Model Checking Geographically Distributed Railway Control Systems

Michel Bøje Randahl Nielsen

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of this project is to investigate model checking as a verification method for analysis of distributed railway control systems wrt. safety.

To drive this investigation an engineering concept of a distributed railway interlocking system is conceived and described. The concept is distilled into an abstract generic model in a model checking language. Furthermore is a tool developed to assist in generating concrete models from the generic model, that are both valid and constrained to help reduce the state space to be searched when model checking the concrete model instances.

The outcome of the project is not only a verified engineering concept, an abstract model of the concept and a tool to assist in exploring concrete instances an abstract model, -but also an example of how an engineering concept can be modeled as an abstract model and verified through model checking.

Summary (Danish)

Målet med dette projekt er at undersøge model checking som metode til verificering og analyse af sikkerheden i distribuerede tog kontrol systemer.

For at motivere undersøgelsen er et konkret engineering koncept udarbejdet og beskrevet. Konceptet er derefter destileret ned til en abstrakt generisk model i et model checker sprog. Yderligere, er et værktøj udviklet til at assistere med at generere instanser af den generiske model som er korrekte og afgrænsede, således at de reducerer det tilstandsrum som skal gennemsøges af et model checker værktøj.

Resultatet af projektet er ikke bare et verificeret engineering koncept, en abstrakt model af konceptet og et værktøj til at hjælpe med at udforske modelen gennem konkrete instanser, men er også et eksempel på hvordan et konkret engineering koncept kan modeleres som en abstrakt model og verificeres gennem model checking.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Computer Science and Engineering.

The thesis deals with the investigation of use of model checking as a means of verification of safety properties in railway interlocking systems.

The thesis has been written in the period from April 1 2016 to October 21 2016 under supervision of associate professor Anne Elisabeth Haxthausen and professor Alessandro Fantechi, and is worth 35 ECTS credits.

The thesis consists of the following report and an associated CD that contains source code files of a tool generated as part of the project, a compiled executable version of the tool and samples of generated models and XML files which can be used in the process of generating models.

Lyngby, 21-October-2016

Michel Bøje Randahl Nielsen

Acknowledgements

I would like to thank my supervisor Anne Elisabeth Haxthausen for all her guidance and support throughout the project.

I would also like to thank Alessandro Fantechi who not only has acted as co-supervisor during the project giving advises, but also has introduced me to the field of model checking prior to the project.

Furthermore would I like to thank Hugo D. Macedo for helping out with the RobustRails tools.

At last would I like to thank my parents and sister for all their support, - especially my dad for assisting with proof reading parts of the thesis and my sister Melanie for helping out printing drafts and helping with producing the final prints.

Contents

Summary (English) **i**

Summary (Danish) **iii**

Preface **v**

Acknowledgements **vii**

1 Introduction **1**
1.1 Content of the thesis work 3

2 Railway Domain **5**
2.1 Terminology and Components of Railway Systems 5
2.1.1 Recent Developments 7
2.2 Safety Measures and Interlocking Systems 7
2.3 Route Reservation Methods 9

3 Formal Specification and Verification of Software Systems **11**
3.1 Common Methods for Ensuring Corectness in Software Systems . 12
3.1.1 Type Checking 12
3.1.2 Testing 13
3.1.3 Peer Reviews and Pair Programming 13
3.1.4 Model Checking 13
3.2 Model Checking 14
3.2.1 Deriving and Specifying Formal Models for Model Checking 14
3.2.2 Temporal Logic and Verification of Properties 15

4 The UMC modeling language 19
4.1 About UMC 19
4.2 Structure and Semantics 20
4.2.1 Class definitions 20
4.2.2 Object declarations 22
4.2.3 Abstraction rules 23
4.3 UCTL properties 23

5 An Engineering Concept of a Geographically Distributed Interlocking System 25
5.1 The overall idea 25
5.2 The Communication Scheme - Two-phase Commit Protocol . . . 26
5.3 Route Reservation 28
5.4 Releasing the Reserved Track Segments - Sequential Release . . . 29
5.5 A Practical Implementation of the System 30
5.6 Discussion 30

6 Modeling the Geographically Distributed Interlocking System in UMC 33
6.1 Defining the Model in UMC 33
6.1.1 The Train Class 34
6.1.2 The Linear Class 39
6.1.3 The Point Class 47
6.2 Model Properties 50
6.2.1 No Collision 51
6.2.2 No Derailments 52
6.2.3 Progress property - arrival of all trains at their destinations 54
6.2.4 No message loss 55
6.3 Scenarios 56
6.3.1 A Successful Route Reservation 56
6.3.2 A train traversing its successfully reserved route 58
6.3.3 Point positioning during reservation 61
6.3.4 Point malfunction during reservation 62
6.3.5 Attempt to reserve a route intersecting with an already reserved route 64
6.4 Discussion 66
6.4.1 Train lengths and movement on track segments 67
6.4.2 Point Lengths 68
6.4.3 Point Machine 68
6.4.4 Repairment of malfunctioning points 68

7 Model generating tool **71**
7.1 Functionality of the Tool 72
7.1.1 Track Layout and Route Extraction from XML files . . . 73
7.1.2 Route Validation 75
7.1.3 Enforcement of length constraints 77
7.1.4 Creation of Object Instantiations and Modeling Language Specific Constructs 80
7.2 Implementation 86
7.2.1 Modules 88
7.3 Extending the Model Generator to support other modeling languages 96

8 Experiments **97**
8.1 Performing the experiments 97
8.2 Experiments performed 98
8.2.1 Two trains and varying route lengths 98
8.2.2 Varying number of trains 102
8.2.3 Experiments with particular layouts from XML files . . . 103
8.3 Discussion 105

9 Future work **107**
9.1 Generic model enhancements 107
9.1.1 Repairment of faults 108
9.1.2 Point Machines 108
9.1.3 Modeling more types of faults 108
9.2 Usability and performance improvements for the tool 109
9.2.1 Enhance the user experience with a route selection GUI . 109
9.2.2 Extend tool to support more modeling languages 109
9.2.3 Model checking of huge railway networks 109

10 Conclusion **111**

A User Guide for the Model Generating Tool **113**
A.1 Model generation through the model generator tool 113
A.2 Model generation using the scripting tools 114
A.3 Model checking the generated models with the UMC web tool . . 121

B UMC BNF **123**

C Generic UMC Model **127**

D UMC model delta **137**

E Tool Source Code **141**
E.1 Compiler version and third party packages . . . 141
E.2 Auxiliary dependency files . . . 142
E.2.1 Project files and compile order . . . 142
E.2.2 Sample XML file used to bootstrap the typeprovider library 143
E.3 Source code . . . 143
E.3.1 Utils.fs . . . 143
E.3.2 InterlockingModel.fs . . . 145
E.3.3 UMCTrainClass.fs . . . 156
E.3.4 UMCLinearClass.fs . . . 158
E.3.5 UMCPointClass.fs . . . 162
E.3.6 UMC.fs . . . 165
E.3.7 XMLExtraction.fs . . . 175
E.3.8 ScriptTools.fs . . . 181
E.3.9 MiniModelGenerator.fs . . . 185
E.3.10 Prelude.fsx . . . 187
E.3.11 Script.fsx . . . 187
E.4 Tests . . . 188
E.4.1 Tests.Utils.fs . . . 188
E.4.2 Tests.InterlockingModel.fs . . . 191

F Experiment Scripts **193**
F.1 SimpleTwoTrains.fsx . . . 193
F.2 BranchManyTrains.fsx . . . 195

G XML sample **197**

Bibliography **203**

Chapter 1

Introduction

Our world is becoming increasingly more automated, improving our living conditions and providing comfort and safety. Today, difficult tasks such as, for example, controlling an aircraft is by large either controlled by or assisted by automation, and currently tech companies and car manufacturers are pushing the limits for autonomous car driving. As we apply automation to more areas and expand responsibilities of already automated systems, -the complexity increases. This increasing complexity becomes a huge task to handle for the engineers designing the systems. Often systems are so complex that it is impossible for the engineers to get a full overview of the given system and confidently predict its behavior.
Model checking is a methodology which was invented to help analyse such complex systems, and has been successfully applied, for example, in the analysis of concurrent systems.

Trains has become an essential part of many peoples life. In big cities millions of people commute daily to school or jobs by different variations of train systems, such as inter city trains, urban railways, streetcars, subways or light railways.
Every day goods and people are transported within and across borders all over the world by trains. And in some places high speed trains are means of transportation which is competitive with other typical ways of fast transportation such as airplanes.

When designed and utilized properly, railway transportation has potential to out-compete many other means of transportation in regards to efficiency. As focus on energy consumption increases around the world, it is very likely that railway transportation will get even more attention in the future.

Even though transportation by train is one of the safest means of transportation today, fatal accidents still happens. In 2016 alone, at least three major train accidents has occurred.
Early 2016 a fatal accident happened in Germany where two trains ended up in a frontal collision on a two-way track. The cause for this incident has been revealed to be a human error caused by a track operator, who accidentally sent warning signals to the wrong recipients instead of the two trains that were on a collision course[mInB16]. Later on in 2016, again two trains ended up in a head-on frontal collision in Italy, and yet again the investigations seems to point to a human track operator error. Yet again later on in 2016 another train accident happened in New Jersey, where a train ran through an end-of-track-barrier and into a wall at high speed, once again a human error happened and there was no fail-safe technology to take over and prevent the accident.

Railway systems are in general very safe and the probability of a train being involved in an accident is quite small compared to other means of transportation. However, guarenteing safety in huge railway systems with many intersections and lots of traffic can be very challenging. Thus the invention of, so called, *interlocking systems* which are systems that serve to ensure safe operations of the trains.
There are many other challenges in railway systems, such as scheduling and liveness of trains, and sometimes solutions to these challenges interfere with how the interlocking system operates. This leads to an interest in optimizing the interlocking systems, which in the end possibly makes them even more complex.

Given that so many people rely on trains for transportation, it is naturally important that an effort is put into ensuring safety, availability and reliability. Railway interlocking systems has indeed been analyzed many times before with regard to both safety and liveness.

In this thesis work an engineering concept of a geographically distributed interlocking system, sporting a sequential release mechanism for increased utility of the given railway network, is explored and analyzed through model checking. The work has been done in the context of the RobustRailS research project[Col, Hax] in which research on formal verification of railway control systems is pursued. An important motivating factor for using model checking, is that it is a verification method recommended in railway signalling safety guidelines for software, specified by the European Committee for Electrotechnical Standardization [CEN11].

The work in this thesis is especially inspired by the work described in [Fan12] and [Pao10], where the idea of a geographically distributed interlocking system using a two-phase commit protocol for route reservation, -is presented and modeled. Furthermore does the work presented in this thesis draw a lot of inspiration from the work in [VHP16], where formal methods are applied to verify safety properties of a new Danish interlocking system that features a sequential release mechanism for increased utility of railway network capacities.

1.1 Content of the thesis work

This section describes the chapters of the thesis work that are to follow this chapter.

chapter 2 - Railway Domain
Briefly explains the basic concepts and terminology of the rail way domain. The chapter serves to prepare the reader for the rest of the thesis work where the terminology will be used extensively.

chapter 3 - Formal Specification and Verification of Software Systems
Gives a brief outline of common methods of ensuring correctness in software systems, and ends up briefly explaining the concept of model checking which is the method used in this thesis work.

chapter 4 - The UMC modeling language
Introduces the modeling language utilized for this project.

chapter 5 - An Engineering Concept of a Geographically Distributed Interlocking System
Describes an engineering concept for a distributed railway interlocking system conceived as part of this project.

chapter 6 - Modeling the Geographically Distributed Interlocking System in UMC
Describes the translation of the engineering concept into an abstract generic model that can be used for model checking.

chapter 7 - Model generating tool
Describes an implementation of a tool for generating concrete model instances based on the generic model.

chapter 8 - Experiments

Presents and elaborates over a set of model checking experiments performed with different concrete model instances.

chapter 9 - Future work
Elaborates over ideas for extensions to the tool and improvements for the abstract generic model are presented.

chapter 10 - Conclusion
Sums up the work and yields conclusions in relation to the project.

Chapter 2

Railway Domain

What cannot be imagined cannot even be talked about.

Ludwig Wittgenstein

2.1 Terminology and Components of Railway Systems

The railway domain is almost two centuries old, and thus it makes sense that a distinctive terminology for talking about railway systems has developed. Fortunately the English terminology has evolved differently in Europe and America. In this thesis work, the European terminology will be used.

The basic elements that make up a railway system, are *points*, *signals*, *interlocking systems*, *track circuits*, *main tracks (linear tracks)*, *loops* and *sidings*.
A *point (switch in American terminology)* is a branching from the main track with a mechanical functionality to switch between the main track and the branch. In old systems points required a human operator to manipulate a hand-operated lever, however in modern railway systems the points are operated by

a point machine which is an electric device that can perform the switching and can be operated from afar.
In context of a railway layout a point can be described as a straight path with a branch into a diverging path, they do however have many different shapes and multiple points can be composed into complex intersections. Generally speaking, a point can be in one of two states (or *positions*) which are referred to as *plus* and *minus*, where plus denotes that the point is positioned such that the the point connects in a straight line, and minus denotes that the point connects to a diverging path.

Following drawings illustrate the described elements.

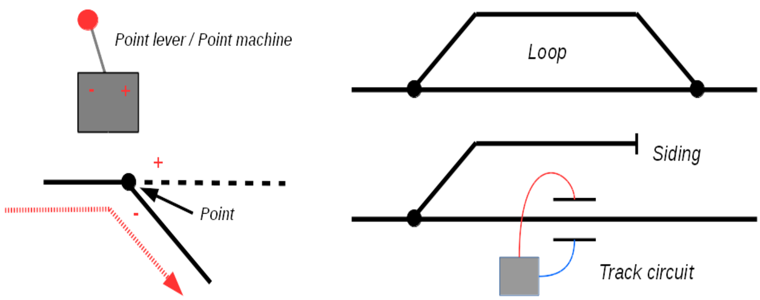

Figure 2.1: Illustration of a point switched to a minus position, a loop, a siding and a track circuit sensor.

A *signal* can either be a physical *signal light* which, for example communicates the occupation status of the coming track segment. A signal can also be a *virtual signal* communicating speed limitations or wait and go messages directly to the operator through an electronic interface in the train, or communicated directly to an autonomous train control system. What is general for the term signal is that it represents a way of communicating things such as stop and go messages or speed up and slow down messages to trains operating on a railway network.

The *main tracks (linear track)* are the regular train tracks, and the term *loop* describes a track which branch out from the main tracks and rejoins them again at a later point. The term *siding* refers to a branching track with a dead end which often is used for maintenance of the trains. A *track circuit* is an electrical sensor which can detect the absence or presence of a train on a track segment. Finally an *interlocking* system, is a control system which is responsible for safe operation of the trains, which at the most basic level involves controlling the signals to avoid conflicting train movements and controlling the points positioning so they are set accordingly for a passing train.

2.1.1 Recent Developments

As communication devices like GSM (Global System for Mobile Communications) become more reliable and cheaper, and other micro processors likewise, it naturally becomes more relevant to use such technologies in systems like railway control systems.

The railway industry is already in progress of moving from *physical light signals* to *virtual signals* in the form of *cab signaling systems*. In a cab signaling system, virtual signals send to the trains are communicated directly to the train operator by some kind of interface.

Another advancement is the automation of train systems, and especially metro systems in big cities. In these systems the trains are made completely autonomous and so is all signaling and point switching. Autonomous systems requires more and advanced sensors, but could potentially factor out some of the human errors that often lead to fatal accidents. As of now, train automation is mostly seen in metro railway systems, because such railway systems are smaller and more confined from the surroundings than typical railway systems are.

2.2 Safety Measures and Interlocking Systems

Train operation is in general a safe way of transportation relative to other means of transportation, which by large is because of the confined nature of trains, since the movement of a train is limited to the given railway track layout. This reduces the safety concerns for operation of individual trains to concerns such as avoiding derailing accidents. This is done by making sure points remain in stable and correct positions, and by making sure the train is operating within the speed limits. However, safety becomes an even bigger concern when multiple trains are utilizing the same railway tracks as collisions becomes possible.

Many measures both in the large and in the small are taken to minimize the risk of accidents. In the small, *track circuit* sensors are for example designed such that they will constantly indicate presence of a train when there is a failure in the sensor.

In the large safety is ensured through an interlocking system. Railway systems are often composed of a central operation control center, an interlocking system and a signaling system which is either fully or partially controlled by the interlocking system.

At the central operation control, the itinerary plan is created and the execution of the plan is carefully monitored by observing the states of signal lights and sensors on the railways. The light signals can be controlled from the central control center, however the control usually goes through an interlocking system which ultimately is responsible for maintaining safety while executing the plan.

The interlocking system will constantly monitor sensor readings and location data for the trains and take actions to ensure that accidents are prevented. It does so by keeping a record the current train routes and by controlling the points and signals, where the signals constitutes of stop and go signals and signals to speed up or slow down. Traditionally the train routes were registered in interlocking control tables, and then the interlocking system would generate a proper execution order for the routes.

In order to guarantee the safety of the trains in the railway system, following requirements must be met.[TV09]

- Track sections in front of the train must be clear from other trains until it has passed.
- Points on the route of the train must be set to the correct positions.
- Speed changes of a train must be applied in sufficient time in order to slow down or speed up to reach permitted speed.

The noble task of the interlocking system is to avoid following situations at all times.

- Head to head collision, which can happen if two trains coming from opposite directions are able to occupy the same track segment. This is probably the most fatal type of error.
- Head to tail collision, this can again happen if two trains are able to occupy the same track segment at the same time.
- Derailment, a derailment of a train can happen if the train traverses a point which is switched to the wrong direction or if the point is performing the mechanical switching while the train is traversing it.

There are multiple ways for interlocking systems to ensure that the above situations are avoided. The simplest way, is to require routes to be fully reserved before permitting a train to traverse it, and not allowing other trains to traverse routes reserved by other trains.

In the case of trains operated by humans, there is still a risk of trains violating reserved routes. To counter this, some interlocking systems uses flank protection, which means that points neighboring other points on the reserved route, will be locked into a positioning such that they are disconnected from the point on the reserved route. This will effectively divert foreign trains from over-running the reserved route.

2.3 Route Reservation Methods

One of the most common methods of ensuring safety along the route of a train, is to fully reserve the whole route and marking it as locked such that no other trains can use it. However, to ensure liveness it is necessary to release the route again at some point. One approach, called *sequential release*, is to release a track segment as soon as the train which reserved it has left it. Another approach is to, strictly, not release anything on the reserved path until the train has completely finished its route. The latter approach is the simplest approach to making the system safe, however it also leads to a poorer utilization of available resources than what can be achieved with sequential release.

Another more extreme version of sequential release, is to define so called *moving safety distance blocks* around the trains based on their braking distance. This approach requires very precise sensory data input, but can in theory optimize the utilization of resources. One problem with this approach is that the moving block is more of a continuous event rather than discrete and this makes it more difficult to model check.

Chapter 3

Formal Specification and Verification of Software Systems

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Edsger W. Dijkstra

Automation has an increasing important role in modern society. From simple tasks such as dispensing sodas to thirsty customers, to more important tasks such as handling money transactions, to more complex tasks such as ensuring safety on railways, in airplanes or even controlling cars autonomously.

The automation is implemented by engineers and software developers who are nothing but humans, and humans tend to make errors. Given the continuous increase in complexity, these systems are bound to contain at least a few errors. We have already experienced many incidents through history where a bug or faulty implementation manifested itself and led to disasters such as fatal overdoses of radiation[1] or huge loss of money and wasted effort[2].

3.1 Common Methods for Ensuring Corectness in Software Systems

Given that the consequences of a faulty implementation can be so severe and have such awful consequences, it is very important that engineers and software developers are able to guarantee the safety of the systems they develop. Thus ensuring correctness and safety in software systems is an important pursuit, and many techniques and methodologies has been investigated and utilized with varying degrees of success through time.

3.1.1 Type Checking

Back in the 1970's, the American military were increasingly concerned about safety, correctness and composeability of the software they produced, and as a result they ended up sponsoring the development of the strongly typed programming language Ada. The strong typing system and the type checker, assisted developers to construct more safe and correct software. Ada quickly came to be one of the preferred languages to be used when developing safety critical software, -not only for the American army but also in industries such as the aerospace industries.

Recently a dependently typed programming language Idris[BRA13] has been developed facilitating types as first-class language constructs and with the goal of making dependent types and proof assistant features more accessible for software developers.

[1]Between 1985 and 1987 patients were given an overdoses of radiation due to a concurrent programming error.[Wikc]

[2]In 1999 a mars orbiter probe crashed uncontrollably into the atmosphere of Mars. Investigation later revealed that two communicating sub-systems were implemented with different units of measures in mind.[Wika]

3.1.2 Testing

Perhaps the most common approach to ensuring safety and correctness in software systems is testing, where especially unit-testing has catched on in popularity. Unit testing is used for both black box and white box testing systems, and has been very popular in the industry since its fairly easy to understand and apply. The popularity of software testing, has even lead to new software development methodologies such as Test Driven Development (TDD), where tests are specified before the actual functionality is implemented.

However, as Edsger Dijkstra famously stated in 1969, testing can only reveal the presence of bugs and not the absence. This drawback is attempted tackled with methods such as property-based-testing, where test oracles are defined as a set of generalized properties, and used together with a randomized testing strategy that gradually attempts to narrow down the randomization to find errors that break the properties.[CH11].

3.1.3 Peer Reviews and Pair Programming

Another commonly applied method of verifying software, is through peer reviews or pair programming. Peer reviews, also known as code reviews, is a simple methodology where code written by one programmer is reviewed by another more experienced programmer who analyses the code for errors and maintainability. Pair programming, involves two programmers sitting together while developing the actual code.
The rationale behind both of these methods, is that drawing from the experience of more than just one developer when producing code will lead to more correct and maintainable software.
However, human verification is costly compared to automated verification, and is rarely enough to fully guarantee correctness in a system.

3.1.4 Model Checking

The last but not least important method of verification to be mentioned here, is model checking. Model checking first started out as a technique for verifying correctness in hardware systems, however it has slowly spread to the domain of software verification as well. Model checking is essentially about verifying a set of *temporal logic properties* by rigorously exploring the state space of a modeled system.

Model checking requires careful modeling of the subject system, since even a normal 32-bit integer in a program will expand the state space to search with a factor of 2^{32}, which quickly becomes very costly to verify. Therefore, to model check a system, it is necessary to distill and abstract the system into a simple model which still reflects the subject system but does so in a way that drastically limits the increase in state space to verify.
Model checking is the type of verification which is studied and described in this thesis work, with one of the motivations being that it is a verification method that promises a complete rigorous verification of the subject modeled system.

3.2 Model Checking

As previously mentioned, model checking is about verifying a set of properties in a system by exploring the state space of the given system. The challenge, however, is to derive a model which represents the behavior of the subject system but with less state space to explore.

Exploring the state space of most systems as they are, is in most cases infeasible. For example, verifying a program dependent on three 32bit integers, would alone require exploration of 2^{96} states. Even by using a computer system capable of exploring $93 * 10^{15}$ states per second, the endeavor of checking all these states would take more than 27000 years. [3] To model check a system, it is therefore necessary to distill the behavior of the subject system into an abstract model which is able to represent the system with fewer states.

3.2.1 Deriving and Specifying Formal Models for Model Checking

In general there are two common methods of deriving such a model. One way is to use the informal description of the system requirements and behavior and formulate it in a formal language. And the other common way is to either automatically extract the model from an existing Software or Hardware implementation or manually derive a formal model through reverse engineering.

The model is typically specified in a formal model checking language as a set of functions, data types and a transition system which describes the behavior of the system and utilizes the defined functions and data types.

[3] The Chinese super computer Sunway TaihuLight, has been bench marked to be able to perform 93 peta FLOPS (floating point operations per second).

3.2.2 Temporal Logic and Verification of Properties

A temporal logic language constitutes of the normal propositional description language (conjunctions, disjunctions and negations) which are used in describing the properties of a given state, and furthermore a set of temporal operators which are used in describing the transitions between the states. The most commonly used temporal languages used in model checking are LTL (Linear Temporal Logic), CTL (Computational Tree Logic) and CTL* which is a less restrictive superset of CTL.

The two most defining operators in temporal logic is the *eventually* operator typically denoted F and the *global* operator which is typically denoted G. The F operator is used to assert that a given property will hold true in some future state, while the G operator specifies that a given property must hold true in every state on a path.
The specified properties are often categorized into one of two categories, where the first is a type of safety property, and the second is a type of liveness property. Safety properties must hold true at all times, so those properties uses the G operator, while the liveness properties must ensure some state eventually is obtained and thus uses the operator F. Since G is used to specify that something must hold true along all states in a path, it can be used specify that a set of properties which are critical for safety, must hold true at all times. Since the F operator is used to specify that some property eventually will become true, it can for example specify that some locked resource will be free again in the future such that a given process wont be waiting forever, and thus the property can be used to specify liveness.

3.2.2.1 Linear Temporal Logic (LTL)

LTL is the simplest of the three above mentioned temporal logic languages. It constitutes the temporal connectives G and F mentioned above and furthermore the connectives X ("next") and U ("until"). The next operator X specifies that the given proposition must hold true in the state following the current. The until operator U is an infix operator written as pUq where p, q are propositions, the until operator states that p must hold true in all states on the path until q is satisfied. LTL is called linear because the properties must hold over a linear path.

3.2.2.2 Computational Tree Logic (CTL)

The logic language of CTL consists of the same operators as in LTL, however CTL is a branching tree logic which means that it is possible to reason about the branches in the tree that unfolds when iteratively expanding all possible states from the transition model to a tree with the initial state as root. CTL therefore also specifies an existential quantifier E and an universal quantifier A. The existential quantifier E is used to specific that the given property hold true along at least one path in the unfolded state space tree, whereas the universal quantifier A specifies that the property must hold true along all branches. CTL is restricted such that any of the other temporal operators in the language must be preceded by a quantifying operator.

Following are illustrations of the various combinations of CTL operators in use.

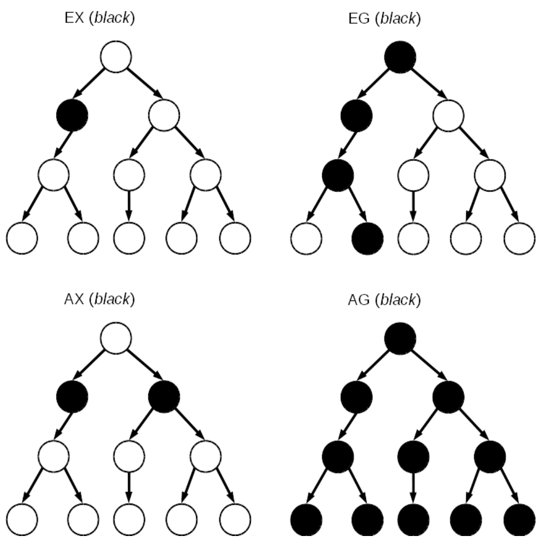

Figure 3.1: EX(*black*) describes that there exists a path where the next state must be black. EG(*black*) describes that there exists a path where all the states must be black. AX(*black*) describes that for all paths the next state must be black. AG(*black*) describes that for all paths all the states must be black.

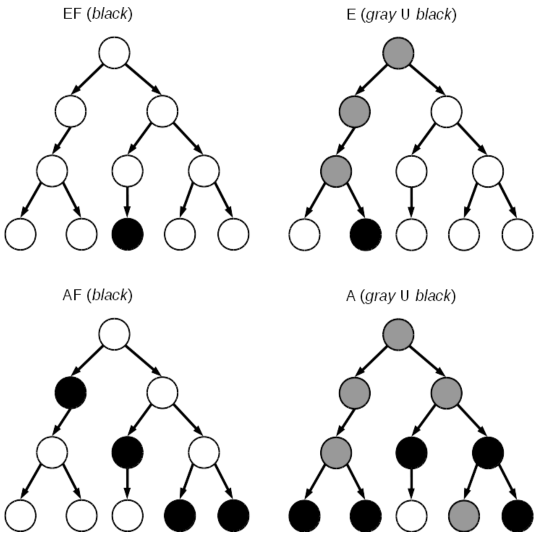

Figure 3.2: EF(*black*) describes that there exists a path where some future state must be black. E(*gray* U *black*) describes that there exists a path where all states must be gray up to a black state. AF(*black*) describes that for all paths there exists some future state which is black. A(*gray* U *black*) describes that for all paths all states must be gray up to a black state.

Chapter 4

The UMC modeling language

This chapter gives a brief introduction to the UMC modeling language[Maz09] and tool-set, which has been used in this project for modeling and verification. It might be useful to refer to Appendix B to get a complete picture of the grammar of the language, and to refer to Appendix C and Appendix D to see a concrete specification made with the language. This chapter solely aims at introducing the subset of the language which has been utilized in this project.

4.1 About UMC

UMC is a modeling language that seeks to make model checking more approachable to non-expert users. The language essentially enables a user to specify textual representations of UML state diagrams, and lends a part of its syntax to the way transitions are described in UML state diagrams.
The language has been designed to be a target language for a more high level language or graphical tool to generate. Therefore, the language itself has been implemented with very limited static type checking capabilities. This also means that the language, for example, doesn't have generalizing functions. The language is object oriented and enables the user to specify a set of generic classes

and a set of object instantiations of the given classes. The classes encapsulates all mutation, such that the only way for objects to manipulate the state of another object is by means of synchronous *operations* or asynchronous *signals* which are queued up in an event queue for each object.

4.2 Structure and Semantics

A full UMC model description consists of a set of class definitions, a set of object instantiations and a set of abstractions. And an abstract skeleton of an UMC model looks as follows.

```
Class classname_1 is
...
end classname_1;

Class classname_n is
...
end classname_n;

Objects
objectname_1 : classname ...;
objectname_n : classname ...;
...

Abstractions {
  Action ... -> ...
  State ... -> ...
  ...
}
```

Following subsections describes the components presented above.

4.2.1 Class definitions

A class definition describes a set of synchronous *operations*, a set of asynchronous *signals*, a set of *variables*, a set of *states* and a set of state *transitions*. The skeleton of a class in UMC looks as follows.

```
Class classname is
Signals
```

```
  ...
Operations
  ...
Vars
  ...
State ... = ...
Transitions:
  State1 -> State2 { ... }
  ...
end classname;
```

The *signals* defines a set of asynchronous messages that can be send to the event queue of an object of the given class that defines the *signals*. A *signal* has a name and an arbitrary number of arguments that it caries along from the sender to the recipient. It is possible to define the types for the arguments, however, they are not statically type checked. The arguments carried along with a signal are simply treated as immutable values making it impossible for a recipient of a signal to manipulate variables of a sender.

The *operations* defines a set of synchronous operations to be invoked on an object of the given class. Just like the *signals*, the *operations* also defines a set of arguments, but furthermore does the *operations* also define a return value to be returned from the *transition* in the object they are invoked upon. Just as with the *signals*, the arguments carried along with an *operation* are treated as immutable values such that the recipient cannot mutate the state of variables on the sender object.

The *vars* defines a set of variables which are used to keep an internal state in an object of the given class. The variables can be integers, booleans, object references or arrays.

The *state* describes a set of modes that an object of the given class can be in. It is possible to define nested states and multiple nested states, however in this project only one layer of states has been used in each of the classes. Note that the first listed state automatically defines the initial starting state of the given class.

The *transitions* describes a set of transitions between the defined *states*. The *transitions* are described with a syntax similar to the syntax used to describe state transitions in UML state diagrams. The syntax for a *transition* looks as follows.

$$eventName[guardExpression]/action$$

Where the *eventName* is either an operation or a signal, which are said to, essentially, trigger the transition. The *guardExpression* is an expression that evaluates to a boolean value, where the expression can be composed of any of the class variables and arguments carried along with the triggering *signal* or *operation*. At last, the *action* is the action carried out when the *transition* is fired. An *action* in this context can be any composition of statements and typical imperative constructs such as if statements, for loops and while loop. The statements permit mutation of the state of the class variables, or sending out *signals* or invoking *operations* upon other objects. Note that if the triggering event is an *operation*, the action will end with a return statement returning whichever type the *operation* defines as return type. Even if the *operation* does not specify a return type, it will still have an implicit return value of zero. It is however valid to omit the return statement in the action, but in that case an implicit return statement returning zero will be executed at the end of an action when model checking the model.

In the expressions used as part of the statements or in the guards, it is possible to use the typical binary operators such as $+$, $-$, $<$, $>$ and $=$ for integers and logical *and* and logical *or* for booleans. *Signals* and *operations* are simply invoked on objects by suffixing the given object reference with $.eventName$ where $eventName$ is the name of the *operation* or *signal* to be invoked on the given object. Array indexes can be accessed and mutated through the classical square bracket notation $array[index]$, however they can also simply be treated as lists by using the *.head* and *.tail* operations.

4.2.2 Object declarations

Once the classes has been defined, a set of *objects* can be declared from the classes. These *object* declarations essentially defines a concrete model, while the classes alone defines the generic model behavior.

