.4 and 5.5 are similar when considering the model without `removeLockTimeout` enabled. By inspection of the formulas we can conjecture that this is because the fundamental formula does not change, but every formula adds a bit to the logical constraints after the `sync` action (i.e. after termination). With `removeLockTimeout` enabled, the execution times for these formulas differ more. Especially in the cases of '1-2' and '1-3'. It is not clear what causes this difference exactly.

By inspection of the timing information provided by mCRL2 we can observe that about a third of the time is spent on the generation of the PBES and then about two thirds on the instantiation and relatively very little on the actual solving of the PBES.

Lastly, the execution times when `removeLockTimeout` is enabled are always higher than whenever it is not simply because the state-space is much larger when it is enabled. This is consistent with the effect of enabling `removeLockTimeout` on the size of the state-space as described in Table 5.2.

5.6 Discussion

The result is that two problems have been identified thanks to the model and verification. The counterexamples that the toolset provided made the exact issues became more clear.

5.6.1 Sharing of results

The response from sharing these results with ProRail was again positive. However, it did not seem to come as a surprise. Apparently, ProRail is aware that the route conflict resolution method is not fail proof. ProRail

states that the purpose is 'mitigation' of problems and not absolute avoidance. However, this notion is not stated explicitly in the documentation.

ProRail states that the configuration where multiple RCs interact with the same EC(s) does not currently exist in the infrastructure. As a consequence, the in this Chapter found problems do not occur in reality.

5.6.2 Notes on delays

When discussing the problem where both RCs end in `fail`, we where quick to conclude that this may have delays for the trains as a result. Let us discuss and refine this.

Obviously, delays can occur to some degree without consequences. One can argue then, that as long as the delay this causes stays below some threshold, then it has no consequences and the occurrence of this problem is acceptable. If a delay exceeds this threshold, then this can cause delays for trains in the infrastructure, which is undesirable. For this, mostly the sum of the duration of the timeouts of each attempt must be below this threshold. The documentation states that the default values are three attempts with a timeout of 500 milliseconds each. Thus, the minimum time before failure of an RC and therefore the minimum length of the delay would then be 1.5 seconds.

However, as mentioned before, this situation does not occur in reality and thus this is not an issue.

5.6.3 Notes on `removeLockTimeout`

One might have noticed that due to the `removeLockTimeout` an RC has to lock all of the elements in the route and do its processing before the `removeLockTimeout` on any element expires. Else, the unlocked element might be wrongfully locked again for a different purpose. We can reason that if the time between the locking and unlocking of an element by an RC is longer than the timeout, `removeLockTimeout` will remove the lock. As mentioned before, this is not a problem because of the notion of 'usage markings' and reservations.

Nevertheless, it is an interesting observation where we can ponder that the timeout of `removeLockTimeout` must be of appropriate length. That is, one must strike a balance where the duration cannot be too short, else the locking mechanism has no purpose, and not too long since it would increase delays. ProRail states that in reality, a `removeLockTimeout` rarely occurs. Thus, there is reason to believe that the default duration of 500 milliseconds is reasonable.

5.6.4 Suggested solutions

Lastly, potential solutions are offered for both of the problems discussed in this chapter.

In the case of the problem where both RCs end in `fail`, the goal is obviously to adapt the conflict resolution method such that always one of the two RC will be able to lock all the elements in the route whenever a conflict occurs. A simple solution for this is to assign the routes a priority number. Intuitively, problems will then occur in the case that two adjacent routes have the same priority number. We might be able to solve this issue by interpreting the routes as a 'four colour problem' where the colours represent the priority number. Then, by the 'four colour theorem', there will never be any adjacent routes with equal priority numbers.

In the case of the problem that because of `removeLockTimeout`, both RCs end in `success`, we know that the notion of `removeLockTimeout` adds non-determinism to the model. The simplest solution would be to remove `removeLockTimeout` altogether. But we can't since it has a clearly important purpose. Mark Bouwman contributed a simple solution to let the RC also run a timeout in parallel with the `removeLockTimeout` such that the RC knows when a lock has expired and can potentially act on this knowledge.

Chapter 6

Coupled elements

This chapter expands on the notion of railway elements by introducing the concept of 'coupled elements' and investigates concerns with how Astris handles these coupled elements in the scope of the RC and EC. In the next section the coupled elements and the concerns will be explained. The following sections will discuss the formal specification and the verification of the raised concerns. The performance of the mCRL2 toolset will be discussed before finally going into the findings, results and observation of this chapter.

6.1 Concerns for deadlock

There exist railway elements that are logically multiple elements, but physically one. There exist configurations where there are physically two points controlled as if it were only one actuator. Thus, whenever one point is actuated the other point will also be actuated since they are 'coupled'. An example of such a configuration is called an *overloopwissel* in Dutch and allows trains to go from the left track to the right track and vice versa. An example is shown in Figure 6.1. In the past, the control for multiple actuators where merged for cost savings purposes.

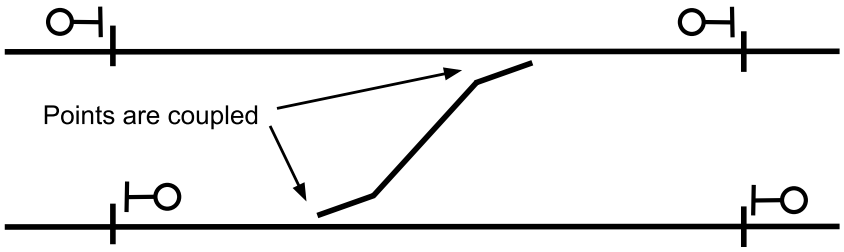


Figure 6.1: Example configuration with coupled elements

In the figure we can see by the signals that on the top track trains travel from left to right and vice versa for the bottom track. One can imagine how actuating the points at the wrong time can have catastrophic consequences. But, there are also times where the points can be actuated without any consequences. To help identify these types of conflicts, the notion of 'usage markings' exist. An element can be marked as 'in use' accompanied with the type of use. Some combinations of types of use are permissible and others are not. For example, if an element is in use for construction and maintenance as part of a work zone, no other use is allowed except for cancellation of the work zone. The checking for conflicts is not of interest in this chapter, but how this process deals with coupled elements is.

Let us consider the above example and assume the responsibility for the points are divided over two ECs. If a command is issued to move one of the points, then we should check if the other point can also move

without potential conflict. However, it is not immediately clear how interactions from one EC to the other EC with the coupled element will behave. ProRail suspected (but did not prove) that this could lead to deadlocks. Specifically in the process of checking 'usage markings' of elements for conflicts. Therefore, a configuration requirement was added that all coupled elements and the elements coupled to these elements should fall under the responsibility of a single EC. Thereby avoiding potential deadlocks.

Thus, in this chapter, we will consider the effects of the addition of coupled elements in the process of checking and marking of usage of elements. By means of verification we can investigate if taking the precaution of the additional configuration requirement is justified.

Additionally we will consider the consequence of coupled elements for the locking mechanism of Chapter 4 since most of the components for this are already available in the model and it will not require much more effort on our part. But also because it allows us to verify ProRail's claim that there should not be any (new) problems with the locking of coupled elements.

6.2 Specification

Similarly to Chapter 5, we take two sub-process of a service request of the RC under consideration. These are discussed in the following Subsection 6.2.1 and 6.2.2 separately.

The notion of coupled elements is interpreted as a relation between two elements. Here, an element cannot be coupled with itself. If element a is coupled with element b, then b is coupled with a. If element a is coupled with element b and b is coupled with c, then a is coupled with c. Each EC has knowledge of the coupled elements for all of the elements it is responsible for. This is modelled in the EC as a process parameter of a list of element pairs. These pairs then denote the coupled elements and obey the stated properties. In the model, elements are arbitrarily coupled.

The mCRL2 specification can be found in Appendix C.

6.2.1 Coupled elements and locking

As we have learned from Chapter 5, the RC must lock all the elements in the route. What was omitted, is that this also includes the coupled elements.

First, an RC must be able to determine the coupled elements for all of the elements in its route. According to the documentation, for this, the EC's `giveCoupledElements` service request is used[1]. Therefore, the `giveCoupledElements` service request is to be added to the repertoire of the EC. The service request determines the coupled elements for some given element and returns the list of coupled elements in the response.

The documentation is not clear on the order in which the elements and coupled elements should be locked. The informal specification simply states that "all the elements in the route, by the route and all their coupled elements"[2] should be locked. The documentation does not clearly define what elements 'by the route' are. It is also not clear if with 'coupled elements' the notion of coupled elements as we consider in this chapter are meant or some other notion of 'connected elements' since in Dutch the used word *gekoppeld* can mean either. This ambiguity is cleared up by verifying with ProRail and recalling the exact definition of 'coupled elements' in the documentation[3] which refers to the notion of coupled elements as we consider in this chapter. This definition was not in the same document however.

Given a list of elements in the route, the process will first build a new list of elements including the coupled elements. This is done by issuing a `giveCoupledElements` service request for every element in the given list and add it to the list of elements that are to be locked. Then, the process will attempt to lock the elements in this newly built list like in Chapter 5. As such, we are able to reuse some parts of the specification of the model of Chapter 5.

[1] [16], Section 3.7.3.1.3, p190-191
[2] [17], RMC-FUNC-004, p81
[3] [16], Section 3.7.2.3, p188; [4], Section "Statisch Domeinmodel gekoppelde elemnten", p17

Since for this chapter we are only interested in how Astris deals with coupled elements split over two ECs we will consider the minimal case of two RCs and two ECs where both ECs are responsible over only one element. This is visualised in Figure 5.3(b) of Chapter 5.

6.2.2 Coupled elements and usage marking

After an RC has locked the elements (and the elements' coupled elements) in the route, it must check and mark the usage of the locked elements to check for conflicts.

By issuing the ECs `takeElementIntoUse` service request an RC can check and mark usage for some element[4]. Then, the EC will check the elements and the coupled elements for conflicts and marks them 'in use'. Hence the name of the service request. Inspecting the description of this service request we can see that the control flow is described by means of a table. Table 6.1 shows a translated version of this table.

Activity		**Paragraph**	**Exception**
Determine coupled elements		3.6.12.7	Exception 1
Check (A) if use with same `OpdrachtID` is already registered		3.6.12.39	
Check (B) for present VHB, also for coupled elements		3.6.12.37	
Check (C) for present VHR, not also for coupled elements		3.6.12.38	
Alternative 1	(no further checks)		
Alternative 2	Check usage of element; check also the coupled element	3.6.12.40	
	Mark usage of element if NOK was registered in the responses in the last sentences	3.6.12.46	
Send response `takeElementIntoUse`		3.6.12.17	

Table 6.1: Translation of the Table describing the behaviour of `takeElementIntoUse` service request

In the table we can observe three main columns. First the activity description. Then a reference to a paragraph where the activity is described in more detail. And lastly, a reference to some other table where the exception to the activity in the same row is described. Upon first inspection of this table, it is not immediately clear what the control flow should be for this service request. Let us assume that the overall control flow is from top to bottom.

First, we begin straightforwardly with determining the coupled elements. But this action apparently has an exception. It is striking that only the first activity has an exception and none of the other activities of the table has one. The table with exception scenarios and the accompanying text is ambiguous in whether the exception holds for the entirety of the service request or only for the activity in the same row. The description of the exception is very general, stating that if something goes wrong, then the EC sends a response with a statement of what went wrong. This can be applied anywhere in the service request. Moreover, the accompanying text for the exceptions states that the table shows the exception scenarios for the `takeElementIntoUse` service request. But this should actually be more precise and state '... the exception scenarios for the activities of the `takeElementIntoUse` service request'.

Second, we go on to do some checks. Accompanying text says that depending on the answer of Check (A) either Alternative 1 or Alternative 2. It is not clear what should happen with the results of Check (B) and (C). Also, the decision point for taking either Alternative 1 or 2 is assumed to be after Check (C) has been done even though it depends on the result of Check (A) that happened two activities ago.

Alternative 1 is clearly empty and we can go straight to the bottom of the table sending the response. Alternative 2 has some contents. Firstly we check the usage of the element and the coupled elements. And then we have to mark the usage of the element if a 'Not OK' response was given in the last sentences. It is unclear what 'sentences' means here and it is assumed that the previous activities are meant. But it is not stated exactly which of the previous actions should be considered. All of them, or only a few?

[4][16], Section 3.6.9.4, p110-111

ProRail states that at the time, there was reason to believe that describing behaviour in a table was favorable compared to other methods like graphs. ProRail admits that this method has not been well received by software contractors. Within ProRail, the department for testing determined they had to draw graphs of the behaviour and verify their correctness with the engineers to be able to test Astris. We can only guess that this may have been quite a lot of work.

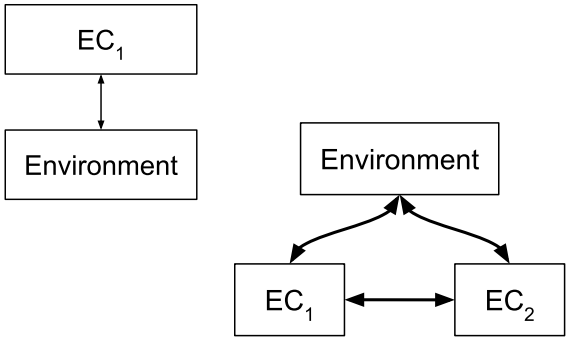
Clearly, there are arguments and observations against the use of describing program behaviour by means of a table. We discuss this further in Section 6.6.

The step of interest for this chapter is the step where the usage of an element and its coupled elements are checked. Thus, all the other steps are omitted in the model or abstractly represented by a dummy action. The documentation does not describe the behaviour of this service request in the case where elements and their coupled elements are not on the same EC. We intuitively expect that this should be done by some service request to the other EC of the coupled element. After inquiring ProRail about this, the conclusion was that the documentation probably assumes that elements and their coupled elements are all local to the same EC per the requirement. Since in this chapter we consider the notion that coupled elements are spread over multiple ECs we have to find and assume a solution for the model.

One solution is to let the issuing of the `takeElementIntoUse` service requests behave similarly as in the case of the locking of elements as described in Subsection 6.2.1. Then, the callee should determine the coupled elements for each element and issue the `takeElementIntoUse` service request for each individually. However, this strays away considerably from the current informal specification and requires a great transfer of responsibilities of the EC to all other components that make use of this service request. Moreover, we do not know of the scope of the effects of such a change. For this reason, this is not a suitable solution.

A different solution is that whenever a `takeElementIntoUse` service request is issued for some element with a coupled element, then to check the usage marking of the coupled elements, the EC issues a `takeElementIntoUse` service request for the coupled element. Coincidentally, upon deliberation with ProRail on this matter, they themselves suggested this same solution. This gives confidence that this is a more suitable solution and therefore this is the solution we consider and assume for our model.

In the model we consider only two elements. These are arbitrarily coupled, i.e. they can be coupled or not. Furthermore, we consider two configurations, i.e. the case of one EC interacting with the environment where it is responsible for both (coupled) elements. And the case of two ECs interacting with the environment and each EC is responsible for one element. This is visualised in Figure 6.2.



(a) One EC (b) Two ECs

Figure 6.2: Coupled elements and usage markings models and interactions

6.3 Verification of coupled elements and locking

One might have noticed that this model is similar to the one discussed in Chapter 5. That is, we consider RCs locking elements in their routes, except that these now also include the coupled elements. As such, we can reuse the formulas of Chapter 5.

The verification of the formulas of Chapter 5 for this model yielded the same results as presented in Table 5.1. This is to be expected since the notion of coupled elements simply add elements to the list of elements in the route of the RC that have to be locked. As such, after identification of the coupled elements, the remainder of the model is equal to the route conflict resolution model and shows the same problems as discussed in Chapter 5.

The most important thing to note is that this model did not contain any deadlocks. Thus, the expectation of ProRail that there where no deadlocks in the locking of coupled elements is correct.

6.4 Verification coupled elements and usage marking

For the verification of the coupled elements and the `takeElementIntoUse` service request we are most interested in proving or disproving ProRail's suspicion of the presence of deadlocks here. For this, we make use of Formula 4.1 of Chapter 4.

Verifying deadlock freedom for the model where the coupled elements are on one EC yields `true`. Verifying deadlock freedom for the model where the coupled elements are split over two ECs yields `false`. Thus, this model does contain a deadlock. Inspection of the counterexample shows an example of a situation leading to a deadlock. The interactions leading to this are described in the sequence diagram of Figure 6.3.

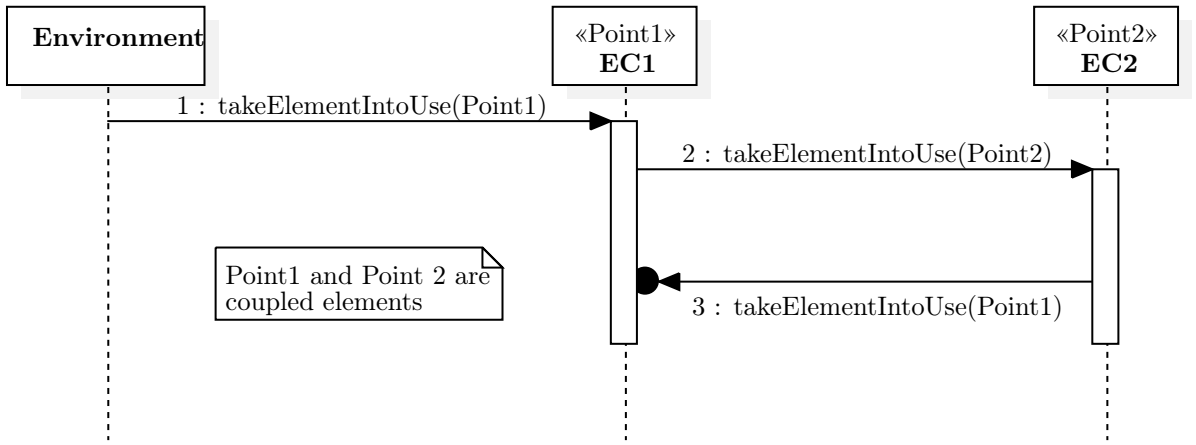


Figure 6.3: Sequence diagram of interactions between components leading to deadlock

In the Figure we can see an RC issuing a `takeElementIntoUse` service request for Point_1. EC_1 receives and processes this service request since it is responsible for Point_1. EC_1 determines the coupled element for Point_1 and finds that Point_2 is coupled to Point_1. Thus, EC_1 issues a `takeElementIntoUse` service request for Point_2. EC_2 receives and processes this service request since it is responsible for Point_2. EC_2 determines the coupled element for Point_2 and finds that Point_1 is coupled to Point_2. Thus, EC_2 issues a `takeElementIntoUse` service request for Point_1. EC_1 should receive and processes this service request since it is responsible for Point_1. However, it is still waiting for the answer of EC_2. And thus, a deadlock occurs.

Now, in reality, there is a timeout making sure that a service request does not wait forever. Nevertheless, any deadlock is undesirable.

This deadlock occurs because a `takeElementIntoUse` service request will always check for coupled elements. In the example this causes an unintended indirect self-directed service request. However, one might

notice that this is not always necessary. In the example, EC_2 does not need to issue a `takeElementIntoUse` service request to the coupled elements of $Point_2$, i.e. $Point_1$, since by transitivity of the binary coupled elements relation EC_1 would have already identified such elements. Thus, the checking of coupled elements should be optional.

One solution is to pass an extra parameter with the `takeElementIntoUse` service request that dictates if an EC must also check the coupled elements, i.e. issue a `takeElementIntoUse` service request for the coupled elements or not. This can be modelled by means of a boolean. The initial call from the environment will always require the receiving EC to also check the coupled elements. Then, any `takeElementIntoUse` service request issued by an EC will never require the checking of coupled elements. With the addition of this condition, the deadlock described in Figure 6.3 should not occur.

We want to verify if this addition to the model resolves the deadlock issue from Figure 6.3 and assume the toolset evaluates to `true`. However, the toolset evaluates to `false`. Thus, this model does contain a deadlock. Inspection of the counterexample shows an example of a situation leading to a deadlock. The sequence diagram in Figure 6.4 describes the order of interactions leading to this deadlock.

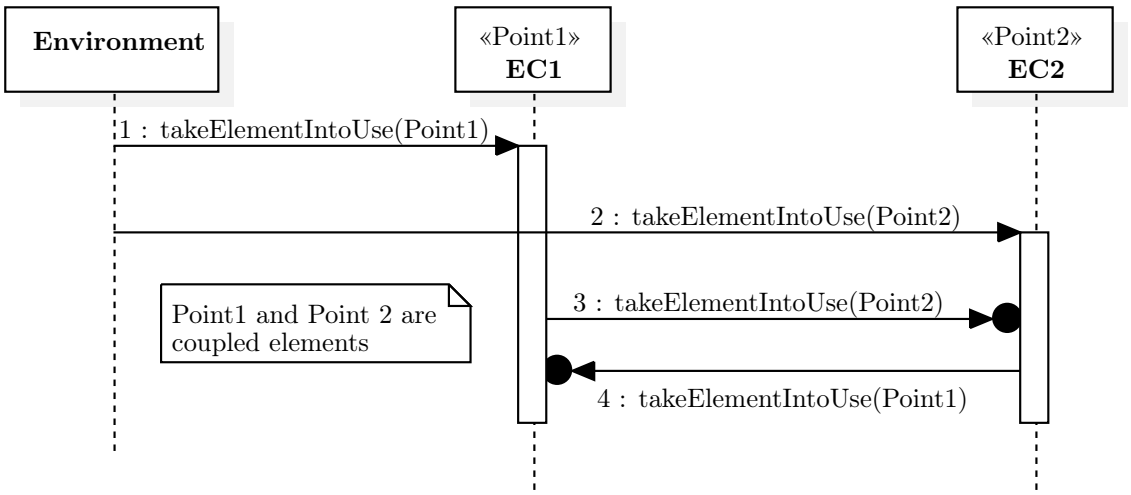


Figure 6.4: Sequence diagram of interaction between components leading to deadlock

In the above sequence diagram we can see how the environment issues a `takeElementIntoUse` service request for both elements at the same time. Then, the deadlock occurs because both ECs want to issue a `takeElementIntoUse` service request to the EC of the coupled element. However, this is not possible since since they are already processing a service request. Thus, we have found one more source of a deadlock.

This issue has no simple solution. However, we still want to find out if there are other ways that reach a deadlock. For this, we exclude the possibility of the above deadlock. One method is to make the assumption that the environment can issue only one `takeElementIntoUse` service request at a time. Or equivalently, the environment can issue the `takeElementIntoUse` service request for only one EC. This is realised in the model by omitting the summation over the domain of `ElementID` for `takeElementIntoUse` and instead passing it one concrete element.

Verification of the model with the addition of this assumption evaluates to `true`. Therefore, we may conclude that we have identified all of the deadlocks in the model.

A summary of the results for this subsection are presented in the below Table 6.2.

	one EC	Two EC
	`true`	`false`
With condition	`true`	`false`
With condition and assumption	`true`	`true`

Table 6.2: Summary of results for verification of coupled elements usage markings model

6.5 Performance

In Table 6.3 the number of states and transitions for the models of this chapter are presented. At the top row we find the model of Subsection 6.2.1. Here, the amount of ECs in the model is not relevant since we consider exactly two. The rest of the rows below are the models we consider in Subsection 6.2.2. The possible configurations are determined by the amount of ECs and the additions presented in the aforementioned subsection.

		Not reduced		Reduced	
#EC	**Additions**	**#states**	**#transitions**	**#states**	**#transitions**
-	Coupled elements and locking model	5389	10026	1215	2587
1		586	918	79	178
1	With condition	586	918	79	178
1	With condition and assumption	532	804	79	160
2		13864	39000	2176	7862
2	With condition	15640	43128	2596	8810
2	With condition and assumption	9628	27564	801	2568

Table 6.3: Number of states and transitions of models in Chapter 6

Firstly, it is clear that the implementation where both coupled elements are on one EC is much less complex than if they are spread over two ECs. We can observe how the addition of the condition does not affect the state-space as expected since with one EC the functionality that it provides is not used.

Second, for the model with two ECs we can see that the introduction of the condition adds some complexity. In the case where the assumption is added to the model we can observe that the model is cut in half. Which is to be expected since we limit the environment to interact with only one EC instead of both.

Finally, there is a great reduction. This is to be expected since this model subsumes the model of Chapter 4 and therefore inherits the reduction benefits. Additionally, the `takeElementIntoUse` service request does not change the state of the EC in contrast to service requests like `LockElement`. It simply adds a few states and transitions to the model of the EC.

The combined execution times of the generation, reduction and verification of the model was always less than a second.

6.6 Discussion

By means of formal specification and verification we have been able to verify two suspicions of ProRail.

1. We were able to verify that the addition of the notion of coupled elements to our model does not cause any problems in the process of locking elements in the route
2. We have identified some deadlocks that can occur with the `takeElementIntoUse` service request and how this deals with coupled elements. Clearly, we where able to resolve the cause of the deadlock described in Figure 6.3. There is not however an easy solution for the deadlock described in Figure

6.4. With this we have been able to prove the suspicion of ProRail that deadlocks may occur in this process. Moreover, we have identified two possibilities for deadlocks.

With this we may conclude that the addition of the requirement that coupled elements should be under the responsibility of the same EC is justified.

The reception of the results by ProRail was positive. The results where found to be clear and convincing. ProRail is of the opinion that it is good to have a definitive answer that their judgement on the matter was right. Their suspicions are proven and are now fact.

Lastly, we share an interesting observation on the specification process. During the process of debugging the model we found that the notion that the environment makes no assumption on the order of the interactions with the ECs caused a problem. Whenever an EC issues a `takeElementIntoUse` service request for another EC, it expects a response. Because the `Respons` actions allows for ambiguity, the environment was able to respond instead of the EC. This caused a deadlock where an EC could no longer respond to the `takeElementIntoUse` service request and this is not the desired behaviour per the informal specification. This was solved by adding a separate interaction `interEC_Respons` that handles responses exclusively between ECs.

Chapter 7

Execution of setting a route

This chapter considers the process of setting of a route in a Route Component. Specifically, the copious set of conditions that must hold before this execution may begin. This set of conditions is described in the informal specification by an elaborate requirement spanning almost three pages. In contrast to the previous chapters, we will not perform any verification. Instead, we will go in-depth on the formal specification of this requirement.

7.1 State of a route

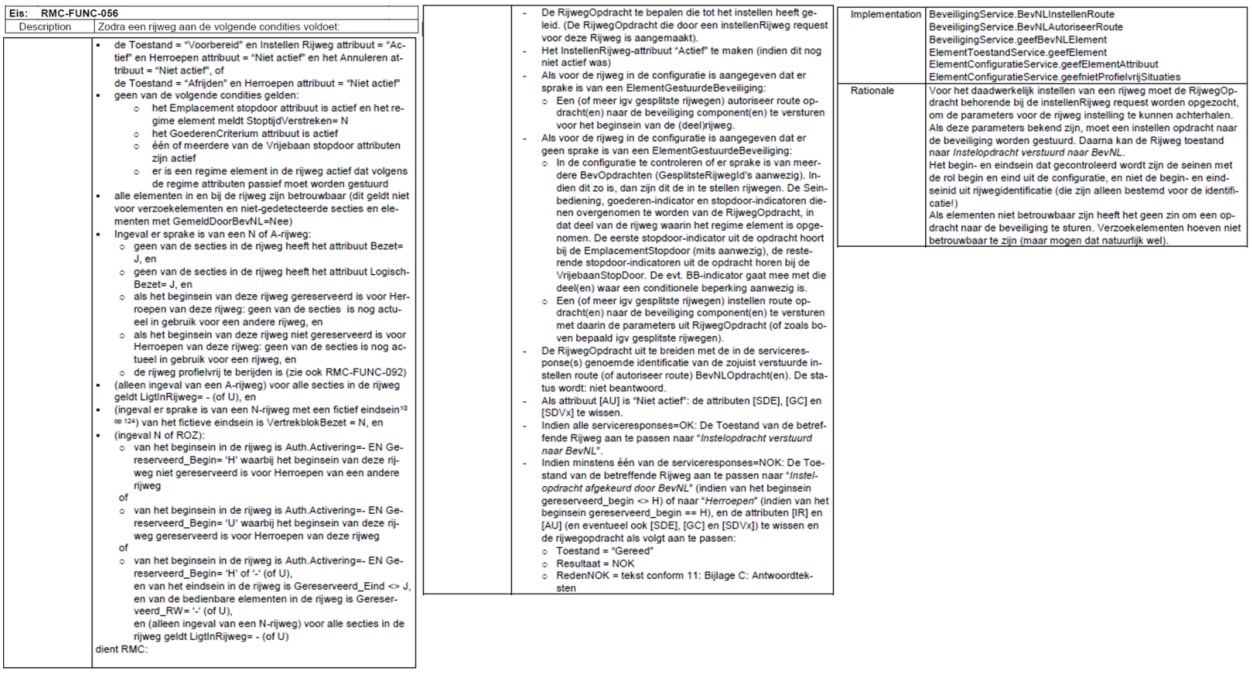
The informal specification for the Route Component describes the functionality of routes by means of a State Transition Diagram, i.e. a state machine. Thus, routes have a state associated to them. Examples of the states are 'rest', 'reserved', 'being prepared', 'prepared', 'cancel route'. Every transition from one state to another has a set of requirements that describe the conditions that must hold before the transition may occur together with the actions associated with this transition. In this chapter we consider the set of conditions of the requirement describing the state change of the route from 'prepared' to 'route setting commands are sent to the infrastructure'. As the names of the states might suggest, this transition will enable the actual sending of commands to the infrastructure. For this reason the requirement is called 'execution of setting a route'. So, if this requirement were not implemented properly, situations could arise where commands are wrongfully sent to the infrastructure.

7.2 Specification

In the informal specification the section on the 'execution of setting a route' consists of two requirements. The subject of the first requirement is the aforementioned state transition of the route. The second requirement takes care of the following corner case: If the state of the route is 'prepared' then all the elements in the route should be in the desired setting. If after preparation the setting has changed, then something wrong has occurred and the route must be cancelled. For the formal specification we disregard this edge case and focus only on the first requirement.

Figure 7.1 shows the requirement in question with trace code `RMC-FUNC-056`[1]. The requirement is split in three parts. The full size requirement text can be found in Appendix F.

[1] [17], Section 3.9.5.26.4, p128-131



Eis: RMC-FUNC-056	
Description	Zodra een rijweg aan de volgende condities voldoet:
• de Toestand = "Voorbereid" en Instellen Rijweg attribuut = "Actief" en Herroepen attribuut = "Niet actief" en het Annuleren attribuut = "Niet actief", of de Toestand = "Afrijden" en Herroepen attribuut = "Niet actief" • geen van de volgende condities gelden: ○ het Emplacement stopdoor attribuut is actief en het regime element meldt StoptijdVerstreken= N ○ het GoederenCriterium attribuut is actief ○ één of meerdere van de Vrijebaan stopdoor attributen zijn actief ○ er is een regime element in de rijweg actief dat volgens de regime attributen passief moet worden gestuurd • alle elementen in en bij de rijweg zijn betrouwbaar (dit geldt niet voor verzoekelementen en niet-gedetecteerde secties en elementen met GemeldDoorBevNL=Nee) • Ingeval er sprake is van een N of A-rijweg: ○ geen van de secties in de rijweg heeft het attribuut Bezet= J, en ○ geen van de secties in de rijweg heeft het attribuut Logisch-Bezet= J, en ○ als het beginsein van deze rijweg gereserveerd is voor Herroepen van deze rijweg: geen van de secties is nog actueel in gebruik voor een andere rijweg, en ○ als het beginsein van deze rijweg niet gereserveerd is voor Herroepen van deze rijweg: geen van de secties is nog actueel in gebruik voor een rijweg, en ○ de rijweg profielvrij te berijden is (zie ook RMC-FUNC-092) • (alleen ingeval van een A-rijweg) voor alle secties in de rijweg geldt LigtInRijweg= - (of U), en • (ingeval er sprake is van een N-rijweg met een fictief eindsein[10 ∞ 124]) van het fictieve eindsein is VertrekblokBezet = N, en • (ingeval N of ROZ): ○ van het beginsein in de rijweg is Auth.Activering=- EN Gereserveerd_Begin= 'H' waarbij het beginsein van deze rijweg niet gereserveerd is voor Herroepen van een andere rijweg of ○ van het beginsein in de rijweg is Auth.Activering=- EN Gereserveerd_Begin= 'U' waarbij het beginsein van deze rijweg gereserveerd is voor Herroepen van deze rijweg of ○ van het beginsein in de rijweg is Auth.Activering=- EN Gereserveerd_Begin= 'H' of '-' (of U), en van het eindsein in de rijweg is Gereserveerd_Eind <> J, en van de bedienbare elementen in de rijweg is Gereserveerd_RW = '-' (of U), en (alleen ingeval van een N-rijweg) voor alle secties in de rijweg geldt LigtInRijweg= - (of U) dient RMC:	• De RijwegOpdracht te bepalen die tot het instellen heeft geleid. (De RijwegOpdracht die door een instellenRijweg request voor deze Rijweg is aangemaakt). • Het InstellenRijweg-attribuut "Actief" te maken (indien dit nog niet actief was) • Als voor de rijweg in de configuratie is aangegeven dat er sprake is van een ElementGestuurdeBeveiliging: ○ Een (of meer igv gesplitste rijwegen) autoriseer route opdracht(en) naar de beveiliging component(en) te versturen voor het beginsein van de (deel)rijweg. • Als voor de rijweg in de configuratie is aangegeven dat er geen sprake is van een ElementGestuurdeBeveiliging: ○ In de configuratie te controleren of er sprake is van meerdere BevOpdrachten (GesplitsteRijwegId's aanwezig). Indien dit zo is, dan zijn dit de in te stellen rijwegen. De Seinbediening, goederen-indicator en stopdoor-indicatoren dienen overgenomen te worden van de RijwegOpdracht, in dat deel van de rijweg waarin het regime element is opgenomen. De eerste stopdoor-indicator uit de opdracht hoort bij de EmplacementStopdoor (mits aanwezig), de resterende stopdoor-indicatoren uit de opdracht horen bij de VrijebaanStopDoor. De evt. BB-indicator gaat mee met die deel(en) waar een conditionele beperking aanwezig is. ○ Een (of meer igv gesplitste rijwegen) instellen route opdracht(en) naar de beveiliging component(en) te versturen met daarin de parameters uit RijwegOpdracht (of zoals boven bepaald igv gesplitste rijwegen). • De RijwegOpdracht uit te breiden met de in de serviceresponse(s) genoemde identificatie van de zojuist verstuurde instellen route (of autoriseer route) BevNLOpdracht(en). De status wordt: niet beantwoord. • Als attribuut [AU] is "Niet actief": de attributen [SDE], [GC] en [SDVx] te wissen. • Indien alle serviceresponses=OK: De Toestand van de betreffende Rijweg aan te passen naar "*Instelopdracht verstuurd naar BevNL*". • Indien minstens één van de serviceresponses=NOK: De Toestand van de betreffende Rijweg aan te passen naar "*Instelopdracht afgekeurd door BevNL*" (indien van het beginsein gereserveerd_begin <> H) of naar "*Herroepen*" (indien van het beginsein gereserveerd_begin == H), en de attributen [IR] en [AU] (en eventueel ook [SDE], [GC] en [SDVx]) te wissen en de rijwegopdracht als volgt aan te passen: ○ Toestand = "Gereed" ○ Resultaat = NOK ○ RedenNOK = tekst conform 11: Bijlage C: Antwoordteksten	Implementation	BeveiligingService.BevNLInstellenRoute BeveiligingService.BevNLAutoriseerRoute BeveiligingService.geefBevNLElement ElementToestandService.geefElement ElementConfiguratieService.geefElementAttribuut ElementConfiguratieService.geefnietProfielvrijSituaties
---	---	---	---
		Rationale	Voor het daadwerkelijk instellen van een rijweg moet de RijwegOpdracht behorende bij de instellenRijweg request worden opgezocht, om de parameters voor de rijweg instelling te kunnen achterhalen. Als deze parameters bekend zijn, moet een instellen opdracht naar de beveiliging worden gestuurd. Daarna kan de Rijweg toestand naar *Instelopdracht verstuurd naar BevNL*. Het begin- en eindsein dat gecontroleerd wordt zijn de seinen met de rol begin en eind uit de configuratie, en niet de begin- en eindseinid uit rijwegidentificatie (die zijn alleen bestemd voor de identificatie!) Als elementen niet betrouwbaar zijn heeft het geen zin om een opdracht naar de beveiliging te sturen. Verzoekelementen hoeven niet betrouwbaar te zijn (maar mogen dat natuurlijk wel).

Figure 7.1: Requirement `RMC-FUNC-056`

The general form of the requirement is as follows: "*If the conjunction of a number of conditions for the route is satisfied, then the RC must do the following sequence of steps*". This form is reflected in the figure. The left part is an exhaustive list of conditions. The middle part describes the steps that must be done if the conditions of the left part are satisfied. Then, the right part gives a listing of the used service requests and a motivation for the purpose of this requirement in the form of a rationale. In this chapter we will only consider the left part of the requirement since we will only specify the conditions of the transition. Note that the requirement does not describe what needs to happen if the condition is not satisfied. Deliberation with ProRail reveals that then nothing should happen. This requirement describes a process where whenever a variable relevant to the described condition changes, the condition is re-evaluated.

We can observe that the method for scoping the individual components of the requirement is by means of bullet points. In the left and middle part of the requirement the style of bullet points is different. It is unclear if this has a meaning or not.

The full size mCRL2 specification can be found in Appendix D.

7.2.1 Data structures

Let us interpret the process described in Figure 7.1 as a 'black box'. Then, we begin by identifying what needs to go 'in' at the beginning of the process and what needs to go 'out' at the end. This is not explicitly described in the requirement and therefore we have to extract this from the requirement itself.

We determine what needs to go 'in' by identifying what needs to be checked in the conjunction of conditions. Here, we find that attributes of the route and the elements in the route are to be checked for certain values. Next, we must track down the data structures from which we can retrieve this information. Since the informal specification does not describe where we can find the relevant data structures, together with ProRail we find the following data structures in the documentation:

- **Information on a route:** The informal specification of the Route Component lists a number of data structures that are to be supported by the RC[2]. We find some of the attributes we are looking for in

[2][17], Section 3.8.2, p35-42

the `Route` data structure.

- **Infrastructural configuration information:** All of the routes of the infrastructure and the elements in these routes are pre-determined immutable 'configuration' information. This information is locally available to an RC in the form of a data structure describing `All elements in and by the route`[3]. Individual elements are described by the `List of elements`[4] data structure containing infrastructural data on elements.

- **Active infrastructural element information:** Information on the current state of a (physical) element are to be retrieved from the infrastructure directly using Astris's 'Signalling' component and its services. We find that for every type of railway element there exists a separate structure[5], e.g. 'section', 'signal', 'fictional signal', 'signal lighting'.

- **Use of elements:** Information on the administrative 'usage markings' of elements, as introduced in Chapter 6, are to be retrieved through Astris's EC and its services[6]. There are three separate structures depending on the type of the element, i.e. 'Section', 'Signal' and more generally 'Operable elements' for the rest of the types.

An overview is given in the class diagram of Figure 7.2. Note that not all of the related data structures are shown, nor all of the attributes for every structure. Mostly the relevant related data structures and attributes are given, i.e. the attributes needed for the formal specification of the requirement. Furthermore, using the stereotype field we make a distinction between structures from within Astris (≪Astris≫), configuration structures (≪Configuration≫) and the active infrastructural element information (≪Infrastructure≫).

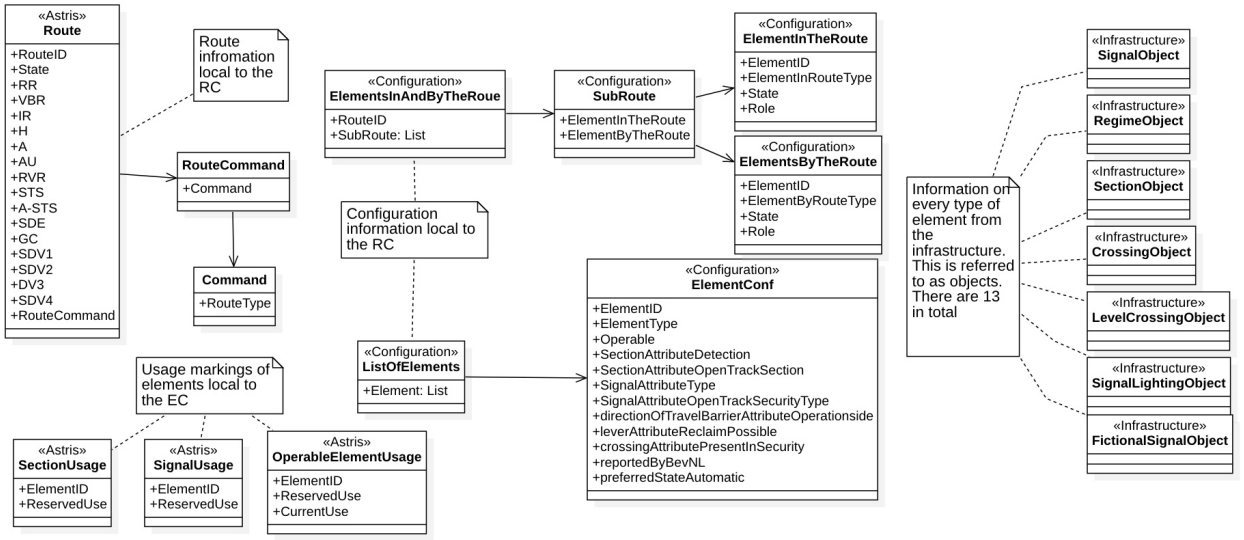


Figure 7.2: Overview of the data structures

From the overview of the data structures we can make some observations:

- Even though we have already omitted the non-relevant parts of the data structures, it is still quite elaborate.

[3][4], p37-41
[4][4], p31-33
[5][18], Appendix A
[6][16], Section 3.5.2.2.2, p19-24

- Data structures are relational. Two data structures with information on the same element are linked by means of the `ElementID`. Likewise for a route by means of the `RouteID`.
- We can observe that, supposedly, the same information can be found in different structures. For example, an element's type is stated in both the "`List of elements`" and the "`All elements in and by the route`" structure. However, type information is not consistent among the structures. For example, a specific element type in one structure does not exist explicitly in the other or are ambiguous (e.g. a type refers to sub-types). This raises the question how we should determine which structure we use to take the necessary information. Moreover, in the case of determining an elements' type, the type information is not consistent. ProRail recognizes this and states that the configuration data is leading for determining an element's type. Therefore, we will do the same in the formal specification. Other ambiguities as to what structure should be used for information is to be validated with ProRail.
- Lastly, there are a great number of element types. However, the requirement considers only a few types and thus, for our specification, we only need these few types.

Formally specifying all of these structures will be a lengthy and tedious process. So, we limit the scope and apply appropriate abstractions. The goal for the abstractions is then to minimize the amount of attributes, element types and data structures we support in our model. Generally, this is realised by including only the relevant attributes and specifying only the necessary element types, i.e. Signal, Fictional Signal, Section and Regime.

The `Route` structure has a reference to the `RouteCommand` structure which has a reference to the `Command` structure. This structure should contain the 'route type' information we need for the specification. The route type is stored in a 'signal operation' (*signaalbediening*) attribute. To minimize the size of the structure, we substitute the attribute with the reference to the `RouteCommand` structure to an attribute holding the 'route type' directly.

The notion that the different structures for individual elements are related by means of the `ElementID` makes the formal specification more tedious since we will have to quantify over multiple lists of structures and then limit the domain to structures with an equal `ElementID`. Instead, we can take an abstract approach by letting the `AllElementsInAndByRoute` structure contain the structures for the elements directly. This eliminates the need for the `ElementID` in the element structures.

Additionally, let us make the assumption that the configuration is leading for determining an elements' type more concrete. That is, the `ElementInRouteType` or `ElementByRouteType` and `role` attributes determine the exact element type. As a consequence, this makes the notion that other structures hold information on the type of the element irrelevant and can therefore be omitted. This is an important realisation since it allows us to more easily make the 'element usage' and 'Active infrastructural' structures more abstract. For example, there are many 'Active infrastructural' structures because each structure differentiates on the element type. However, since this information is not irrelevant in this structure, we can generalise the structures to one structure that contains all of the necessary attributes.

Finally, we can see that a route is partitioned into sub-routes that make the structure more complicated. Since the notion of sub-routes is not used in our process and because it is rarely used in reality, we abstract the notion of sub-routes away and let the `AllElementInAndByRoute` data structure contain the lists of elements in and by the route directly.

Figure 7.3 shows an overview of the data structures of the formal specification resulting from the above described abstractions. Here, all relevant attributes are present.