The object declarations looks as follows.

```
object_name: class_name
 (object_attribute_1 => initial_value_1,
  ...,
  object_attribute_2 => initial_value_2);

object_name: class_name;
```

Here the *object* attributes refers to the local *variables* declared on the given class. Note that it is possible to define default initial values for *variables* in classes, and in that case it is not necessary to specify any initial value for the given *variable* in the *object* instantiation.

4.2.3 Abstraction rules

An *abstraction* is essentially a construct that captures a given situation for the whole model. For example the situation where a variable on a specific object has a certain value, or the situation where a specific object is in a particular *state*.

The *abstractions* can be defined either as a *state-abstraction* or as an *action-abstraction*. *State-abstractions* captures a certain type of state in the model based on a conjunction of propositions that tests the *variables* or *states* of one or more of the defined *objects*. *Action-abstractions* captures the situation in the model where an *operation* or a *signal* are invoked or emitted. In this project only one *action-abstraction* has been defined, and that is an *abstraction* that captures the situation where the UMC model checker determines that a recipient of a *signal* or *operation*, has no handling for the given *signal/operation* for the current *state* of the receiving *object*. When the UMC model checker encounters such a situation, it will emit a *lostevent signal* on an implicit object called *ERR*. This particular *action-abstraction* has been very useful in the modeling process, where it helped track down *states* for which a given *signal* was not handled.

Note that the *state-abstractions* can only be defined as conjunctions. So if the user prefers to use a disjunction, he must exploit De Morgan's Law and specify the *abstraction* as a negation of a conjunction where all the conjunct propositions inside the conjunction are negated.

4.3 UCTL properties

Independently of the actual model, a set of *properties* can be specified in a UMC tailored modal logic syntax called UCTL, which defines the same modal logic operations as defined in CTL.

What is special for UMC and UCTL, is that the previously mentioned *abstractions* are used in the definition of *properties*.

Chapter 5

An Engineering Concept of a Geographically Distributed Interlocking System

This chapter describes the engineering concept of the railway interlocking system which is to be modeled.
The system described is inspired by the idea of an interlocking system where the trains need to reserve their intended route before being permitted to traverse it and, with a distributed communication setup which is dependent on the geographical layout of the railway track circuits.

5.1 The overall idea

The fundamental purpose of the system, is to ensure that no two trains ever end up in a dangerous situation where they both occupy the same track. This situation is avoided by requiring that trains reserve their route before traversing it. To reserve a route, the train must gather *consensus* about the reservation from all the track elements which constitute the route.

In the described railway interlocking system, the network is composed of track elements, consisting of linear track segments and point segments which all are equipped with a track circuit type sensor.

The concept evolves around trains reserving routes through a two-phase commit protocol that attempts at collecting a consensus between the elements that composes a given route. Furthermore does the concept involve a sequential release mechanism for releasing reserved routes as they are traversed by the reserving train.

In the view of the consensus gathering protocol, the track elements assume the role of communication nodes which can be queried for reservation. Each node maintains a local state and is informed about relevant neighboring nodes as a route is reserved. The system is a distributed system, with all communication being propagated through the nodes based on their geographical relationships.

The following sections describes the two-phase commit protocol and the sequential release mechanism.

5.2 The Communication Scheme - Two-phase Commit Protocol

A two phase commit protocol, is a distributed consensus algorithm which coordinates a set of processes that all participate in the same distributed transaction. The protocol helps determine whether to *commit* or *cancel* a given transaction across a set of distributed processes. In the case of the geographically distributed interlocking system, the processes corresponds to the nodes in the railway network, and the agreed transaction is a route reservation for a given train.

The protocol requires an assigned coordinator responsible for initiating the commit, which for the given type of railway interlocking system corresponds to a train.

The protocol has two phases of message correspondences, each consisting of a *request* message being send out to all participating nodes and a *response* message being communicated back to the coordinator. In the case of the geographically distributed interlocking system, the nodes are linked either virtually or physically in a sequential fashion corresponding to the given route. A request message is propagated through all participating nodes, and as the request reaches the very last node, the given node will initiate a *response* message to be propagated back through all the participating nodes.

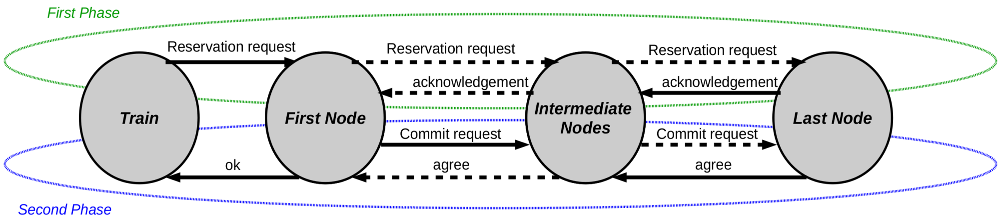

Figure 5.1: Illustrating the concrete two-phase commit protocol used for route reservations. The *First Node, Intermediate Nodes* and *Last Node* represents the track elements in the system.

The **first phase**, commonly referred to as the "voting phase", starts with the coordinator sending out a *query* message which then is propagated through the network of nodes. The initial *query* message will contain details and data of what is to be committed. If all nodes *agree* to the *query*, the last node will initiate an *acknowledgment* message to be propagated back through the network. If, however, one of the nodes *disagrees* to the query, the disagreeing node will initiate a *negative acknowledgment* message to be propagated back to the coordinator.
In this phase, a node might choose to disagree if it detects the presence of a train on the track element, which is not the train that initiated the reservation request, or it might disagree simply because the given node already is involved in a reservation request initiated by another train.

The **second phase**, commonly referred to as the "completion phase", is usually initiated by the coordinator after receiving the *positive acknowledgment* message from the first phase. However, in the case of the geographically distributed interlocking system, the nodes are connected in a sequential fashion and the coordinating train has nothing new to add to the communication. Thus the immediate node following the train in the communication chain, will act as coordinator. Had the system been communicating in a distributed fashion without regards to geographical relationships between the nodes, then the messages would need to be communicated back to the original coordinator (the train).
When the first node in the communication chain receives a *positive acknowledgment* message from the first phase, it immediately initiates a *commit* request, which informs the nodes to commit the awaiting transaction.

As each node receives the *commit* request it will prepare to enforce the transaction. If any node fails the conditions for performing the transaction it will emit a *disagree* message which is to be propagated out to all the nodes involved in the transaction. Upon receiving a disagree message, the nodes which have already committed the transaction will attempt a *rollback* to their previous states, and the nodes which have yet to commit will abandon the pending transaction.
In the case of the geographically distributed interlocking system, the *commit* message will cause the points on the route to apply the positioning requested for the given route. A *disagree* event will simply cause the nodes to abandon the pending reservation and go back to a non reserved state that indicates they are ready to receive new requests.
When the last node in the system has received the *commit* message, it will initiate an *agree* message to be propagated back through the nodes. When the first node, on which the train is located, receives the *agree* message, it will communicate an *ok* message back to the train.

5.3 Route Reservation

The individual nodes need not know the details of the track layout, instead the knowledge of the track layout can be either centralized and/or be known by the trains. In the conceived concept, the nodes are informed of their neighbors with regards to a specific route reservation. The initial route reservation request, which the train sends out, will contain an ordered list of the nodes involved in the route. As the reservation request is propagated through the network to the involved nodes, each node will record its neighbors for the given route.
However, the knowledge about routes in the system is not entirely enough, an overview of the railway network layout must still be duly maintained, and it is assumed that all routes has been verified against the current layout of the railway network.

Alternatively could the nodes be initialized or bootstrapped with knowledge about their immediate neighbors. This configuration would serve the purpose of rejecting impossible routes. However if the routes has already been verified against the concrete track layout prior to even attempting the reservation, then this interconnection check performed in the nodes is redundant and perhaps unnecessary.

One benefit of defining the interconnections between the nodes using routes instead of defining the interconnections statically and locally on the nodes themselves, is that it is easier to change the software representation of layout of the railway network, as the physical network changes over time, as an effect of maintenance work and extensions of the physical railway network.

5.4 Releasing the Reserved Track Segments - Sequential Release

To increase the usage and utility of the given railway track system, a sequential release mechanism can be implemented. The purpose of such a mechanism is to make reserved nodes *available* for other trains to reserve and use, as they are no longer needed by the original train which reserved them. As the individual track segments reserved for a given route, detects first the presence and then the absence of a train -they will go back into a state that defines them as available for new reservations.

The alternative to a sequential release mechanism would be to hold all nodes in a reserved state until the reserving train has reached its route destination. This, however, would reduce the availability of the track elements resulting in a poor usage of the network as a whole as the nodes will be held in a reserved state for a longer time than if they were released immediately after use.

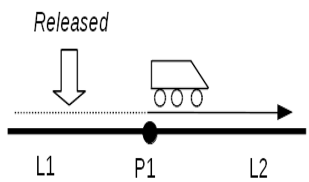

Figure 5.2: Illustration of the sequential release mechanism for a train located at *P1* and *L2* with an original route from *L1* to *L2*. Since the train is no longer occupying *L1*, the track is released.

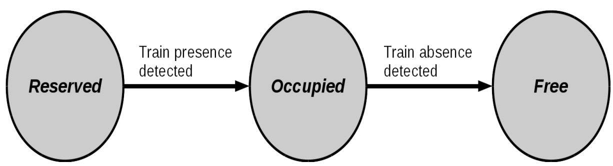

Figure 5.3: The state transitions happening as the presence of a train is detected at a track element by its track circuit sensor, followed by detection of the absence of a train.

5.5 A Practical Implementation of the System

The linear track segments and points in the system would be equipped with track circuit sensors to detect presence or absence of trains. Furthermore can each of the linear segments and points be equipped with radio communication hardware to enable wireless communication with each other, and small processing units for maintaining state wrt. reservation status in the system and handling of the reservation negotiation protocol.
The train can likewise be equipped with radio communication hardware and a processing unit for handling the reservation negotiation protocol. At last, can the train processing unit be responsible for communicating the result of a reservation to the train operator.

For an implementation of the given system to be safe, it would have to somehow rule out human errors. A human train operator could accidentally miss a signal to stop and potentially create a dangerous situation by violating an already reserved route. The obvious way to counter this problem, would be to make the trains fully automated. However, the system could also be implemented with a mechanism in the trains that automatically breaks if it detects that it is about to violate a reserved track section. The protection against human errors could furthermore be enhanced by implementing flank protection, such that points neighboring a reserved route will position themselves to divert trains away from the reserved route.

At the software level, the overall implementation architecture would be a distributed system consisting of isolated processes, each with the responsibility of handling route reservation requests. At least three types of processes would be needed, one for the train, one for the linear track nodes and one for the point nodes.

5.6 Discussion

The core idea behind the presented concept, is that a route must be fully reserved before a train can traverse it, and since it is not possible for other trains to reserve segments of an already reserved route, this should very much prevent collision between trains. This, however, only holds true if either the trains are fully automated or if they implement some sort of fail-safe brake, since human operators potentially still can violate reserved routes.

An important motivation for making the interlocking system distributed instead of centralized, is that it allows on of smaller subsets of the network to be verified independently, as also discussed in [Fan12]. In contrast, changes made to a fully centralized system where everything is highly interdependent, requires the whole network to be verified, which can be quite costly with regard to computational resources, -especially for large networks.

The concept could be extended to support reservations of partial routes or even moving block reservations. However this would require a lot more work to be done with regards to ensuring liveness in the system, in order to avoid trains ending up in deadlock situations. By fully reserving a route and only permitting trains to traverse they have reserved, it is trivial to check the system with regards to liveness.

Chapter 6

Modeling the Geographically Distributed Interlocking System in UMC

This chapter describes the modeling of the Geographic interlocking system described in the previous chapter. The model specifically models the two-phase commit protocol for reserving train routes, and the physical movement of the trains over their respective routes in the railway network layout.

6.1 Defining the Model in UMC

The geographically distributed interlocking system has been specified in the UMC modeling language with initial inspiration drawn from a model originally specified by the student Marco Paolieri from University of Firenze[Pao10].

The model consists overall of the three types of components modeled in UMC classes. The classes constitutes a Train class representing trains, a Linear class representing linear track segments and a Point class points. Each of the UMC classes contain a set of variables, a set of incoming signals, a set of class-states

and a set of state transition definitions. Essentially, it is the state transitions which reflects the behavior of a given component.
The Point class and Linear class each basically models a track element communication node and a physical track circuit sensor, while the Train class models the communication node in a train and the trains physical movement across its route in the railway network.

All the components are modeled such that it is possible to define physical lengths as abstract discretized length units. The purpose of this, is to be able to model situations where a train overlaps multiple track elements, and specifically model train movement transitions over neighboring track segments. All elements use the same abstract unit length, and the train movement is modeled such that a train moves in discretized steps of one unit in each movement step. At all times, the model keeps track of the location of each section that makes up the length of each train in order to model the movement of the trains across multiple segments in the railway network.

The actual use of the classes is described in the next chapter, which describes a tool for instantiating the objects to compose a full model based on the classes.

This section informally describes the specification details of the generic classes and their signals, variables, state transitions and behavior.

Note that the full source code for the following described classes can be found in Appendix C.

6.1.1 The Train Class

The Train class models the behavior of a train in a railway system with a geographically distributed interlocking system. The train is responsible for reserving its own route by sending out a reservation request message, and simulates moving through its route by keeping track of its own position on the route and invoking operations on the track element nodes that it passes on its route.
The Train class supports trains of varied lengths and models the movement of the train such that it is possible for a train to partially cover a track element or even cover multiple track elements.

6.1.1.1 Variables

The train contains *variables* describing its route and for keeping track of its exact position on the route.

- *requested_point_positions* is an array of Boolean values that describes the required position of each point on the route of the train.
- *train_length* is an integer value that determines the length of the train.
- *route_segments* is an array of object references containing references to the track element nodes constituted by communication nodes representing the linear track segments and point segments.
- *route_index* is an integer that indicates how far in the route the train has traveled. The value corresponds to an index in *route_segments*.
- *occupies* is an array of object references. The array has same length as the train and contains references to the nodes which the train currently covers with its length.
- *front_advancement_count* is an integer variable which describes the current location of the front of the train within the track segment that the train currently occupies.
- *track_lengths* is an integer array which describes the lengths of the track segments contained in *route_segments*. The two arrays *track_lengths* and *route_segments* corresponds index-wise.

6.1.1.2 Incoming signals and outgoing signals

The Train class exposes two signals which can be send to the instantiated train objects.

- a *no* signal indicating rejected reservation of its route.
- an *ok* signal indicating successful reservation of its route.

Only one signal is ever emitted from the objects of the Train class, and that is the initial reservation request signal denoted *req*.
The *req* signal has the signature

$$req(sender, route\_index, route\_segments, req\_point\_configurations)$$

where the variables are described as

- *sender* which is a reference to the train itself.
- *route_index* which is used as an index for the *route_segments* array, and is set to a value of zero with the initial request from the train to refer to the first element of the route.
- *route_segments* which is an array containing references to the track elements and points on the route of the train, which constitutes of linear track segments and point track segments.
- *req_point_configurations* which is an array of boolean values indicating the required position of each of the points along the route.

Besides sending and receiving signals, the train also invokes *sensorOff* and *sensorOff* operations on objects of the *Linear class* or *Point class*.
The received and emitted signals and operations of the objects of the *Train class*, are illustrated below.

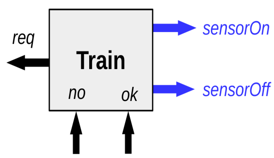

Figure 6.1: The incoming and outgoing signals and operations of objects of the *Train class*.

6.1.1.3 States

The train class has following *states*

- *READY* which denotes that the train is ready to perform a reservation.
- *WAIT_OK* which denotes that the train has send out reservation request for a route and is awaiting response.
- *MOVEMENT* which denotes that the train is currently moving over its route in the network.
- *ARRIVED* which denotes that the train has arrived at its destination.

6.1.1.4 State transitions

Below is a state diagram illustrating the *states* and *transitions* between the *states*. In the diagram, the *transitions* has been simplified as much as possible to improve readability, notably a set of aliases has been declared at the bottom. After the diagram follows a set of high level descriptions of the *state transitions*.

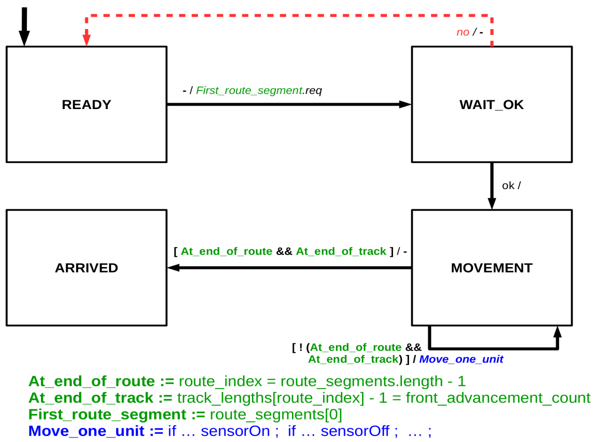

Figure 6.2: State diagram for the *Train class*. The diagram follows the UML convention for state diagrams, where each transition is labeled with "*eventName [guardExpression] / action*", where *eventName* in this case is an incoming signal, *guardExpression*, is the requirement to be able to take the transition and *action* is the action performed when the transition is taken. The symbol - signifies absence or empty statement.

READY -> WAIT_OK
Requires: -
Effect: Sends the initial request to the first node in the route of the train.

WAIT_OK -> READY
Requires: The train has received the signal *no*.
Effect:
The train has received a rejection signal for its reservation request and goes back to its *READY* state.

WAIT_OK -> MOVEMENT
Requires: The train has received the signal *ok*.

Effect:
The train has received the final acknowledgment which indicates that the full reservation of its route has been completed *successfully*, and therefore transitions into the *MOVEMENT* state.

MOVEMENT -> MOVEMENT
Requires: The train has not reached the end of the final node on its route.
Effect:
The train is traversing the track elements on its route and is triggering track circuit sensors as it traverses over the track elements on the route. At each transition the train determines if it has reached the end of a track, by evaluating whether or not the front of the train has reached the end of the length of the current track it occupies. When the front of the train moves into a new track segment, it triggers the the track circuit by invoking the *sensorOn* operation on the given track element. When the rear of the train leaves a track segment, it triggers the track circuit sensor off, by invoking the *sensorOff* operation on the given track element.
At each transition the *occupies* array is updated to reflect which track segments the train covers. The head of the *occupies* array represents the rear of the train and the last element of the array represents the front of the train.

The algorithm used for train movement is presented below in the UMC language with comments in *italics*. Furthermore, to understand the flow of the movement better, it might be beneficial to refer to the Scenario section later in this chapter, where a concrete example is explained in details.

Listing 6.1: Train Movement Transition

```
MOVEMENT -> MOVEMENT {
  -
  [not (route_index = route_segments.length - 1 and
   track_lengths[route_index] - 1 = front_advancement_count)] /
      -- determine if we have reached the end of the current track
      at_end_of_track: bool :=
         track_lengths[route_index] - 1 = front_advancement_count;
      if at_end_of_track = true then {
         front_advancement_count := 0;
         -- if the route index is not the last of the route segments array
         if route_index < route_segments.length - 1 then {
            -- the train enters the next track on the route
            route_index := route_index + 1;
            -- modeling that the track circuit sensor on the next track detects the train
            route_segments[route_index].sensorOn(self);
         };
      } else {
         front_advancement_count := front_advancement_count + 1;
      };
      -- update the occupies array
      rear: obj := occupies.head;
```

```
        next_rear: obj := occupies.tail.head;
        occupies := occupies.tail + [route_segments[route_index]];
        -- if the rear of the train has left a track segment
        if rear != next_rear then {
           – modeling that the past track circuit sensor detects absence of the train
           rear.sensorOff(self);
        };
}
```

MOVEMENT -> ARRIVED
Requires: The train has reached the end of the length of the final track element on its route.
Effect: The train has successfully traversed its route and has reached its destination.

6.1.2 The Linear Class

The *Linear class* models the behavior of a *node* representing a linear track segment equipped with a track circuit sensor. The class models the receival and forwarding of of route reservation negotion messages, and models track circuit sensor responses to a train that's moving across its length.

6.1.2.1 Variables

The *Linear class* contains *variables* describing its immediate neighbors relative to a reserved route, a reference to the train currently occupying the linear track segment, and the length and amount of free capacity of the linear track segment.

- *next* is an object reference to the next neighboring track segment or point on a reserved route. The variable is *null* when the node is in the non reserved state.
- *prev* is an object reference to the previous neighboring track segment or point on a reserved route. The variable is *null* when the node is in the non reserved state.
- *train* is an object reference to the train which is currently occupying the node. This variable is *null* if no train is occupying the node.

6.1.2.2 Incoming and outgoing signals and operations

Incoming signals
The *Linear class* describes a set of *signals* involved in negotiating a full route reservation between the nodes representing the track segments in the network. These signals are:

- *req* which is a reservation request, with the same parameter signature as the *req* signal described under the *Train class.*
- *ack* which is an acknowledgment signal send in response to a request.
- *nack* which is a negative-acknowledgment signal send in response to a failed request.
- *commit* which signifies a request for the current reservation to be enforced.
- *agree* which is an acknowledgment signal send in response to a commit signal.
- *disagree* which is a negative-acknowledgment signal send in response to a failed commit.

Operations
The class specifies two *operations* which are invoked by other objects:

- *sensorOn* which is an *operation* invoked by a *Train class* object. This *operation* models that a sensor has been triggered on by a train when it moves onto the track.
- *sensorOff* which is an *operation* invoked by a *Train class* object. This *operation* models that a sensor has been triggered off by a train when it leaves the track.

The *sensorOn* and *sensorOff* operations, essentially models the triggering of a track circuit sensor.

Outgoing signals
The outgoing *signals* are the same as incoming *signals*, but also includes the train *signals ok* and *no*.
The incoming and outgoing signals and operations are illustrated below.

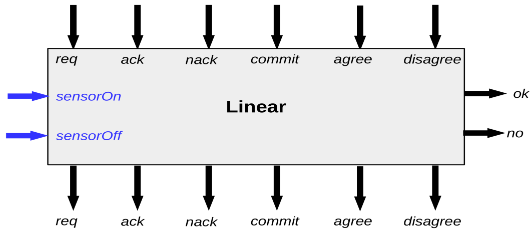

Figure 6.3: The incoming and outgoing *signals* and *operations* of objects of the *Linear class*, with *signals* colored black and *operations* colored blue.

6.1.2.3 States

The *Linear class* contain following states:

- *NON_RESERVED* which denotes that the given node is currently free.
- *WAIT_ACK* which denotes that that the node has received a reservation request and now is awaiting an *ack* response message from the first acknowledgment phase of the two-phase commit protocol.
- *WAIT_COMMIT* which denotes that the node is awaiting a *commit* request message from the second phase of the two-phase commit protocol. In this state, the node is ready to prepare the enforcement of the reservation.
- *WAIT_AGREE* which denotes that the node is awaiting an *agree* response message from the second acknowledgment phase of the two-phase commit protocol. In this state the node is ready to enforce the reservation in the case of consensus among all the nodes.
- *RESERVED* which denotes that a reservation has been enforced, and therefore the node is currently reserved. The node is thus ready for the reserving train to traverse over the track segment associated with the node.
- *TRAIN_IN_TRANSITION* which denotes that a train is currently moving on the track segment associated with the node.

6.1.2.4 State transitions

The set of state transitions involving the states *NON_RESERVED*, *WAIT_ACK*, *WAIT_COMMIT*, *WAIT_AGREE* and *RESERVED*, triggered by the signals *req*, *ack*, *commit* and *agree*, basically describes the behavior of the two-phase commit protocol. In these transitions, an incoming signal is either passed along to a neighboring node, or a new phase of the two-phase commit protocol is started when the node is either first or last on the route.

The transitions involving the states *NON_RESERVED*, *WAIT_ACK*, *WAIT_COMMIT*, *WAIT_AGREE* and *RESERVED*, triggered by the signals *nack* and *disagree*, describes the behavior of the two phase protocol when a route reservation *fails*. A route reservation is *successful* if no other trains are occupying any of the track segments along the route, and when all the track elements on the route are involved in the same two-phase commit protocol. On the other hand, a route reservation *fails* if another train is occupying one of the track elements on the route, or when another train has already engaged one of the track element nodes in a reservation request.

The transitions involving the states *TRAIN_IN_TRANSITION*, *RESERVED*, and *NON_RESERVED* triggered by the operations *sensorOn* and *sensorOff*, describes the movement of a train over the track segment. The transitions also models the *sequential release behavior* of a track segment, where a train first enters a reserved track segment triggering the track circuit sensor on, and then leaves the track segment triggering the track circuit sensor off causing the given track element to be free and non-reserved again.

To give an overview of the state transitions, a state diagram of the *Linear class* is presented below.
Following the diagram, are the individual state transitions described informally.

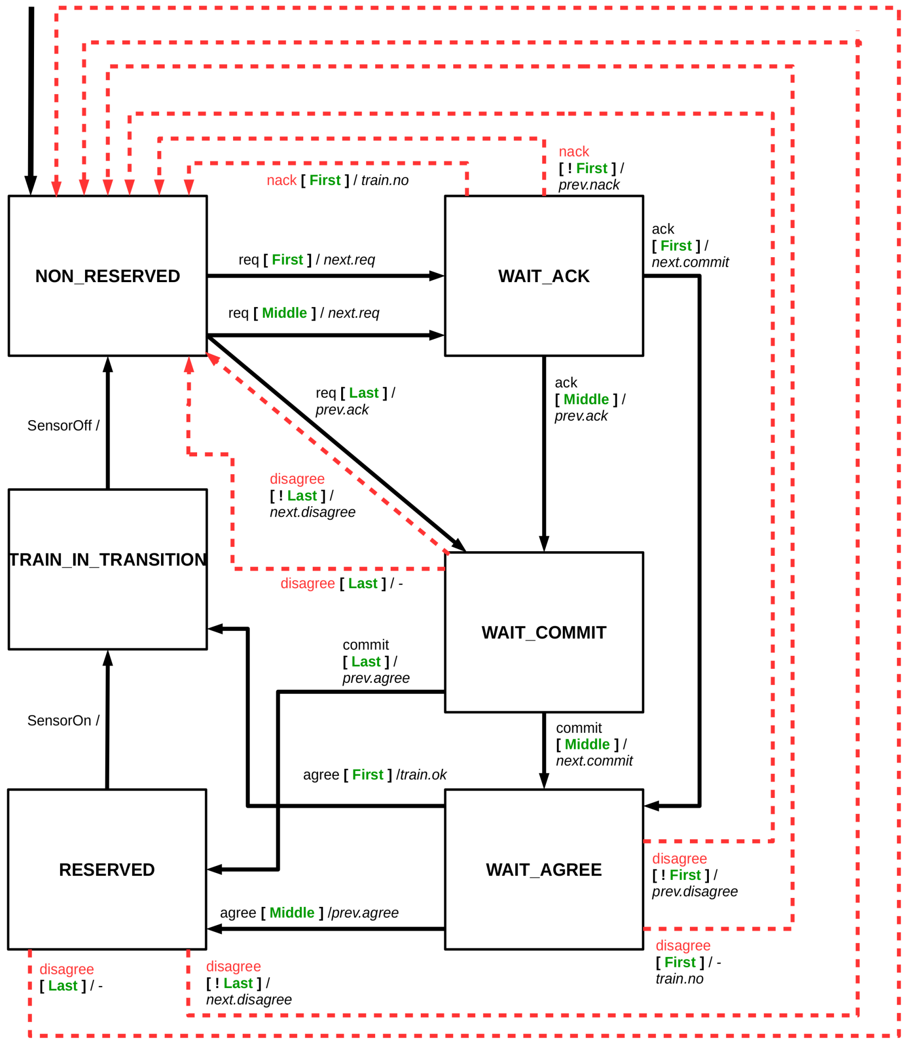

Figure 6.4: State diagram for the *Linear* class. The diagram follows the UML convention for state diagrams, as also described in the figure for the Train state diagram. Furthermore is it important to note that the diagram omits descriptions of the actual performed actions for brevity.

NON_RESERVED -> WAIT_ACK
Requires: a received request where the sender must be the train currently occupying the track segment.

Effect:
The current track element node has received an initial route reservation request. The *next* variable is updated based on the route information and the request is forwarded to the track element node on the route.

NON_RESERVED -> WAIT_ACK
Requires: a received request with a route where the current track element is not the first or last.
Effect:
The *next* and *prev* fields are updated in the current node based on the route information provided with the reservation request. The request is forwarded to the next track element node on the route.

NON_RESERVED -> WAIT_COMMIT
Requires: a received reservation request with a route in which the current track element is the last.
Effect:
The *prev* variable is updated with the route information, and an *ack* signal is send to the previous track element node on the route.

WAIT_ACK -> WAIT_COMMIT
Requires: an *ack* signal has been received, and the current node is not the first track element node on the route, which is verified by checking that the *prev* variable is *not null*.
Effect:
An *ack* signal is passed on to the previous track element node on the route.

WAIT_ACK -> WAIT_AGREE
Requires: an *ack* signal has been received and the track element node is the first on the route, which is verified by checking that the *prev* variable is *null*.
Effect:
A *commit* signal is passed on to the next track element node on the route.

WAIT_COMMIT -> WAIT_AGREE
Requires: a *commit* signal has been received and the track element node is not the last on the route, which is verified by checking that the *next* variable is *not null*.
Effect:
A *commit* signal is passed on to the next track element node on the route.

WAIT_COMMIT -> RESERVED
Requires: a *commit* signal has been received by the current node, and the track segment it represents is the last element on the route, which is verified by checking that the *next* variable is *null*.

Effect:
An *agree* signal is send to the previous track element node on the route.

WAIT_AGREE -> RESERVED
Requires: an *agree* signal has been received by the current node, and the track segment it represents is not the first on the route, which is verified by checking that the *prev* variable is *not null*.
Effect:
An *agree* signal is send to the previous track element node on the route.

WAIT_AGREE -> TRAIN_IN_TRANSITION
Requires: an *agree* signal has been received by the current node, and the track segment it represents is the first on the route, which is verified by checking that the *prev* variable is *null*.
Effect:
An *ok* signal is send to the train, indicating that the route has now been *successfully* reserved.

RESERVED -> TRAIN_IN_TRANSITION
Requires: a sensor on registration which is triggered by a train object invoking the *sensorOn* operation on the current track element, modeling the triggering of a track circuit.
Effect:
The *train* variable is set to the value of the *sender* of the operation.

TRAIN_IN_TRANSITION -> NON_RESERVED
Requires: a sensor off registration which is triggered by a train object invoking the *sensorOff* operation, modeling that the track circuit sensor goes from registering presence of a train to registering absence.
Effect:
This transition models the sequential release functionality, releasing the node as the train leaves it by setting its *train* variable to *null*.

WAIT_ACK -> NON_RESERVED
Requires: a *nack* signal has been received.
Effect:
If the represented track segment is the first on the route, then a *no* signal is send to the train which is occupying the track segment, else a *nack* signal is send to the previous track element node on the route.
Since the node was awaiting an *ack* signal, and since *ack* signals are send in the direction of the *prev* node, the current node only need to forward the *nack* signal in that direction.

WAIT_COMMIT -> NON_RESERVED

Requires: a *disagree* signal has been received.
Effect:
If the represented track segment is not the last on the route, then a *disagree* signal is send to the next node on the route.
Since the node was awaiting a *commit* signal which are send in the direction of the *next* node, it must forward the *disagree* signal in that direction. It is furthermore, only possible for *disagree* signals to occur in the second phase of the two-phase commit protocol, and therefore the *disagree* signals only need handling for the transitions relevant to this part of the phase.

WAIT_AGREE -> NON_RESERVED
Requires: a *disagree* signal has been received.
Effect:
If the current node represents the first track segment on the route, then a *no* signal is send to the train which is occupying the track segment represented by the node, else the *disagree* signal is forwarded to the previous track element node on the route.

RESERVED -> NON_RESERVED
Requires: a *disagree* signal has been received.
Effect:
If the node represents a track segment which is not the last, then a *disagree* signal is send to the next node on the route.

NON_RESERVED -> NON_RESERVED
Requires: a *req* signal has been received, but a train is already occupying the track segment, and the sender of the request is different from the train which is occupying the track segment.
Effect:
A *nack* signal is send to the sender of the request signal.

WAIT_ACK -> WAIT_ACK,
WAIT_COMMIT -> WAIT_COMMIT,
WAIT_AGREE -> WAIT_AGREE,
RESERVED -> RESERVED,
TRAIN_IN_TRANSITION -> TRAIN_IN_TRANSITION
Requires: a request signal has been received.
Effect:
The node is already in the process of being reserved, so it sends a *nack* signal to the sender of the request.

6.1.3 The Point Class

The *Point class* is very similar to the *Linear class*. However, the *Point class* deviates from the *Linear class* because points only can be intermediate nodes on a route, and also because they model the track switching behavior of a point in a railway system. Although the nodes representing points are very similar to the nodes representing linear track segments, the points are required to be in correct position before they can agree to a given reservation. If a positioning of a point *fails* it will emit *disagree* signals to both its neighbors to communicate that the reservation has *failed* and that the current reservation should be canceled.

6.1.3.1 Variables

The *Point class* defines the same variables as the *Linear class*, but with two additional variables which are

- *requested_position* which is a Boolean variable that indicates a requested position for the point for a specific train route.
- *current_position* which is a Boolean variable that indicates the current positioning of the point. A value of True is to be interpreted as a *plus* positioning, and a value of False as a *minus* positioning.

6.1.3.2 Incoming signals, Operations, Outgoing signals

The incoming signals, operations and outgoing signals are the same as for the *Linear class*, with the only exception being that no point will ever communicate directly with a train in the reservation process, so no *ok* or *no* signals are ever sent from points.

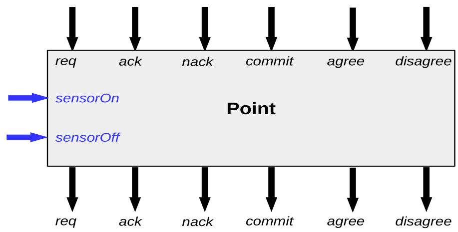

Figure 6.5: The incoming and outgoing signals and operations of the objects of the Point class. With signals colored black and operations colored blue.

6.1.3.3 States

The states are the same as for the *Linear class* including an additional state *POSITIONING*, which indicates that a point is currently performing a positioning switch between tracks, and a state *MALFUNCTION* which indicates that the current point has malfunctioned.

6.1.3.4 State transitions

The set of state transitions are almost identical to the state transitions in the *Linear class*, besides the fact that only transitions relevant for intermediate nodes on a route are implemented, and additional state transitions for the *POSITIONING* state are defined, and a transition for the *MALFUNCTION* state has been added.

To give an overview of the state transitions for the *Point class*, a UML state diagram is presented below. The additional transitions for *Point class* are informally described after the diagram.

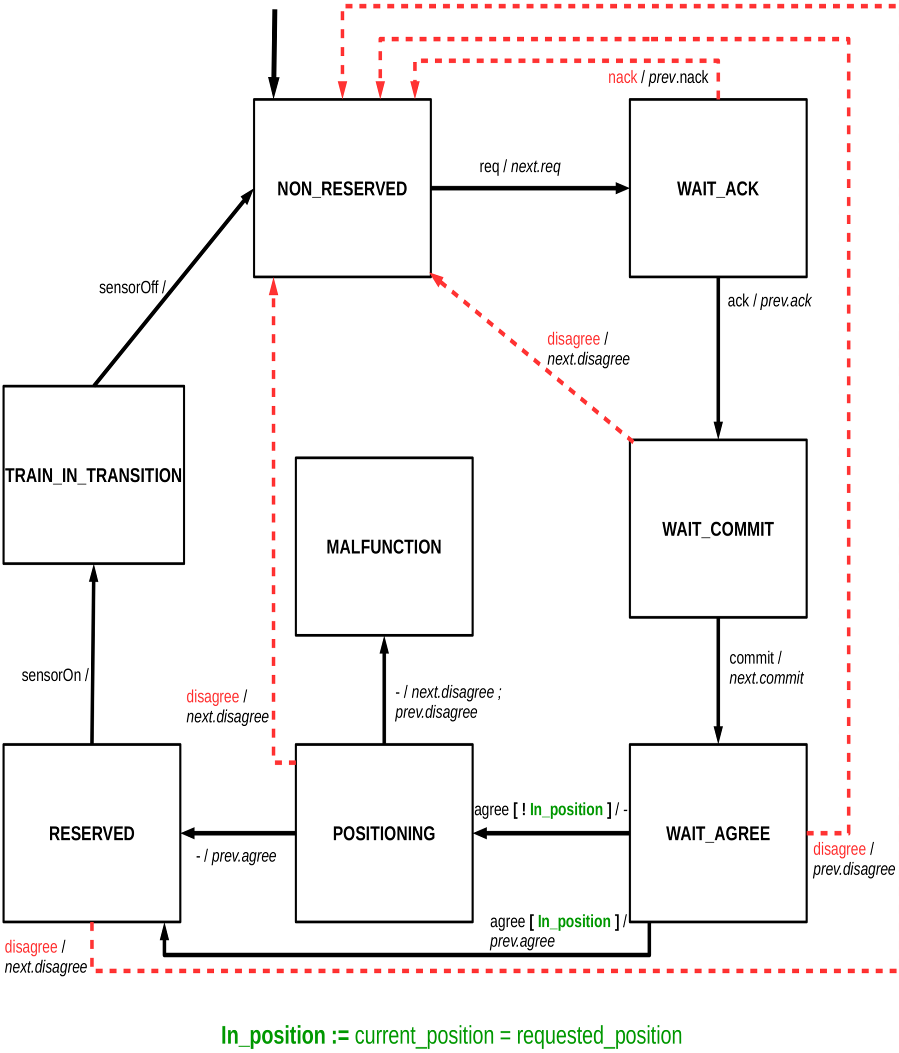

Figure 6.6: State diagram for the *Point* class. The diagram follows the UML convention for state diagrams, as described under the Train class state diagram. Furthermore does the diagram abstract away the actual actions taken in the transitions, and defines an alias In_position of a proposition that tests the current positioning of the point against the requested positioning.

WAIT_AGREE -> POSITIONING
Requires: the node representing the point has received an *agree* signal, but

the requested positioning is different from the current positioning of the point.
Effect: The point goes into the *POSITIONING* state.

POSITIONING -> RESERVED
Requires: -
Effect: The positioning is successfully performed. The point sends an *agree* signal to its previous neighbor.

POSITIONING -> MALFUNCTION
Requires: -
Effect: The point malfunctions while performing the positioning. It emits *disagree* signals to both its neighbors indicating that the current reservation should be canceled.

POSITIONING -> POSITIONING
Requires: the node representing the point has received a *req* signal.
Effect:
The point is currently busy performing its positioning, and therefore the reservation request must be rejected. The node representing the point, sends a *nack* signal to the sender of the request.

6.2 Model Properties

The whole purpose of formally specifying the system, is such that it can be model checked for certain relevant properties. In the case of an interlocking system, the properties of main interest are safety properties. These properties specifies the most critical situations that must be be avoided in the system, no matter what scenarios it is exposed to over its operational time. The two most important safety properties to verify for an interlocking system, is the *no derailment property* and the *no collision property*.

It can also be relevant to check the system for *liveness* or *progress* to verify the absence of deadlocks between trains or messages. However, the modeled system in this project also models possible malfunctions of points, and thus a simple global check for progress in the model would fail, since a malfunction would prevent the trains from passing the malfunctioning point. However, it is still relevant to verify that the system allows the trains to reach their destinations when no malfunctions occur, and thus a *progress* property is defined which verifies the arrival of trains for the state path branches where no malfunction has occurred.

Properties can also be used as a debugging tool and for verifying the consistency of a model. A property which checks for consistent handling of all signals in the model, is an example of such a property. This type of property is mainly of interest as a tool to spot modeling mistakes while deriving the generic model, and to verify modeling assumptions.

Following sub sections describes the individual properties which are verified for a concrete model. Specifically when using UMC as modeling tool, it is possible to define *abstractions* which is a language construct for captures a given situation in the system. These abstractions are core components when specifying model properties in UMC. In this section, each property is formally described together with its associated abstractions.

6.2.1 No Collision

The *no collision property*, is a safety property specifying that no two trains must ever be able to occupy the same space on the modeled railway network. For the modeled trains with discretized lengths, this means that no parts of any two trains must ever be present at the same track segment at the same time.

The *Train class* specifies an array *occupies* which has the same length as the discretized length of the train, and is maintained with references to the track elements which the train occupies when traversing its route. The *occupies* array thus keeps track of the full location at all times for the given train. To verify that no parts of any two trains are ever at the same track segment, it is necessary to check that there is no intersection between the track segments which the two trains occupies.

The *no collision* property uses an abstraction *trains_ at_ diff_positions*, which captures the situation in the model where there is no intersection between any two trains.

The *trains_ at_ diff_positions* abstraction can be formally described with the following predicate, which is a conjunction over all pairs of trains *(t1,t2)* in the system.

$$\bigwedge_{(t1,t2)\in TrainPairs} \left(\bigwedge_{i\in I, j\in J} \mathit{t1.occupies}[i] \neq \mathit{t2.occupies}[j] \right)$$

where $I = \{i | 0 \leq i < \mathit{t1.length}\}$ and $J = \{j | 0 \leq j < \mathit{t2.length}\}$, and *TrainPairs* is the set $\{(t1, t2) | t1 \in trains \wedge t2 \in train \wedge t1 \neq t2\}$ where *trains* is the set of all trains in the system.

The *no collision* property simply verifies that the above abstraction is valid globally over all paths.

no collision

AG (*trains_at_diff_positions*)

6.2.2 No Derailments

The *no derailment property*, is a safety property which specifies that no train must occupy a point which is in progress of performing a mechanical positioning between track segments. In a real life situation if a train were to traverse over a point while the point were performing the mechanical switch, a derailment of the train might occur.

This property requires two types of abstractions. It uses a set of abstractions that specify that a given point is in its positioning state, and a set of abstractions that specify detection of a train on the individual points.

A set of abstractions specifying the absence of trains at points is defined as follows:

For all points ***p***, an abstraction *no_train_on_**p*** is defined as follows:

State ***p***.train = null

The set of abstractions capturing the situations where a point is positioning is formally expressed as follows:

For all points ***p***, an abstraction *positioning_**p*** is defined as follows:

inState(***p***.POSITIONING)

The property can now be defined using the *no_train_on_**p*** and *positioning_**p*** abstractions:

no derailment

$$\text{AG}(\bigwedge_{p \in points} \textit{positioning\_}\boldsymbol{p} \implies \textit{no\_train\_on\_}\boldsymbol{p} \;)$$

where *points* is the set of all points in the system.

When checking this property, a claim could be made that the given property only checks whether or not a train is *detected* on a point while its positioning. To strengthen the claim of the property, another property can be defined to validate that at all times when a train is present on a point it is correctly detected by the point.

To create this property, an abstraction capturing the state where a given train is occupying a specific point, must be specified. The abstraction can be defined as a disjunction over the train parts, where each predicate component checks if the given train part is equal to the given point.

For each pair $(\boldsymbol{t}, \boldsymbol{p})$ in the set $\{(t, p) | t \in trains \wedge p \in points\}$ where *trains* is the set of all trains and *points* is the set of all points in the system, the abstraction $\boldsymbol{t}$*_on_*$\boldsymbol{p}$ is formally defined as follows:

$$\bigvee_{i \in \{0..\boldsymbol{t}.\text{length}-1\}} \boldsymbol{t}.\text{occupies}[i] = \boldsymbol{p}$$

Using the abstractions $\textit{no\_train\_on\_}\boldsymbol{p}$ and $\boldsymbol{t}\textit{\_on\_}\boldsymbol{p}$, the property for verifying the triggering of sensors on points, can now be formally defined as:

Trains correctly detected at points

$$\text{AG}\left(\bigwedge_{(\boldsymbol{t1},\boldsymbol{t2},\boldsymbol{p})} (\boldsymbol{t1}\textit{\_on\_}\boldsymbol{p} \vee \boldsymbol{t2}\textit{\_on\_}\boldsymbol{p}) \implies \neg \textit{no\_train\_on\_}\boldsymbol{p} \right)$$

where $(\boldsymbol{t1},\boldsymbol{t2},\boldsymbol{p})$ is a triple from the set $\{(t1, t2, p) | t1 \in trains \wedge t2 \in trains \wedge t1 \neq t2 \wedge p \in points\}$ where *trains* is the set of all trains and *points* is the set of all points in the system.

This property essentially validates the claim that the points always detects the presence of trains.

In the current model, the trains are always detected, but if the model were to be extended to also model sensor failures, then the *no derailment* property would have to use the $\boldsymbol{t}\textit{\_on\_}\boldsymbol{p}$ abstraction.

6.2.3 Progress property - arrival of all trains at their destinations

It is not enough to verify that a model is safe. For the model to be of any use, it should also be verified that the trains will progress to reach their destinations in the modeled network. This is both to verify that the generic model is specified correctly, and to verify that the trains in a concrete model doesn't just end up being stuck in a deadlock preventing them from reaching their destinations.
Since the generic model models the event that a point may malfunction during the run, the property for verifying arrival of trains must disregard these states.

A simple property can be defined for checking if there is any possibility that the trains will arrive at their destinations. This property simply uses a set of abstractions that captures the situation where each of the trains has changed state to *ARRIVED*.

A set of abstractions $\boldsymbol{t}\_\,arrived$, specifying the arrival of a train $\boldsymbol{t}$ at its destination, can be formally defined as follows:

For each train $\boldsymbol{t}$ in *trains* an abstraction $\boldsymbol{t}\_\,arrived$ is defined as

$$\text{State: inState}(\boldsymbol{t}.\text{ARRIVED})$$

where *trains* represents the set of all trains in the system.

The property can now be defined formally as

Will arrive

$$EF\ AG\left(\bigwedge_{t \in trains} \boldsymbol{t}\_\,arrived\right)$$

With this property it is possible to show that there eventually exist a state in the system in which all the trains have arrived.

However of interest to know in which cases the trains doesn't arrive. The malfunctioning of points is supposedly the only thing that should be stopping the trains from arriving, -disregarding the specification of models with routes that creates deadlocks.

To prove that the malfunction is the only thing stopping the trains from arriving, abstractions for capturing the malfunction states are needed.

For each point $\boldsymbol{p}$, an abstraction $\boldsymbol{p}$_*malfunction* is defined

State: inState($\boldsymbol{p}$.MALFUNCTION)

For the property, what informally needs to be verified, is that there do not exist a state-path to a *final* state where it is *not* the case that both of the trains have *not* arrived and *until* (up to) this final state there has been *no* malfunctions in the points.
Defining the property for verifying the claim, can be done using the CTL construct "until" (U) and the special UMC language construct "final" which denotes a state from which there are no more possible transitions to take.

no malfunctions when trains hasn't arrived

$$\neg \mathrm{E}\left[\neg\left(\bigvee_{\boldsymbol{p}\in points} \boldsymbol{p}\_\mathit{malfunction}\right) \ \mathrm{U} \ \left(\mathbf{final} \wedge \neg\left(\bigwedge_{\boldsymbol{t}\in trains} \boldsymbol{t}\_\mathit{arrived}\right)\right)\right]$$

6.2.4 No message loss

Messages are an important part of the modeled distributed system, and it is therefore of interest to know whether or not all messages are handled for all possible cases. Being able to verify this, has been a great help when defining the generic model, and has helped in tracking down unhandled signals.

The UMC model checker triggers a special *lostevent* action on a special *ERR* object, when an object is incapable of handling an incoming signal in its current state. This action can be captured with an *Action* abstraction *discarded_message*, and can be defined directly in the UMC language as follows

Action: lostevent -> discarded_message

The property for checking for lost messages, can be defined using the UCTL "next" construct. With this construct, a property can be defined which verifies that for all state-paths it will never hold that a message is discarded during a transition from a state to a next state.

The property can be directly specified in UMC as follows

no message loss

AG not (EX discarded_message true)

6.3 Scenarios

To give an idea of how the system functions with regards to movement of the trains and the reservation through the two phase commit protocol, a set of scenarios are described and illustrated in this section.

Each scenario is illustrated with a drawing indicating the intended routes of one or more trains, next the events playing out during the scenario are illustrated through a sequence diagram with state transitions indicated at the relative time they occur.

6.3.1 A Successful Route Reservation

A successful reservation involves a train which sends out a request to reserve a specific route in the railway network. In the following scenario, the train *train1* sends out a request to reserve the route composed of the track segments *L1*, *L2* and the point *P1*. In this scenario the point *P1* is already positioned correctly in relation to the route reservation. A simple drawing of the scenario setup is presented below.

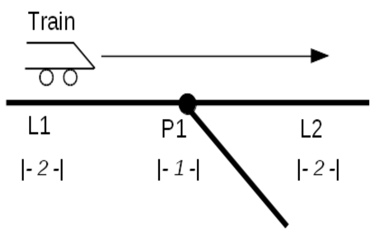

Figure 6.7: A scenario of a successful reservation of a route composed of the linear track segments *L1*, *L2* and the point *P1*. Both the linear elements have a length of two, the point have a length of one and the train a length of two.

The full two phase commit reservation is illustrated in the sequence diagram below.

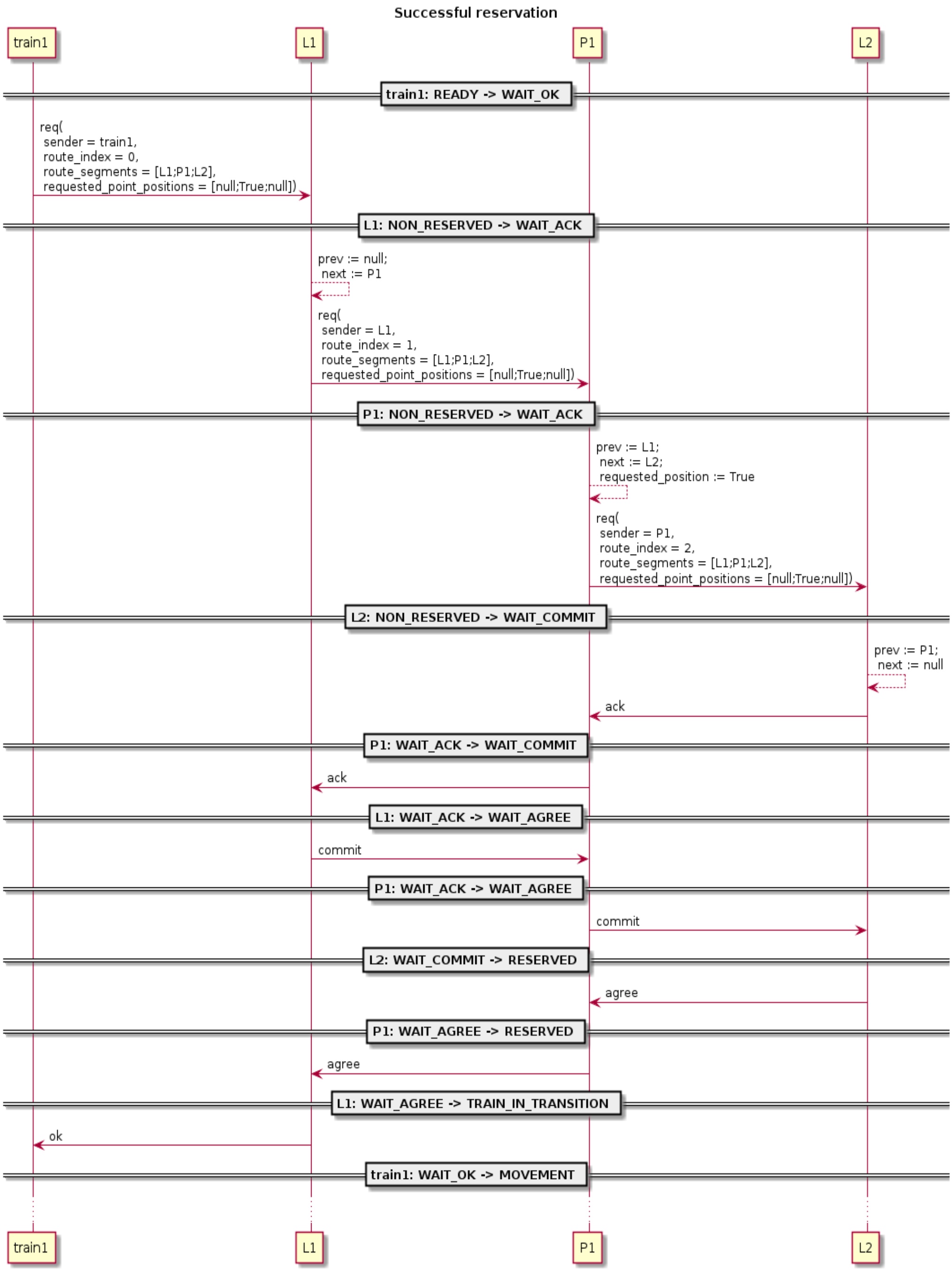

Figure 6.8: Sequence diagram of a successful reservation of a route composed of the linear track segments *L1*, *L2* and the point *P1*.

6.3.2 A train traversing its successfully reserved route

In the current scenario the traversal of the route reserved in the previous scenario, is illustrated and described.

The movement of a train when it traverses its route in the model is rather involved, and a lot of things is going on, therefore the sequence diagram for the movement is followed up by a textual description commenting on the events playing out.

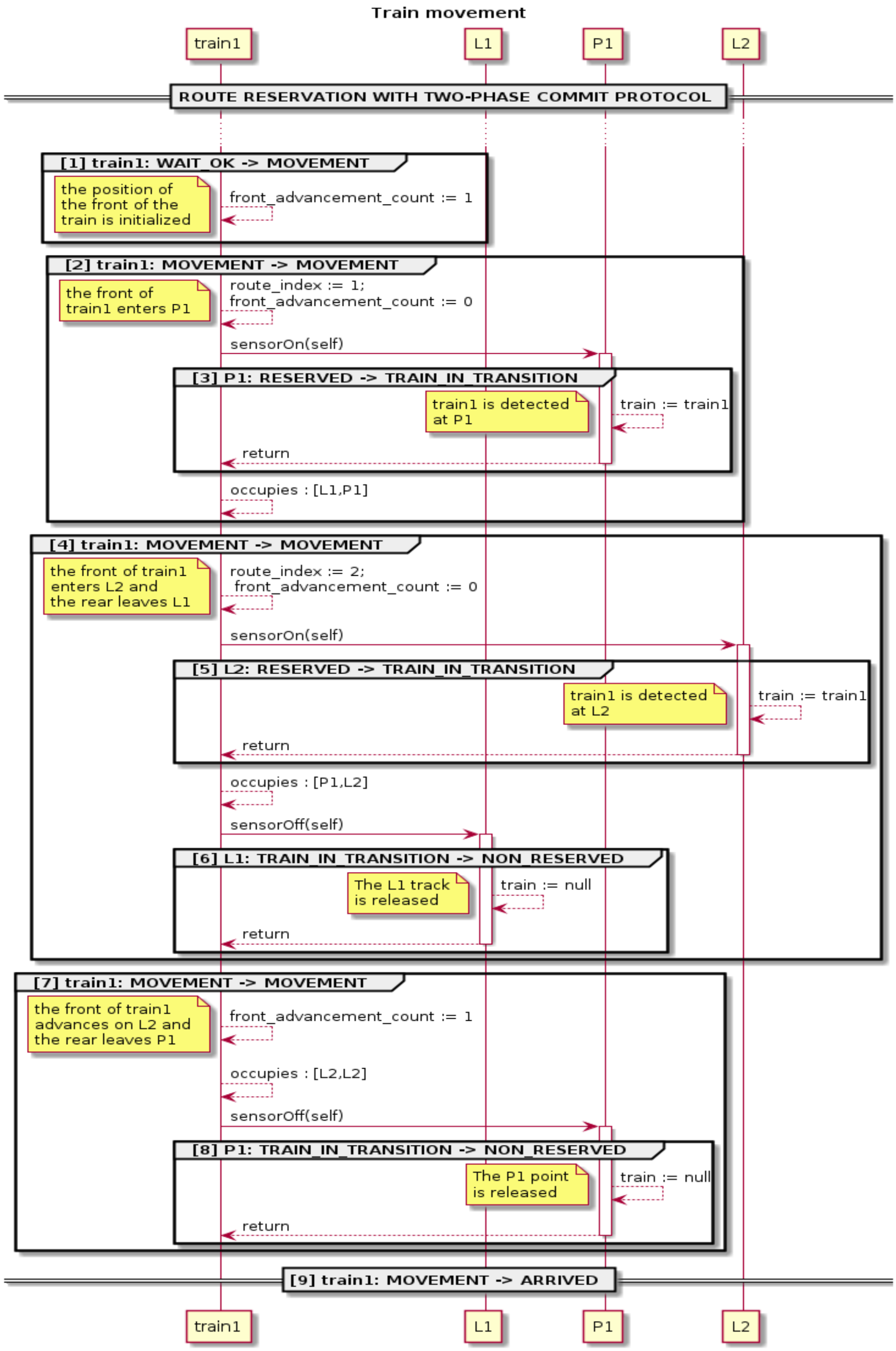

Figure 6.9: Sequence diagram of the events playing out as *train1* traverses its reserved route composed of the linear track segments *L1*, *L2* and the point *P1*.

1. *train1*: *WAIT_OK -> MOVEMENT*
   The train object initializes its *front_advancement_count* variable to reflect where the front of the train is located on the sub-segments of the current linear track (*L1*). The *front_advancement_count* counts index values from zero and the the train has a length of 2, and thus the *front_advancement_count* is initialized to a value of one.

2. *train1*: *MOVEMENT -> MOVEMENT*
   The front of the train enters the next track segment on its route, which is *P1*. During this transition, a *sensorOn* operation is invoked on *P1*.
   It is determined by the value of the *front_advancement_count* that the train has reached the end of *L1*, and therefore the variable *route_index* , which reflects the trains position in relation to its route, -is incremented to a value of one.
   Since the train has entered a new track segment and the front of the train now is located at the beginning *P1*, the *front_advancement_count* variable is updated to a value of zero to reflect this.
   The *sensorOn* operation on the point *P1* is invoked, and when the operation returns, the *occupies* array is updated to [*L1*,*P1*] such that it now reflects that *train1* is occupying *L1* and *P1*.

3. *P1*: *RESERVED -> TRAIN_IN_TRANSITION*
   train1 is detected at *P1*, which is emulated by the invoked *sensorOn* operation. *P1* updates its *train* variable to reflect that *train1* now is occupying the point.

4. *train1*: *MOVEMENT -> MOVEMENT*
   The front of *train1* enters *L2* and the rear of *train1* leaves *L1*. During this transition, operations are invoked by *train1*, on *L2* and on *L1*.
   Since the front of the train has entered a new track segment, the *front_advancement_count* variable is updated to a value of zero. Likewise, the *route_index* variable is incremented to a value of two which, through its route array, reflects that the train now is at *L2*.
   train1 invokes the *sensorOn* operation on *L2*, and when the operation returns, the *occupies* array is updated to reflect that *train1* now covers *P1* and *L2*. After that, *train1* invokes the *sensorOff* operation on *L1*.

5. *L2*: *RESERVED -> TRAIN_IN_TRANSITION*
   train1 is detected at *L2*. The *train* variable is updated to reference *train1* to reflect that a train is now occupying *L2*.

6. *L1*: *TRAIN_IN_TRANSITION -> NON_RESERVED*
   The absence of *train1* is detected, emulated by the invoked *sensorOff* operation on *L1*. Since the model models a system with *sequential release*, *L1* is released by setting the variable *train* to *null* and by entering a *NON_RESERVED* state.

7. *train1*: *MOVEMENT -> MOVEMENT*
   The front of *train1* advances over *L2* and the rear of *train1* leaves *P1*. During this transition, an operation is invoked by *train1*, on *P1*.
   The advancement of the front of the train on *L2*, is reflected in the model by incrementing the *front_advancement_count* variable to a value of one. The *occupies* array is updated to a value of [*L2*,*L2*] to reflect that *train1* now only occupies *L2*.

8. *P1*: *TRAIN_IN_TRANSITION -> NON_RESERVED*
   The absence of *train1* is detected, emulated by the invoked *sensorOff* operation on *P1*. *P1* is released, which is reflected in the model by setting the *train* variable to a value of *null* and by entering a *NON_RESERVED* state.

9. *train1*: *MOVEMENT -> ARRIVED*
   The train has arrived at its final destination.

6.3.3 Point positioning during reservation

During a reservation of a route, points must be positioned correctly in relation to the reserved route, before the route can be fully reserved. In the following scenario, the point *P1* is positioned such that it connects *L1* and *L2*, however the wished route requires the point to be positioned such that it connects *L1* to *L3*. A simple drawing of the scenario setup is presented below.

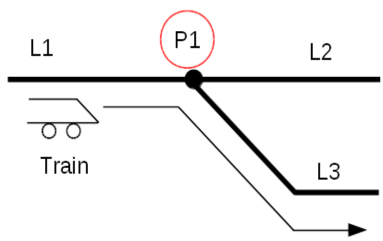

Figure 6.10: A scenario of where a positioning of the point *P1* occurs during reservation a route composed of the linear track segments *L1*, *L3* and the point *P1*.

To simplify the diagram, only the second phase of the two phase commit protocol has been drawn, since this is where the positioning occurs.

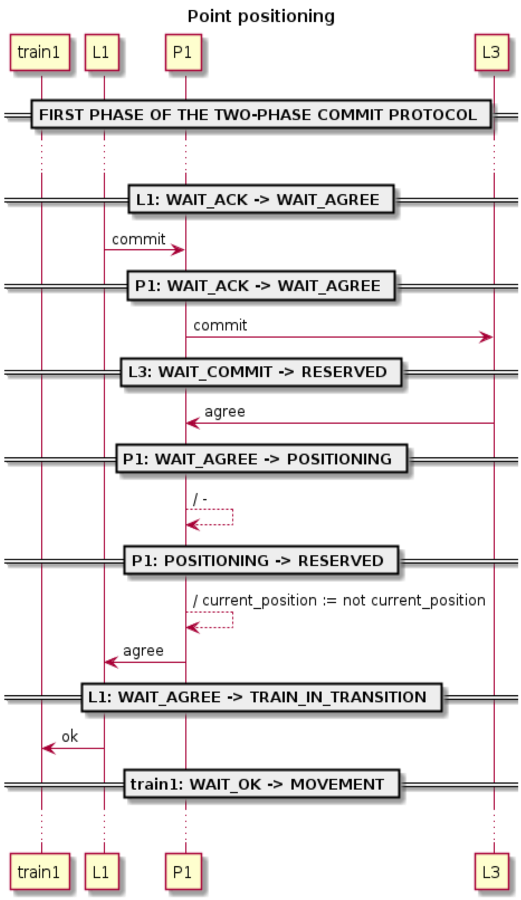

Figure 6.11: Sequence diagram of a scenario where a positioning of the point *P1* occurs during the reservation of a route composed of the linear track segments *L1*, *L3* and the point *P1*.

6.3.4 Point malfunction during reservation

During the reservation of a route, a point might malfunction which consequently causes the route reservation to *fail*. In the following illustrated scenario, the train *train1* attempts to reserve a route consisting of the track segments *[L1, L2]* and the point *P1*, however the point *P1* is positioned such that it doesn't connect to *L2* and therefore it must first be positioned, before the route can be *successfully* reserved. The point therefore enters its positioning state during the reservation process, but fails to perform the mechanical track switching and therefore emits disagree signals to its neighbors to signal that the reservation should be aborted. A simple drawing of the scenario setup is presented below.

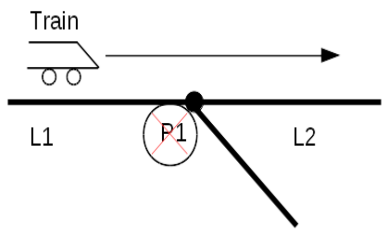

Figure 6.12: A scenario of a malfunctioning point *P1* during reservation a route composed of the linear track segments *L1*, *L2* and the point *P1*.

To simplify the diagram, only the second phase of the two phase commit protocol has been drawn, since this is where the malfunctioning can occur.

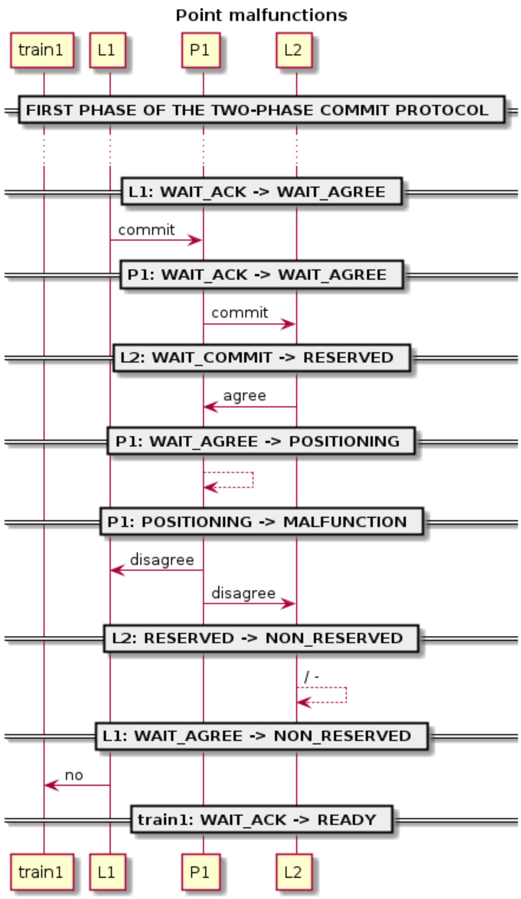

Figure 6.13: Sequence diagram of a scenario where the point *P1* malfunctions during the reservation of a route composed of the linear track segments *L1*, *L2* and the point *P1*.