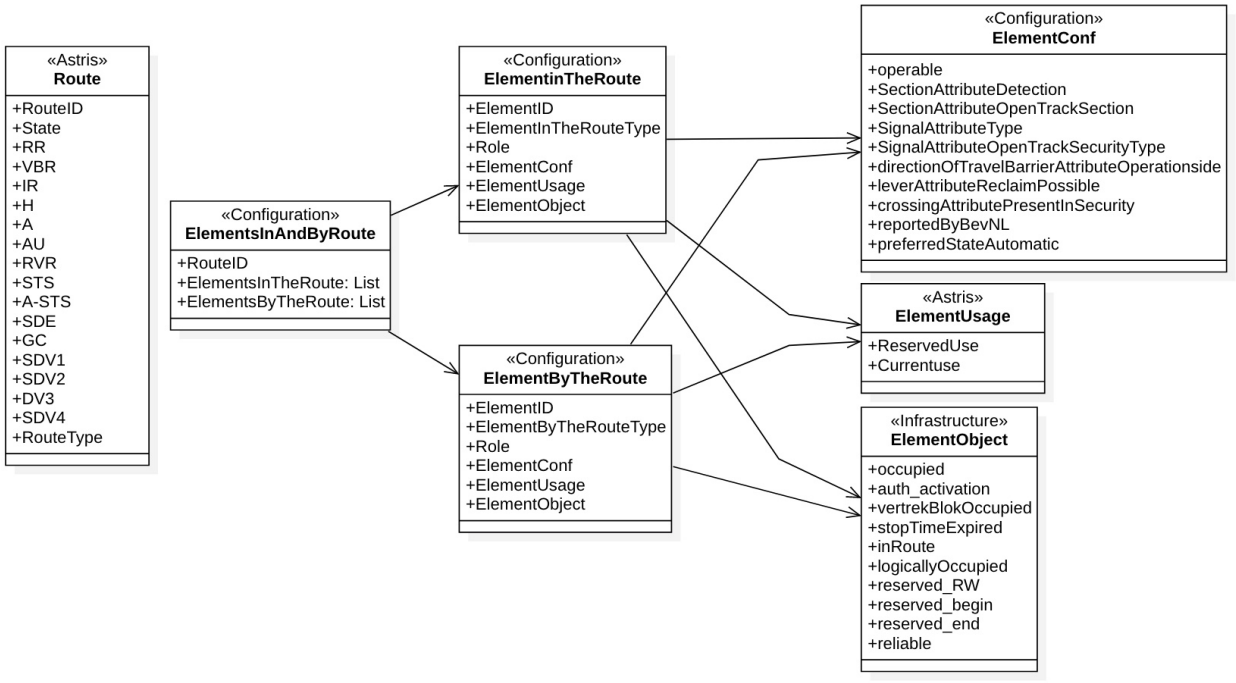


Figure 7.3: Overview of data structures with abstractions

Thus, we now have two data structures for the process specifying requirement `RMC_FUNC_056`. That is, `Route` and "`All elements in and by the route`". Next, we formally specify these data structures. To give a little insight into this, let us consider the specification of the `Route` structure.

mCRL2 allows the user to specify structured data types. Let us take for example the data type `Activity` that we use in the model for a number of the attributes in the `Route` structure. The value of these attributes can be either `Active` or `NotActive`. We specify this as follows.

```
sort
  Activity = struct Active | NotActive;
```

A more involved type is the `RouteType` which is actually the `SignalOperation` (*seinbediening*) attribute which has three possibilities.

```
sort
  SignalOperation = struct Normaal | RijdenOpZicht | Automaat; % SignaalBediening
```

Then, the structure for the `Route` structure is specified as a tuple as shown below.

```
sort
  Route = struct Route( % From Rijweg SRS
    routeID : RouteID,
    state :  RouteState,
    RR : Activity, % Reservation route / Reserveren Rijweg
    VBR : Activity, % Preparing route / VoorBereiden Rijweg
    IR : Activity, % setting route / Instellen Rijweg
    H : Activity, % Cancel Route / Herroepen
    A : Activity, % Cancelling route / annuleren rijweg
```

```
  AU : Activity, % Automatic / Automaat
  RVR : Activity, % Rijweg VrijRijden
  STS : Activity, % STS route
  A_STS : Activity, % Cancelling STS-route / Annuleren STS-route
  SDE : Activity, % Emplacement stopdoor
  GC : Activity, % Cargo criterium / GoederenCriterium
  SDV1 :  Activity, % Vrijebaan stopdoor 1
  SDV2 :  Activity, % Vrijebaan stopdoor 2
  SDV3 :  Activity, % Vrijebaan stopdoor 3
  SDV4 :  Activity, % Vrijebaan stopdoor 4
  signalOperation :  SignalOperation % Route type / seinbediening
);
```

Let us assume we have a route structure `r` of type `Route`, then we can reference the value of a specific attribute, e.g. `state`, with `state(r)` where the value is then of type `RouteType` as per the specification.

7.2.2 Specification of the condition

In the beginning Section 7.2 we had already identified that the form of this process is "*If the conjunction of a number of conditions for the route is satisfied, then the RC must do the following sequence of steps*". Having determined and specified the necessary data structures we can specify the condition. We approach this by letting the general structure of the requirement be our guide. Since we know that the condition is the conjunction of the bullet points, we specify each of the bullet points separately. To illustrate, we map every bullet of the condition to a predicate as visualised in Figure 7.4.

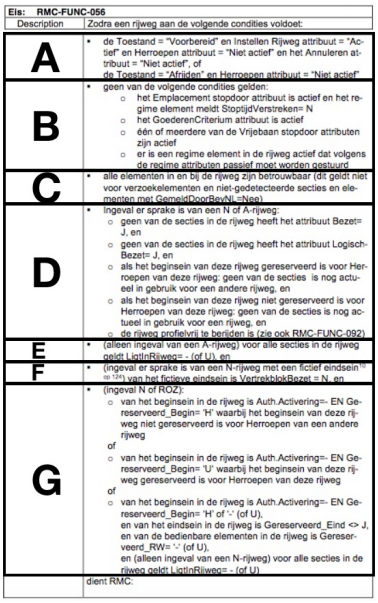


Figure 7.4: Each bullet point has an associated predicate from A to G

Then, the condition in the specification will be of the form $A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G$. Let us take a closer look at the formal specification of all of these predicates.

By translation from Dutch to English, predicate A states the following: "the state = 'Prepared' and Setting Route attribute (IR) = 'Active' and Cancel Route attribute (H) = 'Not Active' and the Cancelling

route attribute (A) = 'Not Active', or the State = "Ride off" en Cancel Route attribute (H) = 'Not Active' ". This requirement makes it very explicit what the values of the attributes must be. One might notice how there are two attributes for 'cancel(ling) route'. This ambiguity is a consequence of translation. The words *herroepen* and *annuleren* have a different meaning in Dutch.

An thus, we come to the following specification of predicate A.

```
map
  pred_A : Route -> Bool;
var
  r :  Route;
eqn
  pred_A(r) = (state(r) == Prepared && IR(r) == Active && H(r) == NotActive &&
    A(r) == NotActive) || (state(r) == Ride_off && H(r) == NotActive);
```

Predicate B states that none of the following clauses must hold:

- "the emplacement *stopdoor* attribute (SDE) is active and the regime element reports StopTimeExpired = N"
- "the CargoCriterium (GC) is active"
- "one or more of the open track *stopdoor* attributes (SDV1, SDV2, SDV3 or SDV4) is active"
- "there is a regime element in the route active that should be passive according to the regime attributes"

So, predicate B is the negation of the conjunction of the above clauses. Here, the first three clauses are simply checking attributes of the `Route` data structure. There is a small remark to make on the first clause though. It speaks of 'the regime element' as if we should assume there is such an element in the route. Moreover, it is not explicitly stated, but the type of the regime element should actually be the regime element sub-type 'Emplacement *stopdoor*'. And thus, we refine this part to 'if there is a regime element of sub-type Emplacement *stopdoor* in the route'. The last clause is more involved. A first observation is that the clause does not state any concrete attributes and the value it should have, which is the case with the other clauses. Then, we find in the documentation that there cannot be regime elements *in* a route, only *by* a route. Furthermore, it is unclear how we should determine a regime element to be 'active' or 'passive'. Inquiring ProRail about this clause did not give a satisfying answer. They conjecture that probably at the time it was clear what was meant with this, but now ProRail cannot give an answer with the desired precision. Because of this, we will not be able to formally specify this correctly, and therefore we omit this clause from the specification.

As a result, we come to the following formal specification of predicate B.

```
map
  pred_B : Route # AllElementsInAndByRoute -> Bool;
var
  r :  Route;
  elementsInAndByRoute :  AllElementsInAndByRoute;
eqn
  pred_B(r, elementsInAndByRoute) = !(SDE(r) == Active &&
    (exists e :  ElementByRoute .  (e in elementByRoute(elementsInAndByRoute) &&
      ElementByRouteType(e) == Regime_EmplacementStopdoor) &&
      !stoptimeExpired(ElementObject(e))) &&
    GC(r) == Active &&
    (SDV1(r) == Active || SDV2(r) == Active || SDV3(r) == Active || SDV4(r) == Active));
```

The text for predicate C is relatively short. It states the following: "All element in and by the route are reliable. This does not hold for request elements, undetected sections and elements with ReportedByBevNL = No".

While it is not explicitly stated, the `ReportedByBevNL` attribute may be used for elements both in and by the route. Hence we check for both in the predicate.

We come to the following formal specification of predicate C:

```
map
  pred_C : Route # AllElementsInAndByRoute -> Bool;
var
  elementsInAndByRoute :  AllElementsInAndByRoute;
eqn
  pred_C(elementsInAndByRoute) =
    forall e_in :  ElementInRoute .  forall e_by :  ElementByRoute .
      ((e_in in elementInRoute(elementsInAndByRoute) &&
        e_by in elementByRoute(elementsInAndByRoute)) &&
      (sectionAttributeDetection(elementConf(e_in)) &&
      !reportedByBevNL(elementConf(e_in)) && !reportedByBevNL(elementConf(e_by)) &&
      elementByRouteRole(e_by) != Element_request)) =>
        (reliable(elementObject(e_in)) && reliable(elementObject(e_by)));
```

Clearly, while the text of the predicate is quite short, the formal specification is much larger than that of the previous predicates.

Predicate D states that in case of the route is of 'A' (*Automaat*) or 'N' (*Normaal*) type, then the following clauses must hold:

- "none of the sections in the route have the attribute Occupied = yes"
- "none of the sections in the route have the attribute Logically Occupied = Yes"
- "if the begin signal of this route is reserved for cancellation (*herroepen*): none of the sections is currently in use for another route"
- "if the begin signal of this route is not reserved for cancellation (*herroepen*): none of the sections is currently in use for a route"
- "the route is drivable profile-free (*profielvrij te berijden*)"

Clearly, the main form of the predicate is that of an implication. This, in contrast to the last few predicates that where a list of clauses.

The first two clauses are straightforwardly of the form 'there does not exists a section in the route with the attribute Occupied = yes' for example.

The next two clauses, i.e. the third and fourth, deviate in form from what we have seen in the previous predicates. First, it is not made explicit what attributes should be evaluated and what their values should be. Second, the semicolon is used to denote a 'if-then' relation between what is in front of the semicolon and what comes after it. Moreover, a subtle difference between the clauses is that in the first the sections may not be in use for a different route than the one we are considering and in the second the section may not be in use for any route at all. At first, it was not clear where the information for reservations and use markings should be retrieved from since both the active infrastructural element information and the use of elements structure contain attributes on reservations. However, apparently, whenever it is not made explicit what attributes should be checked, then the information comes from the usage structure within Astris. Thus, information on reservations is to be retrieved from the usage structure. The specific use contains a `RouteID` for which route it is in use. We can use this to compare with the `RouteID` of the route we are considering, found in the `Route` structure, to determine if the reservation is for this route or another route.

The last clause is more involved and refers to another requirement with its own set of conditions. Since we wish to consider only one requirement in this chapter, we will omit this clause from the specification.

Then, finally, we come to the following formal specification of predicate D.

```
map
  pred_D : Route # AllElementsInAndByRoute -> Bool;
var
  r :  Route;
  elementsInAndByRoute :  AllElementsInAndByRoute;
eqn
  pred_D(r, elementsInAndByRoute) =
    (signalOperation(r) == Automaat || signalOperation(r) == Normaal) =>
    ((!exists e :  ElementInRoute .  ((e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Section && occupied(elementObject(e)))) &&
    (!exists e :  ElementInRoute .  ((e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Section && logicallyOccupied(elementObject(e)))) &&
    (exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == begin_signal &&
      (exists u :  UseTuple .  u in use(elementAstris(e)) &&
        reservedUse(u) == Herroepen && routeID(u) == routeID(r)) &&
        !(forall ep :  ElementInRoute .  (ep in elementInRoute(elementsInAndByRoute) &&
          elementInRouteType(ep) == Section) =>
          (forall u :  UseTuple .  u in use(elementAstris(e)) =>
            (currentUse(u) && routeID(u) != routeID(r))))) &&
    (exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == begin_signal &&
      !(exists u :  UseTuple .  u in use(elementAstris(e)) &&
        reservedUse(u) == Herroepen && routeID(u) == routeID(r)) &&
        !(forall ep :  ElementInRoute .  (ep in elementInRoute(elementsInAndByRoute) &&
          elementInRouteType(ep) == Section) =>
            (forall u :  UseTuple .  u in use(elementAstris(e)) => currentUse(u))))
    );
```

We can see that the specification is quite large. This is mainly because of the many quantifiers and the description of the domain it quantifies over. For example, for a begin signal we quantify over the domain of all elements in a route since signals can only be in the route. Then, we describe the domain by determining that the element is in the route we are considering and that it is of type 'signal' and that it has the role of 'begin signal'. This simply takes some space. Note how we use the existential quantifier for the begin signal since we assume there can be only one in the route.

Next, predicate E is much smaller, both in text and specification. The predicate states the following: "(In case of a route of type 'A' *(Automaat)), for all sections in the route it holds that inRoute = - (or U)*". This predicate is very straightforward, though the evaluation of the attribute inRoute seems odd with the bracket usage. It is interpreted as "(inRoute = -) or (inRoute = U)".

Then, the formal specification of predicate E is as follows.

```
map
  pred_E : Route # AllElementsInAndByRoute -> Bool;
var
  r :  Route;
  elementsInAndByRoute :  AllElementsInAndByRoute;
eqn
  pred_E(r, elementsInAndByRoute) = (signalOperation(r) == Automaat) =>
    (forall e :  ElementInRoute .
      (e in elementsInRoute(elementsInAndByRoute) && ElementInRouteType(e) == Section) =>
        (inRoute(elementObject(e)) == dash_NotPartOfRoute ||
          inRoute(elementObject(e)) == Unknown_IsPartOfRoute));
```

Predicate F states the following: "in case of a route of type 'N' (*Normaal*) with a fictional end signal, for the fictional end signal holds *vertrekblokBezet* = No". Now, a fictional signal is not a type of signal that is available in as a type in the '*All elements in or by the route*' data structure. Instead, it's an attribute of the configuration information of the element itself. It is assumed that an end signal is *in* the route. Since a fictional end signal need not be a physical element nor a signal we do not check if the element is of type 'signal' with role 'end'.

Then, we come to the following formal specification of predicate F.

```
map
  pred_F : Route # AllElementsInAndByRoute -> Bool;
var
  r :  Route;
  elementsInAndByRoute :  AllElementsInAndByRoute;
eqn
  pred_F(r, elementsInAndByRoute) = exists e :  ElementInRoute .
    ((signalOperation(r) == Normaal && (e in elementInRoute(elementsInAndByRoute)) &&
    signalAttributeType(elementConf(e)) == Fictional_End) &&
    vertrekBlokBezet(elementObject(e)));
```

The last predicate, predicate G, states that in if the route is of type 'N' (*Normaal*) or 'ROZ' (*Rijden Op Zicht*), then the disjunction of the following items must hold:

- "for the begin signal of this route it must hold that Auth_activation = - and Reserved_begin = 'H' where the begin signal of this route is not reserved for cancellation (*herroepen*) of another route".
- "for the begin signal of this route it must hold that Auth_activation = - and Reserved_begin = 'U' where the begin signal of this route is reserved for cancellation (*herroepen*) of this route".
- "for the begin signal of this route it must hold that Auth_activation = - and Reserved_begin = 'H' or '-' (or 'U'), and for the end signal in the route it holds that Reserved_end <> yes, and for the operable (*bedienbaar*) elements in the route it holds that Reserved_RW = '-' (or U), and (only in case of a route of type 'N') for all sections in the route it holds that inRoute = - (or U)".

We can observe how the first two items are very similar. Though, the wording is somewhat peculiar. The sentence makes it seem as if the first mentioned begin signal is different from the secondly mentioned begin signal. However, the same begin signal is meant. Thus, the sentence could be refined to: "for the begin signal of this route it must hold that Auth_activation = - and Reserved_begin = 'H' and is not reserved for cancellation (*herroepen*) of another route". Apart from a different value for the 'Reserved_begin' attribute, the items differ in reservation. For the first item we check that it is not the case that the signal is reserved for cancellation and the `RouteID` of the reservation is different from the `RouteID` of the route we are considering. For the second item we check that the reservation is for cancellation and that the `RouteID` of the reservation matches that of the route we are considering.

The last item is a bit larger and consists of a number of clauses. A first observation is that the use of quotes around the values of attributes is not always consistent. It is unclear if this has a meaning or not. Second, we can see the binary operator '<>'. At first it was unclear what this means, but apparently it has the meaning of 'not equal' and, in contrast to the more popular notation '!=', it is ANSI compliant.

Then, we come to the following formal specification of predicate G.

```
map
  pred_G : Route # AllElementsInAndByRoute -> Bool;
var
  r :  Route;
  elementsInAndByRoute :  AllElementsInAndByRoute;
eqn
```

```
  pred_G(r, elementsInAndByRoute) =
    (signalOperation(r) == Normaal || signalOperation(r) == RijdenOpZicht) =>
    ((exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == begin_signal &&
      auth_activation(elementObject(e)) == dash_No_routeCommand &&
      reserved_begin(elementObject(e)) == H_Cancelled &&
      (forall u :  UseTuple .  u in use(elementAstris(e)) =>
        !(reservedUse(u) == Herroepen && routeID(u) != routeID(r)))) &&
    (exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == begin_signal &&
      auth_activation(elementObject(e)) == dash_No_routeCommand &&
      reserved_begin(elementObject(e)) == H_Cancelled &&
      (exists u :  UseTuple .  u in use(elementAstris(e)) &&
        reservedUse(u) == Herroepen && routeID(u) == routeID(r))) &&
    ((exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == begin_signal &&
      auth_activation(elementObject(e)) == dash_No_routeCommand &&
      (reserved_begin(elementObject(e)) == H_Cancelled ||
        reserved_begin(elementObject(e)) == dash_Not_used ||
        reserved_begin(elementObject(e)) == Unknown_Reserved_Begin)) &&
    (exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == end_signal &&
      reserved_end(elementObject(e))) &&
    (forall e :  ElementInRoute .  ((e in elementInRoute(elementsInAndByRoute)) &&
      operable(elementConf(e))) =>
        (reserved_RW(elementObject(e)) == dash_Not_Used ||
          reserved_RW(elementObject(e)) == Unknown_Reserved_RW)) &&
    (signalOperation(r) == Normaal) =>
      (forall e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute) &&
      elementInRouteType(e) == Section) =>
        (inRoute(elementObject(e)) == dash_NotPartOfRoute ||
          inRoute(elementObject(e)) == Unknown_IsPartOfRoute))
    ));
```

7.3 Specification of the process

Having specified all of the predicates in the previous chapter, we can now put them together in a process. The logical consequence whenever the condition is evaluated to true (the middle part of Figure 7.1) is not specified. In the process we will substitute it with a `dummy` action. The process has the `Route` and `AllElementsInAndByRoute` data structures as parameters. Then, the process for `RMC_FUNC_056` is as follows.

```
act
  dummy;
proc
  RMC_FUNC_056(r :  Route, allElementsInAndByRoute :  AllElementsInAndByRoute) =
    (pred_A(r) && pred_B(r, allElementsInAndByRoute) && pred_C(allElementsInAndByRoute) &&
    pred_D(r, allElementsInAndByRoute) && pred_E(r, allElementsInAndByRoute) &&
    pred_F(r, allElementsInAndByRoute) && pred_G(r, allElementsInAndByRoute)) ->
      dummy .  RMC_FUNC_056(r, allElementsInAndByRoute);
```

7.4 Discussion

We have included the reader in the process of formally specifying a part of the requirement `RMC_FUNC_056`. During this process a number of ambiguities, imprecisions and unclarities have been identified.

By clearing up the identified ambiguities and inconsistencies with ProRail we have gained a more precise specification of the requirement. However, we have not been able to formally verify the last clause of predicate B because ProRail cannot currently offer an explanation with the desired precision. If this clause were formally specified, less information would have been lost and the necessity to analyse the implementation to gain the desired precision would disappear.

Upon sharing the results with ProRail, the feedback was positive. The approach to formally specifying the process is clear and the applied abstractions for the data structures where deemed appropriate. Sharing the results per predicate, it became clear how the mapping from the informal specification to the formal specification is established with little additional explanation. An interesting observation is that the difficulty or complexity of the predicates is in increasing order. That is, predicate A deals only with basic logic connectives. Then, predicates B and C introduce the notion of quantifiers. This allows the uninitiated reader get oneself familiarised with the syntax of mCRL2. Predicates D and G exemplify more involved conditions and the intuition of the mapping between the informal specification and the formal specification may be lost due to the formal specification being more complex, but also the informal specification being more general in wording, i.e. not explicitly stating attribute names etc.

It is clear how in the process of formal specification, the necessity for precision reveals imprecisions, ambiguities and inconsistencies in the informal specification. And subsequently, how a formal specification may resolve such issues. While this chapter only deals with data structures and logic statements it is clear how the intuition translates over to the specification of the behaviour of a process. Moreover, it is understood how the formal specification may act as a reference on how functionality is implemented.

Chapter 8

Discussion and conclusion

For the purpose of determining whether or not ProRail may benefit from the use of formal methods in their software development and maintenance process, the use of formal methods has been demonstrated. For this, in the previous chapters, parts of the Train Traffic Control System called Astris have been formally specified and verified.

In the next sections we will discuss the results from the previous Chapters 4, 5, 6 and 7. That is, the results obtained from specification, verification, sharing the results with ProRail, observations from ProRail's informal specification and other general observations. Moreover, we will also discuss observations on the used toolset mCRL2 and its suitability for the formal specification and verification of the traffic control systems. In a final point of discussion we reflect on the work that has been done. After the discussion, some recommendations for future work will be given. Finally, based on the results and the discussion, we conclude with answering the main question whether or not ProRail would benefit from the use of formal specification and verification.

8.1 Discussion

The following subsections provide discussion on the obtained results of this thesis, the mCRL2 toolset and a reflection on the results.

8.1.1 Results

Throughout the previous chapters we have met the goal of demonstrating the use of formal specification and verification. The Chapters 4, 5 and 6 have a focus on showing what formal verification can do. And Chapter 7 demonstrated the use of formal specification. Let us consider the results of each chapter.

In Chapter 4 we have formally specified the Element Component's 'locking mechanism' and verified it against a claim on this mechanism made by ProRail. Through verification we found that the claim was not precise enough. After refinement of the claim we where able to prove the refined claim to be true. It proved to be a good example for introducing ProRail to formal methods and what these can do.

Chapter 5 builds upon this model by introducing the Route Component making use of the Element Component's service requests for the locking of elements in a route. Thus, introducing interactions between RCs and ECs. We verified the model for correctness of the route conflict resolution method and identified two issues where the method proved to not be fail proof. Because of other mechanisms that help resolve these conflicts, this is not a problem.

Then, Chapter 6 extends this model by introducing the notion of 'coupled elements'. ProRail suspected, but had not proven, that under certain conditions a deadlock would occur. For this reason they have imposed a requirement on the system such that these conditions will never occur. By means of verification we where able to prove this suspicion correct by identifying two ways in which a deadlock could occur under said

conditions. And thus, we can conclude that ProRail's suspicion and their decision to impose a requirement to prevent these deadlocks to be justified.

Finally, in Chapter 7 we have taken a lengthy and complex requirement, suggested by ProRail, and included the reader in the process of producing a formal specification from an informal specification. This allowed us to really show the added value that, specifically formal specification, can have for the specification of a system. That is, through formal specification we found a number of ambiguities, inconsistencies and unclarities. Moreover, we found one unclarity that ProRail was not able to elucidate with the precision we required for the formal specification. Thus, the produced formal specification is more precise than the informal specification and it is clear that a formal specification can contribute towards counteracting information lost to time.

While Chapters 4-6 did uncover some ambiguities, inconsistencies and unclarities in the specification, it is clear that their contribution lies in the results of verification. For this reason, Chapter 7 has proven to be a good and relevant addition by highlighting the added value of formal specification.

ProRail's response to the results has been positive. By means of demonstration it has been made clear how formal specification and verification can help in finding and resolving issues, ambiguities, imprecisions and inconsistencies.

8.1.2 Performance of the mCRL2 toolset

In every chapter, except for Chapter 7, we have noted the performance of the mCRL2 toolset. We have observed how the models presented in this thesis are fairly small. As a consequence of this, the generation and verification of the models took little time and hence, nothing remarkable is found.

However, in Section 5.5 we have observed how the execution times of the verification of the model increased as the size of the model increased. ProRail raised the concern what this means for larger models with more functionality. Now, it must be mentioned that with the technology of our time it is not possible to verify a (detailed) specification of the entirety of Astris. Verification is to be applied on isolated parts of a systems if the system as a whole is too large. Nevertheless, this does not mean that one should not specify the entirety of Astris since for simulation generation of the state space is not necessary. And simulation is also a very useful tool for analysis and gain insights.

8.1.3 Using the mCRL2 toolset

Specification in mCRL2 is very different from programming. Some training together with a good understanding of set theory and mathematical logic is required.

Primarily the mCRL2 IDE, mentioned in Subsection 3.2.4, has been used for the formal specification and verification of the models of this thesis. A few issues have been encountered during this process. A number of bugs have been reported to the mCRL2 GitHub issue tracker. As well as a feature request.

One bug of the toolset that has cost a lot of time is where under specific circumstances the evidence generated by the toolset is empty, i.e. consists of a single (initial) state, while we know the evidence is erroneous. This bug is fixed in newer versions of mCRL2.

Aside form the above items of critique, the toolset is very powerful and good to work with.

8.1.4 Reflection

While the added value of formal methods has been shown, it must be noted that it does not solve all problems. The use of formal methods knows its limitations. Formal methods primarily concerns the behaviour of a system. So, there are many types of requirements of a system that the use of formal methods will not cover. Examples are requirements for performance, security or the use of standards.

As we have seen in Chapter 7, a formal specification can get quite complex. One can argue then that a formal specification is harder to understand than its textual counterpart and consequently that the formal specification makes no improvement over the informal specification. However, the purpose of a formal

specification is not necessarily to make a specification easier to understand. The goal is to make a precise specification.

When discussing the results of this thesis with ProRail, it became clear that Chapter 6 was found to be most helpful in understanding what formal verification can do. And 7 was found to be most helpful in understanding what formal specification can do. Especially the latter was found to be most insightful. One can then wonder why, if these chapters are so important, we did not start our work here. We argue that the work in the earlier chapters where necessary to gain the knowledge on Astris and the railway domain that was required for the work in these chapters.

This thesis contains some unanswered questions resulting from decisions based on intuition. These are as follows:

- When considering the basic modelling of Astris, an abstraction was applied that resulted in the omission of message queues for Astris' components. The reasoning that this would not result in a loss of behaviour is based on intuition. Further analysis would be necessary to substantiate this claim.
- In Chapter 4 we consider a model with only one EC and the environment but not more ECs. Again, the reasoning for this is partially on intuition and further analysis is needed to justify this choice.
- Throughout Chapters 4, 5 and 6 we verify the model for the basic notion of deadlock. There are, however, stronger notions of deadlock that the model may be verified for (e.g. livelock). Whether these models require to be verified for these stronger notions of deadlocks needs further analysis.

8.2 Recommendation

In the below subsections are some recommendations on the use of formal methods and future work.

8.2.1 The use of formal methods for Train Traffic Control Systems

The above discussion makes a convincing argument that the use of formal methods is a potential solution for the problem of ProRail. Therefore, ProRail is recommended to make use of formal methods in their software development and maintenance process. Furthermore, we give some suggestions on how formal methods can be used in their current process and documentation structure.

The formal specification is not a replacement for the informal specification. Though, it may be the case that details of a specification are deferred to the formal specification and omitted from the informal specification. The formal specification can then be used as a reference where the exact functionality is described. Another approach is to let the formal specification be the leading specification and a informal specification supports this formal specification. In either case, a (naive) suggestion on how the use of formal methods may be incorporated in the current STD-MIL-498 documentation standard is to make references in the documentation to places in the model (and vice versa) via e.g. the requirement traceability codes.

8.2.2 Future work

ProRail has expressed a desire to look into the use of formal methods for the purpose of testing. If, after this thesis, ProRail remains to find themselves interested in the use of formal methods and how this can help in testing, it is recommended to look into this aspect in the future. For this, mCRL2 can be used since it is possible to derive tests from an LTS using JTorX for example. The combination of mCRL2 and JTorX has already been used in the railway domain for interlockings [6].

In the above reflection subsection we pointed out some unanswered questions that could be the subject of future work.

8.3 Conclusion

This thesis demonstrates how the use of formal specification and verification can help ProRail in their software development and maintenance process. It has been shown how the process of formal specification can help identify ambiguities, inconsistencies and unclarities in the informal specification. It is established how the formal specification can be more precise than an informal specification. Additionally, it has been shown how a model can be verified against properties. By verification we can check if the model exhibits the desired behaviour and analyse this behaviour to find faults.

This thesis also demonstrates the applicability of mCRL2 on a Train Traffic Control System. The mCRL2 toolset has shown to be able to verify meaningful properties and helps in the analysis of found results.

Finally, we answer the main question. To be able to conclude that ProRail will benefit form the use of formal methods in their software development and maintenance process, we are required to show that the use of formal methods is able solve the problem that their informal specification is not precise enough. Since we have shown how the use of formal methods has made an informal specification more precise, we may conclude that ProRail can benefit from the use of formal methods.

Chapter 9

Bibliography

[1] Maarten Bartholomeus, Bas Luttik, and Tim Willemse. Modelling and analysing ERTMS hybrid level 3 with the mCRL2 toolset. In Falk Howar and Jiří Barnat, editors, *Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Proceedings*, Lecture Notes in Computer Science, pages 98–114, Germany, 1 2018. Springer.

[2] Davide Basile, Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, Franco Mazzanti, Andrea Piattino, Daniele Trentini, and Alessio Ferrari. On the industrial uptake of formal methods in the railway domain. In Carlo A. Furia and Kirsten Winter, editors, *Integrated Formal Methods*, pages 20–29, Cham, 2018. Springer International Publishing.

[3] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A successful application of B in a large project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, pages 369–387, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[4] Bert Bossink. Interface Requirements Specification / Design Description Astris Railinfrastructuur Configuratie. Standard, ProRail, Februari 2019.

[5] Bert Bossink. System Subsystem Design Description; Astris. Standard, ProRail, September 2019.

[6] Mark Bouwman, Bob Janssen, and Bas Luttik. Formal modelling and verification of an interlocking using mCRL2. In *Formal Methods for Industrial Critical Systems*, pages 22–39, Cham, 2019. Springer International Publishing.

[7] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39, Cham, 2019. Springer International Publishing.

[8] CENELEC. EN 50128 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. Standard, CENELEC, Brussels, BE, July 2011.

[9] Alessandro Fantechi. Twenty-five years of formal methods and railways: What next? In Steve Counsell and Manuel Núñez, editors, *Software Engineering and Formal Methods*, pages 167–183, Cham, 2014. Springer International Publishing.

[10] J. F. Groote, S. F. M. van Vlijmen, and J. W. C. Koorn. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *COMPASS '95 Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security'*, pages 57–68, 1995.

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

[12] Yi-Ling Hwong, Vincent J. J. Kusters, and Tim A. C. Willemse. Analysing the control software of the compact muon solenoid experiment at the large hadron collider. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, pages 174–189, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] N.N. `https://www.mcrl2.org` - analysing system behaviour, 2019. accessed December 13th, 2019.

[14] U.S. Department of Defense. MIL-STD-498 Military Standard Software Development. Standard, U.S. Department of Defense, USA, December 1994.

[15] Daniela Remenska, Tim A.C. Willemse, Kees Verstoep, Jeff Templon, and Henri Bal. Using model checking to analyze the system behavior of the LHC production grid. *Future Generation Computer Systems*, 29(8):2239 – 2251, 2013.

[16] Martin Renkema. Software Requirements Specification; Astris - Elementen Component. Standard, ProRail, November 2019.

[17] Martin Renkema and Bert Bossink. Software Requirements Specification; Astris - Rijwegen Component. Standard, ProRail, Januari 2020.

[18] F. Reussink. Interface Requirements Specification / Design Description TCS-BEVNL. Standard, ProRail, November 2016.

[19] Ian Sommerville. Formal specification. In *Software Engineering*, chapter 27. Addison-Wesley Publishing Company, USA, 9th edition, 2010.

[20] Wikipedia contributors. Prorail — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=ProRail&oldid=923130527`, 2019. accessed December 4th, 2019.

Appendices

Appendix A

mCRL2 model Chapter 4

Back to the text

```
%% Locking model; Chapter 4 %%
sort
  componentStatus = struct Actief ?  isActief
              | InternActief ?  isInternActief
              | Passief ?  isPassief
              | Geinstantieerd ?  isGeinstantieerd
              | NietActief ?  isNietActief;
  Acceptatie = struct Geaccepteerd | NietGeaccepteerd;
  ElementID = struct Wissel1 | Wissel2 | Sein1 | Sein2;
  ElementSet = Set(ElementID);
  ElementList = List(ElementID);
  OpdrachtID = struct RW1 | RW2;
  Lock = struct Pair(element:ElementID, opdr:OpdrachtID);
  Locks = Set(Lock);

%% equations for the lock and unlock conditions
map
  lockCondition :  ElementID # OpdrachtID # Locks -> Bool;
  unlockCondition :  ElementID # Locks -> Bool;
var
  ep:ElementID;
  op:OpdrachtID;
  ll:Locks;
eqn
  lockCondition(ep, op, ll) =
    !(exists opp:OpdrachtID . (Pair(ep, opp) in ll && !(opp == op)));
  unlockCondition(ep, ll) = exists opp:OpdrachtID . Pair(ep, opp) in ll;

act

%% Respons acties
  Respons, R_Respons, S_Respons :  Acceptatie # OpdrachtID;

%% Placeholder(s) for yet to fill in details
  placeholder;
```

```
  skip;

proc
%% Initialisation of Element Component
initElementComponent(el:ElementSet) = ElementComponent(Actief, {}, el);

ElementComponent(s:componentStatus, l:Locks, el:ElementSet) =
  ElementGebruikService(s, l, el);

act
  LockElement, R_LockElement, S_LockElement :  ElementID # OpdrachtID;
  UnlockElement, R_UnlockElement, S_UnlockElement :  ElementID # OpdrachtID;
  locked, unlocked :  ElementID # OpdrachtID;
proc
ElementGebruikService(s:componentStatus, l:Locks, el:ElementSet) =
  (sum e:ElementID . sum o :  OpdrachtID . (e in el) -> (R_LockElement(e, o) .
    ((isActief(s)||isInternActief(s)) -> ((lockCondition(e, o, l)) -> locked(e, o) .
    S_Respons(Geaccepteerd, o) <> S_Respons(NietGeaccepteerd, o)) <>
    S_Respons(NietGeaccepteerd, o)) .
    ElementComponent(s,if(lockCondition(e,o,l),l+{Pair(e,o)},l), el))) +
  (sum e:ElementID . sum o :  OpdrachtID . (e in el) -> (R_UnlockElement(e, o) .
    ((isActief(s)||isInternActief(s)) -> ((unlockCondition(e, l)) -> unlocked(e, o) <>
    skip) .  S_Respons(Geaccepteerd, o) <> S_Respons(NietGeaccepteerd, o)) .
    ElementComponent(s,l - {a:Lock | element(a) == e}, el)));

proc
%% Environment
initEnvironment = Environment(Actief);

Environment(s:componentStatus) =
  EnvironmentElementService(s) +
  (sum o :  OpdrachtID . sum a :  Acceptatie .  R_Respons(a, o) .  Environment(s)) +
  (sum o :  OpdrachtID . sum a :  Acceptatie .  S_Respons(a, o) .  Environment(s));

EnvironmentElementService(s:componentStatus) =
  (sum e :  ElementID . sum o :  OpdrachtID . S_LockElement(e, o) .  Environment(s)) +
  (sum e :  ElementID . sum o :  OpdrachtID . S_UnlockElement(e, o) .  Environment(s));


init
hide({skip},
allow({
locked,unlocked,
Respons,
LockElement, UnlockElement,
skip, placeholder
},
comm({
S_Respons|R_Respons->Respons,
S_LockElement|R_LockElement->LockElement,
S_UnlockElement|R_UnlockElement->UnlockElement
```

```
},
%% Locking model
%initElementComponent({Wissel1}) || initEnvironment ))); %% one element
%initElementComponent({Wissel1, Wissel2}) || initEnvironment ))); %% two elements
initElementComponent({Wissel1, Wissel2, Sein1}) || initEnvironment ))); %% three
  elements
%initElementComponent({Wissel1, Wissel2, Sein1, Sein2}) || initEnvironment ))); %% four
  elements
```

Appendix B

mCRL2 model Chapter 5

Back to the text

```
%% Route conflict resolution model; Chapter 5 %%
sort
  componentStatus = struct Actief ?  isActief
              | InternActief ?  isInternActief
              | Passief ?  isPassief
              | Geinstantieerd ?  isGeinstantieerd
              | NietActief ?  isNietActief;
  Acceptatie = struct Geaccepteerd | NietGeaccepteerd;
  ElementID = struct Wissel1 | Wissel2 | Sein1 | Sein2;
  ElementSet = Set(ElementID);
  ElementList = List(ElementID);
  OpdrachtID = struct RW1 | RW2;
  OpdrachtIDList = List(OpdrachtID);
  Lock = struct Pair(element:ElementID, opdr:OpdrachtID);
  Locks = Set(Lock);

%% equations for the lock and unlock conditions
map
  lockCondition :  ElementID # OpdrachtID # Locks -> Bool;
  unlockCondition :  ElementID # Locks -> Bool;
var
  ep:ElementID;
  op:OpdrachtID;
  ll:Locks;
eqn
  lockCondition(ep, op, ll) =
    !(exists opp:OpdrachtID . (Pair(ep, opp) in ll && !(opp == op)));
  unlockCondition(ep, ll) = exists opp:OpdrachtID . Pair(ep, opp) in ll;

%% equation to allow no double elements in rijwegen.  So, a list like [Sein1, Sein1] is
  not allowed.
map
  RijwegElementCondition :  ElementList -> Bool;
var
  a:ElementID;
```

```
  b:ElementList;
eqn
  RijwegElementCondition([]) = true;
  RijwegElementCondition(a |> b) = !(a in b) && RijwegElementCondition(b);

%% equation to get the intersection size of two ElementLists
map
  listIntersectSize :  ElementList # ElementList -> Nat;
var
  e1, e2 :  ElementList;
  e, ep :  ElementID;
eqn
  listIntersectSize([], e2) = 0;
  listIntersectSize(e |> e1, e2) = if(e in e2, listIntersectSize(e1, e2) + 1,
    listIntersectSize(e1, e2));

act
%% Respons acties
  Respons, R_Respons, S_Respons :  Acceptatie # OpdrachtID;

%% Placeholder(s) for yet to fill in details
  placeholder, terminate;
  sync, S_sync, R_sync;
  skip;
  fail, success :  OpdrachtID;

proc
initComponent(el:ElementSet) = Component(Actief, {}, el);


Component(s:componentStatus, l:Locks, el:ElementSet) =
  ElementGebruikService(s, l, el);
  %BeheerService(s, l);

act
  LockElement, R_LockElement, S_LockElement :  ElementID # OpdrachtID;
  UnlockElement, R_UnlockElement, S_UnlockElement :  ElementID # OpdrachtID;
  locked, unlocked :  ElementID # OpdrachtID;
  verwijderLockTimeout :  ElementID;
proc
ElementGebruikService(s:componentStatus, l:Locks, el:ElementSet) =
  %(sum e:ElementID . sum o :  OpdrachtID . (e in el) -> (Pair(e,o) in l) ->
    verwijderLockTimeout(e) .  Component(s,l - {a:Lock | element(a) == e}, el)) +
  (sum e:ElementID . sum o :  OpdrachtID . (e in el) -> (R_LockElement(e, o) .
    ((isActief(s)||isInternActief(s)) -> ((lockCondition(e, o, l)) -> locked(e, o) .
    S_Respons(Geaccepteerd, o) <> S_Respons(NietGeaccepteerd, o)) <>
    S_Respons(NietGeaccepteerd, o)) .
    Component(s,if(lockCondition(e,o,l),l+{Pair(e,o)},l), el))) +
  (sum e:ElementID . sum o :  OpdrachtID . (e in el) -> (R_UnlockElement(e, o) .
    ((isActief(s)||isInternActief(s)) -> ((unlockCondition(e, l)) -> unlocked(e, o) <>
    skip) .  S_Respons(Geaccepteerd, o) <> S_Respons(NietGeaccepteerd, o)) .
```

```
    Component(s,l - {a:Lock | element(a) == e}, el)));

proc
%% Environment
initEnvironment = Environment(Actief);

Environment(s:componentStatus) =
  EnvironmentElementService(s) +
  (sum o :  OpdrachtID . R_Respons(Geaccepteerd, o) .  Environment(s)) +
  (sum o :  OpdrachtID . R_Respons(NietGeaccepteerd, o) .  Environment(s)) +
  (sum o :  OpdrachtID . S_Respons(Geaccepteerd, o) .  Environment(s)) +
  (sum o :  OpdrachtID . S_Respons(NietGeaccepteerd, o) .  Environment(s));

EnvironmentElementService(s:componentStatus) =
  (sum e :  ElementID . sum o :  OpdrachtID . S_LockElement(e, o) .  Environment(s)) +
  (sum e :  ElementID . sum o :  OpdrachtID . S_UnlockElement(e, o) .  Environment(s));


%% In the condition below, i.e.  '  (#el >= 1 && #el <= 2)' we may change the amount of
  elements in the route for this RC
LockElementen(opdrID:OpdrachtID) =
  sum el:ElementList .  (#el >= 1 && #el <= 2) -> (RijwegElementCondition(el)) ->
    Poging(1, el, el, opdrID);

Poging(Poging_n:Int, el:ElementList, original:ElementList, opdrID:OpdrachtID) =
  (Poging_n > 0) ->
    S_LockElement(head(el), opdrID) . DoeLock(Poging_n, tail(el), original, opdrID)
  <>
    fail(opdrID).(R_sync + S_sync).LOOP;

DoeLock(Poging_n:Int, el:ElementList, original:ElementList, opdrID:OpdrachtID) =
(#original == 0) ->
  Poging(0, el, original, opdrID) <>
  (sum a :  Acceptatie .  R_Respons(a, opdrID) . (a == NietGeaccepteerd) ->
      DoeUnlock(Poging_n, el, original, opdrID) <>
      (#el == 0 && #original > 0) ->
        success(opdrID).(R_sync + S_sync).LOOP <>
        S_LockElement(head(el), opdrID) . DoeLock(Poging_n, tail(el), original,
          opdrID));

DoeUnlock(Poging_n:Int, el:ElementList, original:ElementList, opdrID:OpdrachtID) =
  (#el + 1 == #original) ->
    Poging((Poging_n - 1), original, original, opdrID)
  <>
    S_UnlockElement(head(original), opdrID) . R_Respons(Geaccepteerd, opdrID) .
      DoeUnlock(Poging_n, el, tail(original), opdrID);

%% Sink state
LOOP = terminate.LOOP;

init
```

```
hide({skip},
allow({
locked,unlocked,
Respons,
LockElement, UnlockElement,
skip, placeholder, verwijderLockTimeout,
fail, success, sync
},
comm({
S_Respons|R_Respons->Respons,
S_LockElement|R_LockElement->LockElement,
S_UnlockElement|R_UnlockElement->UnlockElement,
S_sync|R_sync->sync
},
% Models with elements distributed differently over the ECs.  The LockElementen process
  represents the RC
%initComponent({Wissel1, Wissel2, Sein1, Sein2}) || LockElementen(RW1) ||
  LockElementen(RW2) )));
initComponent({Wissel1, Wissel2}) || initComponent({Sein1, Sein2}) || LockElementen(RW1)
  || LockElementen(RW2) )));
%initComponent({Wissel1}) || initComponent({Wissel2, Sein1, Sein2}) ||
  LockElementen(RW1) || LockElementen(RW2) )));
```