6.3.5 Attempt to reserve a route intersecting with an already reserved route

If a requested route reservation intersect with an already reserved route, the requested route reservation must be aborted. In the following illustrated scenario, the train train2 has successfully reserved a route composed of the linear track segments *L1*, *L3* and the points *P1*, *P2*. The train train1 now attempts to reserve the route consisting of the linear track segments *L2*, *L4* and the points

P1, *P2*, however the route intersects with the already reserved route at *P1* and *P2*, and must therefore be aborted.

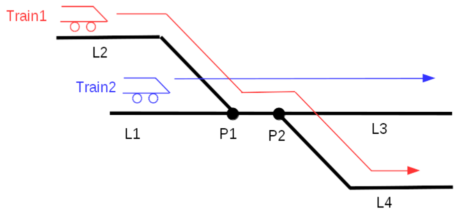

Figure 6.14: A scenario of an attempt to reserve a route that intersects with an already reserved route, where the attempted reservation is a route composed of the linear track segments *L1*,*L3* and points *P1*, *P2*, and the already reserved route is composed of the linear track segments *L2*, *L4* and the points *P1*, *P2*.

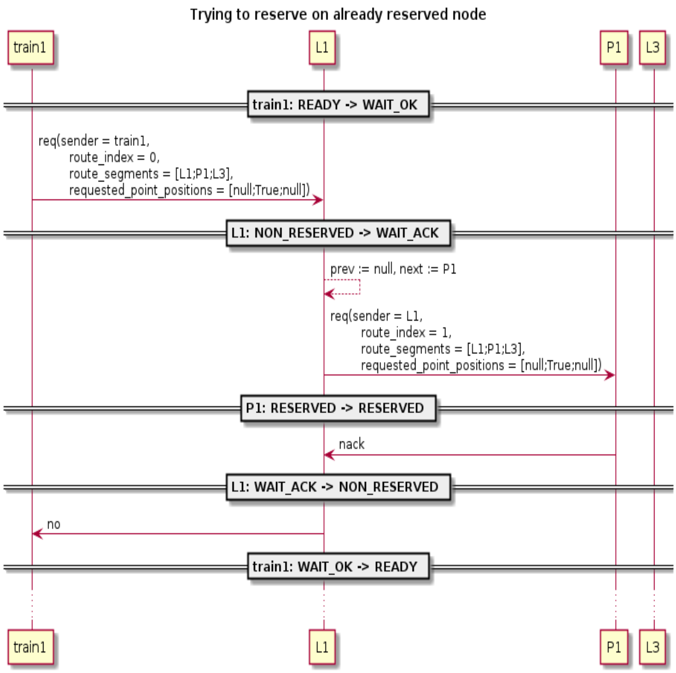

Figure 6.15: Sequence diagram of a scenario of an attempt to reserve a route that intersects with an already reserved route, where the attempted reservation is a route composed of the linear track segments *L1*, *L3* and points *P1*, *P2*, and the already reserved route is composed of the linear track segments *L2*, *L4* and the points *P1*, *P2*.

6.4 Discussion

The developed model, does a good job at modeling the conceived engineering concept presented in the previous chapter, by itself. But improvements can still be made to make the model reflect a real implementation better and to make the model checking more efficient. Especially can constraints be enforced on the model to improve model checking efficiency.

Following is a set of discussions of different aspects of the model that was presented in this chapter.

6.4.1 Train lengths and movement on track segments

In the model, it is possible to specify any length of trains, linear track segments and points. This gives a lot of freedom wrt. modeling choices, for example does it allows modeling of tracks that are physically longer than the trains. This is essentially fine if one wants to model that the trains traverse over very long tracks. However, with respect to how the state tree expands and unfolds during the model checking, the tracks which are longer than trains becomes quite meaningless and generates unnecessary state expansion since, the train has to move in discrete steps over the long track. As illustrated below, a train of length two moving over a linear track segment of length five requires four movement steps which all occur within the given track segment.

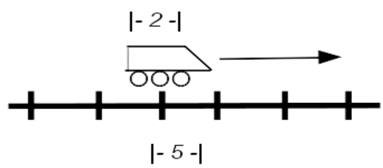

Figure 6.16: A train of length 2 moves over a linear track segment of length 5. This kind of model does not add anything meaningful since the train just will stay in the same movement to movement transition for a few more states.

Yet another complication that can arise as consequence of non-constrained lengths of especially trains and linear track segments, is deadlocks. If a train is much longer than the track segment at its destination, the rear of the train will be located at another track segment or point making it impossible for other trains to pass.

To lessen the state expansion due to trains moving over very long tracks and to avoid obvious deadlocks, a constraint could be made for the length of the track segments such that no track segment can be longer than the train that traverses it.
In the case where a track segment is exactly the same length as the train that traverses it, the model essentially models that the train is in transition over a long track segment, because the train only occupies sections of the given track segment.

The generic model already supports having different length representations of the same track segment between the individual trains, since each *Train class* contains an array *track_lengths* which defines lengths of each of the track segments on its route. However, it is not possible to define constrains directly in UMC for these lengths, so the verification that routes obey these constraints is left over to a tool described in the next chapter.

6.4.2 Point Lengths

In the current model, it is possible to create points of any lengths one desire, however, one could argue that points of lengths higher than one, would be quite meaningless since a point only really represents a track segment containing a branch to two other track segments.

In all experiments conducted in this project based on the model, points of length one has solely been used.

6.4.3 Point Machine

The point has been modeled such that the mechanical action is performed in a synchronized manner. If the model were implemented in a real life system, exactly as specified, the points would block the propagation of the agree message of the two phase commit protocol. In a real system, it would be ideal to let the point node delegate the action of performing the mechanical switching between tracks, to a point machine asynchronously. In a better designed system, the point node would send a request to perform the switching to its point machine asynchronously as soon as the point has received the commit request from the second phase of the two phase commit protocol. This would make the communication in the system faster, since the commit requests wouldn't have to wait at each point for it to perform the mechanical positioning.

6.4.4 Repairment of malfunctioning points

The system get locked down into an impossible state when a point malfunctions, because there is no way for the points to go back into a non reserved state from a malfunctioning state. This makes it less trivial to verify the progress of the system, since the malfunctioning state of a point locks down reservation of any route in the model that includes the given point. So when model checking the model for progress, the malfunction states must be ignored.To fix this problem, a simple state transition could be added to the point, simulating that it has been repaired and is now available for reservation again.

Chapter 7

Model generating tool

Model checking languages, are often made to be target languages for other languages or tools to translate into, and this is also the case for UMC. UMC is great for specifying the generic behavior of components that make up a system, but creating concrete models by instantiating objects from the generic components, can however be cumbersome and can easily result in a wrongly specified model which leads to false model checking results. Furthermore is it easy to specify a model that results in inefficient model checking.
In the case of the model presented in the previous chapter, there are multiple things the user would have to verify manually if specifying a concrete model himself / herself. For example, there is an important relationship between the objects, such as the relationship between overlapping train routes. The model is furthermore very generic, and makes it possible to specify some very inefficient models.

To deal with these issues and making the process of creating models simpler, a model generating tool has been developed in the F# programming language. This chapter describes the goals, workings and implementation of the tool.

7.1 Functionality of the Tool

When manually specifying a geographic interlocking model based on the generic components in UMC, the user must specify initial values such as the routes, track lengths of all the track components, the correct location of the train and the required positioning of the points on the routes. All these values need to be specified correctly when creating the model. If not specified correctly, the model will either not work or give false results. Furthermore is there performance to be gained in the model checking, by enforcing constraints on the models.

The primary goal of the tool is to assist the user in composing and generate a valid model. This is achieved by enabling the user to specify a model in a tiny DSL (Domain Specific Language) in a F# script, or by loading an existing railway network layout and predefined routes from an XML file. The XML files in this case are produced as part of the RobustRails project[**?**] by the **Lyngby Railway Verification Tool-set** (LRVT) [VHP16], which contain a graphical tool that was implemented in another masters project at DTU[Fol15].

The model generating tool described in this chapter validates a given input model specification, rejects it if it contains invalid routes or if it violates constraints, and finally generates a full model with properties ready to be model checked if the model is valid.

The current implementation of the model generating tool is only able to generate models in the UMC model checking language. However, the tool has been implemented with other modeling languages in mind such that it is possible to reuse the core parts of the implementation in extending and adding support for other modeling languages.

The functionalities of the tool are

- **Track Layout and Route Extraction** which is the extraction of relevant data from XML files.
- **Route Composition** which is the ability to compose multiple routes into one.
- **Route validation** which is the validation of the user defined routes against a concrete network layout, and against each other to avoid obvious deadlocks.
- **Enforcement of length constraints** which is about constraining the produced models such that they are more performant during model checking and avoids obvious deadlocks.

- **Object Creation** which is the instantiation of the concrete model objects that together compose a final model.
- **Model Composition** which is the composition of the generated objects and properties, and the merging with a generic model description to form a valid executable model.

7.1.1 Track Layout and Route Extraction from XML files

The model generating tool can use specially formatted XML files generated by the graphical tool of LRVT. These XML files contains a description of a concrete track layout and a set of possible routes for the given track layout.

A snippet of the *network* definition is presented below, and the full version can be found in Appendix G.

```
<network id="mininetwork">
    <trackSection id="b10" length="100" type="linear">
        <neighbor ref="t10" side="up"/>
    </trackSection>
    <trackSection id="t10" length="87" type="linear">
        <neighbor ref="b10" side="down"/>
        <neighbor ref="t11" side="up"/>
    </trackSection>
    <trackSection id="t11" length="26" pointMachine="spskt11" type="point">
        <neighbor ref="t10" side="stem"/>
        <neighbor ref="t12" side="plus"/>
        <neighbor ref="t20" side="minus"/>
    </trackSection>
    <trackSection id="t12" length="3783" type="linear">
        <neighbor ref="t11" side="down"/>
        <neighbor ref="t13" side="up"/>
    </trackSection>
    ...
    <markerboard distance="50" id="mb10" mounted="up" track="b10"/>
    <markerboard distance="50" id="mb11" mounted="down" track="t10"/>
    <markerboard distance="50" id="mb13" mounted="up" track="t12"/>
    ...
</network>
```

The *network* definition contains a list of *trackSection* definitions, each specifying a track section element which can be either a *linear* or a *point*. Each track section also lists its immediate neighbors. This information is essentially what defines a network layout.

The *network* definition also contains a list of *markerboards*, where each marker board contains a field *track* that refers to a *trackSection*. The model generator tool does not use the marker board definitions when extracting and generating a network layout. The marker boards are, however, referenced in the *route* definitions, which each refer to marker boards as the beginning and end of a given route. The *markerboard* definitions are therefore used by the tool to look up start and end *trackSection*s for each route.

Following is a snippet of the *route* definitions in the *routetable* list definition in the XML files.

```
<routetable id="miniroutetable" network="mininetwork">
    <route id="r_1a" source="mb10" destination="mb13" dir="up">
        <condition type='point'  val='plus' ref='t11'/>
        <condition type='point'  val='minus' ref='t13'/>
        <condition type='signal' ref='mb11'/>
        <condition type='signal' ref='mb12'/>
        <condition type='signal' ref='mb20'/>
        <condition type='trackvacancy' ref='t10'/>
        <condition type='trackvacancy' ref='t11'/>
        <condition type='trackvacancy' ref='t12'/>
        <condition type='mutualblocking' ref='r_5b'/>
        <condition type='mutualblocking' ref='r_7_'/>
         ...
    </route>
     ...
</routetable>
```

Each *route* defines a *source* and *destination* which are references to *markerboards* defined in the *network* definition. The model generating tool, uses these references to look up the *trackSection*s for the start and end of the given route. A *route* also has an *id* field, and when specifying a model to be generated, the user will need to refer to these *id* fields.
Each *route* definition furthermore specifies a set of *condition* elements, which is divided by types into a set of *trackvacancy* elements, a set of *point* elements, a set of *mutualblocking* elements and a set of *signal* elements. The *trackvacancy* elements define the route sections between the *source* and *destination* of the route. It is assumed that the set of *trackvacancy* elements always is listed in the same order as the intended route in the network. The *mutualblocking* elements specify which other routes share track sections with the current route, the *signal* elements specify markerboards that must be closed when setting the route, and the *point* elements specify the required positioning of points on the route. Of the *condition* elements, only the elements of type *trackvacancy* and *point* are extracted and used by the model generating tool.

The model generating tool extracts the layout to be used later for validation of the user defined route compositions. Only the *routes* with route *ids* specified as input by the user, will be extracted. It is possible, though, for the user to specify a composition of multiple routes, by providing a list of *route ids*.

7.1.2 Route Validation

Generally speaking, when using the predefined routes specified in the XML file, the individual routes will be well formed since they were generated and verified by the LRVT which generated them.
In the model generating tool, it is, however, still possible for the user to pick two routes that conflicts. The user could for example choose two routes that start out at the same location in the track layout. Furthermore does the model generating tool allow the user to compose multiple routes into one, -so some validation must still be performed even for predefined routes.

Since it is also possible for the user to define a custom network layout and routes in a script, the model generating tool will by default perform validation on all routes specified, such as validating that a given route is a valid route in relation to the provided track layout.

The common validation for both the predefined routes from the XML file, and the custom routes specified in a script, are validations of all the routes to check if they are valid in the provided layout. A set of routes are essentially valid if there is no obvious deadlock or conflict between the routes.

The route validation checks the following set of properties:

- **Different beginning**. No two trains must have routes that start at the same track segment in the network.
  If two trains were to start at exactly the same track segment, there would be a collision right from the beginning.
- **Different end**. No two trains must have routes that end at the same section in the network.
  If two trains were to have the same final destination in a network, an obvious deadlock would occur since only one of the trains would be permitted to reserve and traverse its route.
- **No exact opposite routes**. No two trains must have routes such that the trains start and ends at exact opposite sections in the network.
  An obvious deadlock would occur and none of the trains would be permitted to reserve and traverse its route.

- **No same routes**. No two trains must have exactly the same routes. Either a collision would occur or an obvious deadlock would prevent any progress in the system.

A model obeying any of the first three conditions, will naturally also obey the last condition, which means that the model generating tool don't need to actively validate this condition when it has validated the other mentioned conditions.

The validation required for individual routes, which are either composed predefined routes or custom routes from a script, are

- **Itinerary elements must be transitively wellformed in the layout**. This is a validation that checks that all the neighboring track elements in the specified routes has a legal transitive relation in the track layout. Two neighboring route segments are invalid in a layout if there is no direct connection between them. The route segments must also obey the nature of points such that all routes go through points, from stem to fork or fork to stem, but never from fork to fork (as illustrated in figure 7.1).
- **A train route must start and end at linear track segment types**. Trains should never start or stop on a point track segment, but always start and end at linear track segments.

All of the above conditions are verified for each of the routes specified as input by the user as either a script or input values to the model generating program using XML files.

Following is an illustration of valid and invalid routes.

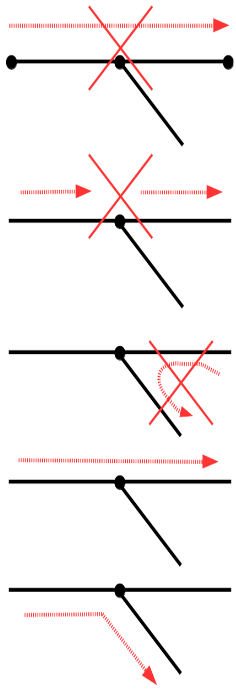

Figure 7.1: Illustrating examples of valid and invalid routes. The first is an *invalid* route that goes from point to point. The second is an *invalid* route that skips an element in the layout. The third is also an *invalid* route that goes from fork to fork through a point. The last two routes are examples of well formed routes.

7.1.3 Enforcement of length constraints

The generic UMC model essentially allows for any length to be defined for all the classes. As mentioned in the discussion in the previous chapter, the modeling freedom in defining lengths, can lead to various problems, where the first described problem is related to the performance of the model checking and the second described problem is a possible deadlock.

Both problems can be solved, by enforcing a set of constraints when generating models.

- **All routes must start and end with a track segment of exactly the same length as the train.**

This constraint must be enforced in order to prevent a train from causing a potential deadlock. If a train is allowed to be longer than its destination track segment, it might occupy more than just the final track segment, which might prevent other trains from progressing to their destination. If a train is allowed to be longer than the start track segment it might lead to a collision between two trains right from the beginning.

- **No track segment length defined in a route must be longer than the train owning the route.**
  Constraining the track segment lengths such that they are exactly the same length or smaller than the train traversing them, improves performance of the model checking since it prevents the expansion of unnecessary movement states within the same track length. A train with a length smaller than the track segment that it traverses, will not add anything meaningful to the model compared to a train traveling over a track segment with exactly the same length as the train.
  If a track segment is supposed to be modeled to be longer than the train, then it is good enough that the track in the model has exactly the same length as the train since this still will represent that the train is in transition over the track when the length of the train is solely on that track.

- **The longest train determines the length of intersecting track segments between routes.**
  For any to trains with intersecting routes, the train with the longest length determines the length of the intersecting route segments.
  If the trains are of equal length, then all their intersecting track segments must be of equal length.
  If given two trains of different lengths with an intersecting track segment described in the route of the longest train, that is shorter than the length of the shortest train, then the intersecting track segment length in the route of the shortest train must be exactly the same length as the length specified in the route of the longest train.
  If given two trains with different lengths with an intersecting track segment described in the route of the longest train to be of the same length as the longest train, then the length of the intersecting track segment in the route of the shortest train, must be exactly the length of the shortest train.

As already pointed out it is not efficient to traverse a track segment longer than the train that traverses it. This essentially justifies the last two constraints.

For the last constraint, the longest train must determine the length of the intersecting track segments for the shorter train, such that if the length of the intersecting segment is equal to the length of the longest train, then the length of the given track segment must be set to the exact value of the shortest train for its own route.

Observe that it is still possible to model tracks that are shorter than the lengths of the trains, this is such that very long trains covering multiple tracks can be modeled as well. Also note that only train lengths of two or higher are meaningful in this model, since a train of size one would be unable to simulate the overlapping transition between two tracks. A train of size one would essentially be reflecting a train jumping between tracks and not transitioning smoothly between them.

Following formula describes the length constraint for an intersecting track segment, which is represented as *L1* in the route of the longest train *T1* and as *L1'* in the route of the shortest train *T2*.

$$(T2.length \leq L1.length \leq T1.length \wedge L1'.length = T2.length) \vee$$
$$(T2.length > L1.length < T1.length \wedge L1'.length = L1.length)$$

Note that the previously mentioned constraints must still hold true. For example if the the intersecting track segment is at the beginning or end of a route it must still obey to the constraint that the given track segment should be exactly as long as the train holding the route. Note also that are points constrained and defaulted to a length of one, which means that they are disregarded in the description of length constraints.

Three scenarios with intersecting routes between two trains *Train1* and *Train2* are illustrated below. In the scenarios *Train2* has a length of two and *Train1* has a length of four, and the relevant intersecting track section is *L3* for the two first scenarios and *L4* in the last. In the two first scenarios *Train1* constraints *Train2*s length representation of *L3*, and *L4* in the last scenario. All the presented scenarios are examples of values that obeys the described constraint.

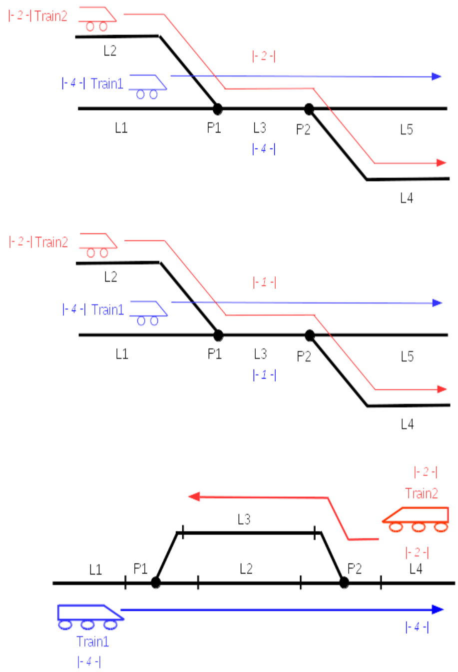

Figure 7.2: Three scenarios with intersecting routes where the length representations are obeying the constraints.

7.1.4 Creation of Object Instantiations and Modeling Language Specific Constructs

The key objective for the tool is to generate wellformed objects that together compose a concrete model.

The most defining element of the models, are the train routes which most of the object initializations depend upon. In the case of generating UMC models, the tool will generate three types of objects which are concrete instantiations of the generic class components described in the previous chapter. These are object instantiations of the *Train* class, the *Linear* class and the *Point* class. Furthermore will the tool generate the UMC specific *abstraction* definitions and *model checking properties*.

The generation of a UMC model is illustrated below, where it can be seen that *objects*, *abstractions* and *properties* are generated based on a railway layout and a set of routes.

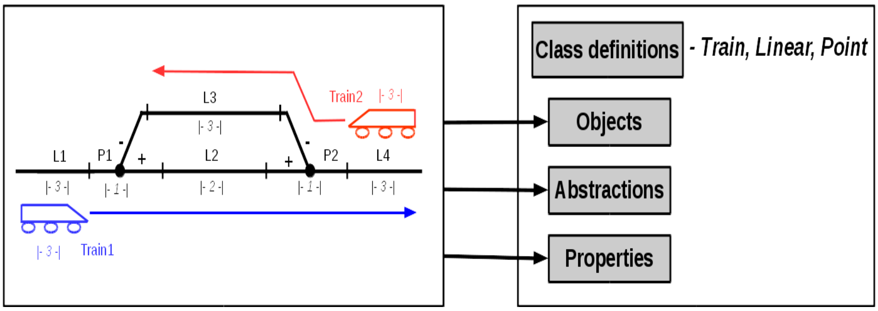

Figure 7.3: A model is generated from a specification of a railway network and a set of routes. The specification is turned into objects, abstractions and properties, using the generic model which the class definitions represents.

In this sub-section, concrete instantiations are illustrated with examples of object instantiations, *abstraction* definitions and *model checking property* definitions, for the UMC model described in the previous chapter. All the instantiation examples are based on the railway network model illustrated below. The model has two trains with routes consisting of the elements [L1, P1, L2, P2, L4] and [L4, P2, L3] where components prefixed L refer to linear track sections and components prefixed P are points. The points in the diagram have been illustrated with explicit *PLUS* and *MINUS* positionings in the form of plus and minus symbols.
A concrete example of a set of object instantiations, abstractions and properties can also be found in Appendix D, using the same example model as presented here, but with slightly different naming.

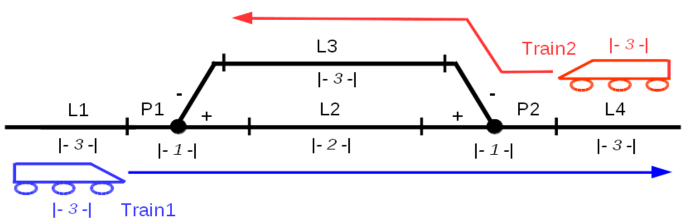

Figure 7.4: Example of a concrete model with two trains and two routes, which is used as running example in all of the following subsections.

7.1.4.1 Object Initializations

Point Objects The *point* objects are the simplest type of object instantiations for the UMC model.
For the given model in the diagram, the point object instantiations are simply as follows

$$\begin{array}{l}\text{P1: Point;}\\ \text{P2: Point;}\end{array} \tag{7.1}$$

The points don't need to be instantiated with any parameters because of the way the *Point* class is specified in the UMC model. As specified in the *Point* class, the point objects will be instantiated with a default positioning value of true ($PLUS$).

Linear Objects The *linear* objects are also quite simple, as all that is needed in the instantiation, is to instantiate the linears with information about the absence or presence of a train.

For the model in the diagram, the linear object instantiations would be as follows.

$$\begin{array}{l}\text{L1: Linear (train => train1);}\\ \text{L2: Linear;}\\ \text{L3: Linear;}\\ \text{L4: Linear (train => train2);}\end{array} \tag{7.2}$$

The *train* variables are only initialized for the linears which are at the beginning of a given train route. When no value is specified, the *train* variable simply defaults to *null*.

Train objects The *train* objects are the most involved objects as they contain information about track segment lengths which must be specified consistently across the trains, the initial *occupies* variables used for for simulating train movement, and finally the routes which must be valid between the trains.

The instantiation based on the running example model in figure 7.4 is as follows

```
train1: Train(
            route_segments => [L1,P1,L2,P2,L4],
            track_lengths => [3,1,2,1,3],
            train_length => 3,
            occupies => [L1, L1, L1],
            requested_point_positions => [null,True,null,True]);
train2: Train(
            route_segments => [L4,P2,L3],
            track_lengths => [3,1,3],
            train_length => 3,
            occupies => [L4, L4, L4],
            requested_point_positions => [null,False]);
```

(7.3)

The array *route_segments* must be initialized with the ordered train route, such that the routes are valid between the trains in the same layout.
Likewise must the *track_lengths* array be initialized with the length value of the index-wise corresponding elements of the *route_segments* array, in a way such that the length of a given element obeys the length constraints between the route descriptions of the trains. As previously mentioned, points are always instantiated with a length of one.
The *occupies* array must be exactly same length as the train, and it must be filled with references to the first track segment of the given train route.
The *requested_point_positions* array, must be defined such that it refers index-wise to the *route_segments* array. The *requested_point_positions* array is simply filled with *null* values for the indexes which does not correspond to points on the route, and it only needs to define values up to the index of the last point element in the route array.

7.1.4.2 Generation of Language Specific Constructs

Specific for the UMC modeling language is that it has special constructs called *abstractions* which are used in the definition of the model checking properties. The concrete definitions of the abstractions are, like the the objects, dependent on the given model layout and routes.

Following are concrete examples of definitions based on the formally specified abstractions and properties that were formally described in the model checking section of the previous chapter.

The *no collision* property
The no-collision property is generic since it simply validates the abstraction *trains_at_diff_positions* globally over all paths. For the running model example, the property is simply defined as follows

$$\text{AG (trains\_at\_diff\_positions)}$$

The *trains_at_diff_positions* abstraction
The *trains_at_diff_positions* abstraction is rather involved, and cannot be generalized so well in the language of UMC. In the tool, a cross product of the indexes of any two trains *occupies* arrays are generated, and based on this a set of conjunctions are composed, each specifying that two any two indexes of the *occupies* arrays must be different.
The instantiation based on the running example model is presented below.

$$\begin{array}{llll} \text{State} & \text{train1.occupies[0]} & /= & \text{train2.occupies[1] and} \\ & \text{train1.occupies[0]} & /= & \text{train2.occupies[2] and} \\ & \text{train1.occupies[1]} & /= & \text{train2.occupies[2] and} \\ & \text{train1.occupies[2]} & /= & \text{train2.occupies[1] and} \\ & \text{train1.occupies[2]} & /= & \text{train2.occupies[0] and} \\ & \text{train1.occupies[1]} & /= & \text{train2.occupies[0] and} \\ & \text{train1.occupies[0]} & /= & \text{train2.occupies[0] and} \\ & \text{train1.occupies[1]} & /= & \text{train2.occupies[1] and} \\ & \text{train1.occupies[2]} & /= & \text{train2.occupies[2] -> trains\_at\_diff\_positions} \end{array} \tag{7.4}$$

The *no derailment* property
As mentioned in the previous chapter, the *no derailment* property uses a set of abstractions defining absence of trains on points, and a set of abstractions defining positioning of points.

Using the running example, the absence of trains on points are declared as follows by the model generator tool.

$$\begin{array}{lll} \text{State} & \text{p1.train = null} & \text{-> no\_train\_on\_p1} \\ \text{State} & \text{p2.train = null} & \text{-> no\_train\_on\_p2} \end{array} \tag{7.5}$$

And the abstractions specifying points in positioning states, are declared as follows, using the running example model.

$$\begin{array}{lll} \text{State} & \text{inState(p1.POSITIONING)} & \text{-> position\_p1} \\ \text{State} & \text{inState(p2.POSITIONING)} & \text{-> position\_p2} \end{array} \tag{7.6}$$

The *no derailment* property can now be specified, for the running model example, using above abstractions.

```
AG(
        position_p1   implies   no_train_on_p1 and
        position_p2   implies   no_train_on_p2        )
```
(7.7)

The *progress* property
The progress property which specifies that trains eventually will arrive at their destinations, simply uses one abstraction which captures the the global state where a train has arrived in the system.
Using the running example model, an abstraction is simply defined for each of the trains as follows.

```
State   inState(train1.ARRIVED) -> train1_arrived
State   inState(train2.ARRIVED) -> train2_arrived
```
(7.8)

The property can now be declared as follows, for the running example model.

EF AG (train2_arrived and train1_arrived)

As mentioned in the previous chapter, the above property only verifies that there exist states where the trains will reach their destinations. However, the claim is that the trains will arrive in all cases where the *points* on the routes functions correctly without malfunctions. And to verify this claim for the current model, a special property can be created as described in the previous chapter.

The property requires a set of abstractions, each capturing the situation where a point has entered a *MALFUNCTION* state.

```
State   inState(p2.MALFUNCTION) -> p2_malfunction
State   inState(p1.MALFUNCTION) -> p1_malfunction
```
(7.9)

The property for verifying the claim, can then be defined as follows

```
not E[   not (p2_malfunction or p1_malfunction) U
         (final and not (train2_arrived and train1_arrived))]
```
(7.10)

(For a more elaborate explanation of this property, refer to the formal specification of the given property in the previous chapter)

The *no message loss* property
The property for checking that all messages are handled in the system, is independent of the actual model, and therefore the abstraction *discarded_message*, and the property that verifies that for all state paths no message will ever be lost, -can both be used directly as they are defined in the previous chapter.
Thus, for the running example model, the abstraction *discarded_message* is simply defined as

Action: lostevent -> discarded_message

and the property itself is defined as

AG not (EX discarded_message true)

7.2 Implementation

The model generating tool has been implemented in the programming language F#. F# was, first of all, chosen due to familiarity and experience, but also because F# provides a special set of tools for conveniently parsing data files such as XML files with minimal effort.[PGS16]
Other considered languages and frameworks were the Idris programming language, which is a functional language that provides dependent types as first class citizen as a core facility[BRA13], and RAISE (Rigorous Approach to Formal Software Engineering) RSL (RAISE Specification Language)[Hax14].

Using Idris was discarded due to lack of practical experience with the language, and using RSL was discarded due to the lack of out-of-the-box facilities for, for example, working with XML files.

Both an executable model generating program and simple scripting DSL (Domain Specific Language) library has been implemented, both using the same set of core functions for validation, constraint verification and internal model representation and composition.

The executable program can be given an XML file path as input together with a set of route definitions to generate a complete model, but has been simplified to only generate models where trains and linear segments all have a length of two and points a length of one. The scripting DSL gives full freedom in specifying lengths of trains and track segments, but still validates wrt. the constraints.

The source code used in the program and scripting DSL tools is divided into four main modules which are *Utils*, *InterlockingModel*, *UMC*, *XMLExtraction* and *MiniModelGenerator*.

Below an informal diagram is presented, showing the dependency relationship between all of the files containing the main modules, and where in the model generating process they are used.

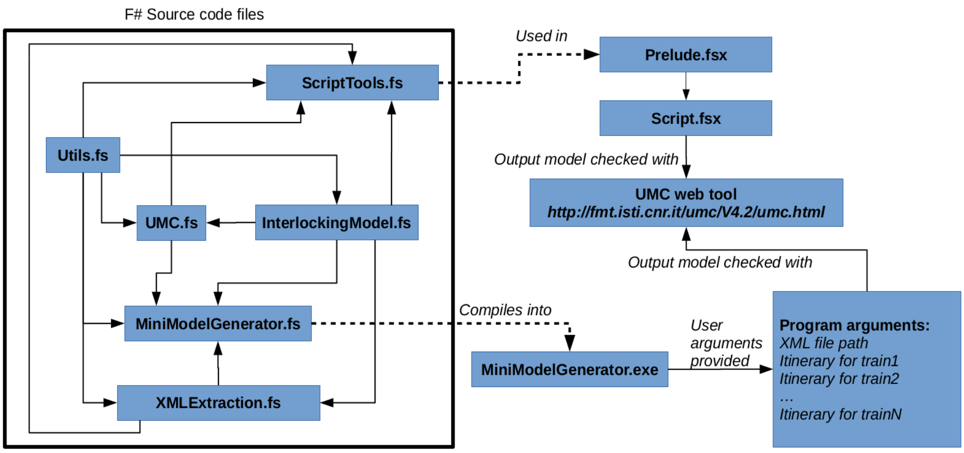

Figure 7.5: Overview of the Tool components. Arrows points to where a given module is used.