Appendix C

mCRL2 model Chapter 6

Back to the text

```
%% Coupled elements model(s); Chapter 6 %%

sort
  componentStatus = struct Actief ?  isActief
              | InternActief ?  isInternActief
              | Passief ?  isPassief
              | Geinstantieerd ?  isGeinstantieerd
              | NietActief ?  isNietActief;
  Acceptatie = struct Geaccepteerd | NietGeaccepteerd;
  ElementID = struct Wissel1 | Wissel2;% | Sein1 | Sein2;
  ElementSet = Set(ElementID);
  ElementList = List(ElementID);
  OpdrachtID = struct RW1 | RW2;
  OpdrachtIDList = List(OpdrachtID);
  Lock = struct Pair(element:ElementID, opdr:OpdrachtID);
  Locks = Set(Lock);

%% equations for the lock and unlock conditions
map
  lockCondition :  ElementID # OpdrachtID # Locks -> Bool;
  unlockCondition :  ElementID # Locks -> Bool;
var
  ep:ElementID;
  op:OpdrachtID;
  ll:Locks;
eqn
  lockCondition(ep, op, ll) =
    !(exists opp:OpdrachtID . (Pair(ep, opp) in ll && !(opp == op)));
  unlockCondition(ep, ll) = exists opp:OpdrachtID . Pair(ep, opp) in ll;

%% equation to allow no double elements in rijwegen.  So, a list like [Sein1, Sein1] is
  not allowed.
map
  RijwegElementCondition :  ElementList -> Bool;
var
```

```
  a:ElementID;
  b:ElementList;
eqn
  RijwegElementCondition([]) = true;
  RijwegElementCondition(a |> b) = !(a in b) && RijwegElementCondition(b);

%% equation to get the intersection size of two ElementLists
map
  listIntersect :  ElementList # ElementList -> ElementList;
  listUnion :  ElementList # ElementList -> ElementList;
  listIntersectSize :  ElementList # ElementList -> Nat;
var
  e1, e2 :  ElementList;
  e :  ElementID;
eqn
  listIntersect([], e2) = [];
  listIntersect(e |> e1, e2) = if(e in e2, listIntersect(e1, e2) ++ [e],
    listIntersect(e1, e2));
  listUnion([], e2) = e2;
  listUnion(e |> e1, e2) = if(e in e2, listUnion(e1, e2), listUnion(e1, e2) ++ [e]);
  listIntersectSize([], e2) = 0;
  listIntersectSize(e |> e1, e2) = if(e in e2, listIntersectSize(e1, e2) + 1,
    listIntersectSize(e1, e2));

%% sorts, maps and such for coupled elements (gekoppelde elementen)
sort
  ElementPair = struct Pair(element1:ElementID, element2:ElementID);
  CoupledElements = List(ElementPair);
map
  getCoupledElement :  CoupledElements # ElementID -> ElementList;
  noMultiplePairs :  CoupledElements -> Bool;
var
  ce :  CoupledElements;
  pair :  ElementPair;
  e :  ElementID;
eqn
  getCoupledElement([], e) = [];
  getCoupledElement(pair |> ce, e) = if(element1(pair)==e, % If e is in the first part
    of the pair
                      element2(pair) |> getCoupledElement(ce, e),
                    if(element2(pair)==e, % Else if e in the second part of the pair
                      element1(pair) |> getCoupledElement(ce, e),
                    getCoupledElement(ce, e))); % Else, just go on with the 'search'
  %% We learned that [] |> [] is not allowed
%% equation that says the CoupledElements set may not contain doubles of pair.
%% e.g.  for every (a,b) in '(a,b) |> c', it is not the case that (a,b) in c or (b,a) in
  c
  noMultiplePairs([]) = true;
  noMultiplePairs(pair |> ce) = !(pair in ce || Pair(element2(pair), element1(pair)) in
    ce || element2(pair) == element1(pair)) && noMultiplePairs(ce);
```

```
act
%% Respons acties
  Respons, R_Respons, S_Respons :  Acceptatie # OpdrachtID;
  InterECRespons, R_InterECRespons, S_InterECRespons :  Acceptatie # OpdrachtID;
  Respons_GekoppeldeElementen, R_Respons_GekoppeldeElementen,
    S_Respons_GekoppeldeElementen :  Acceptatie # OpdrachtID # ElementList;

%% Placeholder(s) for yet to fill in details
  placeholder, terminate;
  sync, S_sync, R_sync;
  skip;
  fail, success :  OpdrachtID;

proc
initComponent(el:ElementSet) = sum ce :  CoupledElements .  (noMultiplePairs(ce)) ->
  ElementComponent(Actief, {}, el, ce);


ElementComponent(s:componentStatus, l:Locks, el:ElementSet, ce:CoupledElements) =
  ElementGebruikService(s, l, el, ce) +
  ElementConfiguratieService(s, l, el, ce);

act
  LockElement, R_LockElement, S_LockElement :  ElementID # OpdrachtID;
  UnlockElement, R_UnlockElement, S_UnlockElement :  ElementID # OpdrachtID;
  NeemElementInGebruik, R_NeemElementInGebruik, S_NeemElementInGebruik :  ElementID #
    OpdrachtID # Bool;
  MeldGebruik, R_MeldGebruik, S_MeldGebruik;
  locked, unlocked, verwijderLockTimeout :  ElementID # OpdrachtID;
proc %% The process containing all of the service requests belonging in the 'usage' set
ElementGebruikService(s:componentStatus, l:Locks, el:ElementSet, ce:CoupledElements) =
%% neemElementInGebruik / takeElementIntoUse service request
  (sum e:ElementID . sum o :  OpdrachtID . sum b:Bool .  (e in el) ->
    (R_NeemElementInGebruik(e, o, b) .
    (((#getCoupledElement(ce, e) > 0) && !(head(getCoupledElement(ce, e)) in el) && !b)
      ->
      placeholder .  S_NeemElementInGebruik(head(getCoupledElement(ce, e)), o, true) .
        sum a :  Acceptatie .  R_InterECRespons(a, o) <> skip) .
  ((!b) -> S_Respons(Geaccepteerd, o) <> S_InterECRespons(Geaccepteerd, o)) .
    ElementComponent(s, l, el, ce))) +
%% verwijderLockTimeout / deleteLockTimeout - comment/uncomment to disable/enable
  % (sum e:ElementID . sum o :  OpdrachtID . (e in el) -> (Pair(e,o) in l) ->
    verwijderLockTimeout(e, o) .  ElementComponent(s,l - {a:Lock | element(a) == e}, el,
    ce)) +
%% LockElement servcie request
  (sum e:ElementID . sum o :  OpdrachtID . (e in el) -> (R_LockElement(e, o) .
    ((isActief(s)||isInternActief(s)) -> ((lockCondition(e, o, l)) -> locked(e, o) .
    S_Respons(Geaccepteerd, o) <> S_Respons(NietGeaccepteerd, o)) <>
    S_Respons(NietGeaccepteerd, o)) .
    ElementComponent(s,if(lockCondition(e,o,l),l+{Pair(e,o)},l), el, ce))) +
%% Unlock Element service request
```

```
  (sum e:ElementID . sum o :  OpdrachtID . (e in el) -> (R_UnlockElement(e, o) .
    ((isActief(s)||isInternActief(s)) -> ((unlockCondition(e, l)) -> unlocked(e, o) <>
    skip) .  S_Respons(Geaccepteerd, o) <> S_Respons(NietGeaccepteerd, o)) .
    ElementComponent(s,l - {a:Lock | element(a) == e}, el, ce)));

act
  geefGekoppeldElement, R_geefGekoppeldElement, S_geefGekoppeldElement :  ElementID #
    OpdrachtID;

proc %% The process containing all the service requests belonging in the 'configuration'
  set
ElementConfiguratieService(s:componentStatus, l:Locks, el:ElementSet,
  ce:CoupledElements) =
  (sum e:ElementID . sum o:OpdrachtID . (e in el) -> R_geefGekoppeldElement(e, o) .
    ((isActief(s)||isInternActief(s)) -> S_Respons_GekoppeldeElementen(Geaccepteerd, o,
    getCoupledElement(ce, e)) <> S_Respons_GekoppeldeElementen(NietGeaccepteerd, o, []))
    .  ElementComponent(s, l, el, ce) );

proc
%% Environment
initEnvironment = Environment(Actief);

Environment(s:componentStatus) =
  EnvironmentElementService(s) +
  EnvironmentElementConfiguratieService(s) +
  %% General responses
  (sum a :  Acceptatie .  sum o :  OpdrachtID . R_Respons(a, o) .  Environment(s)) +
  (sum a :  Acceptatie .  sum o :  OpdrachtID . S_Respons(a, o) .  Environment(s));

EnvironmentElementService(s:componentStatus) =
  (sum e :  ElementID . sum o :  OpdrachtID . S_NeemElementInGebruik(e, o, false) .
    Environment(s)) +
%% Comment the above and uncomment the below for the assumption of chapter 6
  %(sum e :  ElementID . sum o :  OpdrachtID . S_NeemElementInGebruik(Wissel1, o, false) .
    Environment(s)) +
  (sum e :  ElementID . sum o :  OpdrachtID . S_LockElement(e, o) .  Environment(s)) +
  (sum e :  ElementID . sum o :  OpdrachtID . S_UnlockElement(e, o) .  Environment(s));

EnvironmentElementConfiguratieService(s:componentStatus) =
  (sum e :  ElementID . sum o :  OpdrachtID . S_geefGekoppeldElement(e, o) .
    Environment(s)) +
  (sum a :  Acceptatie .  sum o :  OpdrachtID . sum el :  ElementList .
    R_Respons_GekoppeldeElementen(a, o, el) .  Environment(s));

% #el > 0 && #el < 3
LockElementsInRoute(opdrID:OpdrachtID) = %% If the ElementList el satisfies the
  criteria, then make an attempt to lock the list el
  sum el:ElementList .  (#el == 1) -> (RijwegElementCondition(el)) -> Attempt(1, el, el,
    opdrID);

proc
```

```
%% An attempt at locking elements
Attempt(Attempt_n:Int, el:ElementList, original:ElementList, opdrID:OpdrachtID) =
  (Attempt_n > 0) ->
    S_LockElement(head(el), opdrID) . LockElementList(Attempt_n, tail(el), original,
      opdrID)
  <>
    fail(opdrID).(R_sync + S_sync).LOOP;

%% Lock a list of Elements.  Attempt_n is potentially used in UnlockElementList
LockElementList(Attempt_n:Int, el:ElementList, original:ElementList, opdrID:OpdrachtID)
  =
(#original == 0) ->
  Attempt(0, el, original, opdrID) <>
  (sum a :  Acceptatie .  R_Respons(a, opdrID) . (a == NietGeaccepteerd) ->
      UnlockElementList(Attempt_n, el, original, opdrID) <>
      (#el == 0 && #original > 0) ->
        success(opdrID) . (R_sync + S_sync).LOOP <>
        S_LockElement(head(el), opdrID) . LockElementList(Attempt_n, tail(el), original,
          opdrID));

%% Unlock a list of Elements
UnlockElementList(Attempt_n:Int, el:ElementList, original:ElementList,
  opdrID:OpdrachtID) =
  (#el + 1 == #original) ->
    Attempt((Attempt_n - 1), original, original, opdrID)
  <>
    S_UnlockElement(head(original), opdrID) . R_Respons(Geaccepteerd, opdrID) .
      UnlockElementList(Attempt_n, el, tail(original), opdrID);

%% Sub-routine for locking all elements in some list and all the gekoppelde elementen
  /coupled elements.  This is a naive version where we consider only one initial element
%% This is a prototype process for the below
LockElementAndCoupledElement(opdrID:OpdrachtID) = sum e:ElementID .
  S_geefGekoppeldElement(e, opdrID) . sum ce :  ElementList .
  R_Respons_GekoppeldeElementen(Geaccepteerd, opdrID, ce) .  (#(e|>ce) >= 1) ->
  (RijwegElementCondition(e|>ce)) -> Attempt(1, e|>ce, e|>ce, opdrID);

LockElementsInRouteAndCoupledElements(opdrID:OpdrachtID) =
  sum el:ElementList .  (#el > 0 && #el < 3 && RijwegElementCondition(el)) ->
    BuildListOfElementAndCoupledElements(opdrID, el, []);

% In this process, some arbitrary list el is given.  Then, the list of elements that is
  to be locked is build , i.e.  list elp.  When el has been exhausted (empty) we can lock
  the elements in the elp list (i.e.  the list of elements including all the coupled
  elements.  List elp must be initially empty
BuildListOfElementAndCoupledElements(opdrID:OpdrachtID, el :  ElementList,
  elp:ElementList) =
(#el > 0) ->
  S_geefGekoppeldElement(head(el), opdrID) .
    sum ce :  ElementList .  R_Respons_GekoppeldeElementen(Geaccepteerd, opdrID, ce) .
    BuildListOfElementAndCoupledElements(opdrID, tail(el),
```

```
      listUnion(listUnion([head(el)], elp), ce) )
  <>
  (RijwegElementCondition(elp) && #elp >= 1) -> Attempt(1, elp, elp, opdrID);

%% LOOP process for a sink state
LOOP = terminate.LOOP;

init
hide({skip},
allow({
locked,unlocked,
Respons, InterECRespons, Respons_GekoppeldeElementen,
LockElement, UnlockElement, NeemElementInGebruik,
skip, placeholder, verwijderLockTimeout,
fail, success, sync,
geefGekoppeldElement
},
comm({
S_Respons|R_Respons->Respons,
S_InterECRespons|R_InterECRespons->InterECRespons,
S_Respons_GekoppeldeElementen|R_Respons_GekoppeldeElementen->
  Respons_GekoppeldeElementen,
S_LockElement|R_LockElement->LockElement,
S_UnlockElement|R_UnlockElement->UnlockElement,
S_NeemElementInGebruik|R_NeemElementInGebruik->NeemElementInGebruik,
S_sync|R_sync->sync,
S_geefGekoppeldElement|R_geefGekoppeldElement->geefGekoppeldElement
},
%%Lock coupled elements in route model (similar to Route conflict model)
initComponent({Wissel1}) || initComponent({Wissel2}) ||
  LockElementsInRouteAndCoupledElements(RW1) ||
  LockElementsInRouteAndCoupledElements(RW2) )));

% Take coupled elements into use model
%initComponent({Wissel1, Wissel2}) || initEnvironment ))); %% no deadlock
%initComponent({Wissel1}) || initComponent({Wissel2}) || initEnvironment ))); %%
  deadlock
% Same model but with control over parameters (for debugging purposes)
%ElementComponent(Actief, {}, {Wissel1, Wissel2}, [Pair(Wissel1, Wissel2)]) ||
  initEnvironment ))); %% no deadlock
%ElementComponent(Actief, {}, {Wissel1}, [Pair(Wissel1, Wissel2)]) ||
  ElementComponent(Actief, {}, {Wissel2}, [Pair(Wissel1, Wissel2)]) || initEnvironment
  ))); %% deadlock
```

Appendix D

mCRL2 model Chapter 7

Back to the text

```
sort
RouteID = struct R1;

Activity = struct Active | NotActive;

RouteState = struct
  Rest | % Rust
  Reserved | % Gereserveerd
  Being_prepared | % Wordt voorbereid
  Prepared | % Voorbereid
  CommandsRegimes | % StuurtRegimes
  Wait_for_stoptijd_expired | % Wacht op Stoptijd Verstreken
  Setting_command_sent_to_BevNL | % Instelopdracht verstuurd naar BevNL
  Setting_command_refuesed_by_BevNL | % Instelopdracht afgekeurd door BevNL
  Setting_and_cancelRoute_command_sent_to_BevNL | % Instel- en herroepopdracht verstuurd
    naar BevNL
  Route_bBeing_set | % Wordt ingesteld
  Route_set| % Ingesteld (sein uit de stand stop)
  Ride_off| % Afrijden
  Revoking | % Herroepen
  STS_route_being_set | % STS-wordt ingesteld
  STS_route |
  VCVL_route_set; % VCVL-rijweg ingesteld

SignalOperation = struct Normaal | RijdenOpZicht | Automaat; % SignaalBediening

Route = struct Route( % From Rijweg SRS
  routeID : RouteID,
  %followNumber :  DummySort,
  state :  RouteState,
  RR : Activity, % Reservation route / Reserveren Rijweg
  VBR : Activity, % Preparing route / VoorBereiden Rijweg
  IR : Activity, % setting route / Instellen Rijweg
  H : Activity, % Cancel Route / Herroepen
  A : Activity, % Cancelling route / annuleren rijweg
```

```
  AU : Activity, % Automatic / Automaat
  RVR : Activity, % Rijweg VrijRijden
  STS : Activity, % STS route
  A_STS : Activity, % Cancelling STS-route / Annuleren STS-route
  SDE : Activity, % Emplacement stopdoor
  GC : Activity, % Cargo criterium / GoederenCriterium
  SDV1 :  Activity, % Vrijebaan stopdoor 1
  SDV2 :  Activity, % Vrijebaan stopdoor 2
  SDV3 :  Activity, % Vrijebaan stopdoor 3
  SDV4 :  Activity, % Vrijebaan stopdoor 4
  %RouteCommand :  DummySort % optional
  signalOperation :  SignalOperation
  %ActiveRouteCommand :  DummySort, % optional
  %InfraUser :  DummySort % optional
);

ElementID = struct E1;
% Translation of the below items is not entirely correct
%ElementType = struct
  %Point | % Wissel
  %Crossing | % Kruising
  %CatenaryCounter | % Bovenleidingsteller
  %Stopdoor | % Stopdoor
  %CargoCriterium | % Goederencriterium
  %Section | % Sectie
  %Signal | % Sein
  %StopDerailmentBlock | % Stopontspoorblok
  %DerailedTongue | % Ontspoortong
  %SignalLighting | % SeinVerlichting
  %DirectionOfTravelBarrier | % Rijrichtingkering
  %LevelCrossing | % Overweg
  %Lever; % Grendel

SignalAttributeType = struct Normal | Fictional_Begin | Fictional_End;

ElementConf = struct ElementConf( % From CONF IRS ‘‘Lijst van elementen’’
  %elementId :  ElementID,
  %elementType :  DummySort, %ElementType,
  operable :  Bool,
  sectionAttributeDetection :  Bool,
  %sectionAttributeVrijebaansectie :  Bool,
  %sectionAttributeSplitCombine :  Bool,
  signalAttributeType :  SignalAttributeType,
  %signalAttributeOpenTrackSecurityType :  DummySort,
  %directionOfTravelBarrierAttributeOperationside :  Bool,
  %leverAttributeReclaimPossible :  Bool,
  %crossingAttributePresentInSecurity :  Bool,
  reportedByBevNL : Bool
  %preferredStateAutomatic :  Bool
);
```

```
ReservedUse = struct Herroepen;
UseTuple = struct UseTuple(reservedUse :  ReservedUse, currentUse :  Bool, routeID :
  RouteID); % The ActueelGebruik has only one possible value 'InRijweg'.  As such, it
  would never be anything else.  But intuitively we understand it can also not be this.
  Therefore we model it as a boolean
UseList = List(UseTuple);

ElementAstris = struct ElementAstris( % From Element SRS
  element :  ElementID,
  use :  UseList
);

% Element object types of only the types we are considering in the model
% ElementObjectType = struct Section | Signal | FictionalSignal | Regime;

%RouteConstraint = struct N | P | T | C;
InRoute = struct East | South | West | North | ESWIN | dash_NotPartOfRoute |
  Unknown_IsPartOfRoute;
Activation = struct dash_No_routeCommand | N_Normal_routeCommand |
  R_OnSight_routeCommand;
Reserved_Begin = struct dash_Not_used | N_Proceed | R_DriveOnSight | A_Automatic | S_STS
  | H_Cancelled | Unknown_Reserved_Begin;
Reserved_RW = struct dash_Not_Used | Off_rail | Unknown_Reserved_RW;

ElementObject = struct ElementObject( % From infrastructure (BEVNL-IRS)
% Assumed relational referencing of object via ElementID
  %elementID : ElementID, % used by all
  % Abstraction of different types of objects by type now centralised in one
    datastructure
  %type :  ElementObjectType,
% value properties
  occupied :  Bool, % used by Section,
  auth_activation :  Activation, % used by Signal, FictionalSignal
  vertrekBlokBezet :  Bool, % used by FictionalSignal
  stoptimeExpired :  Bool, % used by Regime
  disturbed :  Bool, % used by Section, Singal, FictionalSignal, Regime
% usage properties
  inRoute :  InRoute, % used by Section
  logicallyOccupied :  Bool, % used by Section
  reserved_RW : Reserved_RW, % used by operable elements, though not all elements have
    this attribute
  reserved_begin :  Reserved_Begin, % used by Signal, FictionalSignal
  reserved_end :  Bool, % used by Signal, FictionalSignal
% constraint properties
  %routeConstraint :  RouteConstraint, % used by section
  %constraintRoute_active :  DummySort, % used by Regime
  %constraintRoute_passive :  DummySort, % used by Regime
  reliable :  Bool % used by Signal Section, Regime
);

ElementInRouteType = struct Section | Signal; % Only elements that we consider
```

```
ElementByRouteType = struct Regime_EmplacementStopdoor | Regime_OpenTrackStopdoor |
  Regime_CargoCriterium; % Only element that we consider.  Regime has sub-types
  Emplacement Stopdoor, vrijebaan stopdoor and GoederenCriterium

ElementInRouteRole = struct begin_signal | end_signal;
ElementByRouteRole = struct Element_required | Element_request | Regime_Control |
  Regime_Regime; % Eis element, verzoek element, regime controle, regime regime

ElementInRoute = struct ElementInRoute(
  elementID : ElementID,
  elementConf :  ElementConf,
  elementObject :  ElementObject,
  elementAstris :  ElementAstris,
  elementInRouteType :  ElementInRouteType,
  elementInRouteRole :  ElementInRouteRole);
ElementByRoute = struct ElementByRoute(
  elementID : ElementID,
  elementConf :  ElementConf,
  elementObject :  ElementObject,
  elementAstris :  ElementAstris,
  elementByRouteType :  ElementByRouteType,
  elementByRouteRole :  ElementByRouteRole);

ElementInRouteList = List(ElementInRoute);
ElementByRouteList = List(ElementByRoute);

AllElementsInAndByRoute = struct AllElementsInAndByRoute(
routeID : RouteID,
% The below is an abstraction.  The Notions of elements in and around the route are
  actually part of a list of sub-routes that contains single elementInRouteID and
  ElementAroundRouteID
elementInRoute :  ElementInRouteList,
elementByRoute :  ElementByRouteList
);

map
  pred_A : Route -> Bool;
  pred_B : Route # AllElementsInAndByRoute -> Bool;
  pred_C : AllElementsInAndByRoute -> Bool;
  pred_D : Route # AllElementsInAndByRoute -> Bool;
  pred_E : Route # AllElementsInAndByRoute -> Bool;
  pred_F : Route # AllElementsInAndByRoute -> Bool;
  pred_G : Route # AllElementsInAndByRoute -> Bool;
var
  r :  Route;
  elementsInAndByRoute :  AllElementsInAndByRoute;
  dummy :  Bool;
eqn
  pred_A(r) = (state(r) == Prepared && IR(r) == Active && H(r) == NotActive && A(r) ==
    NotActive) ||
    (state(r) == Ride_off && H(r) == NotActive);
```

```
pred_B(r, elementsInAndByRoute) = !(SDE(r) == Active &&
  (exists e :  ElementByRoute .  (e in elementByRoute(elementsInAndByRoute) &&
    elementByRouteType(e) == Regime_EmplacementStopdoor) &&
      !stoptimeExpired(elementObject(e)))
  && GC(r) == Active &&
  (SDV1(r) == Active || SDV2(r) == Active || SDV3(r) == Active || SDV4(r) == Active)
    );

pred_C(elementsInAndByRoute) = forall e_in :  ElementInRoute .  forall e_by :
  ElementByRoute .
  ((e_in in elementInRoute(elementsInAndByRoute) &&
  e_by in elementByRoute(elementsInAndByRoute)) &&
  (sectionAttributeDetection(elementConf(e_in)) && !reportedByBevNL(elementConf(e_in))
    && !reportedByBevNL(elementConf(e_by)) && elementByRouteRole(e_by) !=
    Element_request)) =>
  (reliable(elementObject(e_in)) &&
  reliable(elementObject(e_by)));

  pred_D(r, elementsInAndByRoute) = (signalOperation(r) == Automaat ||
    signalOperation(r) == Normaal) => ((!exists e :  ElementInRoute .
  ((e in elementInRoute(elementsInAndByRoute)) && elementInRouteType(e) == Section &&
    occupied(elementObject(e)))) &&
  (!exists e :  ElementInRoute .
  ((e in elementInRoute(elementsInAndByRoute)) && elementInRouteType(e) == Section &&
    logicallyOccupied(elementObject(e)))) &&
  (exists e :  ElementInRoute .
  (e in elementInRoute(elementsInAndByRoute)) && elementInRouteType(e) == Signal &&
    elementInRouteRole(e) == begin_signal && (exists u :  UseTuple .  u in
    use(elementAstris(e)) && reservedUse(u) == Herroepen && routeID(u) == routeID(r))
    && !(forall ep :  ElementInRoute .  (ep in elementInRoute(elementsInAndByRoute) &&
    elementInRouteType(ep) == Section) => (forall u :  UseTuple .  u in
    use(elementAstris(e)) => (currentUse(u) && routeID(u) != routeID(r))) ) ) &&
  (exists e :  ElementInRoute .
  (e in elementInRoute(elementsInAndByRoute)) && elementInRouteType(e) == Signal &&
    elementInRouteRole(e) == begin_signal && !(exists u :  UseTuple .  u in
    use(elementAstris(e)) && reservedUse(u) == Herroepen && routeID(u) == routeID(r))
    && !(forall ep :  ElementInRoute .  (ep in elementInRoute(elementsInAndByRoute) &&
    elementInRouteType(ep) == Section) => (forall u :  UseTuple .  u in
    use(elementAstris(e)) => currentUse(u)) ) )
  );

pred_E(r, elementsInAndByRoute) = (signalOperation(r) == Automaat) => (forall e :
  ElementInRoute .
  (e in elementInRoute(elementsInAndByRoute) && elementInRouteType(e) == Section) =>
    (inRoute(elementObject(e)) == dash_NotPartOfRoute || inRoute(elementObject(e)) ==
    Unknown_IsPartOfRoute));

pred_F(r, elementsInAndByRoute) = exists e :  ElementInRoute .  ((signalOperation(r) ==
  Normaal && (e in elementInRoute(elementsInAndByRoute)) &&
  signalAttributeType(elementConf(e)) == Fictional_End) &&
```

```
    vertrekBlokBezet(elementObject(e)));

  pred_G(r, elementsInAndByRoute) = (signalOperation(r) == Normaal || signalOperation(r)
    == RijdenOpZicht) =>
    ((exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == begin_signal &&
      auth_activation(elementObject(e)) == dash_No_routeCommand &&
      reserved_begin(elementObject(e)) == H_Cancelled && (forall u :  UseTuple
      .  u in use(elementAstris(e)) => !(reservedUse(u) == Herroepen && routeID(u) !=
      routeID(r)))) &&
    (exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == begin_signal &&
      auth_activation(elementObject(e)) == dash_No_routeCommand &&
      reserved_begin(elementObject(e)) == H_Cancelled && (exists u :  UseTuple .  u in
      use(elementAstris(e)) && reservedUse(u) == Herroepen && routeID(u) == routeID(r)))
      &&
    ((exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == begin_signal &&
      auth_activation(elementObject(e)) == dash_No_routeCommand &&
      (reserved_begin(elementObject(e)) == H_Cancelled ||
      reserved_begin(elementObject(e)) == dash_Not_used ||
      reserved_begin(elementObject(e)) == Unknown_Reserved_Begin)) &&
    (exists e :  ElementInRoute .  (e in elementInRoute(elementsInAndByRoute)) &&
      elementInRouteType(e) == Signal && elementInRouteRole(e) == end_signal &&
      reserved_end(elementObject(e))) &&
    (forall e :  ElementInRoute .  ((e in elementInRoute(elementsInAndByRoute)) &&
      operable(elementConf(e))) => (reserved_RW(elementObject(e)) == dash_Not_Used ||
      reserved_RW(elementObject(e)) == Unknown_Reserved_RW)) &&
    (signalOperation(r) == Normaal) => (forall e :  ElementInRoute .  (e in
      elementInRoute(elementsInAndByRoute) && elementInRouteType(e) == Section) =>
      (inRoute(elementObject(e)) == dash_NotPartOfRoute || inRoute(elementObject(e)) ==
      Unknown_IsPartOfRoute))
    ));

act
  dummy;

proc
RMC_FUNC_056(r :  Route, allElementsInAndByRoute :  AllElementsInAndByRoute) =
  (pred_A(r) && pred_B(r, allElementsInAndByRoute) && pred_C(allElementsInAndByRoute) &&
    pred_D(r, allElementsInAndByRoute) && pred_E(r, allElementsInAndByRoute) &&
    pred_F(r, allElementsInAndByRoute) && pred_G(r, allElementsInAndByRoute)) ->
  dummy .  RMC_FUNC_056(r, allElementsInAndByRoute);

init
sum r :  Route .  sum a :  AllElementsInAndByRoute .  RMC_FUNC_056(r,a);
```

Appendix E

Script

The functions in the following script are used for the generation of the state-space and the verification of the modal μ-calculus formulas. First, there is a function `generate_state_space` to generate a state-space and reduce it. Then, the `verify_property` function can be used to verify a property for the previously generated state-space. Below, an example is shown where we generate one state-space and subsequently verify two properties for this model. The references to the files are paths relative to the script. `#!/bin/bash`

```
function generate_state_space() {
  echo Generating $1
  time mcrl22lps -verbose -timings=time.txt $1 model.lps
  time lps2lts -verbose -timings=time.txt -cached model.lps model.lts
  time ltsconvert model.lts model.min.lts -verbose -timings=time.txt -equivalence=bisim
    -out=model.min.lts
}

function verify_property() {
  echo Verifying $2
  time lts2pbes model.min.lts -verbose -timings=time.txt $2.pbes -counter-example
    -formula=$1 -lps=model.lps
  time pbessolve -verbose -timings=time.txt $2.pbes -in=pbes -search=breadth-first
    -strategy=2
}

generate_state_space example_model.mcrl2
verify_property example_property1.mcf property_name1
verify_property example_property2.mcf property_name2
```

Appendix F

Requirement RMC-FUNC-056

Back to the text

	- Indien alle serviceresponses=OK: De Toestand van de betreffende Rijweg aan te passen naar "*Instelopdracht verstuurd naar BevNL*" - Indien minstens één van de serviceresponses=NOK: De Toestand van de betreffende Rijweg aan te passen naar "*Instelopdracht afgekeurd door BevNL*" (indien van het beginsein gereserveerd_begin <> H) of naar "*Herroepen*" (indien van het beginsein gereserveerd_begin == H), en de attributen [IR] en [AU] (en ook [SDE], [GC] en [SDVx]) te wissen en de rijwegopdracht als volgt aan te passen: o Toestand = "Gereed" o Resultaat = NOK o RedenNOK = tekst conform 11: Bijlage C: Antwoordteksten
Implementation	BeveiligingService.BevNLAutoriseerRoute BeveiligingService.geefBevNLElement ElementConfiguratieService.geefElementAttribuut ElementConfiguratieService.geefnietProfielvrijSituaties
Rationale	Als alle regime element opdrachten ten behoeve van het instellen van een rijweg succesvol zijn uitgevoerd, BevNL meldt dat de stoptijd verstreken is (als goederen criterium actief), dan zal de daadwerkelijk route instelling geautoriseerd worden.

3.9.5.26.4 Uitvoeren Instellen Rijweg (zonder regime sturing)

Eis: RMC-FUNC-156	
Description	- Vevallen
Implementation	
Rationale	

Als in de toestand "Voorbereid" het Instellen Rijweg attribuut actief is en er hoeven geen regime elementen gestuurd te worden, dan dient de toestandsovergang naar "Instelopdracht verstuurd naar BevNL" plaats te vinden, nadat de opdracht ten behoeve van het instellen naar de beveiliging verstuurd is. Het mag echter niet zo zijn dat de rijweg al herroepen wordt (attribuut Herroepen actief), of dat de er nog bezette secties zijn in de rijweg, als het geen ROZ-rijweg betreft.
Er moet tevens een rijweginstelopdracht naar BevNL gestuurd worden in de toestand "Afrijden", indien Herroepen "Niet actief" is.
In beide gevallen mogen er geen secties meer in-rijweg liggen volgens de gegevens die BevNL levert, en moet de rijweg nog steeds profielvrij zijn.
Deze toestandovergang is niet beschreven in het toestanddiagram in Figuur 3-1.

Merk op dat deze eis niet getriggerd hoeft te worden door een toestandswijziging van BevNL, maar uitgevoerd dient te worden zodra een bepaalde situatie zich voordoet.

Eis: RMC-FUNC-056	
Description	Zodra een rijweg aan de volgende condities voldoet:

	▪ de Toestand = “Voorbereid” en Instellen Rijweg attribuut = “Actief” en Herroepen attribuut = “Niet actief” en het Annuleren attribuut = “Niet actief”, of de Toestand = “Afrijden” en Herroepen attribuut = “Niet actief” ▪ geen van de volgende condities gelden: o het Emplacement stopdoor attribuut is actief en het regime element meldt StoptijdVerstreken= N o het GoederenCriterium attribuut is actief o één of meerdere van de Vrijebaan stopdoor attributen zijn actief o er is een regime element in de rijweg actief dat volgens de regime attributen passief moet worden gestuurd ▪ alle elementen in en bij de rijweg zijn betrouwbaar (dit geldt niet voor verzoekelementen en niet-gedetecteerde secties en elementen met GemeldDoorBevNL=Nee) ▪ Ingeval er sprake is van een N of A-rijweg: o geen van de secties in de rijweg heeft het attribuut Bezet= J, en o geen van de secties in de rijweg heeft het attribuut LogischBezet= J, en o als het beginsein van deze rijweg gereserveerd is voor Herroepen van deze rijweg: geen van de secties is nog actueel in gebruik voor een andere rijweg, en o als het beginsein van deze rijweg niet gereserveerd is voor Herroepen van deze rijweg: geen van de secties is nog actueel in gebruik voor een rijweg, en o de rijweg profielvrij te berijden is (zie ook RMC-FUNC-092) ▪ (alleen ingeval van een A-rijweg) voor alle secties in de rijweg geldt LigtInRijweg= - (of U), en ▪ (ingeval er sprake is van een N-rijweg met een fictief eindsein[10] op [124]) van het fictieve eindsein is VertrekblokBezet = N, en ▪ (ingeval N of ROZ): o van het beginsein in de rijweg is Auth.Activering=- EN Gereserveerd_Begin= ‘H’ waarbij het beginsein van deze rijweg niet gereserveerd is voor Herroepen van een andere rijweg of o van het beginsein in de rijweg is Auth.Activering=- EN Gereserveerd_Begin= ‘U‘ waarbij het beginsein van deze rijweg gereserveerd is voor Herroepen van deze rijweg of o van het beginsein in de rijweg is Auth.Activering=- EN Gereserveerd_Begin= ‘H’ of ‘-‘ (of U), en van het eindsein in de rijweg is Gereserveerd_Eind <> J, en van de bedienbare elementen in de rijweg is Gereserveerd_RW= ‘-‘ (of U), en (alleen ingeval van een N-rijweg) voor alle secties in de rijweg geldt LigtInRijweg= - (of U) dient RMC:

	- De RijwegOpdracht te bepalen die tot het instellen heeft geleid. (De RijwegOpdracht die door een instellenRijweg request voor deze Rijweg is aangemaakt). - Het InstellenRijweg-attribuut “Actief” te maken (indien dit nog niet actief was) - Als voor de rijweg in de configuratie is aangegeven dat er sprake is van een ElementGestuurdeBeveiliging: o Een (of meer igv gesplitste rijwegen) autoriseer route opdracht(en) naar de beveiliging component(en) te versturen voor het beginsein van de (deel)rijweg. - Als voor de rijweg in de configuratie is aangegeven dat er geen sprake is van een ElementGestuurdeBeveiliging: o In de configuratie te controleren of er sprake is van meerdere BevOpdrachten (GesplitsteRijwegId’s aanwezig). Indien dit zo is, dan zijn dit de in te stellen rijwegen. De Seinbediening, goederen-indicator en stopdoor-indicatoren dienen overgenomen te worden van de RijwegOpdracht, in dat deel van de rijweg waarin het regime element is opgenomen. De eerste stopdoor-indicator uit de opdracht hoort bij de EmplacementStopdoor (mits aanwezig), de resterende stopdoor-indicatoren uit de opdracht horen bij de VrijebaanStopDoor. De evt. BB-indicator gaat mee met die deel(en) waar een conditionele beperking aanwezig is. o Een (of meer igv gesplitste rijwegen) instellen route opdracht(en) naar de beveiliging component(en) te versturen met daarin de parameters uit RijwegOpdracht (of zoals boven bepaald igv gesplitste rijwegen). - De RijwegOpdracht uit te breiden met de in de serviceresponse(s) genoemde identificatie van de zojuist verstuurde instellen route (of autoriseer route) BevNLOpdracht(en). De status wordt: niet beantwoord. - Als attribuut [AU] is “Niet actief”: de attributen [SDE], [GC] en [SDVx] te wissen. - Indien alle serviceresponses=OK: De Toestand van de betreffende Rijweg aan te passen naar “*Instelopdracht verstuurd naar BevNL*”. - Indien minstens één van de serviceresponses=NOK: De Toestand van de betreffende Rijweg aan te passen naar “*Instelopdracht afgekeurd door BevNL*” (indien van het beginsein gereserveerd_begin <> H) of naar “*Herroepen*” (indien van het beginsein gereserveerd_begin == H), en de attributen [IR] en [AU] (en eventueel ook [SDE], [GC] en [SDVx]) te wissen en de rijwegopdracht als volgt aan te passen: o Toestand = “Gereed” o Resultaat = NOK o RedenNOK = tekst conform 11: Bijlage C: Antwoordteksten

Implementation	BeveiligingService.BevNLInstellenRoute BeveiligingService.BevNLAutoriseerRoute BeveiligingService.geefBevNLElement ElementToestandService.geefElement ElementConfiguratieService.geefElementAttribuut ElementConfiguratieService.geefnietProfielvrijSituaties
Rationale	Voor het daadwerkelijk instellen van een rijweg moet de RijwegOpdracht behorende bij de instellenRijweg request worden opgezocht, om de parameters voor de rijweg instelling te kunnen achterhalen. Als deze parameters bekend zijn, moet een instellen opdracht naar de beveiliging worden gestuurd. Daarna kan de Rijweg toestand naar *Instelopdracht verstuurd naar BevNL*. Het begin- en eindsein dat gecontroleerd wordt zijn de seinen met de rol begin en eind uit de configuratie, en niet de begin- en eindseinid uit rijwegidentificatie (die zijn alleen bestemd voor de identificatie!) Als elementen niet betrouwbaar zijn heeft het geen zin om een opdracht naar de beveiliging te sturen. Verzoekelementen hoeven niet betrouwbaar te zijn (maar mogen dat natuurlijk wel).

2592
2593

Eis: RMC-FUNC-157	
Description	o Vervallen
Implementation	
Rationale	

2594
2595

Eis: RMC-FUNC-057	
Description	Zodra een rijweg aan de volgende condities voldoet: ▪ de Toestand = “Voorbereid” en Instellen Rijweg attribuut = “Actief” en Herroepen attribuut = “Niet actief”, en ▪ alle elementen in en bij de rijweg zijn betrouwbaar (Dit geldt niet voor verzoekelementen en niet-gedetecteerde secties en elementen met GemeldDoorBevNL=Nee) ▪ Ingeval er sprake is van een N of A-rijweg: o geen van de secties in de rijweg heeft het attribuut Bezet = J, en o geen van de secties in de rijweg heeft het attribuut LogischBezet= J, en o als het beginsein van deze rijweg gereserveerd is voor Herroepen van deze rijweg: geen van de secties is nog actueel in gebruik voor een andere rijweg, en o als het beginsein van deze rijweg niet gereserveerd is voor Herroepen van deze rijweg: geen van de secties is nog actueel in gebruik voor een rijweg in geval van een N-rijweg, en ▪ (alleen ingeval van een A-rijweg) voor alle secties in de rijweg geldt LigtInRijweg= - (of U), en ▪ (ingeval er sprake is van een N-rijweg met een fictief eindsein[10] op [124]) van het fictieve eindsein is VertrekblokBezet = N, en ▪ (ingeval N of ROZ):

Appendix G

Preparation report

Report feasibility study

Preparation graduation project

Project name: Formal verification of train control systems

January 27th 2020

Author & student	Jasper H.P. van Meer (1190029)
Supervisor(s)	S.B. Luttik, TU/e; E. Bossink, ProRail; E. de Boer, ProRail
Tutor	M.S. Bouwman, TU/e and ProRail
Company	ProRail, Department: ICT Logistiek-Treinbeheersing

Contents

Version history

Version	Date	Description
0.1	Nov 13, 2019	Initial version
0.2	Nov 25, 2019	First draft; to be discussed with Bossink
0.3	Nov 26, 2019	Second draft; processed Bossink's comments; to be discussed with Luttik and Bouwman
0.4	Dec 9, 2019	Third draft; processed some of Luttik and Bouwman's comments; some parts to be discussed with and filled in by Bossink
0.5	Dec 10, 2019	Fourth draft; processed some of Bossink's comments; to be discussed with Luttik and Bouwman; to be sent to Bossink and de Boer
0.6	Jan 7, 2020	Fifth draft; processed some of Luttik and Bouwman's comments. Added section with more information on the system under consideration. Added subsection with overview of formal methods. Expanded on Proof-of-Concept section.
0.7	Jan 21, 2020	Sixth (pre-final) draft; processed some of Luttik and Bouwman's comments; to be sent to Bossink; Expanded on Proof-of-Concept and planning sections
0.8	Jan 22, 2020	Seventh (pre-final) draft; finished some minor left over work that was not finished in time for 0.7; (small) changes in sections, 1.0, 2.2 paragraph 4, 5, 8
1.0	Jan 27, 2020	Final version; processed some of Luttik, Bouwman and Bossink's comments; to be presented

1 Introduction

As part of the preparation for my MSc graduation project a feasibility study has been conducted. In the sections below, the problem statement will be defined. Then, the result of a state-of-the-art analysis will be presented after which the system under consideration is discussed. Then, a Proof-of-Concept and its results will be presented. Hereafter the expected end result is discussed followed up by a research and development proposal. Penultimate, a project planning is proposed before finally concluding whether or not this project is feasible.

1.1 About ProRail

This project is done at ProRail, a government task organisation that takes care of maintenance and extensions of the national railway network infrastructure (not the metro or tram), of allocating rail capacity, and of traffic control [14]. Within ProRail, there is the department of ICT with a subdivision focused on train control from the perspective of logistics, traffic and routes. They are responsible for the development and maintenance of software in their domain.

1.2 About the department of ICT

The department of ICT makes a great variety of applications. From custom made real-time safety-critical control systems to Commercial off-the-shelf supporting applications. Some applications are made in-house, others are created with software contractors. The subdivision for train control creates large custom made applications where safety is highly valued. They do so with the help of software contractors.

1.3 Astris

The system under consideration for this project is called Astris. Astris is an acronym originating from "***A****ansturen en* ***ST****atusmelding* ***R****ail****I****nfra****S****tructuur*" which roughly translates to "**D**irecting and **ST**atus reporting **R**ail**I**nfra**S**tructure". So, in english "Dstris" would be an apt acronym.

Obviously, as the rail infrastructure evolves, a lot of railway elements, e.g. train switches, interlockings, from different vendors are used. As a consequence, a number of different interfaces must be supported to interact with these elements. Astris aims to solve this problem by providing a uniform interface for applications that want to control and/or request information from these elements.
A more in-depth explanation can be found in Section 4.

2 Problem statement

The department of ICT wishes to have a potential solution investigated that pertains to the development and maintenance process of software applications. Below, we will elaborate on the problem, the cause for this project and state a concrete problem statement.

2.1 The problem

The development and maintenance process follows the well-known waterfall model (also known or similar to the V-model or V-cycle) supported by the MIL-STD-498 standard documentation structure. For ease of explanation, let us consider the scenario of the creation of a new application.

When the concept and high level goals of a new application are clear, engineers and domain experts will meticulously write down all requirements to the point where almost a detailed design is dictated.

In the below Figure 1 a moderately extreme example is given of the description of a single requirement spanning about three pages where the description is in essence one large implication.