The source code for all of the files can be found in Appendix E.

- The *Utils* module contains general types and functions which has no concrete relationship to the model generation.
- The *InterlockingModel* module defines generalized types and functions for defining a model, furthermore does it contain an higher order function *validateAndGenerateModel* which performs all validation and constraint verification on a model, and can be applied to a specific model generating function, such as for example, a function for generating a UMC model.
- The *UMC* module defines a set of functions for instantiating models in the UMC language, and defines a specific UMC model generating function which can be passed on to the higher order function *validateAndGenerateModel* defined in the *InterlockingModel* module.
- The *XMLExtraction* module defines functionality for reading XML files and constructing an internal F# data type representation of the model to be generated.
- The *MiniModelGenerator* module contains the entry point for the executeable program, and performs basic validation of the user inputs.

The scripting library tools are implemented in the *ScriptTools* module which exposes a set of simple types and functions that can be used as a small DSL directly from an F# script file.

Each of the mentioned modules are defined in its own file, and are divided into several sub modules that each contain functions related to the same aspect of the model generator.

A model can be generated either based on a specification in a script, or from the compiled *MiniModelGenerator* program which can be provided with a set of user input arguments including a file path to an XML file containing a network layout and predefined routes.

Descriptions on how to use the tool to generate models, can be found in Appendix A.

7.2.1 Modules

This subsection provides a high level description of each of the main modules involved in the model generation.

7.2.1.1 Utils module

This module contains common functionality such as functions for generating all possible paired combinations of values from a list or cross products from two lists, and types and functionality for error handling.
The module notably defines a type *Result*, a computation expression *resultFlow* and a computation expression *maybe*, -which all are used extensively throughout all of the other modules.

The type *Result<'T1, 'T2>* and the Computation Expression *resultFlow*
The Result type carries a generic success type (*'T1*) and a generic error type (*'T2*). This type is used throughout the whole program and libraries to handle error cases and to collect and propagate errors out to the user.
The Result type is defined as follows

```
type Result<'success,'error> = Ok of 'success
                             | Error of 'error
```

The *Result* type is extended with a set of operators and functions for composability of values and functions using the Result type. The type is extended with functions for common list operations such as *reduce* and *fold*, and a special F# construct called a *Computation Expression*[PS14].
The type and its associated *Computation Expression* together form a Monad-pattern[wikb]. The implemented monad pattern, used for the *Result* type, is more commonly known as the *Either Monad* pattern.[1]

Monad patterns makes it syntactically convenient to compose value wrapping types and functions that returns types of the same value wrapping type.

To implement a Monad pattern, a function *bind* and a function *return* must be defined.
The *return* function is simply a function that takes a simple value of an arbitrary type as input and returns the same value wrapped in the type used for the monad pattern (eg. the Result type). The type used in a monad pattern is commonly referred to as a monadic type.
The *bind* function is a higher order function that takes two arguments as input, a value of arbitrary type wrapped in the monadic type, and a continuation function that takes a value of the same arbitrary type, as the first argument, -as input and returns a result which is another arbitrary type wrapped in the monadic type. The *bind* function essentially unwraps the first argument from the monadic type, and applies the result to the continuation function, after which the result of the continuation function is returned.
The abstract type signatures for the bind and return functions are sketched below

$$return: \ 'a -> M \ 'a$$
$$bind: M \ 'a -> (\ 'a -> M \ 'b) -> M \ 'b$$

where *'a* and *'b* are arbitrary types and *M* is the monadic type. The bind function is also commonly implemented as the binary operator »=, and such an operator is also defined for the Result type in the model generating tool implementation.
In order for a type to be classified as monadic and to be optimally composable, the bind and return functions must obey the three Monad Laws[wikb], which essentially describes how to compose the *bind* and *return* functions.

[1] F# 4.1 will be released late 2016 and will implement a type *Result* with the exact same signature. Therefore the name *Result* has been chosen in this implementation, instead of the name Either.

The Computation Expression for the *Result* type is instantiated into a construct named *resultFlow*, and is used extensively throughout the implementation for error handling and propagation of error messages.

The *maybe* Computation Expression construct
F# defines a standard library type *Option<'T>* for constructing optional types that can be either be *None* or a value of type *'T*.
The *Utils* module implements a Computation Expression construct *maybe* to help composing values of type *Option* and functions returning the type *Option*. The defined Monad is often referred to as the Maybe Monad, thus the name *maybe* for the construct in this implementation.

7.2.1.2 InterlockingModel module

This module contains all the basic type definitions necessary for describing an interlocking system model. Furthermore does the module define sub-modules containing functions for verifying that a given model is sound with regards to the routes and with regards to the length constraints.
At last the module define an higher order function *validateAndGenerateModel* which takes a model generating function as input and generates a textual repressentation of a valid model.
The module contains the sub modules *TypeDefinitions*, *RouteConstruction*, *RouteValidation*, *LengthConstraints* and *ModelGeneration*. The *TypeDefinitions* module contains all the basic types for representing a concrete railway model in F#. The module *RouteConstruction* contains functions for composing multiple routes together to form one route. The module *RouteValidation* contains functions for validating a set of routes against each other, and against a railway network layout. The module *LengthConstraints* contains functions for checking that the length constraints within and between between intersecting routes, are obeyed. The *ModelGeneration* contains functions that uses the functions from the other modules, to validate and compose valid models, most important, the sub module defines the *validateAndGenerateModel* function.

The core type definitions representing the internal representation of the concrete railway model components, are presented below.

```
type TrainId = TrainId of string
type TrainIds = TrainId list

type LinearId = LinearId of string

type PointId = PointId of string

type PointPosition = Plus | Minus
```

```
type RouteSegment =
    | LinearRouteSegment of LinearId * length : int
    | PointRouteSegment of PointId * required_position : PointPosition
type RouteSegments = RouteSegment list

type RouteDirection = Up | Down

type Route = Route of RouteSegments * RouteDirection
type Routes = Route list

type Train = { id : TrainId
               route : Route
               length : int }
type Trains = Train list

type Linear = { id : LinearId
                train : Train option }
type Linears = Linear list

type Point = { id : PointId
               position : PointPosition }
type Points = Point list

type LayoutSegment =
    | LinearLayoutSegment of id : string
    | PointStemLayoutSegment of id : string
    | PointForkLayoutSegment of id : string * position : PointPosition

type RailwayNetworkLayout = Map< LayoutSegment, LayoutSegment>

type ModelObjects = { trains  : Map<TrainId, Train>
                      linears : Map<LinearId, Linear>
                      points  : Map<PointId, Point> }
type ValidatedModelObjects = Validated of ModelObjects

type ModelGeneratorFunction = ValidatedModelObjects -> string
```

These types together describe the internal representation of a given model in the tool implementation, and the functions implemented in the *InterlockingModel* module all deal with one or more of these types.

Notably, is the type *ModelObjects* used to represent the objects that must be initialized in the modeling language to compose the concrete model. The simple wrapper type *ValidatedModelObjects* is defined to represent a model that has been validated.

The *RailwayNetworkLayout* type describes a railway network as a mapping from a *LayoutSegment* to a *LayoutSegment*. Consequently, the network layout is naturally only represented as connections from left to right, which also represents the *Up* direction. This means that when a route in the *Down* direction is to be verified against the layout, it is reversed before the verification.

The last type definition *ModelGeneratorFunction*, is a signature description of a general function that can produce a concrete model in a string representation, based on a valid model. The *InterlockingModel* module itself implements such a function for outputting the 'raw' internal F# model representation as a string, but more importantly does the *UMC* module implement such a function as well for generating a textual representation of a full UMC model. The type is used in the signature of the function *validateAndGenerateModel* which is the only function in the module used from the other modules. The type signature and implementation of the function is presented below.

```
val validateAndGenerateModel : ModelGeneratorFunction ->
                               RailwayNetworkLayout ->
                               ModelObjects ->
                               Result<string, string>

let validateAndGenerateModel : modelGenFun = fun layout objects ->
    validateTrainRoutes layout objects
    >>= checkLengthContraints
    >>= updateTrainLocations
    >>= (modelGenFun >> Ok)
```

Note that signatures and functions normally aren't defined together like this in F#, but for the sake of presentation the evaluated type of the function is shown above the function in this code fragment.

As can be seen from the presented code, the function makes use of the bind operator described in the *Utils* module, for chaining together functions that produce *Result* types.
The function first makes sure the routes are validated by applying the *validateTrainRoutes* function, the result from that function is applied through the *bind* operator to the *checkLengthContraints* which verifies that the length constraints are obeyed. Next the result is applied through the bind operator to the *updateTrainLocations* function which updates the track elements in the model with regards to presence of trains. At last the result is applied through the bind operator to the *modelGenFun* function which for example can be a reference to the UMC model generating function. If at any of the bindings an error is returned, the error will simply be propagated out as an error result of the *validateAndGenerateModel* function.

The last thing to note about the *InterlockingModel* module, is that it defines a class interface *ModelCheckingPropertyDefinitions* which simply act as a template for the actual properties to be implemented. The interface definitions is presented below.

```
type ModelCheckingPropertyDefinitions =
```

```
    abstract NoCollision : string
    abstract AllTrainsArrived : string
    abstract NoDerailment : string
    abstract TrainsDetectedOnPoints : string
    abstract AllMessagesHandled : string
```

7.2.1.3 UMC module

The *UMC* module imports the source files *UMCLinearClass.fs*, *UMCPointClass.fs* and *UMCTrainClass.fs*, which each simply contains a string representation of the three generic classes defined in UMC (The Linear class, the Point class and the Train class). At the end of the model generation, these string representations are simply concatenated and prepended to the generated objects and abstractions of the model. The motivation for containing the generic UMC class information this way instead of having them in separate files, is simply such that the type checker will yield an error if the files doesn't exists.

The *UMC* module basically contains functions for generating a full textual UMC model. The module contains the sub modules *AbstractionDefinitions*, *ModelObjectInstantiations* and *Properties*. The *AbstractionDefinitions* module defines functions for generating concrete UMC abstractions, the *ModelObjectInstantiations* module defines functions for generating the set of UMC object instantiations for a final model, and the *Properties* module define the functions for generating and composing UMC model properties.

The *UMC* module furthermore implements the *ModelCheckingPropertyDefinitions* interface (described in the *InterlockingModel* module) in a class that has a constructor which takes a set of validated model objects and instantiates each of the model properties.

At last the *UMC* module defines a function *composeModel*, which utilizes functions from the *UMC* module to implement a function of the type *ModelGeneratorFunction*, that generates and composes a full textual UMC model. Since the function has the type signature *ModelGeneratorFunction*, it can be used directly together with the *validateAndGenerateModel* function from the *InterlockingModel* module, to generate a valid full UMC model.

7.2.1.4 XMLExtraction module

This module defines functionality for extracting and composing an internal model representation, based on a given XML file. The module notably uses

the special F# Type provider library for parsing and handling XML files.
The module contains the sub modules *BasicObjectExtraction*, *LayoutExtraction*, *RouteExtraction* and *ModelGenerationFromXML*. The module *BasicObjectExtraction*, defines functions for extracting and creating the basic components required for creating an internal model. The *LayoutExtraction* module defines functions for extracting and constructing an internal railway network layout representation from an XML file. The module *RouteExtraction* defines functions for extracting a set of routes from an XML file and generating an internal model representation. The *ModelGenerationFromXML* module, finally defines a type *ModelGenerationParameters* and the function *generateModelFromXML*. The *ModelGenerationParameters* type is a record type holding a reference to a concrete model generating function with the type signature *ModelGeneratorFunction*, a file path to an XML file and a field *routes* which is a list of lists of route ids to be extracted from the XML file. Each list of ids describes a composition of multiple routes.

```
    type ModelGenerationParameters =
        { modelGeneratorFunction : ModelGeneratorFunction
          xml_file_path : string
          routes : string list list }
```

The *generateModelFromXML* function has following type signature, which returns a result that can be either a string representation of a final composed model, or an error message describing for example the violation of a constraint during the validation of a model.

val generateModelFromXML : ModelGenerationParameters -> Result<string,string>

The function uses the extraction functions described in the other sub modules to extract and compose an internal representation of the model, which it then proceeds to apply to the *validateAndGenerateModel* function from the *InterlockingModel* module to validate and generate a final concrete model instantiation in the modeling language of interest.

7.2.1.5 MiniModelGenerator module

The *MiniModelGenerator* module defines the entry point for the compiled model generator tool. The entry point is defined as a function *main* that takes an array of user provided arguments, parses them, generates a model and outputs the string representation of the given model instantiation in the console.
The module defines functionality for parsing user input such as lists of route ids, referring to the route ids of routes in an XML file, and a type *ModelOutput* with following type signature

```
type ModelOutput = UMC
                 | Raw // representing the raw F# objects
```

which describes the available textual model outputs, and are either a raw string representation of the internal model in F#, or a concrete UMC model which can be used directly as input to a UMC model checking tool.

7.2.1.6 ScriptingTools module

The scripting tools module describes a set of very simple types for specifying a concrete model, and exposes one function *generateUMCModel* which validates, composes and generates a concrete UMC model based on a simple type representation of the model.
The simple types for defining a model are listed below

```
type SimpleTrackSegment =
    | LLinear of name : string
    | LPointFork of name : string * PointPosition
    | LPointStem of name : string

let (<+>) (el1 : SimpleTrackSegment)
          (el2 : SimpleTrackSegment) = el1, el2

type SimpleLayout = (SimpleTrackSegment * SimpleTrackSegment) list

type SimpleRouteElement =
    | RLinear of name : string * length : int
    | RPoint of name : string * position : PointPosition
type SimpleRoute = SimpleRouteElement list

type SimpleTrain =
    { id : string
    ; length : int
    ; route : SimpleRoute
    ; route_direction : RouteDirection }
type SimpleTrains = SimpleTrain list

type LayoutType = CustomLayout of SimpleLayout
                | XMLLayout of path : string

type SimpleModelArgs =
    { trains : SimpleTrain list
    ; layout : LayoutType
    ; show_stats : bool
    ; output_file : string option }
```

The types and operator basically forms a small DSL like language which the user can use to define a set of trains with routes and a network layout, and apply these to the *generateUMCModel* function to produce a full and valid textual UMC model.

A guide on how to compose a script can be found in Appendix A.

7.3 Extending the Model Generator to support other modeling languages

The code has been structured such that it can be used as a library for generating models. To implement support for more model checking languages, one could define a module similar to the *UMC* module, for generating the textual string representations in the given modeling language. The module must furthermore implement a function with the signature of the type *ModelGeneratorFunction*, which then can be used as input to the *validateAndGenerateModel* function from the *InterlockingModel* module, to produce validated concrete models in the target modeling language.

Chapter 8

Experiments

Logic takes care of itself; all we have to do is to look and see how it does it.

Ludwig Wittgenstein

During the project, many experiments has been performed with concrete models based on the presented generic UMC model.
This chapter describes a few of the experiments and presents the results from performing model checking on concrete models. Specifically, a set of experiments has been set up to investigate the scalability of the model with regards to the size of the railway network and number of trains. Investigating the model checking performance with regards to scalability, essentially serves to benchmark the performance of the generic model devised in this project.

8.1 Performing the experiments

The experiments performed in this project, refers to the generation of a concrete model instantiations with a set of train routes based on a specific type of network

layout, and at last the model checking of said model instantiation.
The three first types of model instantiations used in the following sections, has been generated using the scripting tools described in the previous chapter, and the concrete scripts used can be found in Appendix F.
The last set of experiments was performed with models generated from XML files by using the executable model generating tool.

The generated UMC models are loaded and model checked through an online service with a web-interface[Mazb]. The online service is hosted on a server with an Intel Xeon 2x2.66 GHz Quad-Core processor and 24 GB of Memory. The online service might be used simultaneously by other clients, and furthermore does the server host other applications, which all might impact the execution time of the model checking.

8.2 Experiments performed

This section describes to sets of different experiments performed to test the scalability of the model.
One set of experiments seeks to investigate the scalability of the model with regards to the length of train routes, and the other set of experiments seeks to investigate the scalability of the model with regards to number of active trains in the model.

8.2.1 Two trains and varying route lengths

The first set of experiments seek to investigate the performance of the model when it is instantiated with long train routes.
All the model instantiations contains two trains, each with a route as long as possible. The model instantiations are based on the layout below, which contains a set of reconnecting branch loops which can be perceived as stations.
For the given set of experiments, models from one to ten stations has been generated. Out of these ten models, it was possible to model check the first eight within reasonable time.

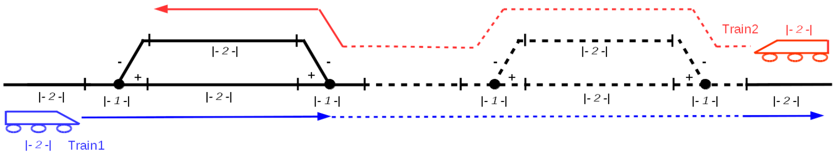

Figure 8.1: Drawing of the model type which is generated for the experiments. The generated layouts have a varying number of reconnecting loops, or stations.

The results from model checking the first eight models, are presented in the table below, where the model names refer to the number of stations in the given model.

	One	Two	Three	Four	Five	Six	Seven	Eight
Route lengths	[5;3]	[9;7]	[13;11]	[17;15]	[21;19]	[25;23]	[29;27]	[33;31]
Number of linears	4	7	10	13	16	19	22	25
Number of points	2	4	6	8	10	12	14	16
Number of route sub-segments	13	25	37	49	61	73	85	97
Number of shared points	1	3	5	7	9	11	13	15
Number of shared linears	1	2	3	4	5	6	7	8
*Time for **Trains correctly detected on points***	*0.3*	*5.7*	*48.4*	*242.1*	*1005.0*	-	-	-
Time for **No collisions**	0.2	1.7	7.8	25.2	72.6	176.8	381.2	952.1
Time for **No derailments**	0.2	1.3	7.5	25.0	74.2	180.0	394.9	795.1
Time for **Will arrive**	0.2	0.9	6.0	14.1	67.7	140.2	273.3	567.3
Time for **No message loss**	0.1	0.9	4.8	14.9	42.3	96.7	212.4	408.6
Total time used checking properties	0.7	4.9	26.2	79.2	256.8	593.7	1261.9	2723.2
Number of states explored	736	6795	28768	84314	198868	406409	750228	1283696

Table 8.1: Resulting data from experiments performed with two trains and varying number of stations from one station to eight stations. All time values are given as seconds and the route lengths of the involved trains are presented as a list.

All the listed properties evaluated to true in the model checking.

Following listing describes the variables presented in the table.

- *Route lengths* are presented as a list of values separated by ';', where each value refers to the length of a route. The length of a route describes how many track segments, consisting of points and linear segments, which the route covers.
- *Number of linears* is the total number of linear segments in the network.
- *Number of points* is the total number of points in the network.

- *Number of route sub-segments* is the sum of all of the sub-segments which both of the routes covers, where a sub-segment is either a point or a part of a linear segment.
- *Number of shared points* is the total number of points shared between all of the routes.
- *Number of shared linears* is the total number of linears shared between all of the routes.
- *Time for **Trains correctly detected on points*** presents the time used for verifying the property **Trains correctly detected on points** through model checking. As can be seen in the table, the property is significantly more time consuming to model check, and therefore it has been skipped for the last three models.
- *Time for **No collisions*** is the time used for model checking the property **No collision**.
- *Time for **No derailments*** is the time used for model checking the property **No derailments**.
- *Time for **Will arrive*** is the time used for model checking the property **Will arrive**.
- *Time for **No message loss*** is the time used for model checking the property **No message loss**.
- *Total time used checking properties* is the sum of the time used for model checking each property. However, to ease the comparison of the models, -the sum excludes the time used for checking the property **Trains correctly detected on points** since this property has only been checked for the first five models.
- *Number of states explored* is the total number of states explored by the model checker in its pursuit of verifying the properties.

All the described properties are described in chapter 6.

Following are two plots of the obtained data. First a plot of the number of states explored in relation to the number of stations, and next a plot illustrating the time spend on model checking the properties in relation to the number of stations.

From the plot below it can be observed that the number of states increases with an exponential growth as the number of stations increases and the routes gets longer.

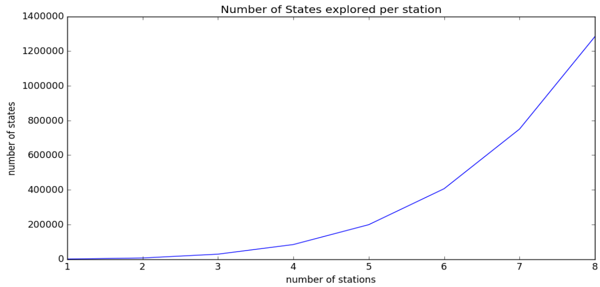

Figure 8.2: Plot of the states explored for each number of stations

The following plot, clearly show that the two safety properties **No collision** and **No derailments** are guilty of the majority of time spend on model checking properties.

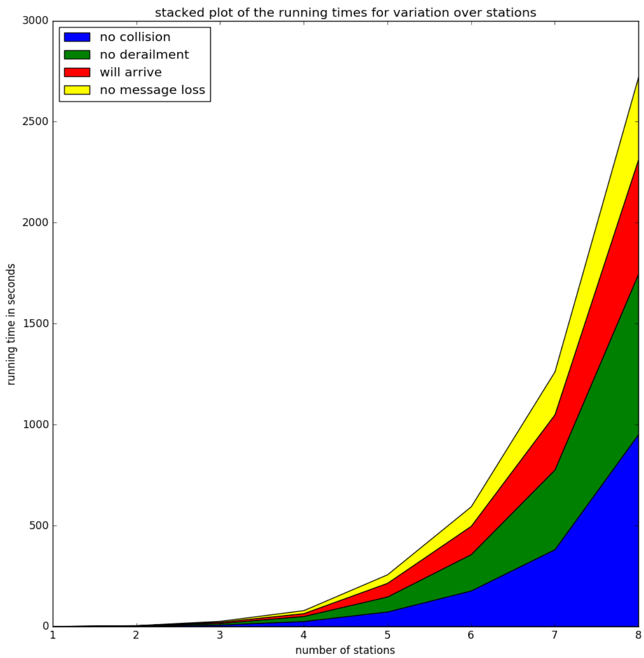

Figure 8.3: Stacked plot of the running times of each property for each number of stations

8.2.2 Varying number of trains

This set of experiments seeks to investigate how the model performs with regards to the number of active trains in the model, and specifically with regards to trains with intersecting routes.

The concrete models to be used, are based on the branching network layout presented in the drawing below.

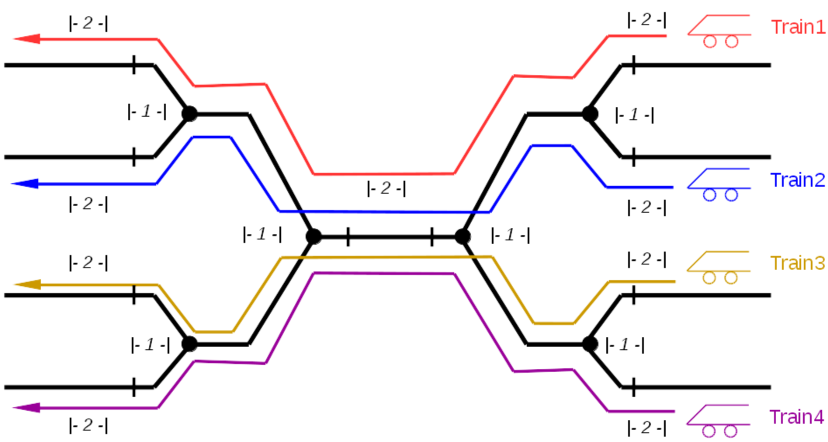

Figure 8.4: Drawing of the model used as basis for creating experiments with multiple trains.

Even though the devised script for generating the models is able to create bigger networks with room for more trains, only the layout presented in the drawing above was used, since model checking couldn't even be performed within reasonable time for just four trains with intersecting routes.

A table of the data obtained from model checking concrete model scenarios with **two**, **three** and **four** trains, are presented below.
The model with four trains didn't succeed in being model checked within reasonable time, however was reported by the model checker that it had explored more than seven million states.
Also note that the shared points and shared linears here describes the total number of intersecting points and linears between any two routes.

	Two	Three	Four
Route lengths	[7;7]	[7;7;7]	[7;7;7;7]
Number of linears	9	9	9
Number of points	6	6	6
Number of route sub-segments	20	30	40
Number of shared points	4	8	16
Number of shared linears	1	3	6
Time for **No collisions**	0.9	78.6	-
Time for **No derailments**	0.7	79.8	-
Time for **Will arrive**	0.5	40.5	-
Time for **No message loss**	0.4	49.1	-
Total time used checking properties	2.5	248.0	-
Number of states explored	2750	234691	7000000+

Table 8.2: Resulting data from experiments with varying number of trains.

All the listed properties evaluated to true in the model checking.

8.2.3 Experiments with particular layouts from XML files

Following experiments were performed with models generated by the executable model generating program with XML files containing layouts and routes.

	Mini	**Twist**	**Threelines**
Route lengths	[6;4]	[5;5;5]	[6;6;7]
Number of linears	6	8	13
Number of points	2	2	6
Number of route sub-segments	17	24	28
Number of shared points	1	3	5
Number of shared linears	1	2	2
Time for **No collisions**	0.4	8.4	91.9
Time for **No derailments**	0.3	8.1	96.3
Time for **Will arrive**	0.2	3.7	40.9
Time for **No message loss**	0.2	6.2	68.1
Total time used checking properties	1.1	26.4	297.2
Number of states explored	1473	34978	223294

Table 8.3: Resulting data from experiments with models generated from XML files.

	Twist	**Three trains**	**Threelines**
Route lengths	[5;5;5]	[7;7;7]	[6;6;7]
Number of linears	8	9	13
Number of points	2	6	6
Number of route sub-segments	24	30	28
Number of shared points	3	8	5
Number of shared linears	2	3	2
Time for **No collisions**	8.4	78.6	91.9
Time for **No derailments**	8.1	79.8	96.3
Time for **Will arrive**	3.7	40.5	40.9
Time for **No message loss**	6.2	49.1	68.1
Total time used checking properties	26.4	248.0	297.2
Number of states explored	34978	234691	223294

Table 8.4: Resulting data from experiments with models generated from XML files.

Here it is interesting to observe the dramatic difference between the two models Twist and Threelines which both involves three trains and three routes. Comparing the data from the Threelines model with the model using three trains from the experiment in the previous subsection, it seems that its the number of linears and points that make the difference. The Threelines model takes longer to model check and, has a total count of linears and points of 19, while the model with three trains from the previous subsection has a total count of linears and points of 15. The number of shared points and linears and the lengths of the routes are all smaller for the Threelines model. So it must be the number of points and linears that makes the difference here.

8.3 Discussion

As has been shown through the experiments, model checking concrete models with just two trains at the time is feasible for routes of limited lengths.
The increase in states to explore is however exponential which displays an important limitation of the model with regards to length of routes. It is certainly feasible to model check routes with lengths shorter than the ones presented in the experiments, but if they get much longer the state space explosion will be too significant to perform any model checking.

The experiments performed with more than two trains displayed an even more drastic explosion in states to explore, and the model with four trains was impossible to model check within a reasonable time frame.

It is worth reflecting over the fact that one of the reasons why model checking becomes much harder as the number of trains with intersecting routes increases, is by large due to the reservation protocol. When model checking any model with more than one train and with intersecting routes, one of the trains will always be the first to succeed in reserving a route and the rest of the trains will keep attempting to reserve until they succeed as well. This results in a lot of fruitless attempts at reserving routes without success.

One way to improve the models performance, might be to abstract away the concept of route reservation rejection, such that trains will patiently wait for their routes to be reserved instead of reattempting reservations until success. The nodes representing the track segments and points, could for instance be equipped with a queue for reservations, in which it will queue up reservation requests while it is reserved, and as soon it is free it will pick up the first reservation request from the queue and complete that reservation. The downside of this model, though, is that it moves further away from the original engineering concept, and perhaps the abstract model would be too far from anything realistically possible.

Another way to solve the problem of model checking models with multiple trains, might simply be to use the model as it is, and model check each pair of two trains with intersection routes by themselves. This opens up for model checking the model in parallel, by assigning the verification of each pair of intersecting

routes to its own process.
A challenge with this approach might be to verify progress and liveness properties for models with many trains. in order to confidently verify the whole model as sub-models in parallel, one would somehow have to prove that checking for liveness and progress for each pair would be the same as checking the whole model for liveness and progress.

Chapter 9

Future work

Even though a satisfying generic UMC model and a tool for generating concrete models, has been implemented, there is still possible work to be done. Notably the the generic UMC model can still be improved to better reflect a real life implementation, and the model generating tool can be extended with support for generating models in other modeling languages and be improved wrt. usability.

This chapter briefly elaborates over ideas for tool extensions, model improvements and usability enhancements for the tool.

9.1 Generic model enhancements

This section describes a few ideas for enhancement of the generic UMC model specification. Even though the ideas are described in the context of the generic UMC model conceived in this project, the ideas are more general as such and would also apply to a model specified in any other model checking language.

9.1.1 Repairment of faults

As mentioned in the discussion section of chapter 6, malfunctioning of a point will cause the point go into a MALFUNCTION state with no further transitions to take, and this essentially blocks trains from reserving and passing the given point. The fact that trains, in some cases, are prevented from reserving and traversing their route, makes checking the system for liveness impossible since liveness per definition requires the system to be deadlock free.

The current model could simply be enhanced such that points are able to transition back into a functioning state of availability again. This transition could represent that the point has been repaired.

9.1.2 Point Machines

In the discussion section of chapter 6, it is mentioned that the current model of the points is slightly detached from how a point would realistically operate in a life implementation.

The UMC model could simply be enhanced with a *Point machine class* and each *Point class* object would then have responsibility for one point machine, as described in the discussion of chapter 6. Modeling the points this way would also be more true to the nature of the two phase commit protocol, where changes ideally are to be enforced upon receiving the commit request -and not upon receiving the final agree response.

9.1.3 Modeling more types of faults

The model could be extended such that it models more types of faults and malfunctions. For instance, it could model failing track circuit sensors, such that all points and linear segments are able to randomly emit a false presence of a train, and letting reservation attempts fail as a consequence.

However, one should be careful about how the faults are modeled, since modeling of even a simple failure, leads to an increase of the size of the state space to be explored when model checking the properties.

9.2 Usability and performance improvements for the tool

9.2.1 Enhance the user experience with a route selection GUI

As the tool is implemented now, all user interaction either goes through the console or through user defined scripts. A user might greatly benefit from having a graphical user interface where he can define a fine grained route in a track layout diagram and define the lengths of the individual tracks.

9.2.2 Extend tool to support more modeling languages

Even though the UMC language does a great job at modeling especially distributed systems such as the one presented in this work, it could still be of interest to explore other types of modeling languages and approaches.
Another modeling language might for example have better performance characteristics.

9.2.3 Model checking of huge railway networks

As briefly discussed in the experiments chapter, the current model might not be feasible if applied to large railway networks with many trains.

A way to solve this problem, could perhaps be to parallelize the model checking such that every pair of intersecting routes are model checked in their own process. And If any of the processes find any of the properties to be false, the given property has failed for the model as a whole.

It seems intuitive for the current model, that if it is verified for any pair of two trains with intersecting routes that no collision will happen, then there shouldn't be any collisions for any pairs of three trains in the system either. And the same could be said for verifying for derailment. This seems intuitive with the current model, since it require routes to be fully reserved from start to end. However, checking for liveness or progress properties might not be as trivial to check in parallel and would perhaps require other methods of verification.

Chapter 10

Conclusion

In this thesis an engineering concept of a distributed railway control system has been devised. The concept uses a two-phase commit protocol as a means for trains to reserve routes, and trains are required to fully reserve their assigned route before being permitted to traverse it.

The engineering concept has been distilled into an abstract model in the object oriented modeling language UMC. The abstract model describes three classes, each encapsulating the generic behavior of a communication node representing either a train, point or linear track segment. The model has been defined such that only the trains are aware of their routes and the connection layout in the railway network. The model has been defined such that all the physical length of all the involved elements can be described.

To ease the generation of objects from the generic classes, and to compose concrete models that obey a set of constraints, a tool has been developed in the programming language F#. The tool enables a user to generate concrete models that are valid and has the best possible characteristics with regards to performance under model checking. The user can use the tool as a library and small DSL to define models in a script, or specify a set of routes to be used from an XML file containing both routes and a railway network layout.

Using the developed model generating tool, a set of models has been generated and analyzed by model checking of safety and progress properties, and the running times and number of explored states has been recorded and compared.

For the analyzed models, the model checking has indeed revealed that all the specified safety and progress properties are valid. However, the recorded performance characteristics resulting from the model checking, also shows that the model suffers from state space explosion and doesn't scale particularly well for multiple trains and very long routes.
As discussed, the solution to this problem of scalability, could be to change the model itself with the risk of ending up with a model that doesn't reflect the engineering concept. However, as also mentioned, a better solution might be parallelize the model checking in a divide and conquer fashion such that all pairs of trains are verified by themselves in parallel.