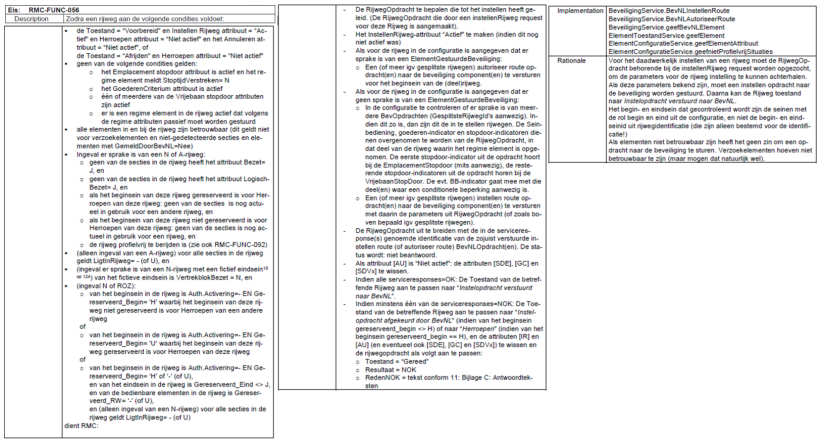


Figure 1: One of the longest descriptions for a requirement

The requirement explains the execution of the setting of routes (*instellen rijweg*). It says that whenever a route meets a conjunction of conditions, then a number of

sequential steps must be done to set the route (including exception scenarios whenever a step fails). In addition to some general conditions that must hold, a distinction is made in conditions per route type. Let us consider the set of route types $\{N, ROZ, A, H\}$, then there are conditions that must hold in case the type is N or A, in case the type is N or ROZ, in case the route is of type A, in case the route is of type N with a fictional end signal. This explains why the requirement is so large. Furthermore, the scoping of conditions is done by bullet points and commas, which is not as precise or intuitive as using brackets in mathematical formulas.

Having written down the requirements in a specification, this specification is handed over to (one or more) software contractors that will create the specified application. During this process the ICT department regularly meets with the contractors to discuss progress and answer questions.

However, they find that the fact that the specifications are written in natural language gives freedom in interpretation by the ambiguity inherent to natural language. This leads to many questions, leading to discussions, leading to refinements of the specifications. And these refinements may itself be cause for more questions. From the above discussed example one can see why this is the case. Despite the efforts to deal with the complexity of the requirement, it can still be difficult to understand.

Moreover, this problem is not limited to the scope of the contact between the ICT department and the software contractors, but also internally within ProRail, e.g. the department responsible for testing or maintaining knowledge in the face of employee churn. Especially the latter can be important considering the aging workforce of ProRail.

Clearly, this is undesirable and the ICT department seeks a solution to specify system requirements more precisely.

2.2 The cause for this project

Once upon a time, the managers of the departments of ICT and the department of Train signalling systems (*Treinbeveiliging*, responsible for systems like interlocking, train detection and signals) came together and the department of Train signalling systems explained that they are experiencing similar problems as ICT is facing. As a potential solution they are looking into the use of formal methods.

As will be discussed in the next section when we take a look at the state-of-the-art of formal specification, the use of formal methods in the domain of responsibilities of the department of Train signalling systems is not new (e.g. formal methods have been applied many times in the analysis of interlocking systems). So, the department has some basis for confidence in the successful application of these methods.

The department of ICT was excited about the prospects too and wants to look into the use of formal methods as well. The department of ICT has experienced to

some degree the benefits of formal specification in a project with a company called Axini which employs formal specification for the purposes of automated Model Based Testing. The reception of this project was predominantly positive, claiming that the software quality increased. In this collaboration Axini suggested that the department of ICT should consider making formal specification themselves. Hence, the motivation for this project.

During the discussions for identifying the goals of this project, the subject of the automation that the use of formal specification enables came up. Obviously, the ICT department is already familiar with test automation as explained in the above paragraph. However, additionally, a formal specification can be verified (validated) quite extensively. The ICT department was pleasantly intrigued by this notion since quite a bit of effort is put into verification and validation of their systems to reaffirm their systems are safe. Thus, while verification/validation was not mentioned in ProRail's original project description, it is part of the project now.

2.3 The ICT department's goals

So, the initial goal of the ICT department is to explore the use of formal specification. More concisely, they want special attention paid to:

- How these methods can help improve communicability and/or transferability of the specification to (sub-)contractors and colleagues within ProRail
- How these methods can help with verification/validation of (safety) properties of a system
- How these methods can help with testing

In addition to the above described three major goals, there are two minor goals:

- Seeing how the ICT department stringently upholds their documentation structure, they wish to know what would have to change to the documentation structure to accommodate the use of formal specification
- The ICT department wants to know how, where and if the use of the methods considered in this project can reinforce their conformance to Safety Integrity Level (SIL)-1 with respect to the CENELEC EN 50128 [4] and 50126 [5] standard(s)

2.4 Project goals

Unfortunately, we are not able to answer the questions of the ICT department directly since the scope of communicability is immense and not properly defined. As such,

we are not able to directly address their major goal. So, the purpose of this project must be adjusted. Instead, the aim is to provide evidence how formal specification can help in the development and maintenance process of the ICT department such that they are able to answer these questions themselves.

2.5 Concrete problem statement

Finally, we can conclude with the following concrete problem statement:

"Can the use of formal specification be used effectively in the development and maintenance process of the ICT department. Specifically in the area of specification, verification and testing."

Here, 'effectively' is subjective with respect to the ICT department. It depends on how many inconsistencies or faults we find and if these are relevant or not. For example, one may find issues that are theoretically possible, but practically will never happen and are not worth the time and effort to investigate any further.

The contribution of this project is to demonstrate the applicability of formal methods for railway train control applications. Specifically for the use of specification, verification and testing.

3 State-of-the-art analysis

In the sections below we will elaborate on findings on the current or recent developments and experience in the industrial use of formal methods with the goal of explaining where this project is similar or differs from previously documented projects.

For this, a number of articles from the past 30 years have been examined that describe the state-of-the-art at that moment in time. These either consider formal methods across many domains and different types of applications or constrained within the railway domain. These articles from 1994 [7], 2000 [9], 2009 [15, 13], 2014 [6] and 2018 [2] vary in size depending on if a survey has been conducted or not. We find that the problems with the application of formal methods remain the same. Now, the amount of successful projects has increased. But, it is repeatedly written that despite the long and arguably successful history of formal methods, not one universally accepted formal method or tool has emerged. There is a lot written on the B method though.

3.1 About formal methods

Formal methods is a collective name for methods or techniques like formal specification and formal verification, where 'formal' refers to the foundation in mathematics

in the areas of e.g. mathematical logic and set theory. This allows us to apply mathematical techniques on the models for the purpose of analysis, proving properties etc.

The key points mentioned in [13] are clear on the benefits of the use of formal specification. "They may be used with requirements defined in natural language to clarify any areas of potential ambiguity. Since they are precise and unambiguous they remove doubt in the specification and avoid problems of language interpretation." To this end it seems to almost exactly tackle the problems that the ICT department is facing. These methods also enable automation in e.g. formal verification, test case generation, code generation and simulation. Though, formal methods cannot solve all problems. It has some drawbacks.

"Non-specialists find it hard to understand and work with formal specifications." It is not clear how highly trained the personnel needs to be for this. But there is a general consensus that without proper training it is not easy to work with formal methods.

[13] explains that formal methods can be cost-effective when used for core parts of critical systems where safety, reliability and security are particularly important. Not much is known about non- or less-critical systems. However, that does not mean that it can't be. Perhaps in the specific case of the ICT department it will be.

Additionally, the methods can have a positive impact on the development process in terms of time. The time investment can be shortened, potentially even in the phase(s) of the development process where ProRail would want it to be. The use of formal methods can save time in the later phases of the process (e.g. testing, integration, validation) at the expense of investing more time in the specification phase. Talking with people of ProRail and overhearing conversations has shown that the software development is perceived as slow in these later phases. Some people of ProRail agree and they acknowledge that this is a problem that should be addressed. The use of formal methods may or may not be helpful here.

3.2 Formal methods in the railway domain

The use of formal methods and therefore also formal specification, are not new in the railway domain. They have been applied successfully many times over the last few decades in e.g. France, the Netherlands, Italy and the UK. The CENELEC EN 50128 [4] standard that ProRail also must conform to, mentions formal methods as a highly recommended practice for SIL3-4 platforms. To that extent the transportation sector and specifically the railways benefit a lot from formal methods and the potential for more is high.

The survey [2] gathered information on the use of formal methods for industrial projects in the railway domain. From this survey we discuss some interesting results.

Firstly, the type of products that formal methods are used for are mostly lower level (i.e. close to the hardware/railway elements) cyber-physical safety-critical SIL3-4 products like interlockings and axle counters. Barely any products pertain to higher level purely software less-critical SIL-1 train control applications in the railway domain that are the subject of this project. So, the type of applications formal methods are to be applied to in this project is somewhat new.
Second, formal methods are mostly used for formal specification, formal verification and model analysis. To a lesser, but still substantial, extent formal methods are used quite a bit in the areas of simulation, testing and code generation. These are exactly some of the areas where the ICT department wants to have support from formal methods.

Third, the most relevant quality aspects of a formal tool were investigated. By far the most relevant aspect was maturity specified as stable and industry-ready. It is not really clear what the author meant with 'mature'. Perhaps the author meant the important notion that formal methods do not scale well to larger systems. Or maybe maturity is confused with a longing for a universally accepted tool. A factor in the latter could be caused by poor usability of tools since the most relevant quality aspects are associated to this. From other quality aspects it can be observed that there is some interest in how formal methods relates to the CENELEC standard, where the ICT department shares this interest.

Lastly, the second most relevant feature for formal methods was found to be modelling. It is not clear how to interpret this since the author does not elaborate on this. Nevertheless, one can guess that this pertains to tools that can e.g. visualize a model. One can imagine that the usually textual based models with a particular syntax can be hard to understand without visual support. From people at ProRail I have heard that visual representations of systems, e.g. state machines, would probably help immensely in communicating a specification. They could not however provide me an answer why they are not doing so right now. The most relevant feature was formal verification. Clearly, verification is found to be useful. Thus, confirming the validity of our objective in this project to investigate this.

3.3 Overview software tools

We will take a look at a number of formal methods software tools to help determine in Section 7.1 which tools are relevant for this project and to get an idea of the formal methods landscape. Since there are a plethora of tools available, the scope is limited to the tools described in CENELEC EN 50128 [4] and the mCRL2 tool.

The reader is referred to appendix D.28 of [4] for a description of the tools described in [4]. Below, a brief description is given of mCRL2.

mCRL2 is a formal specification language with an associated toolset, that can be

used for modelling, validation and verification of concurrent systems and protocols. mCRL2 is developed by the Mathematics and Computer Science department of the Eindhoven University of Technology in collaboration with the University of Twente [10]. mCRL2 has three languages: the data language, which is based on the theory of abstract data types, the process specification language, which is an ACP-style process algebra, and the requirement language, which is a highly expressive extension of the modal μ-calculus. mCRL2 has been used successfully in a number of case studies including the modelling and analysis of ERTMS Hybrid level 3 [1] and of interlockings [3]. mCRL2 is also used in projects outside of the railway domain such as CERN's Large Hadron Collider [8, 11].

#	**Formal method**
1	Communicating Sequential Processes (CSP)
2	Calculus of Communicating Systems (CCS)
3	Higher Order Logic (HOL)
4	Language for Temporal Ordering Specification (LOTOS)
5	OBJ
6	Temporal Logic
7	Vienna Development Method (VDM)
8	Z method
9	B method
10	Model Checking
11	mCRL2

Table 1: List of formal methods under consideration

The formal methods under consideration are listed in Table 1. The numbering scheme will be consistent throughout all tables. It is interesting to see (6) and (10) in this list. Model checking is a method for checking if a system modelled by a finite-state machine meets a specification. Temporal logics such as CTL or LTL are used to describe such specifications. Since most methods incorporate (6) and (10) (which will become clear in Table 3) we can omit them from the list. Furthermore, (3) is a method suitable for the specification and verification of hardware. Thus, we also omit this method from the list henceforth.

Table 2 shows the underlying techniques of each formal method and whether the modelling of concurrent systems is supported in addition to the modelling of sequential systems.

#	Formal method	Technique	Concurrent
1	CSP	process algebra	✓
2	CCS	process algebra	✓
4	LOTOS	process algebra	✓
5	OBJ	term rewriting	
7	VDM	model based	in VDM++
8	Z method	model based	
9	B method	model based	
11	mCRL2	process algebra	✓

Table 2: Underlying technique and concurrency support per method

Simply put, in an algebraic (process algebra) specification the system behaviour is captured by describing sequences of actions. In a model-based specification a system is modelled by describing its states and the operations to change the state. The term rewriting technique of (5) can also be considered an algebraic technique.

In [9] the distinction between algebraic and model-based methods is avoided and instead a distinction is made in the specification paradigm. Here, the methods (7 - 9) are described as state-based specification methods, (5) is an algebraic functional specification and (1)(2)(4)(11) rely on the operational paradigm.

Having identified the fundamental differences between the considered formal methods we will take a look at their functionalities for verification, testing and code generation in Table 3 below.

#	Formal method	Verification	Test gen.	Code gen.
1	CSP	✓		
2	CCS	✓		
4	LOTOS	✓	via CADP	
5	OBJ			✓
7	VDM	✓		
8	Z method	✓		✓
9	B method	✓		✓
11	mCRL2	✓	via JTorX	

Table 3: Functionality per formal method

From the above table we can clearly see that almost every method can do some form of verification. However, test and code generation functionalities are more scarce.

Now, for test and code generation it is very clear what they entail. However, there is some difference with verification.

In the case of the 'process algebra' methods (1)(2)(4)(11) verification is done by the aforementioned notion of model checking. This checking can be done using model checkers like nuSMV, mCRL2 or CADP. However, model checkers are not equal. Most notably, they are distinguished by the expressive power of their 'properties language'. The aforementioned temporal logics such as CTL and LTL are such properties languages. However, there are languages with more expressive power, i.e. subsumes and extends upon the properties that can be described. The order of languages is as follows: $CTL, LTL \subseteq CTL^* \subseteq \mu\text{-}calculus \subseteq mCRL2\ \mu\text{-}calculus$.

In the case of the 'model-based' methods (7 - 9) the user can model the system of interest "in terms of set-theoretic structures on which are defined invariants (predicates), and operations on that state are modelled by specifying their pre- and post-conditions in terms of the system state. Operations can be proved to preserve the system invariants." [4].

It is not immediately obvious what the differences are between the algebraic and model based methods. Some consider this difference to be a matter of style and personal preference. But it is clear that the differences allow some systems to be modelled by one language but not by the other and vice versa.

4 System under consideration

As explained in the introductory Section 1.3 the system under consideration for this project is Astris. The below sections will expand upon this and explain its place and purpose in the complex composition of software systems that (help) make the Dutch railways function.

4.1 Purpose

As mentioned in Section 1.3 Astris aims to provide a uniform interface for applications that need to interact with railway elements. An overview of these interacting components supported by Figure 2 is presented below.

Astris's interface towards the variety of these elements (or actually the interlocking that interacts with the physical elements) is abbreviated to **BevNL**, a shorthand for '***Bev****eiliging* ***N****eder****l****and*'. Astris also provides interfaces towards applications that want to control and/or request information from these elements. The major applications that do are the following two:

PRL, an abbreviation for '***Pr****oces****l****eiding Rijwegen*' which roughly translates to 'process management of routes'. This application supports the *treindienstleiders* (train dispatchers) in their logistic and safety tasks for the process of controlling rail traffic.

TROTS, an abbreviation for '**T**rain **O**bservation and **T**racking **S**ystem' where its function is self-explanatory from the name. Its purpose is to provide information to ProRail and users of the infrastructure (e.g. NS, Arriva, SBB Cargo) on where trains are on the infrastructure and by extension geographically. This is necessary for normal operation and for example in the case of NS, to reliably predict arrival times of passenger trains to keep travellers informed of the departure schedule.

So, Astris connects TROTS and PRL to the variety of railway elements unified under BevNL.

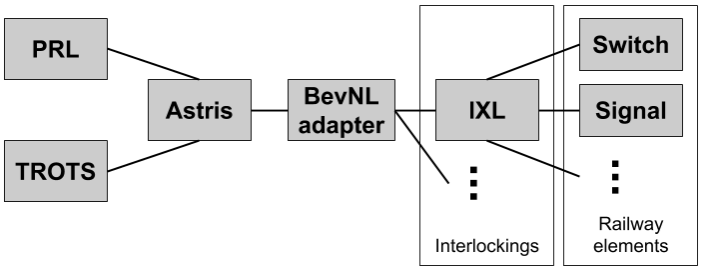


Figure 2: Overview of Astris and connected components

4.2 Architecture

Astris is built on a Service Oriented Architecture (SOA) consisting of the following components: *Element*, *Rijweg* (Route), *Gebied* (Region), *Beveiliging* (Signalling), *Systeem* (System) and *Beheer* (Management). This is reflected in Figure 3 below. These components provide their services through a Request/Reply and Notification Message Exchange protocol. Every component is at least connected to the *Beheer* (Management) component since it is responsible for managing the other components within Astris.

4.3 Component under consideration

Since Astris is a very large software system the scope and focus of this project is shifted from a 'system under consideration' to a 'component under consideration'. Here, the aforementioned Element component has been selected since it has interesting logic and is fairly self-contained. Within this component the responsibilities are logically divided in three parts: a configuration part, the data management part and the core part responsible for offering functionality. We will focus on the last part which is

named the Element manager component (EMC). Its place within Astris is shown in the below Figure.

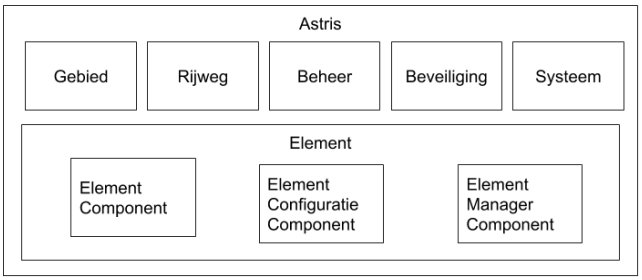


Figure 3: The EMC in the context of Astris

It must be noted that within every instance of Astris, there can be one or more instances of each component displayed in the above Figure. With the exception of the *Beheer* (Management) component of which only one may exist. So, within an Astris instance, there can be multiple Element Components running in parallel.

4.4 Element manager component

The EMC provides the elementary 'Element' service outside that support high level service operations to, for example, set routes or secure zones for construction on the infrastructure. Internally, within the logical scope of Astris the EMC provides a number of services for the use and retrieval of information on railway elements. Additionally, there is the service shared with the *Beheer* component for management purposes.

The EMC has the task to receive service requests and execute them. The EMC must safeguard the execution of these requests such that it is not possible that two applications act on the same railway element at the same time, e.g. to prevent race conditions.

5 Proof of Concept

To be able to strengthen our conclusion on the feasibility of this project a small start has been made on the actually formally specifying and verifying of the Element Manager Component discussed in Section 4.4.

Though, we will do so with a higher abstraction level and limited scope since we obviously want to form our conclusion on feasibility sooner rather than later.

A higher abstraction level can be achieved by omitting details and only covering the coarse grain functionality. The scope can be limited by considering only one instance of the EMC and specifying only a few service operations, which is possible since they are independent. Later in the project, other service operations can be added.

This small start is henceforth dubbed a Proof-of-Concept since the goal is to show, by example, the feasibility of the project. Thus, this is a Proof-of-Concept within the scope of determining the feasibility of this project. The Proof-of-Concept will be elaborated on in the subsequent Sections.

5.1 Expected end result

At the end of the Proof-of-Concept we will have a thorough understanding of the EMC. We have shown that the EMC can be formally specified and that some properties can be checked. With this Proof-of-Concept we are able to incorporate our findings and experience to better describe the expected end result, research and development approach and planning of the project as a whole. This, in conjunction with the conclusion of this Proof-of-Concept, which will be presented in Section 5.3, we are able to make a more firm conclusion on the feasibility of this project.

5.2 Approach

To gain the required understanding of the EMC we want to identify the EMC's purpose and responsibilities, the components it interacts with and the used interfaces. The formal specification should (at a high level) reflect these findings. This can be achieved by analysing the Software Requirements Specification (SRS) of the Element Component [12] and its referenced documents. The aim is to omit, if possible, the data structures and capture the general behaviour through actions.

Additionally, a goal is to have identified some properties that are interesting to check against the formal specification. Do not though that the formal specification must be of an appropriate level of abstraction such that we can meaningfully check these properties.

For the Proof-of-Concept the mCRL2 toolset is used for formal specification and verification. The reader is referred to Section 7.1 for the exact rationale on why mCRL2 is used for this project.

5.3 Results

By analysing [12] and its referenced documents an important fact was found that allows the application of abstractions, that is, the EMC processes service operation

sequentially (in a queue actually). This allows the EMC to be modelled as a single entity. But it also allows all other components that the EMC interacts with to be considered a single entity that is henceforth denoted as 'the environment'. Thus, the formal specification is the parallel composition of the 'EMC' and the 'environment'.

As mentioned before the Proof-of-Concept will cover a limited amount of service requests (8 out of 53 total) for changing the mode of operation (e.g. Active, Not active, Internally active, passive etc.) of the EMC and the logically locking and unlocking of railway elements. Where the latter is especially important since it describes the important functionality where applications can get exclusive access to a railway element and thus prevent race conditions from occurring.

5.3.1 Specification

The entirety of the mCRL2 formal specification of the EMC can be found in Appendix A. In the below code snippet the Element Component process `Component` is shown.

```
Component(s:componentStatus, l:Locks) =
   ElementGebruikService(s, l) +
   BeheerService(s, l);
```

We can observe how the process is essentially the sum of all possible service operations such that any service operation can occur in any order.

5.3.2 Properties

The checked properties described in modal μ-calculus formulas can be found below to get an impression of the Proof-of-Concept.
'Absence of deadlock' is described by (1)

$$\texttt{[true*]<true>true} \tag{1}$$

'A service operation (here `LockElement` and `UnlockElement`) will inevitably receive a `Respons`' is described by (2) and (3)

$$\begin{aligned}&\texttt{[true*.}\exists\texttt{e:Element.LockElement(e)]}\\&\mu\texttt{X.([}\neg\exists\texttt{a:Acceptatie.Respons(a)]X}\wedge\texttt{<true>true)}\end{aligned} \tag{2}$$

$$\begin{aligned}&\texttt{[true*.}\exists\texttt{e:Element.UnlockElement(e)]}\\&\mu\texttt{X.([}\neg\exists\texttt{a:Acceptatie.Respons(a)]X}\wedge\texttt{<true>true)}\end{aligned} \tag{3}$$

Using the mCRL2 IDE we checked if the above formulas evaluated to `true` as expected. Table 4 below shows the results.

Formula	Expected result	Actual result
(1)	`true`	`true`
(2)	`true`	`true`
(3)	`true`	`true`

Table 4: Results verification of properties

Since for every formulas the actually achieved result is equal to the expected result, we can claim that the properties described by these formulas hold.