A division of a concrete model into sub-models might be a feasible solution for verifying safety properties, however more research must be performed in this regard, especially with regards to verifying progress and liveness properties which might be the biggest challenge.

All in all, it has been shown in this project how an engineering concept can be turned into an abstract model specification, and verified through model checking. The UMC language has shown to be a great tool for describing distributed systems, by enabling the user to essentially describe a set of state machines and the communication between them.
Model checking, is a very useful tool for creating program specifications that can be verified. However, model checking by itself is merely a way to develop an validated abstract specification, and the software implementation itself must in many cases still be developed in traditional ways based on the specification. And so, model checking does not completely eliminate the need for rigorous testing, strong type systems and so on.
What model checking most importantly brings to the table, is a verified specification for the system to be implemented, thus a good way to implement safe systems might be to use a combination of all mentioned methods together.

Appendix A

User Guide for the Model Generating Tool

This appendix chapter serves to give a brief introduction on how to use the model generating tool developed as part of the project.

First a section that covers model generation through the compiled program is presented, and after that a section which covers how to compose a model through a script. At short guide on how to model check the generated models is presented.

A.1 Model generation through the model generator tool

Generating a model with the model generator tool, requires either Linux with the open source .Net platform Mono or a Windows platform with .Net. Furthermore does the tool require that the user has an XML file of the type generated from the LRVT tool set mentioned in chapter 7.
In the following examples it will be assumed that the user is using the Mono platform to execute the tool, and the examples will use the sample.xml file from

Appendix G.

To generate a model, the user must first decide upon a set of routes which the model should evolve around. Examining the layout and routes listed in Appendix G, we decide to make a route that goes from b10 to t14 and another route that goes from t14 to t20.
Examining the route lists and markerboard definitions in the XML file, we conclude that a route from b10 to t14 can be composed of the two routes with ids 'r_1a' and 'r_4_', and furthermore that the route from t14 to t20 is defined by the route with id 'r_5a'.

An UMC model with the chosen routes can now be generated by executing following command.

```
mono MiniModelGenerator.exe sample.xml umc [r_1a,r_4_] [r_5a]
```

If everything goes right, the UMC model will be written out in the console.
To save it in a file, you can simply use the Linux pipeline operator '>' to pipe the resulting model into an output file.

A.2 Model generation using the scripting tools

There are multiple ways of generating a model from a script, and in fact the user has the full freedom of the F# core libraries at his disposal when creating scripts since the scripts are just normal F# scripts. This has indeed been exploited when generating multiple models for the experiments (see the experiments scripts in Appendix F).

However, the scripting tools also defines a very simple set of types that can be used like a simple DSL to specify a model, furthermore is it possible to load a layout from an XML file and use that in a script as well. These two approaches will be described here.

The scripts are executed using the F# interactive program which is bundled with all installations of F#, and is usually a program called *fsharpi*.

When creating a new script, it must have the file extension of '.fsx', and in the first lines of the script one must import the tools and types to be used.
A Prelude.fsx script has been created with the purpose of simplifying the process of creating new scripts. Loading the Prelude script will simply cause the required files to be loaded. (Note that the new script must be defined at the same

location as the Prelude script). Next the namespaces *InterlockingModel* and *ScriptingTools* must be imported. Thus a basic starting script looks as follows.

```
(* loading the prelude script *)
#load "Prelude.fsx"

(* importing required modules *)
open InterlockingModel
open ScriptingTools
```

The type like used in scripts to describe models, is presented below.

```
type SimpleTrackSegment = LLinear of name : string
                        | LPointFork of name : string *
                          PointPosition
                        | LPointStem of name : string
let (+>) (el1 : SimpleTrackSegment) (el2 : SimpleTrackSegment) =
    el1 , el2
type SimpleLayout = (SimpleTrackSegment * SimpleTrackSegment) list

type SimpleRouteElement =
    | RLinear of name : string * length : int
    | RPoint of name : string * position : PointPosition
type SimpleRoute = SimpleRouteElement list

type SimpleTrain =
    { id : string
    ; length : int
    ; route : SimpleRoute
    ; route_direction : RouteDirection }
type SimpleTrains = SimpleTrain list

type LayoutType = CustomLayout of SimpleLayout
                | XMLLayout of path : string

type SimpleModelArgs =
    { trains : SimpleTrain list
    ; layout : LayoutType
    ; show_stats : bool
    ; output_file : string option }
```

The script tools furthermore exposes two simple functions to be used by a user.

$$printRawLayout(path : string)$$

$$generateUMCModel(model\_args : SimpleModelArgs) : unit$$

The *printRawLayout* permits the user to explore the layout in an XML file, by printing out a simple representation of the layout in the same format as layouts can be defined by the user. The function *generateUMCModel* takes record of arguments as input, and based on the requirements specified in the record, the function will generate a model.

An example of using the *printRawLayout* in a file script.fsx is illustrated in the following listing (using the sample.xml file)

```
#load "Prelude.fsx"
open InterlockingModel
open ScriptingTools
let path = "sample.xml"
printRawLayout path
```

Executing this file with the F# interactive *fsharpi* program.

fsharpiscript.fsx

Prints following output in the console.

```
LLinear "b10"              <+> LLinear "t10"
LLinear "t10"              <+> LPointStem "t11"
LLinear "t12"              <+> LPointFork ("t13",Plus)
LLinear "t14"              <+> LLinear "b14"
LLinear "t20"              <+> LPointFork ("t13",Minus)
LPointStem "t11"           <+> LLinear "t10"
LPointStem "t13"           <+> LLinear "t14"
LPointFork ("t11",Plus)    <+> LLinear "t12"
LPointFork ("t11",Minus)   <+> LLinear "t20"
LPointFork ("t13",Plus)    <+> LLinear "t12"
LPointFork ("t13",Minus)   <+> LLinear "t20"
```

Which illustrates the connections between linear segments (*LLinear*), point stems (*LPointStem*) and point forks (*LPointFork*). It is now trivial to look at the mappings and choose a route in the layout.

Based on the presented layout, a user could for example decide to define a route from "b10" to "t14", going through the elements "b10", "t10", "t11", "t12", "t13" and "t14". And another route from "b14" to "t20", going through "b14", "t14", "t13" and "t20". These routes must now each be defined for a train and the user must decide the lengths of the train and segments.

The user can define a train using the record type *SimpleTrain*.

```
{ id = "1"
; length = 3
; route = [ RLinear(name = "b10", length = 3)
          ; RLinear(name = "t10", length = 3)
          ; RPoint(name = "t11", position = Plus)
          ; RLinear(name = "t12", length = 3)
          ; RPoint(name = "t13", position = Plus)
```

```
              ; RLinear(name = "t14", length = 3) ]
; route_direction = Up }
```

Here the user have chosen that the length of the modeled train should be a value of three and all its linear segments likewise, furthermore has the user decided that the route should go in the *Up* direction, since the route connects left to right in the described layout. Furthermore, has the user determined the required positionings of the points should both be *Plus*.

The user chooses to define the other train with a length of two and assigns it to the remaining route.

```
{ id = "2"
; length = 2
; route = [ RLinear(name = "b14", length = 2)
          ; RLinear(name = "t14", length = 2)
          ; RPoint(name = "t13", position = Minus)
          ; RLinear(name = "t20", length = 2) ]
; route_direction = Down }
```

The two train definitions can simply be represented together as a list, as follows.

```
let trains : SimpleTrains = [
  { id = "1" ; length = 3
  ; route = [ RLinear(name = "b10", length = 3)
            ; RLinear(name = "t10", length = 3)
            ; RPoint(name = "t11", position = Plus)
            ; RLinear(name = "t12", length = 3)
            ; RPoint(name = "t13", position = Plus)
            ; RLinear(name = "t14", length = 3) ]
  ; route_direction = Up }
  { id = "2"
  ; length = 2
  ; route = [ RLinear(name = "b14", length = 2)
            ; RLinear(name = "t14", length = 2)
            ; RPoint(name = "t13", position = Minus)
            ; RLinear(name = "t20", length = 2) ]
  ; route_direction = Down } ]
```

At last to produce an UMC model, the user can apply the trains together with a set of arguments in a record to the function *generateUMCModel*, as follows.

```
generateUMCModel {
    trains = trains
    layout = XMLLayout path
    show_stats = true
    output_file = Some "mymodel.txt" }
```

Here the input arguments to the functions specifies that the defined trains should be used, the layout should be extracted from an XML file previously specified by the path value. Furthermore does the inputs describe that a set of summary statistics must be shown together with the model that has been generated, and at last that the resulting model should be saved to the file “mymodel.txt”. If a None type was given instead of a Some type with a filename, the script would simply print the model to the console instead of saving it to a file.

Composing all the described script fragments into one script, results in the following script.

```
#load "Prelude.fsx"
open InterlockingModel
open ScriptingTools
let path = "sample.xml"
printRawLayout path
let trains : SimpleTrains = [
    { id = "1" ; length = 3
    ; route = [ RLinear(name = "b10", length = 3)
              ; RLinear(name = "t10", length = 3)
              ; RPoint(name = "t11", position = Plus)
              ; RLinear(name = "t12", length = 3)
              ; RPoint(name = "t13", position = Plus)
              ; RLinear(name = "t14", length = 3) ]
    ; route_direction = Up }
    { id = "2"
    ; length = 2
    ; route = [ RLinear(name = "b14", length = 2)
              ; RLinear(name = "t14", length = 2)
              ; RPoint(name = "t13", position = Minus)
              ; RLinear(name = "t20", length = 2) ]
    ; route_direction = Down } ]
generateUMCModel {
    trains = trains
    layout = XMLLayout path
    show_stats = true
    output_file = Some "mymodel.txt" }
```

And executing the script using the F# interactive program results in the following output.

```
LLinear "b10"          <+> LLinear "t10"
LLinear "t10"          <+> LPointStem "t11"
LLinear "t12"          <+> LPointFork ("t13",Plus)
LLinear "t14"          <+> LLinear "b14"
LLinear "t20"          <+> LPointFork ("t13",Minus)
LPointStem "t11"       <+> LLinear "t10"
LPointStem "t13"       <+> LLinear "t14"
LPointFork ("t11",Plus)  <+> LLinear "t12"
LPointFork ("t11",Minus) <+> LLinear "t20"
LPointFork ("t13",Plus)  <+> LLinear "t12"
LPointFork ("t13",Minus) <+> LLinear "t20"
model written to file mymodel.txt
```

(Note that it is not required to print the layout again)

The script could also have been defined such that a custom layout is defined in the script. In that case, the script could instead look as follows.

```
#load "Prelude.fsx"
(* importing required modules *)
open InterlockingModel
open ScriptingTools

let network : SimpleLayout =
    [ LLinear "1"            <+> LPointFork("1", Plus)
      LLinear "3"            <+> LPointFork("1", Minus)
      LPointStem "1"         <+> LPointStem "2"
      LPointFork("2", Plus)  <+> LLinear "2"
      LPointFork("2", Minus) <+> LLinear "4" ]

let trains : SimpleTrains =
    [ { id = "1"
      ; length = 2
      ; route = [ RLinear("1", 2)
                  RPoint("1", Plus)
                  RPoint("2", Plus)
                  RLinear("2", 2) ]
      ; route_direction = Up }
      { id = "2"
      ; length = 3
      ; route = [ RLinear("3", 3)
                  RPoint("1", Minus)
                  RPoint("2", Minus)
                  RLinear("4", 3) ]
      ; route_direction = Up } ]

generateUMCModel { trains = trains
                 ; layout = CustomLayout(network)
                 ; show_stats = true
                 ; output_file = Some "mymodel.txt" }
```

Notice that for this script the layout argument for the *generateUMCModel* function is of of type *CustomLayout*, and in the previous script the type *XMLLayout* was used.

The produced delta (-omitting the generic classes) of the model, is presented in following listing.

```
STATS:
{num_of_trains = 2;
 train_lengths = [2; 3];
 route_lengths = [4; 4];
 total_route_sub_segments = 14;
 total_linears = 4;
 total_points = 2;
 shared_points = 2;
```

```
 shared_linears = 0;}

MODEL:

... omitted ...

Objects
 train_1: Train(
route_segments => [linear_1,point_1,point_2,linear_2],
track_lengths => [2,1,1,2],
train_length => 2,
occupies => [linear_1,linear_1],
requested_point_positions => [null,True,True,null]);

train_2: Train(
route_segments => [linear_3,point_1,point_2,linear_4],
track_lengths => [3,1,1,3],
train_length => 3,
occupies => [linear_3,linear_3,linear_3],
requested_point_positions => [null,False,False,null]);

linear_1: Linear(train => train_1);

linear_2: Linear(train => null);

linear_3: Linear(train => train_2);

linear_4: Linear(train => null);

point_1: Point;

point_2: Point;
Abstractions {
State: inState(train_1.ARRIVED) -> train_1_arrived
State: inState(train_2.ARRIVED) -> train_2_arrived
State: point_1.train = null -> no_train_on_point_1
State: point_2.train = null -> no_train_on_point_2
State: inState(point_1.POSITIONING) -> positioning_point_1
State: inState(point_2.POSITIONING) -> positioning_point_2
State: point_1.current_position = True -> point_1_in_plus
State: point_2.current_position = True -> point_2_in_plus
State: point_1.current_position = True -> point_1_in_minus
State: point_2.current_position = True -> point_2_in_minus
State: train_1.occupies[0] /= train_2.occupies[0] and
train_1.occupies[0] /= train_2.occupies[1] and
train_1.occupies[0] /= train_2.occupies[2] and
train_1.occupies[1] /= train_2.occupies[0] and
train_1.occupies[1] /= train_2.occupies[1] and
train_1.occupies[1] /= train_2.occupies[2] ->
    trains_at_diff_positions
State: train_1.occupies[0] /= point_1 and
train_1.occupies[1] /= point_1 -> train_1_not_on_point_1
State: train_1.occupies[0] /= point_2 and
train_1.occupies[1] /= point_2 -> train_1_not_on_point_2
State: train_2.occupies[0] /= point_1 and
```

```
train_2.occupies[1] /= point_1 and
train_2.occupies[2] /= point_1 -> train_2_not_on_point_1
State: train_2.occupies[0] /= point_2 and
train_2.occupies[1] /= point_2 and
train_2.occupies[2] /= point_2 -> train_2_not_on_point_2
Action: lostevent -> discarded_message
}

-- safety property:
-- no incident
-- no trains occupy the same location node at the same time
AG (trains_at_diff_positions);

-- safety property:
-- no trains are located at any 'point' while it is changing its
    position
AG (positioning_point_1 implies no_train_on_point_1 and
positioning_point_2 implies no_train_on_point_2);

-- property to verify that all trains are correctly detected at
    points
AG ((not (train_1_not_on_point_1 and train_2_not_on_point_1)
    implies not no_train_on_point_1) and
(not (train_1_not_on_point_2 and train_2_not_on_point_2) implies
    not no_train_on_point_2));

-- progress property that specifies that
-- all trains has arrived at their destinations
EF AG (train_1_arrived and train_2_arrived);

-- no signal is ever lost in the system
AG not (EX {discarded_message} true);
```

As can be seen in the above listing, the stats are presented at the beginning followed by the generic model which has been omitted for the presented code, and thereafter comes the object instantiations, abstraction definitions and at last the properties for the generated model.

A.3 Model checking the generated models with the UMC web tool

The produced model from a script or from the tool program, can now be model checked with the UMC model checking tool.

This tool is exposed as a service at the site `http://fmt.isti.cnr.it/umc/V4.2/umc.html`.

When entering the site, choose 'Model Definition' from the menu at the left, and after that choose 'Edit a new Model' from the new menu point. An online editor is now displayed in the browser with an existing model skeleton. Delete the skeleton before proceeding. After deleting the skeleton open the generated model file and copy everything after the **MODEL:** except for the properties at the end of the file. Insert the model into the online editor and click the 'Load Current Model' button.

When the model is done loading, a new menu is presented and options for exploring the model are shown. To model check the generated properties of the generated model, simply copy the properties from the file containing the model, and click the 'Modelcheck L2TS ..' button. A small box opens at the bottom of the screen. Paste in the properties into the box and click the 'Check the Formula' button on the right side of the screen. Now all the properties will be model checked for the given model, and if they are all evaluated to be true, then the model has been verified with regards to the given properties. In the event that one or more properties evaluates to false, then the model has failed the verification. In this case it is possible to get a trace to the failing states by clicking the 'Explain the Result' button at the right side of the screen.

Appendix B

UMC BNF

This Appendix chapter presents the BNF grammar for the UMC modeling language.
The BNF is extracted from the UMC User Guide and presented here for completeness.[Maza]

{item} denotes 0 or more occurrences of the item
[item] denotes 0 or 1 occurrence of the item
“item” denotes a terminal character sequence
“item | item ” denotes indicates alternative items

```

Model ::= { Class } { Object }

Class ::= "class" ClassName "is"
            [ "Signals"
            Signal , {"," Signal} ]
            [ "Operations" Operation , {"," Operation} ]
            [ "Vars" Attribute {"," Attribute}]
            [ "State" "top" "=" Composite
            {"State" Statepath "=" State } ]
            [ "Transitions" {Transition} ]
            "end;" [ClassName]

Signal ::= SignalName ["(" ParamName [":" TypeName]
                      {"," ParamName [":" TypeName] }
                  ")" ]
```

```
Operation ::= OpName ["(" Name [":" TypeName]
                     {"," Name [":" TypeName] }
                      ")" ] [ ":" TypeName]

Attribute ::= AttrName [":" TypeName] [":=" StaticExpr ] ";"

State ::= Composite | Parallel

Composite ::= StateName { "," StateName}
                 ["Defers" Defer {"," Defer} ]

Parallel ::= Name { "/" Name}
                 ["Defers" Defer {"," Defer}]

StateName ::= Name | "final" | "initial"

Defer ::= EventName ["(" ParamName {"," ParamName } ")"]

Transition ::= Statepaths "-(" Trigger[Guard] ["/" Actions] ")->"
    Statepaths

Statepaths ::= Statepath | "(" Statepath {"," Statepath} ")"

Statepath ::= ["top."] Name { "." Name}

Trigger ::= "-" | EventName ["(" ParamName {"," ParamName } ")"]

Guard ::= "[" BoolBoolExpr "]"

Actions::= [ Stm {";" Stm} ]

Object ::= "Object" ObjName ":" ClassName [ Initializations ]

Initializations ::= "(" AttrName "=>" StaticExpr
                     { "," AttrName "=>" StaticExpr } ")"

-- action statements

Stm ::= Assignment
      | SignalSending
      | OperationCall
      | FunctionCall
      | ConditionalStm
      | LoopStm
      | VarDecl
      | ReturnStm
      | ExitStm

Assignment ::= TargetExpr ":=" Expr

SignalSending ::= ObjExpr "." SignalName ["(" Expr {"," Expr} ")"]

OperationCall ::= ObjExpr "." OpName ["(" Expr {"," Expr} ")"]
```

```
FunctionCall ::= TargetVar ":=" ObjExpr "." OpName ["(" Expr {"," Expr} ")"]

ConditionalStm ::= "if" BoolBoolExpr [ "then" ] "{" Actions "}" [ "else {" Actions "}" ]

LoopStm ::= "for" LoopIndex "in" IntExpr ".." IntExpr "{" Actions "}"

VarDecl ::= VarName ":" TypeName

ReturnStm ::= "return" ["(" Expr {"," Expr} ")"]

ExitStm ::= "exit"

TargetExpr ::= AttrName [ Selection ] | VarName [Selection]

Selection ::= "[" IntExpr "]"

-- names and expressions

Expr ::= "("Expr")" | BoolBoolExpr | IntExpr | ObjExpr | VectorExpr

BoolBoolExpr ::= BoolExpr {"and" BoolExpr}
               | BoolExpr {"or" BoolExpr}
               | "not" BoolExpr
               | BoolExpr

BoolExpr ::= "true" | "false"
           | AttrName [Selection]| VarName [Selection]
           | Expr "=" Expr
           | Expr "/=" Expr
           | IntExpr relop IntExpr

ObjExpr ::= "null" | AttrName [Selection]| VarName [Selection]
          | ObjName | "self" | "this"

IntExpr ::= Number | AttrName [Selection] | VarName [Selection]
          | (Intexpr intop IntExpr ")" | VectorExpr ".head"

VectorExpr ::= "[]" | AttrName | VarName | VectorExpr "+" VectorExpr
             | VectorExpr ".tail"

StaticExpr ::= Number | ObjName | "null" | "self" | "this"

relop ::= ">" | ">=" | "<" | "<="

intop ::= "+" | "-" | "*" | "/" | "mod"

TypeName ::= "int" | "bool" | "obj" | ClassName
             "int[]" | "bool[]" | "obj[]" | ClassName"[]"
```

Appendix C

Generic UMC Model

This chapter contains the source code for the three generic classes that together defines the modeled concept.

```
Class Train is
Signals: ok, no;

Vars:
requested_point_positions : bool[];
train_length : int = 2; -- how many track segments does the train occupy
route_segments : obj[];
route_index : int := 0; -- current location on the route
occupies : obj[]; -- the tc and pt objects which the train currently occupies
front_advancement_count : int; -- a variable for keeping track of the trains front
    advancement over a track
track_lengths : int[]; -- same number of elements as route_segments

State Top = READY, WAIT_OK, MOVEMENT, ARRIVED

Transitions:
-- send out initial reservation request to the first node on route
READY -> WAIT_OK {
  - /
  route_segments[0].req(self, 0, route_segments,
      requested_point_positions);
}

-- when the train reservation is rejected we just keep cycling between
    WAIT_OK and READY
```

```
WAIT_OK -> READY { no }

-- train receives acknowledgment that the route has been reserved successfully
-- the front_advancement_count variable is initialized to reflect the trains front location on the track
WAIT_OK -> MOVEMENT { ok / front_advancement_count := train_length - 1; }

MOVEMENT -> MOVEMENT {
-
[not (route_index = route_segments.length - 1 and
  track_lengths[route_index] - 1 = front_advancement_count)] / -- at end of track
  at_end_of_track: bool := track_lengths[route_index] - 1 = front_advancement_count; -- determine if we have reached the end of the current track
  if at_end_of_track = true then { -- the train has reached the end of its current track
    front_advancement_count := 0;
    if route_index < route_segments.length - 1 then { -- the route_index is not the last
      -- train enters next track
      route_index := route_index + 1;
      route_segments[route_index].sensorOn(self); -- the next track detects the train
    };
  } else {
    front_advancement_count := front_advancement_count + 1;
  };
  -- update the occupies array
  rear: obj := occupies.head;
  next_rear: obj := occupies.tail.head;
  occupies := occupies.tail + [route_segments[route_index]];
  if rear != next_rear then { -- determine if the rear of the train has left a track
    rear.sensorOff(self); -- the past track detects that the train does not occupy it anymore
  };
}

MOVEMENT -> ARRIVED {
-
[route_index = route_segments.length - 1 and -- at last track segment of route
  track_lengths[route_index] - 1 = front_advancement_count] -- at end of track
}
end Train


Class Linear is
Signals:
req(sender: obj, route_index: int, route_elements: obj[], requested_point_positions: bool[]);
ack(sender: obj);
nack(sender: obj);
```

```
commit(sender: obj);
agree(sender: obj);
disagree(sender: obj);

Operations:
sensorOn(sender: obj);
sensorOff(sender: obj);

Vars:
next: obj;
prev: obj;
train: obj := null;

State Top = NON_RESERVED, WAIT_ACK, WAIT_COMMIT, WAIT_AGREE,
    RESERVED, TRAIN_IN_TRANSITION

Transitions
-- first node receive request
NON_RESERVED -> WAIT_ACK {
  req(sender, route_index, route_elements,
      requested_point_positions)
  [route_index = 0 and sender = train and route_elements.length >
      0] /
  prev := null;
  next := route_elements[1];
  next.req(self, 1, route_elements, requested_point_positions);
}

-- intermediate node receive request
NON_RESERVED -> WAIT_ACK {
  req(sender, route_index, route_elements,
      requested_point_positions)
  [train = null and (route_index > 0 and route_index+1 <
      route_elements.length)] /
  prev := route_elements[route_index - 1];
  next := route_elements[route_index + 1];
  next.req(self, route_index + 1, route_elements,
      requested_point_positions);
}

-- initial reservation request for last node
-- starts ack phase
NON_RESERVED -> WAIT_COMMIT {
  req(sender, route_index, route_elements,
      requested_point_positions)
  [train = null and route_elements.length = route_index+1] /
  prev := route_elements[route_index - 1];
  next := null;
  prev.ack(self);
}

-- intermediate node receive ack
WAIT_ACK -> WAIT_COMMIT {
  ack(sender)
  [prev /= null] /
```

```
    prev.ack(self);
  }

  -- first node receive ack
  -- and starts commit phase
  WAIT_ACK -> WAIT_AGREE {
    ack(sender)
    [prev = null] /
    next.commit(self);
  }

  -- intermediate node receives commit
  WAIT_COMMIT -> WAIT_AGREE {
    commit(sender)
    [next /= null] /
    next.commit(self);
  }

  -- last node receive commit
  -- and starts agree phase
  WAIT_COMMIT -> RESERVED {
    commit(sender)
    [next = null] /
    prev.agree(self);
  }

  -- intermediate node receive agree
  WAIT_AGREE -> RESERVED {
    agree(sender)
    [prev /= null] /
    prev.agree(self);
  }

  -- first node receive agree
  -- and sends ok to the train
  WAIT_AGREE -> TRAIN_IN_TRANSITION {
    agree(sender)
    [prev = null and train /= null] /
    train.ok;
  }

  -- train moves onto current node
  RESERVED -> TRAIN_IN_TRANSITION {
    sensorOn(sender) /
    train := sender;
  }

  -- sequential release
  -- reset train
  TRAIN_IN_TRANSITION -> NON_RESERVED {
    sensorOff(sender) /
    train := null;
  }

  -- nack received
```

```
-- forwards and goes into non-reserved
WAIT_ACK -> NON_RESERVED {
   nack(sender) /
   if  prev = null  then { -- is first node on itinerary
      train.no
   } else { -- is not first node
      prev.nack(self)
   };
}

-- disagree received
-- forwards and goes into non-reserved
WAIT_COMMIT -> NON_RESERVED {
   disagree(sender) /
   if next /= null  then { -- not last node on itinerary
      next.disagree(self)
   };
}

-- disagree received
-- forwards and goes into non-reserved
WAIT_AGREE -> NON_RESERVED {
   disagree(sender) /
   if prev /= null then { -- not first node on itinerary
      prev.disagree(self)
   } else { -- is first node
      train.no
   };
}

-- disagree received
-- forwards (if there is someone to forward to) and goes into non-reserved
RESERVED -> NON_RESERVED {
   disagree(sender) /
   if next /= null then { -- not last node on itinerary
      next.disagree(self)
   };
}

-- reservation request received
-- however a train is already on the track, so a nack is returned to sender
NON_RESERVED -> NON_RESERVED {
   req(sender, route_index, route_elements,
        requested_point_positions)
   [train /= null and sender /= train] /
   sender.nack(self);
}

-- reservation request received
-- however, the node is already in wait-ack, so it returns a nack to sender
WAIT_ACK -> WAIT_ACK {
   req(sender, route_index, route_elements,
        requested_point_positions) /
   sender.nack(self);
}
```

```

-- reservation request received
-- however, the node is already in wait-commit, so it returns a nack to sender
WAIT_COMMIT -> WAIT_COMMIT {
   req(sender, route_index, route_elements,
         requested_point_positions) /
   sender.nack(self);
}

-- reservation request received
-- however, the node is already in wait-agree, so it returns a nack to sender
WAIT_AGREE -> WAIT_AGREE {
   req(sender, route_index, route_elements,
         requested_point_positions) /
   sender.nack(self);
}

-- reservation request received
-- however, the node is already reserved, so it returns a nack to sender
RESERVED -> RESERVED {
   req(sender, route_index, route_elements,
         requested_point_positions) /
   sender.nack(self);
}

-- reservation request received
-- however, a train is in transition on the node, so it returns a nack to sender
TRAIN_IN_TRANSITION -> TRAIN_IN_TRANSITION {
   req(sender, route_index, route_elements,
         requested_point_positions) /
   sender.nack(self);
}
end Linear


-- points are always intermediate nodes
-- so we don't need to check if they are first or last in the guards
Class Point is
Signals:
req(sender: obj, route_index: int, route_elements: obj[],
      requested_point_positions: bool[]);
ack(sender: obj);
nack(sender: obj);
commit(sender: obj);
agree(sender: obj);
disagree(sender: obj);

Operations:
sensorOn(sender: obj);
sensorOff(sender: obj);

Vars:
next: obj;
prev: obj;
requested_position: bool;
```

```
current_position: bool := True;
train: obj := null;

State Top = NON_RESERVED, WAIT_ACK, WAIT_COMMIT, WAIT_AGREE,
    POSITIONING, RESERVED, TRAIN_IN_TRANSITION, MALFUNCTION

Transitions:
-- initial reservation request
NON_RESERVED -> WAIT_ACK {
  req(sender, route_index, route_elements,
      requested_point_positions) /
  prev := route_elements[route_index - 1];
  next := route_elements[route_index + 1];
  requested_position := requested_point_positions[route_index];
  next.req(self, route_index + 1, route_elements,
      requested_point_positions);
}

-- receiving and forwarding ack
WAIT_ACK -> WAIT_COMMIT {
  ack(sender) /
  prev.ack(self);
}

-- receiving and forwarding commit
WAIT_COMMIT -> WAIT_AGREE {
  commit(sender) /
  next.commit(self);
}

-- if the point is positioned as required for the given route reservation
-- receiving and forwarding agree
WAIT_AGREE -> RESERVED {
  agree(sender)
  [current_position = requested_position]/
  prev.agree(self);
}

-- if the point is not positioned as required for the given route
-- goes into positioning state
WAIT_AGREE -> POSITIONING {
  agree(sender)
  [current_position /= requested_position] /
  -
}

-- successfully performing positioning
POSITIONING -> RESERVED {
  - /
  current_position := not current_position;
  prev.agree(self);
}

-- simulating sudden malfunction of positioning system
-- and sending disagrees to neighbor nodes
```

```
POSITIONING -> MALFUNCTION {
  - /
  prev.disagree(self);
  next.disagree(self);
}

-- train moves onto current node
RESERVED -> TRAIN_IN_TRANSITION {
  sensorOn(sender) /
  train := sender;
}

-- sequential release
-- reset all train
TRAIN_IN_TRANSITION -> NON_RESERVED {
  sensorOff(sender) /
  -- [sender = train] /
  train := null;
}

-- nack received and forwarded
WAIT_ACK -> NON_RESERVED {
  nack(sender) /
  prev.nack(self);
}

-- disagree received and forwarded
WAIT_COMMIT -> NON_RESERVED {
  disagree(sender) /
  next.disagree(self);
}

-- disagree received and forwarded
WAIT_AGREE -> NON_RESERVED {
  disagree(sender) /
  prev.disagree(self);
}

-- disagree received and forwarded
POSITIONING -> NON_RESERVED {
  disagree(sender) /
  next.disagree(self);
}

-- disagree received and forwarded
RESERVED -> NON_RESERVED {
  disagree(sender) /
  next.disagree(self);
}

-- reservation request received
-- however, the node is already in wait-ack, so it returns a nack to sender
WAIT_ACK -> WAIT_ACK {
  req(sender, route_index, route_elements,
      requested_point_positions) /
```

```
    sender.nack(self);
  }

  -- reservation request received
  -- however, the node is malfunctioning
  MALFUNCTION -> MALFUNCTION {
    req(sender, route_index, route_elements,
        requested_point_positions) /
    sender.nack(self);
  }

  -- reservation request received
  -- however, the node is already in wait-commit state and returns nack to the sender
  WAIT_COMMIT -> WAIT_COMMIT {
    req(sender, route_index, route_elements, requested_position) /
    sender.nack(self);
  }

  -- reservation request received
  -- however, the node is already in wait-agree state and returns nack to the sender
  WAIT_AGREE -> WAIT_AGREE {
    req(sender, route_index, route_elements, requested_position) /
    sender.nack(self);
  }

  -- reservation request received
  -- however, the node is already in positioning state and returns nack to the sender
  POSITIONING -> POSITIONING {
    req(sender, route_index, route_elements, requested_position) /
    sender.nack(self);
  }

  -- reservation request received
  -- however, the node is already reserved and returns nack to the sender
  RESERVED -> RESERVED {
    req(sender, route_index, route_elements, requested_position) /
    sender.nack(self);
  }

  -- reservation request received
  -- however, the node is occupied by a train and returns nack to the sender
  TRAIN_IN_TRANSITION -> TRAIN_IN_TRANSITION {
    req(sender, route_index, route_elements, requested_position) /
    sender.nack(self);
  }
end Point
```

Appendix D

UMC model delta

Example of the appended delta part of a concrete UMC model generated by the tool.
In the generated model, the presented code would be appended to the code from the previous appendix chapter, thus the code presented here can be seen as a delta of a generated model.

```
Objects
  train_0: Train(
    route_segments =>
        [linear_b10,linear_t10,point_t11,linear_t12,point_t13,linear_t14],
    track_lengths => [2,2,1,2,1,2],
    train_length => 2,
    occupies => [linear_b10,linear_b10],
    requested_point_positions => [null,null,True,null,True,null]);

  train_1: Train(
    route_segments => [linear_b14,linear_t14,point_t13,linear_t20],
    track_lengths => [2,2,1,2],
    train_length => 2,
    occupies => [linear_b14,linear_b14],
    requested_point_positions => [null,null,False,null]);

  linear_b10: Linear(train => train_0);

  linear_b14: Linear(train => train_1);

  linear_t10: Linear(train => null);
```

```
  linear_t12: Linear(train => null);

  linear_t14: Linear(train => null);

  linear_t20: Linear(train => null);

  point_t11: Point;

  point_t13: Point;

Abstractions {
  State: inState(train_0.ARRIVED) -> train_0_arrived
  State: inState(train_1.ARRIVED) -> train_1_arrived

  State: point_t11.train = null -> no_train_on_point_t11
  State: point_t13.train = null -> no_train_on_point_t13

  State: inState(point_t11.POSITIONING) -> positioning_point_t11
  State: inState(point_t13.POSITIONING) -> positioning_point_t13

  State: point_t11.current_position = True -> point_t11_in_plus
  State: point_t13.current_position = True -> point_t13_in_plus
  State: point_t11.current_position = False -> point_t11_in_minus
  State: point_t13.current_position = False -> point_t13_in_minus

  State: inState(point_t11.MALFUNCTION) -> point_t11_malfunction
  State: inState(point_t13.MALFUNCTION) -> point_t13_malfunction

  State: train_0.occupies[0] /= train_1.occupies[0] and
    train_0.occupies[0] /= train_1.occupies[1] and
    train_0.occupies[1] /= train_1.occupies[0] and
    train_0.occupies[1] /= train_1.occupies[1] ->
        trains_at_diff_positions

  State: train_0.occupies[0] /= point_t11 and
    train_0.occupies[1] /= point_t11 -> train_0_not_on_point_t11

  State: train_0.occupies[0] /= point_t13 and
    train_0.occupies[1] /= point_t13 -> train_0_not_on_point_t13

  State: train_1.occupies[0] /= point_t11 and
    train_1.occupies[1] /= point_t11 -> train_1_not_on_point_t11

  State: train_1.occupies[0] /= point_t13 and
    train_1.occupies[1] /= point_t13 -> train_1_not_on_point_t13

  Action: lostevent -> discarded_message
}

-- safety property:
-- no incident
-- no trains occupy the same location node at the same time
AG (trains_at_diff_positions);
```

```
-- safety property:
-- no trains are located at any 'point' while it is changing its position
AG (positioning_point_t11 implies no_train_on_point_t11 and
    positioning_point_t13 implies no_train_on_point_t13);


-- property to verify that all trains are correctly detected at points
AG ((not (train_0_not_on_point_t11 and train_1_not_on_point_t11)
    implies not no_train_on_point_t11) and (not
    (train_0_not_on_point_t13 and train_1_not_on_point_t13)
    implies not no_train_on_point_t13));

-- progress property that specifies that
-- all trains has arrived at their destinations
EF AG (train_0_arrived and train_1_arrived);

-- property that specifies that
-- there does not exist a final state where at least one train has not arrived
-- and in all states leading to this final state, no points have malfunctioned
not E[not (point_t11_malfunction or point_t13_malfunction) U
    (final and not (train_0_arrived and train_1_arrived))];

-- no signal is ever lost in the system
AG not (EX {discarded_message} true);
```

Appendix E

Tool Source Code

This chapter lists the source code for the implemented model generator tool and related files. All the code has been developed on a linux platform using emacs as editor and executed with Mono, however F# is primarily a .Net language, and therefore it should be possible to run the code on a windows platform with the latest Visual Studio installation. (even though this hasn't been tested)

The code also defines a small set of unit tests defined using the testing library XUnit[mad] and a few property based tests using the FsCheck[maa] library. The tests are listed last in this chapter in its own section.

E.1 Compiler version and third party packages

As mentioned, the tool has been developed on a linux platform using the open source .Net platform Mono and an fsharp compiler targeted at Mono.
The tool has been developed using the following versions.

- Mono JIT compiler version 4.6.1
- F# 4.0 (Open Source Edition)

The tool utilizes a set of third party libraries which have been installed through the package .Net manager Nuget[mac]. Most notably, the package FSharp.Data[mab] has been used as it contains the F# type provider library for parsing XML files.

Following is a listing of the packages and version numbers used for this project.

- FsCheck (2.6.2)
- FsCheck.Xunit (2.6.2)
- FSharp.Data (2.3.2)
- xunit (2.1.0)
- FSharp.Core (4.0.0.1)

E.2 Auxiliary dependency files

E.2.1 Project files and compile order

F# source code files must be compiled in a certain order, this order is usually defined in a project file with the suffix 'fsproj', however these files also contains a lot of noise and irrelevant information. Here a snippet of the so called Item-Groups are listed. This information essentially contains the compile order of the source code files where the files must be compiled in the listed order.

The source code project is represented in the following listing.

```
<ItemGroup>
  <Compile Include="Utils.fs" />
  <Compile Include="InterlockingModel.fs" />
  <Compile Include="UMCTrainClass.fs" />
  <Compile Include="UMCLinearClass.fs" />
  <Compile Include="UMCPointClass.fs" />
  <Compile Include="UMC.fs" />
  <Compile Include="XMLExtraction.fs" />
  <Compile Include="ScriptTools.fs" />
  <Compile Include="MiniModelGenerator.fs" />
  <None Include="App.config" />
</ItemGroup>
```

The test project is represented by following listing.

```
<ItemGroup>
  <Compile Include="Tests.Utils.fs" />
  <Compile Include="Tests.InterlockingModel.fs" />
</ItemGroup>
```

E.2.2 Sample XML file used to bootstrap the type-provider library

See Appendix G

E.3 Source code

E.3.1 Utils.fs

```
module Utils

open System

/// Generate all unique products with values from a given list
/// where no product contains a pair of identical values
let rec uniqueProducts (xs : 'a list) : ('a * 'a) seq = seq {
    match xs with
    | x::xs ->
        for y in xs do
            yield x,y
        yield! uniqueProducts xs
    | _ -> () }

/// Generate all cross products of two sequences
let crossProductOfLists xs ys = seq {
    for x in xs do
        for y in ys do
            yield x,y }

/// Generate 'n choose k' combinations of values from list xs
/// where n is length of xs and each combination is of length k
let combinations (k : int) (xs : 'a list) : ('a list) seq =
    let rec loop (k : int) (xs : 'a list) : ('a list) seq = seq {
        match xs with
        | [] -> ()
        | xs when k = 1 -> for x in xs do yield [x]
        | x::xs ->
            let k' = k - 1
            for ys in loop k' xs do
                yield x :: ys
            yield! loop k xs }
```

```
    loop k xs
    |> Seq.filter (List.length >> (=)k)

// In F# 4.1 the Result type will be in the core library with exactly the same definition
type Result<'success,'error> = Ok of 'success
                             | Error of 'error
    with
    static member map (f : 'a -> 'b) (x : Result<'a,'error>) : Result<'b, 'error> =
        match x with
        | Ok x -> Ok (f x)
        | Error err -> Error err
    /// Binding value in result to parameter of a continuation function
    static member bind (x : Result<'a, 'error>) (continueationFun : 'a -> Result<'b, 'error>)
        : Result<'b, 'error> =
        match x with
        | Error err -> Error err
        | Ok x -> continueationFun x


// Computation-Expression definition for the Result type.
// The defined Result type has same functionality as
// the more commonly known Either monad defined in Haskell and other languages.
// The reason why it's called Result in this code, is because
// (soon to-be-released) F# 4.1 will have a Result type defined in its core library.
type ResultBuilder() =
    member this.Bind (x, f) = Result<_,_>.bind x f
    member this.Return (x : 'a) : Result<'a,'error> = Ok x
let resultFlow = new ResultBuilder()

let private traverseResults (f : 'a -> Result<'b, 'error>) (xs : 'a list)
    : Result<'b list, 'error> =
    let folderFun (head : 'a) (tail : Result<'b list, 'error>) : Result<'b list, 'error> =
        resultFlow { let! (h : 'b) = f head
                     let! (t : 'b list) = tail
                     return h :: t }
    let initial_val = Ok []
    // folding from right to left in order to maintain original order of the input list (xs)
    List.foldBack folderFun xs initial_val

let private sequenceResults (xs : Result<'a, 'error> list) : Result<'a list, 'error> =
    traverseResults id xs

let private reduceResults (f : 'a -> 'a -> Result<'a,'error>) (xs : Result<'a, 'error> seq)
    : Result<'a,'error> =
```

```
    let reducerFun = fun x y -> resultFlow {
        let! x' = x
        let! y' = y
        let! combined = f x' y'
        return combined }
    xs |> Seq.reduce reducerFun

let private foldResults
    (f : 'a -> 'b -> Result<'b,'error>) (initial : Result<'b,'error>) (xs : 'a list)
    : Result<'b,'error> =
    let folderFun (state : Result<'b,'error>) (x : 'a) : Result<'b, 'error> =
        Result<_,_>.bind state (f x)
    Seq.fold folderFun initial xs

// Extending the Result type with a set of functions for handling Result types
type Result with
    /// Applies a function ('a -> Result) on all elements in a list
    /// and lifts the list of results to a Result containing a list
    static member traverse f xs = traverseResults f xs

    /// Lifts a list of Results to a Result with a list
    static member sequence xs = traverseResults id xs

    /// Reduces a list of Result<a,..> to a Result<a,..>
    /// using a function a -> a -> Result<a,..>
    static member reduce f xs = reduceResults f xs

    /// Fold over a list of 'a with an initial value of 'b
    /// and a function 'a -> 'b -> Result<'b,..>
    static member fold f initial xs = foldResults f initial xs

    /// A bind operator for conveniently chaining together
    /// functions that produce Result types from non-Result types
    static member (>>=) (a: Result<'a, 'error>, f : 'a -> Result<'b, 'error>) : Result<'b, 'error> =
        Result<_,_>.bind a f


// computation expression definition for the Option type (called Maybe in other languages)
type MaybeBuilder() =
    member this.Bind (m : 'a option, f : 'a -> 'b option) : 'b option =
        Option.bind f m
    member this.Return (x : 'a) : 'a option = Some x
let maybe = new MaybeBuilder()
```

E.3.2 InterlockingModel.fs

```
namespace InterlockingModel

```

```
open Utils

[<AutoOpen>]
module TypeDefinitions =
    type TrainId = TrainId of string
    type TrainIds = TrainId list

    type LinearId = LinearId of string

    type PointId = PointId of string

    type PointPosition = Plus | Minus

    type RouteSegment = LinearRouteSegment of LinearId * length :
        int
                      | PointRouteSegment of PointId *
                        required_position : PointPosition
    type RouteSegments = RouteSegment list

    type RouteDirection = Up | Down

    type Route = Route of RouteSegments * RouteDirection
    type Routes = Route list

    type Train = { id : TrainId
                   route : Route
                   length : int }
    type Trains = Train list

    type Linear = { id : LinearId
                    train : Train option }
    type Linears = Linear list

    type Point = { id : PointId
                   position : PointPosition }
    type Points = Point list

    /// Elements for composing a railway network layout
    type LayoutSegment = LinearLayoutSegment of id : string
                       | PointStemLayoutSegment of id : string
                       | PointForkLayoutSegment of id : string *
                         position : PointPosition

    type RailwayNetworkLayout = Map<LayoutSegment, LayoutSegment>

    /// A collection of objects to be instantiated in the end model
    type ModelObjects = { trains  : Map<TrainId, Train>
                          linears : Map<LinearId, Linear>
                          points  : Map<PointId, Point> }
    type ValidatedModelObjects = Validated of ModelObjects
    type ModelGeneratorFunction = ValidatedModelObjects -> string


    (* Extending the types with field access functions *)
    type TrainId with
```

```
        static member value : TrainId -> string = fun (TrainId v)
            -> v

    type LinearId with
        static member value : LinearId -> string = fun (LinearId
            v) -> v

    type PointId with
        static member value : PointId -> string = fun (PointId v)
            -> v

    type RouteSegment with
        static member length : RouteSegment -> int = function
            | LinearRouteSegment(_,len) -> len
            // All points has been simplified to have a length of
                one
            | PointRouteSegment _ -> 1

    type Route with
        static member segments : Route -> RouteSegments =
            fun (Route (segments,_)) -> segments
        static member direction : Route -> RouteDirection =
            fun (Route (_,dir)) -> dir

    type ModelObjects with
        member this.trainList : Trains =
            this.trains |> Map.toList |> List.map snd
        member this.pointList : Points =
            this.points |> Map.toList |> List.map snd
        member this.linearList : Linears =
            this.linears |> Map.toList |> List.map snd

    (* Creating static access functions for record fields,
        with the functions having the same name as its field *)

    [<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
    module Train =
        let id : Train -> TrainId = fun t -> t.id
        let route : Train -> Route = fun t -> t.route
        let length : Train -> int = fun t -> t.length

    [<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
    module Linear =
        let id : Linear -> LinearId = fun tc -> tc.id
        let train : Linear -> Train option = fun tc -> tc.train

    [<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
    module Point =
        let id : Point -> PointId = fun p -> p.id
        let position : Point -> PointPosition = fun p -> p.position

    [<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
    module LayoutElement =
        let id : LayoutSegment -> string = function
            | LinearLayoutSegment id -> id
```

```
                | PointStemLayoutSegment id -> id
                | PointForkLayoutSegment (id,_) -> id

    /// Helper type used in the validation process.
    /// The type is used where routes or route segment lengths need validation
    /// and can represent a simple success or an error case with a description
    type SuccessResult = Result<unit,string>
    /// Infix operator for combining simple unit results
    let (&&&) (a : SuccessResult) (b : SuccessResult) : SuccessResult =
        match a with
        | Ok () -> b
        | Error _ -> a

/// Functions for constructing routes
[<AutoOpen>]
module RouteConstruction =
    /// Stitch two route fragments together to one route
    let stitchRoutePair (route1 : Route) (route2 : Route) : Result<Route, string> =
        let stitch : RouteSegments -> RouteSegments -> RouteDirection -> Route =
            fun route1_elements route2_elements direction ->
            let stitched = List.concat [route1_elements; List.tail route2_elements]
            Route (stitched, direction)

        let must_have_same_direction : Route -> Route -> string =
            sprintf """
            route1 and route2 must have same direction
            route1: %A
            route2: %A
            """
        let (|Diff_directions|_|) (r1 : Route, r2 : Route) =
            let (Route(_, dir1)) = r1
            let (Route(_, dir2)) = r2
            if dir1 <> dir2
            then Some(must_have_same_direction r1 r2)
            else None

        let r1_must_end_where_r2_starts : Route -> Route -> string =
            sprintf "route1 must end where route2 starts\nroute1: %A\nroute2: %A"
        let (|NoCommonStichPoint|_|) (r1 : Route, r2 : Route) =
            let (Route(r1_elements, _)) = r1
            let (Route(r2_elements, _)) = r2
            match r1_elements, r2_elements with
            | r1_elements, (r2_first::_)
              when r1_elements <> []
              && List.last r1_elements = r2_first -> None
            | _ -> Some (r1_must_end_where_r2_starts r1 r2)

```

```
        match route1, route2 with
        | Diff_directions(error_msg) -> Error error_msg
        | NoCommonStichPoint(error_msg) -> Error error_msg
        | Route (route1_elements,dir), Route (route2_elements,_) ->
            Ok (stitch route1_elements route2_elements dir)

    let stitchRoutes (routes : Routes) : Result<Route, string> =
        routes
        |> Seq.map Ok
        |> Result<_,_>.reduce stitchRoutePair

/// Functions for checking if routes are valid together in a layout
module RouteValidation =
    /// Checks that a route is has a linear as start and end
    let private routeHasLinearStartAndEnd (route : Route) : SuccessResult =
        let route_segments = Route.segments route
        match Seq.head route_segments, Seq.last route_segments with
        | LinearRouteSegment _, LinearRouteSegment _ -> Ok ()
        | _ -> route_segments
               |> sprintf "route is must have a linear as start and end\n%A"
               |> Error

    /// Checks that a given route is legal in a given layout
    let private routeIsValidInLayout (layout : RailwayNetworkLayout) (route : Route)
                                     : SuccessResult =
        // helper function for looking up an element in the layout
        let tryFind = fun from_element ->
            Map.tryFind from_element layout

        // helper function that evaluates if two segments are connected
        let areConnected : LayoutSegment -> LayoutSegment -> bool =
            fun from_element to_element ->
            match tryFind from_element with
            | Some(to_element') when to_element' = to_element -> true
            | _ -> false

        let linearToLinear : string -> string -> bool = fun from_id to_id ->
            let from_linear, to_linear = LinearLayoutSegment from_id, LinearLayoutSegment to_id
            areConnected from_linear to_linear

        let linearToPoint : string -> string -> PointPosition -> bool =
            fun from_id to_id pos ->
            let from_linear = LinearLayoutSegment from_id
            let to_stem = PointStemLayoutSegment to_id
            let to_fork = PointForkLayoutSegment (to_id, pos)
            areConnected from_linear to_stem || areConnected from_linear to_fork
```

```

    let pointToLinear : string -> PointPosition -> string -> bool =
        fun from_id pos to_id ->
        let from_stem = PointStemLayoutSegment from_id
        let from_fork = PointForkLayoutSegment(from_id, pos)
        let to_linear = LinearLayoutSegment to_id
        // Since the Route-elements doesn't specify stem or fork
        // -we have to try both to see if they exists in the layout
        areConnected from_stem to_linear || areConnected from_fork to_linear

    let pointToPoint : string -> PointPosition -> string -> PointPosition -> bool =
        fun from_id from_pos to_id to_pos ->
        let from_stem = PointStemLayoutSegment from_id
        let from_fork = PointForkLayoutSegment(from_id, from_pos)
        let to_stem = PointStemLayoutSegment to_id
        let to_fork = PointForkLayoutSegment(to_id, to_pos)
        areConnected from_stem to_stem
        || areConnected from_stem to_fork
        || areConnected from_fork to_stem
        || areConnected from_fork to_fork

    /// checks that a connection between two given elements exist in the current layout
    /// basically we look up in the layout map to see if there exist a mapping from
    /// a route element to the other given route element
    let connectionExistInLayout : (RouteSegment * RouteSegment) -> bool =
        function
        | LinearRouteSegment(LinearId from_id, _),
          LinearRouteSegment(LinearId to_id, _) ->
              linearToLinear from_id to_id
        | LinearRouteSegment(LinearId from_id, _),
          PointRouteSegment(PointId to_id, pos) ->
            linearToPoint from_id to_id pos
        | PointRouteSegment(PointId from_id, pos),
          LinearRouteSegment(LinearId to_id, _) ->
            pointToLinear from_id pos to_id
        | PointRouteSegment(PointId from_id, from_pos),
          PointRouteSegment(PointId to_id, to_pos) ->
              pointToPoint from_id from_pos to_id to_pos

    let verifySegmentPair : RouteSegment [] -> SuccessResult =
        fun segment_pair ->
        let segment1, segment2 = segment_pair.[0], segment_pair.[1]
        match connectionExistInLayout(segment1, segment2) with
        | true  -> Ok ()
        | false ->
```

```
                sprintf "no connection between tracks [ %A -> %A ]" segment1 segment2
                |> Error

        match Route.direction route with
        | Up -> Route.segments route
        | Down -> List.rev (Route.segments route)
        |> Seq.windowed 2
        |> Seq.map verifySegmentPair
        |> Seq.reduce (&&&)

    /// verifying that two given routes can be used together in a model without obvious deadlock
    let private noObviousConflict (Route (route1_segments, _)) (Route (route2_segments, _))
                                  : SuccessResult =
        let diffStart : RouteSegments -> RouteSegments -> SuccessResult =
            fun r1_segments r2_segments ->
            let diff_start = Seq.head r1_segments <> Seq.head r2_segments
            if diff_start then Ok ()
            else Error "The given routes start at the same place"

        let diffEnd : RouteSegments -> RouteSegments -> SuccessResult =
            fun r1_segments r2_segments ->
            let diff_end = Seq.last r1_segments <> Seq.last r2_segments
            if diff_end then Ok ()
            else Error "The given routes have the same destination"

        let validStartAndEnd : RouteSegments -> RouteSegments -> SuccessResult =
            fun r1_segments r2_segments ->
            let route1_start = Seq.head r1_segments
            let route2_start = Seq.head r2_segments
            let route1_end   = Seq.last r1_segments
            let route2_end   = Seq.last r2_segments
            let diff_start_end =
                not(route1_start = route2_end && route2_start = route1_end)
            if diff_start_end then Ok ()
            else Error "The given routes start and end in exact opposite locations"

        diffStart route1_segments route2_segments
        &&& diffEnd route1_segments route2_segments
        &&& validStartAndEnd route1_segments route2_segments

    let verifyRoutes (layout: RailwayNetworkLayout) (routes : Routes) : SuccessResult =
        let individual_routes_valid : SuccessResult seq =
            routes
            |> Seq.map (fun route ->
```

```
                        routeHasLinearStartAndEnd route
                        &&& routeIsValidInLayout layout route)

        let routes_are_valid_together : SuccessResult seq =
            uniqueProducts routes
            |> Seq.map (fun (r1,r2) -> noObviousConflict r1 r2)

        [ individual_routes_valid
        ; routes_are_valid_together ]
        |> Seq.concat
        |> Seq.reduce (&&&)

/// Validation functions for verifying that the route segments
/// obey the constraints defined by the lengths of the trains
module LengthConstraints =
    type Intersection =
        { train1 : Train
          train2 : Train
          train1_track_length : int
          train2_track_length : int
          track_id : LinearId }

    /// Checks that a route segment is shorter or equal to the
        train that holds the route
    let private routeSegmentIsShorterOrEqual (train : Train) :
        RouteSegment -> SuccessResult =
        function
        | LinearRouteSegment(segment_id,length)
            when length > train.length ->
                let err_msg =
                    sprintf "%A in the route of train %A must be
                        equal to or smaller than %i"
                err_msg segment_id train.id train.length
                |> Error
        | _ -> Ok ()

    /// Verify that lengths of intersecting track segments
    /// are obeying the following rule:
    ///
    /// if t2.length <= t1.track_length
    /// then t2.track_length == t2.length
    /// else t2.track_length == t1.track_length
    ///
    /// where t1.length >= t2.length
    let private shortestConstrainedByLongest (intersection :
        Intersection) : SuccessResult =
        // dividing the trains of the intersection into longest
            and shortest trains
        // together with the length of the intersecting track
            representation in their route.
        // train1 is the longest train and train2 is the shortest
            train
        let [ t1, t1_track_length
            ; t2, t2_track_length ] =
```

```
              [ intersection.train1, intersection.train1_track_length
              ; intersection.train2, intersection.train2_track_length ]
              |> List.sortByDescending (fst >> Train.length)

        let track_id = intersection.track_id

        // When the shortest train is shorter than the length of
        // the track representation in the longest train,
        // then the shortest trains track representation must be
        // exactly the length of the shortest train.
        //
        // Eg. if the shortest train length is 2 and the longest train length is 4
        // and the longest train track representation is 3,
        // then the shortest train track representation must be equal to 2
        if t2.length <= t1_track_length then
            if t2_track_length = t2.length then Ok ()
            else
                let err_msg = sprintf """
                    the intersecting track %A, between train %A and train %A
                    must have a length of %i in the route of train %A
                    """
                Error(err_msg track_id t1.id t2.id t2.length t2.id)
        else// shortest_train.length > longest_train_track_length
            // for example the length of shortest train is 2
            // and the length of the longest train track representation is 1,
            // then the length of the shortest trains track representation
            // must also be 1
            if t2_track_length = t1_track_length then Ok ()
            else
                let err_msg = sprintf """
                    intersecting track %A between train %A and train %A
                    must have a length of %i in the route of train %A
                    """
                Error(err_msg track_id t1.id t2.id t1.length t2.id)

    // Collect all intersections between two train routes
    let private getIntersections (t1 : Train, t2 : Train) : Intersection seq =
        let getLinearSegment : RouteSegment -> (LinearId * int) option =
            function
            | LinearRouteSegment(id,length) -> Some(id,length)
            | _ -> None

        let getLinearSegments : Route -> (LinearId * int) seq =
```

```
            fun (Route(segments,_)) ->
            segments |> Seq.choose getLinearSegment

        let t1_linear_lengths : Map<LinearId, int> =
            getLinearSegments t1.route |> Map.ofSeq

        getLinearSegments t2.route
        |> Seq.choose (
            fun (track_id, t2_track_len) ->
            Map.tryFind track_id t1_linear_lengths
            |> Option.map (fun t1_track_len ->
                          { train1 = t1
                            train2 = t2
                            train1_track_length = t1_track_len
                            train2_track_length = t2_track_len
                            track_id = track_id }))

    let checkLengthContraints (trains : Trains) : SuccessResult =
        let all_individual_routes_are_valid : SuccessResult seq =
            trains
            |> Seq.collect (fun train ->
                            let (Route(route_segments,_)) = train.route
                            route_segments
                            |> Seq.map (routeSegmentIsShorterOrEqual train))
        let all_route_intersections_are_valid : SuccessResult seq =
            uniqueProducts trains
            |> Seq.collect getIntersections
            |> Seq.map shortestConstrainedByLongest

        [ all_individual_routes_are_valid
        ; all_route_intersections_are_valid ]
        |> Seq.concat
        |> Seq.reduce (&&&)

[<AutoOpen>]
module ModelGeneration =
    /// Interface describing the properties to be generated for the model
    type ModelCheckingPropertyDefinitions =
        abstract NoCollision : string
        abstract AllTrainsArrived : string
        abstract NoDerailment : string
        abstract TrainsDetectedOnPoints : string
        abstract AllMessagesHandled : string
        abstract NoMalfunctionsWhenTrainHasNotArrived : string

    /// Validates that all trains have a route and that all the routes are valid in the layout
    let private validateTrainRoutes (layout : RailwayNetworkLayout) (objects : ModelObjects)
                                      :
                                        Result<ValidatedModelObjects,
```

```
                                        string> =
        /// gets a route if the train have one
        let getRoute : Train -> Result<Route, string> = fun train ->
            match train.route with
            // there exist a route with at least one element
            | Route(head::tail, direction) as route -> Ok (route)
            | _ -> Error (sprintf "no route for train %A" train.id)

        let routes_valid : SuccessResult =
            objects.trainList
            |> Result<_,_>.traverse getRoute
            >>= (RouteValidation.verifyRoutes layout)

        match routes_valid with
        | Ok ()      -> Ok(Validated objects)
        | Error msg -> Error msg

    let checkLengthContraints (valid_objects : ValidatedModelObjects)
                                : Result<ValidatedModelObjects, string> =
        let (Validated objects) = valid_objects
        let length_constraints_ok =
            objects.trainList
            |> LengthConstraints.checkLengthContraints
        match length_constraints_ok with
        | Ok ()      -> Ok(Validated objects)
        | Error msg -> Error msg

    // Updates all the linears, which are first on a route, to reflect presence of a train
    let updateTrainLocations (Validated objects) : Result<ValidatedModelObjects, string> =
        let updateTrainLocation (train : Train) (objects : ModelObjects) = resultFlow {
            let! linear_id =
                match train.route with
                | Route(LinearRouteSegment(linear_id,_)::_,_) -> Ok linear_id
                | _ -> Error (sprintf """
                                possibly malformed route for train %A
                                perhaps validation is missing?
                                """ train)
            let! linear =
                match Map.tryFind linear_id objects.linears with
                | Some linear -> Ok linear
                | None -> Error (sprintf "linear %A doesnt exist" linear_id)
            let updated_linear = { linear with train = Some train }
            let updated_linears = objects.linears |> Map.add linear_id updated_linear
            return { objects with linears = updated_linears } }
        objects.trainList
        |> Result<_,_>.fold updateTrainLocation (Ok objects)
```

```
        |> Result<_,_>.map Validated

    let generateRawModel : ModelGeneratorFunction = fun (Validated objects) ->
        sprintf "%A" objects

    /// Validates the routes,
    /// updates the train locations in the network based on their first route segment,
    /// and generates the final model instantiation using the generateModel function
    let validateAndGenerateModel (modelGenFun : ModelGeneratorFunction)
        : RailwayNetworkLayout -> ModelObjects -> Result<string, string> =
        fun layout objects ->
        validateTrainRoutes layout objects
        >>= checkLengthContraints
        >>= updateTrainLocations
        >>= (modelGenFun >> Ok)
```

E.3.3 UMCTrainClass.fs

```
module UMCTrain

let class_definition = """
Class Train is

  Signals:
    ok, no;

  Vars:
    requested_point_positions : bool[];
    train_length : int = 2; -- how many track segments does the train occupy
    route_segments : obj[];
    route_index : int := 0; -- current location on the route
    occupies : obj[]; -- the tc and pt objects which the train currently occupies
    front_advancement_count : int; -- a variable for keeping track of the trains front advancement over a track
    track_lengths : int[]; -- same number of elements as route_segments

  State Top = READY, WAIT_OK, MOVEMENT, ARRIVED

  Transitions:
    -- send out initial reservation request to the first node on route
    READY -> WAIT_OK {
                 - /
                 route_segments[0].req(self, 0, route_segments, requested_point_positions);
          }
```

```

        -- when the train reservation is rejected we just keep cycling between WAIT_OK and READY
    WAIT_OK -> READY { no }

    -- train receives ackknowledgement that the route has been reserved succesfully
    -- the front_advancement_count variable is initialized to reflect the trains front location on the track
    WAIT_OK -> MOVEMENT { ok / front_advancement_count := train_length - 1; }

    MOVEMENT -> MOVEMENT {
                -
                [not (route_index = route_segments.length - 1 and
                 track_lengths[route_index] - 1 = front_advancement_count)] / -- at end of track
        at_end_of_track: bool := track_lengths[route_index] - 1 = front_advancement_count; -- determine if we have reached the end of the current track
                if at_end_of_track = true then { -- the train has reached the end of its current track
          front_advancement_count := 0;
                  if route_index < route_segments.length - 1 then { -- the route_index is not the last
            -- train enters next track
            route_index := route_index + 1;
                    route_segments[route_index].sensorOn(self); -- the next track detects the train
                  };
                } else {
          front_advancement_count := front_advancement_count + 1;
                };
        -- update the occupies array
                rear: obj := occupies.head;
        next_rear: obj := occupies.tail.head;
        occupies := occupies.tail + [route_segments[route_index]];
            if rear != next_rear then { -- determine if the rear of the train has left a track
          rear.sensorOff(self); -- the past track detects that the train does not occupy it anymore
            };
        }

    MOVEMENT -> ARRIVED {
                -
                [route_index = route_segments.length - 1 and -- at last track segment of route
                 track_lengths[route_index] - 1 = front_advancement_count] -- at end of track
        }

end Train
"""
```