5.4 Conclusion

A formal specification of a part of the Element manager component has been made. Furthermore, we have been able to verify a few simple (but not trivial) properties of the model.

These results have also been shared with the people of ProRail in a monthly team meeting and the response was positive. Using the mCRL2 IDE to show the state-space and what properties could be verified raised many questions on applicability and how formal methods can help. These serious questions clearly indicate some interest and motivate the purpose of this project.

For some remarks it should be mentioned that the taken approach was very helpful in the process of understanding the component in a structured manner and coming to a conclusion on how to formally specify the component.

However, it did not help with finding properties to formally verify. From reading the informal requirements it is evident that these where not written with formal methods in mind. For example, the locking and unlocking of railway elements is an important mechanism. But there is no concrete requirement stating that the goal of this mechanism is to prevent race conditions. So, creating the appropriate and interesting properties is inevitably a creative process.

The experience and knowledge gained in this proof-of-Concept was not all captured in the specification of Section 5.3. However, it can be used to think about goals for the future of the project.

Clearly, the model presented in Section 5.3 for the EMC and the verified properties are somewhat simple. So, the complexity lies not in the interactions of independent systems. This is because we have considered the minimal configuration of only one instance of the EMC and some environment. However, as described in Section 4.3, there can be more than one instance of the EMC in one Astris instance. This introduces complexity in the interaction of systems and opens up the possibilities to check

more interesting properties where the answer is not immediately obvious. Possibly because of the interleaving of actions. Some examples:

- Is it possible that two applications lock the same element?
- Is it possible that two applications use (reserve for use) the same element?
- Is it possible that an element can be reserved for use forever?
- What happens when the EMC is in the process of operating an element and at the same time that element sends a conflicting state change?

There are also other areas worth looking into but require some more research. For example: Is it possible to send conflicting instructions to railway elements? But first we must then know what these 'conflicts' are exactly.

6 Expected end result

At a high level we would like it if the ICT department knows what formal specification is and has gained some experience and knowledge through this project. This experience and the produced results should help them to make decisions on whether or not they want to apply formal specification in their processes or not. Part of these results is a recommendation.

Concretely, at the end we hope to have an answer to the problem statement. That is, a formal specification of a part of a system has been made and we have demonstrated how verification and testing is possible with formal specification.

From the state-of-the-art analysis section we have gained some confidence that formal specification is able to solve the problems the ICT department hopes it solves.

7 Proposed research and development approach

For the purpose of answering the question posed by the problem statement the in Section 4.4 discussed component, i.e. the Element Manager Component, will be formally specified. Moreover, a number of requirements (e.g. safety properties, liveness, deadlock) will be formally verified against this formal model.

Before the Proof-of-Concept, described in Section 5, was conducted it was unclear how the translation from the ICT department's informal requirements to formal specifications and verification requirements should be done. The approach taken for the Proof-of-Concept proved to be very useful for this purpose. Therefore, we will adopt this method for the project. The difference is that much of the groundwork has

already been done in the Proof-of-Concept. The focus will change to, as predicted, the details of the service operations.

7.1 Tooling

From Section 3.3 it is clear that a number of formal method tools can do specification and verification, but not a lot of tools can do test generation or code generation. In fact, not one tool does both. Since one of ProRail's main goals is seeing how formal methods can help with testing, it would be wise to choose a tool which supports this. From Table 3 it is clear that only LOTOS and mCRL2 meet these criteria.

mCRL2 is a formal methods toolkit that can be used for this project. Not only are we already familiar with this language and the toolkit, we can also expect support from the University. Whereas if we choose a different language, we do not have these advantages. Moreover, learning a new formal language to a level sufficient for this project may take weeks. Also, we can investigate the suitability of the mCRL2 toolkit in the train control domain.

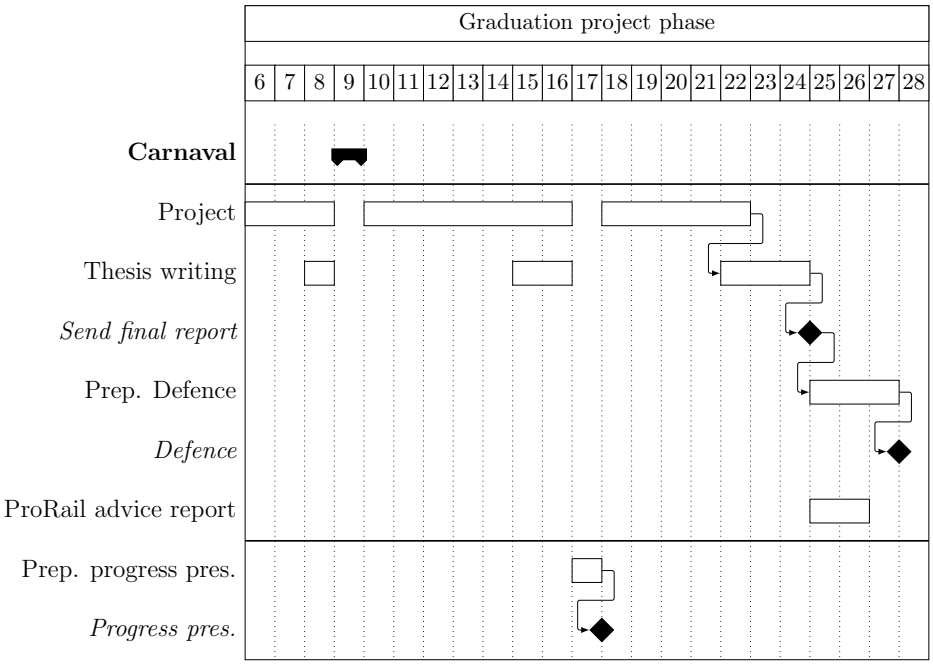
The above, in conjunction the purpose of this project to investigate specification, verification and testing, mCRL2 is the best pick.

If in a future project a focus on e.g. code generation is required, then mCRL2 is probably not a suitable language for that project.

8 Project planning

In this section the a coarse grain planning for the future of this project is presented. The frequent meetings are listed as well.

The planning of coarse grain activities for the upcoming weeks (Monday Feb 3rd - Monday July 6th) is shown in the Gantt chart below up until approximately the Master thesis defence.



The Gantt chart is divided into three segments denoted by the horizontal lines. In the first segment the Carnaval vacation is denoted. In the second segment the activities for the project are displayed. The last segment contains the preparation for a presentation on the progress of the project. The exact week may vary.

One might notice how the *Project* is not subdivided into multiple tasks. This is because from the Proof-of-Concept we have learned this is an iterative process of specification and verification. Concretely, during this time, we want to adapt the formal specification to include at least two instances of the EMC and further expand on the service requests. Then we can verify at least the properties from the examples described in Section5.4 and some more.

Additionally there are some tasks to complete for the graduation process:

- As soon as the preparation phase is finished successfully a *Graduation plan form* must be handed in to formally start the graduation project
- As reflected in the chart a good quality copy of the final report must be sent to the members of the assessment committee at least 15 working days before the defence

- At least four weeks before the defence an *Assessment committee form* must be handed in formally confirming the members of the assessment committee
- Around the time of the defence, an appointment must be made with the exam committee at least four weeks before their meeting to evaluate the Master thesis
- Book a room for the defence for at least two hours

In the below listing all frequently scheduled meetings are described in terms of frequency, participants (other than myself) and purpose. A distinction is made between meetings from ProRail and TU/e.

From ProRail:

- At least once every week; Bert Bossink; Meeting for in-depth questions and discussing issues
- Once per month; Bert Bossink, the team manager and the internship ambassador of the ICT department; General progression of the project
- Once per month; ProRail Team; Optional opportunity to present the progress of the project to colleagues

From TU/e:

- Once every two weeks; Bas Luttik and Mark Bouwman; Discuss the progression of the project
- Once every week; Mark Bouwman; Questions and support on mCRL2

9 Conclusion

A feasibility study has been conducted. In the above sections the problem has been discussed and a concrete problem statement has been defined. Afterwards we have looked into the state-of-the-art of the use of formal methods with a focus on the applications in the railway domain.

Then, the system under consideration has been discussed and it was determined the Element Manager Component will be considered for this project. Moreover, a Proof-of-Concept has been produced a small part of the EMC is formally specified and some properties have been successfully formally verified.

Finally, with the results of the Proof-of-Concept as our guide we have been able to define a concrete and realistic expected end result, propose a research and development approach and an apt planning denoting the steps in time needed to reach the expected end result within this project's allotted time.

Thus, we can conclude that the project is feasible.

10 Bibliography

[1] Maarten Bartholomeus, Bas Luttik, and Tim Willemse. Modelling and analysing ertms hybrid level 3 with the mcrl2 toolset. In Falk Howar and Jiří Barnat, editors, *Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Proceedings*, Lecture Notes in Computer Science, pages 98–114, Germany, 1 2018. Springer.

[2] Davide Basile, Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, Franco Mazzanti, Andrea Piattino, Daniele Trentini, and Alessio Ferrari. On the industrial uptake of formal methods in the railway domain. In Carlo A. Furia and Kirsten Winter, editors, *Integrated Formal Methods*, pages 20–29, Cham, 2018. Springer International Publishing.

[3] Mark Bouwman, Bob Janssen, and Bas Luttik. Formal modelling and verification of an interlocking using mcrl2. In Kim Guldstrand Larsen and Tim Willemse, editors, *Formal Methods for Industrial Critical Systems*, pages 22–39, Cham, 2019. Springer International Publishing.

[4] CENELEC. EN 50128 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. Standard, CENELEC, Brussels, BE, July 2011.

[5] CENELEC. EN 50126 - Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS). Standard, CENELEC, Brussels, BE, October 2017.

[6] Alessandro Fantechi. Twenty-five years of formal methods and railways: What next? In Steve Counsell and Manuel Núñez, editors, *Software Engineering and Formal Methods*, pages 167–183, Cham, 2014. Springer International Publishing.

[7] Marie-Claude Gaudel. Formal specification techniques (extended abstract). In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 223–227, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[8] Yi-Ling Hwong, Vincent J. J. Kusters, and Tim A. C. Willemse. Analysing the control software of the compact muon solenoid experiment at the large hadron collider. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, pages 174–189, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[9] Axel van Lamsweerde. Formal specification: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, New York, NY, USA, 2000. ACM.

[10] N.N. www.mcrl2.org - analysing system behaviour, 2019. accessed December 13th, 2019.

[11] Daniela Remenska, Tim A.C. Willemse, Kees Verstoep, Jeff Templon, and Henri Bal. Using model checking to analyze the system behavior of the lhc production grid. *Future Generation Computer Systems*, 29(8):2239 – 2251, 2013.

[12] Martin Renkema. Software Requirements Specification; Astris - Elementen. Standard, ProRail, November 2019.

[13] Ian Sommerville. Formal specification. In *Software Engineering*, chapter 27. Addison-Wesley Publishing Company, USA, 9th edition, 2010.

[14] Wikipedia contributors. Prorail — Wikipedia, the free encyclopedia, 2019. accessed December 4th, 2019.

[15] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.

Appendices

A mCRL2 model

```
%% Notities over action notatie:  Neem een actie 'actie'.  Dan is ' actie'
de gesynchroniseerde actie van 'R_actie' en 'S_actie' welke Receive en Send
voorstellen.

sort
```

```
componentStatus = struct Actief ?  isActief
| InternActief ?  isInternActief
| Passief ?  isPassief
| Geinstantieerd ?  isGeinstantieerd
| NietActief ?  isNietActief;
Acceptatie = struct Geaccepteerd | NietGeaccepteerd;
ElementID = struct Wissel | Sein;
OpdrachtID = struct opdr1 | opdr2;
Lock = struct Pair(element:ElementID, opdr:OpdrachtID);
Locks = Set(Lock);

map
lockCondition :  ElementID # OpdrachtID # Locks -> Bool;
var
ep:ElementID;
op:OpdrachtID;
ll:Locks;
eqn
lockCondition(ep, op, ll) =
!(exists opp:OpdrachtID . (Pair(ep, opp) in ll && !(opp == op)));

act
%% initiele acties
instantieerComponent, R_instantieerComponent, S_instantieerComponent;
startComponent, R_startComponent, S_startComponent :  componentStatus;

%% Beheer service request acties
verwerkComponentToestandWijziging, R_verwerkComponentToestandWijziging,
S_verwerkComponentToestandWijziging :  componentStatus;
verwerkParameterWijziging, R_verwerkParameterWijziging,
S_verwerkParameterWijziging;
verwerkInterfaceToestandWijziging, R_verwerkInterfaceToestandWijziging,
S_verwerkInterfaceToestandWijziging;

%% Respons acties
Respons, R_Respons, S_Respons :  Acceptatie;

%% Placeholder(s) for yet to fill in details
placeholder;
skip;
```

```
proc
initComponent = R_instantieerComponent .
S_verwerkComponentToestandWijziging(Geinstantieerd) .
(R_startComponent(Actief) .  S_Respons(Geaccepteerd) .
Component(Actief, {}) +
R_startComponent(InternActief) .  S_Respons(Geaccepteerd) .
Component(InternActief, {}) +
R_startComponent(Passief) .  S_Respons(Geaccepteerd) .
Component(Passief, {}));

Component(s:componentStatus, l:Locks) =
ElementGebruikService(s, l) +
BeheerService(s, l);

act
LockElement, R_LockElement, S_LockElement :  ElementID # OpdrachtID;
UnlockElement, R_UnlockElement, S_UnlockElement :  ElementID # OpdrachtID;
ControleerBedienbevoegdheid, R_ControleerBedienbevoegdheid,
S_ControleerBedienbevoegdheid;
NeemElementInGebruik, R_NeemElementInGebruik, S_NeemElementInGebruik;
GeefElementGebruikVrij, R_GeefElementGebruikVrij, S_GeefElementGebruikVrij;
NeemElementActueelInGebruik, R_NeemElementActueelInGebruik,
S_NeemElementActueelInGebruik;
GeefElementActueelGebruikVrij, R_GeefElementActueelGebruikVrij,
S_GeefElementActueelGebruikVrij;
wijzigenBeginseinReservering, R_wijzigenBeginseinReservering,
S_wijzigenBeginseinReservering;
wijzigenBedienbevoegdheid, R_wijzigenBedienbevoegdheid,
S_wijzigenBedienbevoegdheid;
VerwerkBevNLAntwoord, R_VerwerkBevNLAntwoord, S_VerwerkBevNLAntwoord;
VerwerkBevNLElementWijziging, R_VerwerkBevNLElementWijziging,
S_VerwerkBevNLElementWijziging;
MeldGebruik, R_MeldGebruik, S_MeldGebruik;
locked, unlocked;
proc
ElementGebruikService(s:componentStatus, l:Locks) =
(sum e :  ElementID . sum o :  OpdrachtID . R_LockElement(e, o) .
((isActief(s)||isInternActief(s)) -> ((lockCondition(e, o, l)) -> locked
<> skip)
```

```
.  S_Respons(Geaccepteerd) <> S_Respons(NietGeaccepteerd)) .
Component(s,if(lockCondition(e,o,l),l+{Pair(e,o)},l))) +
(sum e :  ElementID . sum o :  OpdrachtID . R_UnlockElement(e, o) .
((isActief(s)||isInternActief(s)) -> unlocked.S_Respons(Geaccepteerd)
<> S_Respons(NietGeaccepteerd)) .
Component(s,l - {a:Lock | Pair(e, opdr(a)) in l})) +
R_ControleerBedienbevoegdheid.Component(s,l) +
R_NeemElementInGebruik.Component(s,l) +
R_GeefElementGebruikVrij.Component(s,l) +
R_NeemElementActueelInGebruik.Component(s,l) +
R_GeefElementActueelGebruikVrij.Component(s,l) +
R_wijzigenBeginseinReservering.Component(s,l) +
R_wijzigenBedienbevoegdheid.Component(s,l) +
R_VerwerkBevNLAntwoord.Component(s,l) +
R_VerwerkBevNLElementWijziging.Component(s,l) +
R_MeldGebruik.Component(s,l);

act
activeerComponent, R_activeerComponent, S_activeerComponent;
activeerinternComponent, R_activeerinternComponent, S_activeerinternComponent;
passiveerComponent, R_passiveerComponent, S_passiveerComponent;
stopComponent, R_stopComponent, S_stopComponent;
geefComponentToestand, R_geefComponentToestand, S_geefComponentToestand
:  componentStatus;
wijzigParameter, R_wijzigParameter, S_wijzigParameter;
geefParameterToestand, R_geefParameterToestand, S_geefParameterToestand;
checkParameter :  Bool;
proc
BeheerService(s:componentStatus,l:Locks) =
%% BeheerFramework %
R_activeerComponent .  S_Respons(Geaccepteerd) .
S_verwerkComponentToestandWijziging(Actief)
.  (R_Respons(Geaccepteerd) + R_Respons(NietGeaccepteerd)) .  Component(Actief,l)
+
R_activeerinternComponent .  S_Respons(Geaccepteerd) .
S_verwerkComponentToestandWijziging(InternActief)
.  (R_Respons(Geaccepteerd) + R_Respons(NietGeaccepteerd)) .
texttt      Component(InternActief,l) +
R_passiveerComponent .  S_Respons(Geaccepteerd) .
S_verwerkComponentToestandWijziging(Passief) .
```

```
(R_Respons(Geaccepteerd) + R_Respons(NietGeaccepteerd)) .  Component(Passief,l)
+
R_stopComponent .  S_Respons(Geaccepteerd) .
S_verwerkComponentToestandWijziging(NietActief) .
(R_Respons(Geaccepteerd) + R_Respons(NietGeaccepteerd)) .  Component(NietActief,l)
+
R_geefComponentToestand(s) .  S_Respons(Geaccepteerd) .  Component(s,l);

proc
%% Environment
initEnvironment = S_instantieerComponent .
(sum s:componentStatus .  R_verwerkComponentToestandWijziging(s) .
(S_startComponent(Actief) .  R_Respons(Geaccepteerd) +
S_startComponent(InternActief) .  R_Respons(Geaccepteerd) +
S_startComponent(Passief) .  R_Respons(Geaccepteerd) ) .  Environment(s)
);
act
activeerInterface, R_activeerInterface, S_activeerInterface;
deactiveerInterface, R_deactiveerInterface, S_deactiveerInterface;
proc
Environment(s:componentStatus) =
EnvironmentElementService(s) +
EnvironmentBeheerService(s);

EnvironmentElementService(s:componentStatus) =
(sum e :  ElementID . sum o :  OpdrachtID . S_LockElement(e, o) .
(R_Respons(Geaccepteerd) + R_Respons(NietGeaccepteerd)) .  Environment(s))
+
(sum e :  ElementID . sum o :  OpdrachtID . S_UnlockElement(e, o) .
(R_Respons(Geaccepteerd) + R_Respons(NietGeaccepteerd)) .  Environment(s));

EnvironmentBeheerService(s:componentStatus) =
S_activeerComponent .  (R_Respons(Geaccepteerd) +
R_Respons(NietGeaccepteerd)) .  Environment(s) +
S_activeerinternComponent .  (R_Respons(Geaccepteerd) +
R_Respons(NietGeaccepteerd)) .  Environment(s) +
S_passiveerComponent .  (R_Respons(Geaccepteerd) +
R_Respons(NietGeaccepteerd)) .  Environment(s) +
S_stopComponent .  (R_Respons(Geaccepteerd) +
R_Respons(NietGeaccepteerd)) .  Environment(s) +
```

```
(sum t:componentStatus .  R_verwerkComponentToestandWijziging(t)
.  S_Respons(Geaccepteerd) .  Environment(t));


init
hide({skip},
allow({
locked,unlocked,
instantieerComponent, startComponent, verwerkComponentToestandWijziging,
verwerkParameterWijziging,
Respons,
LockElement, UnlockElement, ControleerBedienbevoegdheid, NeemElementInGebruik,
GeefElementGebruikVrij, NeemElementActueelInGebruik,
GeefElementActueelGebruikVrij,
wijzigenBeginseinReservering, wijzigenBedienbevoegdheid,
VerwerkBevNLAntwoord, VerwerkBevNLElementWijziging, MeldGebruik,
activeerComponent, activeerinternComponent,
passiveerComponent, stopComponent,
geefComponentToestand,
wijzigParameter, geefParameterToestand,
activeerInterface, deactiveerInterface,
skip, placeholder
},
comm({
S_instantieerComponent|R_instantieerComponent->instantieerComponent,
S_startComponent|R_startComponent->startComponent,
S_verwerkComponentToestandWijziging|R_verwerkComponentToestandWijziging
->verwerkComponentToestandWijziging,
S_verwerkParameterWijziging|R_verwerkParameterWijziging
->verwerkParameterWijziging,
S_Respons|R_Respons->Respons,
S_LockElement|R_LockElement->LockElement,
S_UnlockElement|R_UnlockElement->UnlockElement,
S_ControleerBedienbevoegdheid|R_ControleerBedienbevoegdheid->
ControleerBedienbevoegdheid,
S_NeemElementInGebruik|R_NeemElementInGebruik->NeemElementInGebruik,
S_GeefElementGebruikVrij|R_GeefElementGebruikVrij->GeefElementGebruikVrij,
S_NeemElementActueelInGebruik|R_NeemElementActueelInGebruik->
NeemElementActueelInGebruik,
S_GeefElementActueelGebruikVrij|R_GeefElementActueelGebruikVrij->
```

```
GeefElementActueelGebruikVrij,
S_wijzigenBeginseinReservering|R_wijzigenBeginseinReservering->
wijzigenBeginseinReservering,
S_wijzigenBedienbevoegdheid|R_wijzigenBedienbevoegdheid->
wijzigenBedienbevoegdheid,
S_VerwerkBevNLAntwoord|R_VerwerkBevNLAntwoord->VerwerkBevNLAntwoord,
S_VerwerkBevNLElementWijziging|R_VerwerkBevNLElementWijziging->
VerwerkBevNLElementWijziging,
S_MeldGebruik|R_MeldGebruik->MeldGebruik,
S_activeerComponent|R_activeerComponent->activeerComponent,
S_activeerinternComponent|R_activeerinternComponent->activeerinternComponent,
S_passiveerComponent|R_passiveerComponent->passiveerComponent,
S_stopComponent|R_stopComponent->stopComponent,
S_geefComponentToestand|R_geefComponentToestand->geefComponentToestand,
S_wijzigParameter|R_wijzigParameter->wijzigParameter,
S_geefParameterToestand|R_geefParameterToestand->geefParameterToestand,
S_activeerInterface|R_activeerInterface->activeerInterface,
S_deactiveerInterface|R_deactiveerInterface->deactiveerInterface
},
initComponent || initEnvironment)));
```