E.3.4 UMCLinearClass.fs

```
module UMCLinear

let class_definition = """
Class Linear is

  Signals:
        req(sender: obj, route_index: int, route_elements: obj[],
                requested_point_positions: bool[]);
        ack(sender: obj);
        nack(sender: obj);
        commit(sender: obj);
        agree(sender: obj);
        disagree(sender: obj);

  Operations:
    sensorOn(sender: obj);
    sensorOff(sender: obj);

  Vars:
    next: obj;
    prev: obj;
    train: obj := null;

  State Top = NON_RESERVED, WAIT_ACK, WAIT_COMMIT, WAIT_AGREE,
      RESERVED, TRAIN_IN_TRANSITION

  Transitions

        -- first node receive request
    NON_RESERVED -> WAIT_ACK {
                req(sender, route_index, route_elements,
                    requested_point_positions)
                [route_index = 0 and sender = train and
                    route_elements.length > 0] /
                prev := null;
                next := route_elements[1];
                next.req(self, 1, route_elements,
                    requested_point_positions);
        }

        -- intermediate node receive request
    NON_RESERVED -> WAIT_ACK {
                req(sender, route_index, route_elements,
                    requested_point_positions)
                [train = null and (route_index > 0 and
                    route_index+1 < route_elements.length)] /
                prev := route_elements[route_index - 1];
                next := route_elements[route_index + 1];
                next.req(self, route_index + 1, route_elements,
                    requested_point_positions);
        }

        -- initial reservation request for last node
```

```
        -- starts ack phase
    NON_RESERVED -> WAIT_COMMIT {
                    req(sender, route_index, route_elements,
                        requested_point_positions)
                    [train = null and route_elements.length =
                        route_index+1] /
                    prev := route_elements[route_index - 1];
                    next := null;
                    prev.ack(self);
        }

        -- intermediate node receive ack
    WAIT_ACK -> WAIT_COMMIT {
                    ack(sender)
                    [prev /= null] /
                    prev.ack(self);
        }

        -- first node receive ack
        -- and starts commit phase
    WAIT_ACK -> WAIT_AGREE {
                    ack(sender)
                    [prev = null] /
                    next.commit(self);
        }

        -- intermediate node receives commit
    WAIT_COMMIT -> WAIT_AGREE {
                    commit(sender)
                    [next /= null] /
                    next.commit(self);
        }

        -- last node receive commit
        -- and starts agree phase
    WAIT_COMMIT -> RESERVED {
                    commit(sender)
                    [next = null] /
                    prev.agree(self);
        }

        -- intermediate node receive agree
    WAIT_AGREE -> RESERVED {
                    agree(sender)
                    [prev /= null] /
                    prev.agree(self);
        }

        -- first node receive agree
        -- and sends ok to the train
    WAIT_AGREE -> TRAIN_IN_TRANSITION {
                    agree(sender)
                    [prev = null and train /= null] /
                    train.ok;
        }
```

```

        -- train moves onto current node
    RESERVED -> TRAIN_IN_TRANSITION {
                sensorOn(sender) /
                train := sender;
        }

        -- sequential release
        -- reset train
    TRAIN_IN_TRANSITION -> NON_RESERVED {
                sensorOff(sender) /
                train := null;
        }

        -- nack received
        -- forwards and goes into non-reserved
    WAIT_ACK -> NON_RESERVED {
                nack(sender) /
                if  prev = null  then { -- is first node on
                    itinerary
                        train.no
                } else { -- is not first node
                        prev.nack(self)
                };
        }

        -- disagree received
        -- forwards and goes into non-reserved
    WAIT_COMMIT -> NON_RESERVED {
                disagree(sender) /
                if next /= null  then { -- not last node on
                    itinerary
                        next.disagree(self)
                };
        }

        -- disagree received
        -- forwards and goes into non-reserved
    WAIT_AGREE -> NON_RESERVED {
                disagree(sender) /
                if prev /= null then { -- not first node on
                    itinerary
                        prev.disagree(self)
                } else { -- is first node
                        train.no
                };
        }

        -- disagree received
        -- forwards (if there is someone to forward to) and goes
            into non-reserved
    RESERVED -> NON_RESERVED {
                disagree(sender) /
                if next /= null then { -- not last node on
                    itinerary
```

```
                        next.disagree(self)
                    };
            }

            -- reservation request received
            -- however a train is already on the track, so a nack is
                returned to sender
        NON_RESERVED -> NON_RESERVED {
                    req(sender, route_index, route_elements,
                        requested_point_positions)
                    [train /= null and sender /= train] /
                    sender.nack(self);
            }

            -- reservation request received
            -- however, the node is already in wait-ack, so it returns
                a nack to sender
        WAIT_ACK -> WAIT_ACK {
                    req(sender, route_index, route_elements,
                        requested_point_positions) /
                    sender.nack(self);
            }

            -- reservation request received
            -- however, the node is already in wait-commit, so it
                returns a nack to sender
        WAIT_COMMIT -> WAIT_COMMIT {
                    req(sender, route_index, route_elements,
                        requested_point_positions) /
                    sender.nack(self);
            }

            -- reservation request received
            -- however, the node is already in wait-agree, so it
                returns a nack to sender
        WAIT_AGREE -> WAIT_AGREE {
                    req(sender, route_index, route_elements,
                        requested_point_positions) /
                    sender.nack(self);
            }

            -- reservation request received
            -- however, the node is already reserved, so it returns a
                nack to sender
        RESERVED -> RESERVED {
                    req(sender, route_index, route_elements,
                        requested_point_positions) /
                    sender.nack(self);
            }

            -- reservation request received
            -- however, a train is in transition on the node, so it
                returns a nack to sender
        TRAIN_IN_TRANSITION -> TRAIN_IN_TRANSITION {
```

```
                req(sender, route_index, route_elements,
                    requested_point_positions) /
                sender.nack(self);
        }

end Linear
"""
```

E.3.5 UMCPointClass.fs

```
module UMCPoint

let class_definition = """
-- points are always intermediate nodes
-- so we don't need to check if they are first or last in the
    guards
Class Point is

  Signals:
    req(sender: obj, route_index: int, route_elements: obj[],
        requested_point_positions: bool[]);
    ack(sender: obj);
    nack(sender: obj);
    commit(sender: obj);
    agree(sender: obj);
    disagree(sender: obj);

  Operations:
    sensorOn(sender: obj);
    sensorOff(sender: obj);

  Vars:
    next: obj;
    prev: obj;
    requested_position: bool;
    current_position: bool := True;
    train: obj := null;

  State Top = NON_RESERVED, WAIT_ACK, WAIT_COMMIT, WAIT_AGREE,
                          POSITIONING, RESERVED,
                              TRAIN_IN_TRANSITION, MALFUNCTION

 Transitions:

        -- initial reservation request
    NON_RESERVED -> WAIT_ACK {
                req(sender, route_index, route_elements,
                    requested_point_positions) /
                prev := route_elements[route_index - 1];
                next := route_elements[route_index + 1];
                requested_position :=
                    requested_point_positions[route_index];
                next.req(self, route_index + 1, route_elements,
                    requested_point_positions);
```

```
        }

        -- receiving and forwarding ack
    WAIT_ACK -> WAIT_COMMIT {
                ack(sender) /
                prev.ack(self);
        }

        -- receiving and forwarding commit
    WAIT_COMMIT -> WAIT_AGREE {
                commit(sender) /
                next.commit(self);
        }

        -- if the point is positioned as required for the given route reservation
        -- receiving and forwarding agree
    WAIT_AGREE -> RESERVED {
                agree(sender)
                [current_position = requested_position]/
                prev.agree(self);
        }

        -- if the point is not positioned as required for the given route
        -- goes into positioning state
    WAIT_AGREE -> POSITIONING {
                agree(sender)
                [current_position /= requested_position] /
                -
        }

        -- successfully performing positioning
    POSITIONING -> RESERVED {
                - /
                current_position := not current_position;
                prev.agree(self);
        }

        -- simulating sudden malfunction of positioning system
        -- and sending disagrees to neighbor nodes
    POSITIONING -> MALFUNCTION {
                - /
                prev.disagree(self);
                next.disagree(self);
        }

        -- train moves onto current node
    RESERVED -> TRAIN_IN_TRANSITION {
                sensorOn(sender) /
                train := sender;
        }

        -- sequential release
        -- reset all train
```

```
    TRAIN_IN_TRANSITION -> NON_RESERVED {
                sensorOff(sender) /
                -- [sender = train] /
                train := null;
        }

        -- nack received and forwarded
    WAIT_ACK -> NON_RESERVED {
                nack(sender) /
                prev.nack(self);
        }

        -- disagree received and forwarded
    WAIT_COMMIT -> NON_RESERVED {
                disagree(sender) /
                next.disagree(self);
        }

        -- disagree received and forwarded
    WAIT_AGREE -> NON_RESERVED {
                disagree(sender) /
                prev.disagree(self);
        }

        -- disagree received and forwarded
    POSITIONING -> NON_RESERVED {
                disagree(sender) /
                next.disagree(self);
        }

        -- disagree received and forwarded
    RESERVED -> NON_RESERVED {
                disagree(sender) /
                next.disagree(self);
        }

        -- reservation request received
        -- however, the node is already in wait-ack, so it returns
            a nack to sender
    WAIT_ACK -> WAIT_ACK {
                req(sender, route_index, route_elements,
                    requested_point_positions) /
                sender.nack(self);
        }

        -- reservation request received
        -- however, the node is malfunctioning
    MALFUNCTION -> MALFUNCTION {
                req(sender, route_index, route_elements,
                    requested_point_positions) /
                sender.nack(self);
        }

        -- reservation request received
```

```
        -- however, the node is already in wait-commit state and
            returns nack to the sender
    WAIT_COMMIT -> WAIT_COMMIT {
                req(sender, route_index, route_elements,
                    requested_position) /
                sender.nack(self);
        }

        -- reservation request received
        -- however, the node is already in wait-agree state and
            returns nack to the sender
    WAIT_AGREE -> WAIT_AGREE {
                req(sender, route_index, route_elements,
                    requested_position) /
                sender.nack(self);
        }

        -- reservation request received
        -- however, the node is already in positioning state and
            returns nack to the sender
    POSITIONING -> POSITIONING {
                req(sender, route_index, route_elements,
                    requested_position) /
                sender.nack(self);
        }

        -- reservation request received
        -- however, the node is already reserved and returns nack
            to the sender
    RESERVED -> RESERVED {
                req(sender, route_index, route_elements,
                    requested_position) /
                sender.nack(self);
        }

        -- reservation request received
        -- however, the node is occupied by a train and returns
            nack to the sender
    TRAIN_IN_TRANSITION -> TRAIN_IN_TRANSITION {
                req(sender, route_index, route_elements,
                    requested_position) /
                sender.nack(self);
        }

end Point
"""
```

E.3.6 UMC.fs

```
namespace UMC

open InterlockingModel
open UMCTrain
open UMCPoint
```

```
open UMCLinear
open System
open Utils

[<AutoOpen>]
module UMCDefinitions =
    /// the UMC classes defining the behavior of the model
    let train_class = UMCTrain.class_definition
    let point_class = UMCPoint.class_definition
    let linear_class = UMCLinear.class_definition

    /// Model specific string representations
    /// used as object identifiers in the UMC model
    type TrainId with
        static member modelRepresentation train_id =
            TrainId.value train_id |> sprintf "train_%s"
    type LinearId with
        static member modelRepresentation linear_id =
            LinearId.value linear_id |> sprintf "linear_%s"
    type PointId with
        static member modelRepresentation point_id =
            PointId.value point_id |> sprintf "point_%s"
    type PointPosition with
        static member modelRepresentation : PointPosition ->
            string = function
            | Plus -> "True"
            | Minus -> "False"
    type RouteSegment with
        static member modelRepresentation : RouteSegment -> string
            = function
            | LinearRouteSegment(lin_id,_)  ->
                LinearId.modelRepresentation lin_id
            | PointRouteSegment(point_id,_) ->
                PointId.modelRepresentation point_id

/// Functions for generation of UMC abstractions
[<AutoOpen>]
module AbstractionDefinitions =
    type AbstractionType = Action | State
    type Abstraction<'a> =
        { name : 'a  -> string
          abstraction_type : AbstractionType
          predicate : 'a -> string }

    let getAbstractionName : 'a -> Abstraction<'a> -> string = fun
        args abstraction ->
        abstraction.name args

    let abstractionDefinition (abstraction : Abstraction<'a>)
        (args : 'a) : string =
        let predicate = abstraction.predicate args
        let name = abstraction.name args
        match abstraction.abstraction_type with
        | Action -> sprintf "Action: %s -> %s" predicate name
        | State  -> sprintf "State: %s -> %s" predicate name
```

```

    let withPointModelRep : Point -> (string -> string) -> string =
        fun {id=id} stringGenerator ->
        PointId.modelRepresentation id
        |> stringGenerator

    let withTrainModelRep : Train -> (string -> string) -> string =
        fun {id=id} stringGenerator ->
        TrainId.modelRepresentation id
        |> stringGenerator

    let pointIn : PointPosition -> Abstraction<Point> = fun position ->
        let name = fun point ->
            match position with
            | Plus -> withPointModelRep point (sprintf "%s_in_plus")
            | Minus -> withPointModelRep point (sprintf "%s_in_minus")
        let predicate = fun point ->
            match position with
            | Plus -> withPointModelRep point (sprintf "%s.current_position = True")
            | Minus -> withPointModelRep point (sprintf "%s.current_position = False")
        { name = name
        ; abstraction_type = State
        ; predicate = predicate }

    let pointInPlus : Abstraction<Point> = pointIn Plus
    let pointInMinus : Abstraction<Point> = pointIn Minus

    let noTrainDetectedOnPoint : Abstraction<Point> =
        let name = fun point ->
            withPointModelRep point (sprintf "no_train_on_%s")
        let predicate = fun point ->
            withPointModelRep point (sprintf "%s.train = null")
        { name = name
        ; abstraction_type = State
        ; predicate = predicate }

    let all_pairs_of_trains_at_diff_positions : Abstraction<Trains> =
        let name = fun _ -> "trains_at_diff_positions"
        let predicate = fun trains ->
            let trainPairAtDiffPositions : (Train * Train) -> string =
                fun (train1,train2) ->
                let train1_id = TrainId.modelRepresentation train1.id
                let train2_id = TrainId.modelRepresentation train2.id

                let trainPartsAtDiffPositions : (int * int) -> string = fun (id1,id2) ->
```

```
                    sprintf "%s.occupies[%d] /= %s.occupies[%d]"
                            train1_id
                            id1
                            train2_id
                            id2

                crossProductOfLists [0..train1.length-1]
                    [0..train2.length-1]
                |> Seq.map trainPartsAtDiffPositions
                |> String.concat " and \n"

            uniqueProducts trains
            |> Seq.map trainPairAtDiffPositions
            |> String.concat " and \n"
        { name = name
        ; abstraction_type = State
        ; predicate = predicate }

    let positioning : Abstraction<Point> =
        let name = fun point ->
            (sprintf "positioning_%s")
            |> withPointModelRep point
        let predicate = fun point ->
            (sprintf "inState(%s.POSITIONING)")
            |> withPointModelRep point
        { name = name
        ; abstraction_type = State
        ; predicate = predicate }

    let trainArrived : Abstraction<Train> =
        let name = fun train ->
            withTrainModelRep train (sprintf "%s_arrived")
        let predicate = fun train ->
            withTrainModelRep train (sprintf "inState(%s.ARRIVED)")
        { name = name
        ; abstraction_type = State
        ; predicate = predicate }

    let trainNotOnPoint : Abstraction<Train * Point> =
        let name : Train * Point -> string = fun (train, point) ->
            let train_model_rep = TrainId.modelRepresentation
                train.id
            let point_model_rep = PointId.modelRepresentation
                point.id
            sprintf "%s_not_on_%s" train_model_rep point_model_rep
        let predicate : Train * Point -> string = fun (train,
            point) ->
            let point_model_rep = PointId.modelRepresentation
                point.id
            let train_model_rep = TrainId.modelRepresentation
                train.id
            let trainPartNotOnPoint : int -> string = fun index ->
                sprintf "%s.occupies[%d] /= %s"
                        train_model_rep
                        index
```

```
                        point_model_rep
            [0 .. train.length - 1]
            |> Seq.map trainPartNotOnPoint
            |> String.concat " and\n"
        { name = name
        ; abstraction_type = State
        ; predicate = predicate }

    let discarded_msg : Abstraction<unit> =
        { name = fun _ -> "discarded_message"
        ; abstraction_type = Action
        ; predicate = fun _ -> "lostevent" }

    let pointMalfunction : Abstraction<Point> =
        let pointModelRep = fun p -> PointId.modelRepresentation p.id
        { name = fun p -> sprintf "%s_malfunction" (pointModelRep p)
        ; abstraction_type = State
        ; predicate = fun p -> sprintf "inState(%s.MALFUNCTION)" (pointModelRep p) }

    let all_abstraction_declarations (validated_objects : ValidatedModelObjects) : string =
        let (Validated objects) = validated_objects
        let trains : Trains = objects.trainList
        let points : Points = objects.pointList

        let no_trains_detected_on_points =
            points |> List.map (abstractionDefinition noTrainDetectedOnPoint)
        let points_positioning =
            points |> List.map (abstractionDefinition positioning)
        let points_in_plus =
            points |> List.map (abstractionDefinition pointInPlus)
        let points_in_minus =
            points |> List.map (abstractionDefinition pointInMinus)
        let trains_arrived =
            trains |> List.map (abstractionDefinition trainArrived)
        let trains_at_diff_positions =
            (all_pairs_of_trains_at_diff_positions, trains)
            ||> abstractionDefinition
        let trains_not_on_points =
            crossProductOfLists trains points
            |> List.ofSeq
            |> List.map (abstractionDefinition trainNotOnPoint)
        let discarded_message =
            abstractionDefinition discarded_msg ()
        let points_malfunction =
            points |> List.map (abstractionDefinition pointMalfunction)

        [ trains_arrived
        ; no_trains_detected_on_points
        ; points_positioning
```

```
          ; points_in_plus
          ; points_in_minus
          ; points_malfunction
          ; [ trains_at_diff_positions ]
          ; trains_not_on_points
          ; [ discarded_message ] ]
          |> List.concat
          |> String.concat "\n"

//// Extension functions for generating object instantiations in UMC
[<AutoOpen>]
module ModelObjectInstantiations =
    type Linear with
        static member modelObjectInstantiation : Linear -> string = fun linear ->
            let train_id =
                match linear.train with
                | Some train -> TrainId.modelRepresentation train.id
                | None       -> "null"
            let linear_id = LinearId.modelRepresentation linear.id
            sprintf "%s: Linear(train => %s);" linear_id train_id

    type Point with
        static member modelObjectInstantiation : Point -> string = fun point ->
            let model_point_id = PointId.modelRepresentation point.id
            sprintf "%s: Point;" model_point_id

    type Train with
        static member modelObjectInstantiation : Train * ModelObjects -> string =
            fun (train,objects) ->
            let train_id : string = TrainId.modelRepresentation train.id

            let pointPosition : RouteSegment -> string = function
                | PointRouteSegment (_,pos) -> PointPosition.modelRepresentation pos
                | _ -> "null"

            let route_segments : RouteSegments = Route.segments train.route

            let route_element_ids : string =
                route_segments
                |> Seq.map RouteSegment.modelRepresentation
                |> String.concat ","

            let track_lengths : string =
                route_segments
                |> Seq.map (RouteSegment.length >> string)
                |> String.concat ","
```

```
            let requested_point_positions : string =
                route_segments
                |> Seq.map pointPosition
                |> String.concat ","

            let occupies : string option =
                Seq.tryHead route_segments
                |> function
                    | Some(LinearRouteSegment(linear_id,_)) -> Some
                        linear_id
                    | _ -> None
                |> Option.map (LinearId.modelRepresentation
                              >> Seq.replicate train.length
                              >> String.concat ",")

            match occupies with
            | Some occupies ->
                [ sprintf "%s: Train(" train_id
                ; sprintf "route_segments => [%s]," 
                    route_element_ids
                ; sprintf "track_lengths => [%s]," track_lengths
                ; sprintf "train_length => %i," train.length
                ; sprintf "occupies => [%s]," occupies
                ; sprintf "requested_point_positions => [%s]);" 
                    requested_point_positions ]
                |> String.concat "\n"
            | None -> sprintf """
                              %s: Train(
                              route_segments => [],
                              track_lengths => [],
                              train_length => %i,
                              occupies => [],
                              requested_point_positions => []);
                              """
                              train_id train.length

    let modelObjectInstantiation (Validated objects) : string =
        let trains : string seq =
            Map.toSeq objects.trains
            |> Seq.map (fun (_,train) ->
                Train.modelObjectInstantiation (train,objects))
        let linears : string seq =
            Map.toSeq objects.linears |> Seq.map (snd >>
                Linear.modelObjectInstantiation)
        let points : string seq =
            Map.toSeq objects.points |> Seq.map (snd >>
                Point.modelObjectInstantiation)
        [trains; linears; points]
        |> Seq.concat
        |> String.concat "\n\n"

[<AutoOpen>]
module Properties =
    /// Class containing generated properties
```

```
    /// (implements the ModelCheckingPropertyDefinitions interface)
    type ModelCheckingProperties(validated_objects : ValidatedModelObjects) =
        let (Validated objects) = validated_objects
        let trains = objects.trainList
        let points = objects.pointList

        interface ModelCheckingPropertyDefinitions with
            member this.NoMalfunctionsWhenTrainHasNotArrived : string =
                let p_malfunctions : string =
                    points
                    |> Seq.map (fun p -> getAbstractionName p pointMalfunction)
                    |> String.concat " or "
                let t_arrivals : string =
                    trains
                    |> Seq.map getAbstractionName
                    |> Seq.map ((|>) trainArrived)
                    |> String.concat " and "
                sprintf "not E[not (%s) U (final and not (%s))];"
                        p_malfunctions t_arrivals

            member this.NoCollision : string =
                all_pairs_of_trains_at_diff_positions
                |> getAbstractionName trains
                |> sprintf "AG (%s);"

            member this.AllTrainsArrived : string =
                trains
                |> Seq.map getAbstractionName
                |> Seq.map ((|>) trainArrived)
                |> String.concat " and "
                |> sprintf "EF AG (%s);"

            member this.NoDerailment : string =
                let positioningImpliesNoTrain = fun abstractionName ->
                    let point_is_positioning = abstractionName positioning
                    let no_train_detected =
                        abstractionName noTrainDetectedOnPoint
                    [ point_is_positioning; " implies "; no_train_detected ]
                    |> String.concat ""
                points
                |> Seq.map (getAbstractionName >> positioningImpliesNoTrain)
                |> String.concat " and\n"
                |> sprintf "AG (%s);"

            member this.TrainsDetectedOnPoints : string =
                let trainsAtPointImpliesTrainsDetected = fun (train1, train2, point) ->
                    let train1_not_on_point =
```

```
                        trainNotOnPoint |> getAbstractionName (train1,point)
                    let train2_not_on_point =
                        trainNotOnPoint |> getAbstractionName (train2,point)
                    let no_train_on_point =
                        noTrainDetectedOnPoint |> getAbstractionName point
                    [ "(not ("; train1_not_on_point; " and "; train2_not_on_point; ") "
                    ; "implies "; no_train_on_point; ")" ]
                    |> String.concat ""
                points
                |> Seq.collect (fun point ->
                                uniqueProducts trains
                                |> Seq.map (fun (t1,t2) -> t1,t2,point))
                |> Seq.map trainsAtPointImpliesTrainsDetected
                |> String.concat " and\n"
                |> sprintf "AG (%s);"

        member this.AllMessagesHandled : string =
                getAbstractionName () discarded_msg
                |> sprintf "AG not (EX {%s} true);"

    /// Collects all the model properties into one string
    let collectModelProperties (properties : ModelCheckingPropertyDefinitions) =
        let trains_at_diff_positions =
            properties.NoCollision
            |> sprintf """
                -- safety property:
                -- no incident
                -- no trains occupy the same location node at the same time
                %s
                """
        let no_derailment =
            properties.NoDerailment
            |> sprintf """
                -- safety property:
                -- no trains are located at any 'point' while it is changing its position
                %s
                """
        let all_trains_arrived =
            properties.AllTrainsArrived
            |> sprintf """
                -- progress property that specifies that
                -- all trains has arrived at their destinations
                %s
                """
        let no_malfunction_when_train_has_not_arrived =
            properties.NoMalfunctionsWhenTrainHasNotArrived
            |> sprintf """
```

```
                -- property that specifies that
                -- there does not exist a final state where at least one train has not arrived
                -- and in all states leading to this final state, no points have malfunctioned
                %s
                """
        let trains_detected_on_points =
            properties.TrainsDetectedOnPoints
            |> sprintf """
                -- property to verify that all trains are correctly detected at points
                %s
                """
        let all_messages_handled =
            properties.AllMessagesHandled
            |> sprintf """
                -- no signal is ever lost in the system
                %s
                """

        [ trains_at_diff_positions
        ; no_derailment
        ; trains_detected_on_points
        ; all_trains_arrived
        ; no_malfunction_when_train_has_not_arrived
        ; all_messages_handled ]
        |> String.concat "\n"

/// Functions for composing a UMC model
module UMCModelConstruction =
    let private objects_section objects =
        modelObjectInstantiation objects
        |> sprintf "Objects\n %s"

    let private abstractions_section objects =
        objects
        |> AbstractionDefinitions.all_abstraction_declarations
        |> sprintf "Abstractions {\n%s\n}"

    let private properties_section objects =
        ModelCheckingProperties(objects)
        |> collectModelProperties

    let composeModel : ModelGeneratorFunction = fun objects ->
        [ train_class
        ; linear_class
        ; point_class
        ; objects_section objects
        ; abstractions_section objects
        ; properties_section objects ]
        |> String.concat "\n"
```

E.3.7 XMLExtraction.fs

```
namespace XMLExtraction

open FSharp.Data
open Utils
open InterlockingModel

/// Required types and extensions of existing types
[<AutoOpen>]
module TypeDefinitions =
    [<Literal>]
    let sample_xml_file = "sample.xml"
    // Loading the type provider with a sample file
    type RailwayXML = XmlProvider<sample_xml_file>

    (* Modules and functions for more convenient access of
       values extracted by the type provider *)

    module XMLRoute =
        let Id (route : RailwayXML.Route) : string = route.Id

    module TrackSection =
        let Id (track_section : RailwayXML.TrackSection) : string = track_section.Id
        let Type (track_section : RailwayXML.TrackSection) : string = track_section.Type

    module Neighbor =
        let Id (neighbor : RailwayXML.Neighbor) : string = neighbor.Ref
        let Side (neighbor : RailwayXML.Neighbor) : string = neighbor.Side

    module MarkerBoard =
        let Id (markerboard : RailwayXML.Markerboard) : string = markerboard.Id
        let Track (markerboard : RailwayXML.Markerboard) : string = markerboard.Track

    module Condition =
        let Type (condition : RailwayXML.Condition) : string = condition.Type
        let Ref (condition : RailwayXML.Condition) : string = condition.Ref

    type PointPosition with
        static member fromString : string -> PointPosition = function
            | "plus"  -> Plus
            | "minus" -> Minus

    type RouteDirection with
        static member fromString : string -> RouteDirection = function
```

```
            | "up"   -> Up
            | "down" -> Down

    type LayoutSegment with
        static member fromString (id : string) : string * string
            -> LayoutSegment =
            function
            | "linear", _       -> LinearLayoutSegment id
            | "point", "stem"   -> PointStemLayoutSegment id
            | "point", "plus"   -> PointForkLayoutSegment (id, Plus)
            | "point", "minus" -> PointForkLayoutSegment (id,
                Minus)

[<AutoOpen>]
module BasicObjectExtraction =
    type ModelObjectsExtractionParameters =
        { xml_file_path : string
          train_ids : TrainIds
          train_length : int
          routes : Routes }
    /// Extracts all trains, points and linears from xml file
    let extractBasicModelObjectsFromXML :
        ModelObjectsExtractionParameters -> ModelObjects =
        fun parameters ->
        let xml = RailwayXML.Load parameters.xml_file_path
        let linears : (LinearId * Linear) seq =
            xml.Interlocking.Network.TrackSections
            |> Seq.choose
                (fun track_section ->
                 match track_section.Type with
                 | "linear" ->
                     Some(LinearId track_section.Id,
                          { id = LinearId track_section.Id
                            train = None })
                 | _ -> None)
        let points : (PointId * Point) seq =
            xml.Interlocking.Network.TrackSections
            |> Seq.choose
                (fun track_section ->
                 match track_section.Type with
                 | "point" ->
                     Some(PointId track_section.Id,
                           { id = PointId track_section.Id
                             position = Plus})
                 | _ -> None )
        let trains =
            List.zip parameters.train_ids parameters.routes
            |> List.map (fun (id,route) ->
                            id,{ id = id; route = route; length =
                                parameters.train_length })
        let basic_objects : ModelObjects =
            { trains  = trains |> Map.ofList
              points  = points |> Map.ofSeq
              linears = linears |> Map.ofSeq }
        basic_objects
```

```

[<AutoOpen>]
module LayoutExtraction =
    /// Retrieves all connection pairs from a given xml file.
    /// Assumes that the layout defined in the xml file is well-formed
    let createLayoutFromXML (path : string) : Result<RailwayNetworkLayout, string> =
        let xml = RailwayXML.Load path
        let elements = xml.Interlocking.Network.TrackSections
        /// getting the adjacent neighbor type
        /// if its a linear then its straightforward
        /// if its a point then we have to figure out if its the stem or one of the forks
        let getNeighborType : string -> string -> Result<LayoutSegment, string> =
            fun from_id to_id -> resultFlow {
            let! element =
                elements
                |> Seq.tryFind (TrackSection.Id >> (=)to_id)
                |> function
                    | None -> Error (sprintf "no tracksection with id %s" to_id)
                    | Some element -> Ok element
            let! from_neighbor =
                element.Neighbors
                |> Seq.tryFind (Neighbor.Id >> (=)from_id)
                |> function
                    | Some neighbor -> Ok neighbor
                    | None -> sprintf "could not find %s in neighbors of %s" to_id from_id
                                |> Error
            return LayoutSegment.fromString to_id (element.Type, from_neighbor.Side) }

        /// All connection pairs going from linear to other
        let linears : Result<LayoutSegment * LayoutSegment, string> seq  =
            let isLinearWithUpNeighbor : RailwayXML.TrackSection -> bool =
                fun element ->
                let element_is_linear = element.Type = "linear"
                let element_has_up_neighbor =
                    element.Neighbors |> Seq.exists (Neighbor.Side >> (=)"up")
                (element_is_linear && element_has_up_neighbor)

            let getFromIdAndToId : RailwayXML.TrackSection -> string * string =
                fun element ->
                let from_id = element.Id
                let neighbor = element.Neighbors |> Seq.find (Neighbor.Side >> (=)"up")
                let to_id = neighbor.Ref
                from_id, to_id
```

```

            let getLayoutElementPair
                : string * string -> Result<LayoutSegment * LayoutSegment, string> =
                fun (from_id, to_id) ->
                let from_element = LinearLayoutSegment from_id
                let to_element = getNeighborType from_id to_id
                Result<_,_>.map (fun to_el -> from_element, to_el) to_element

            elements
            |> Seq.filter isLinearWithUpNeighbor
            |> Seq.map (getFromIdAndToId >> getLayoutElementPair)

        let getLayoutElementPair
            : string -> RailwayXML.Neighbor -> Result<LayoutSegment * LayoutSegment, string> =
            fun from_id neighbor ->
            let neighbor_id = neighbor.Ref
            let from_element =
                LayoutSegment.fromString from_id ("point", neighbor.Side)
            getNeighborType from_id neighbor_id
            |> Result<_,_>.map (fun to_element -> from_element, to_element)

        /// All connection pairs going from point to other
        let points : Result<LayoutSegment * LayoutSegment, string> seq = seq {
            for element in elements |> Seq.filter (TrackSection.Type >> (=)"point") do
                let from_id = element.Id
                for neighbor in element.Neighbors do
                    yield getLayoutElementPair from_id neighbor }
        [linears; points]
        |> Seq.concat
        |> List.ofSeq
        |> Result<_,_>.sequence
        |> Result<_,_>.map Map.ofList

[<AutoOpen>]
module RouteExtraction =
    let extractRouteFragmentFromXML (path : string) (linear_length : int) (route_id : string)
                                    : Result<Route, string> = resultFlow {
        let xml = RailwayXML.Load path

        let route = xml.Interlocking.Routetable.Routes
                    |> Seq.tryFind (XMLRoute.Id >> (=)route_id)

        let! (route : RailwayXML.Route) =
            xml.Interlocking.Routetable.Routes
            |> Seq.tryFind (XMLRoute.Id >> (=)route_id)
            |> function
```

```
                | Some route -> Ok route
                | None -> Error (sprintf "no route with id %s" route_id)

        let markerboard_src_id : string = route.Source

        let! (linear_src_id : string) =
            xml.Interlocking.Network.Markerboards
            |> Seq.tryFind (MarkerBoard.Id >> (=)markerboard_src_id)
            |> function
                | Some markerboard -> Ok markerboard.Track
                | None -> Error (sprintf "no markerboard for id %s" markerboard_src_id)

        let points : (string * PointPosition) seq =
            route.Conditions
            |> Seq.filter (Condition.Type >> (=)"point")
            |> Seq.map (fun condition ->
                            let pos = Option.get condition.Val
                            let id = condition.Ref
                            let point_pos = PointPosition.fromString pos
                            (id, point_pos) )

        // if the id exists as a point id then the id is a point
        let tryGetPoint : string -> (string * PointPosition) option =
            fun id -> points |> Seq.tryFind (fst >> (=)id)

        let vacancies : RouteSegments =
            let routeElement : RailwayXML.Condition -> RouteSegment =
                fun condition ->
                let id = condition.Ref
                match tryGetPoint id with
                | Some (id,pos) -> PointRouteSegment(PointId id, pos)
                | None          -> LinearRouteSegment(LinearId id, linear_length)
            route.Conditions
            |> Seq.filter (Condition.Type >> (=)"trackvacancy")
            |> Seq.map routeElement
            |> List.ofSeq

        let! (direction : RouteDirection) =
            xml.Interlocking.Network.Markerboards
            |> Seq.tryFind (MarkerBoard.Id >> (=)markerboard_src_id)
            |> function
                | None -> Error (sprintf "no markerboard with id %s" markerboard_src_id)
                | Some markerboard ->
                    markerboard.Mounted
                    |> RouteDirection.fromString
```

```
                        |> Ok

            let first_element = LinearRouteSegment(LinearId linear_src_id, linear_length)
            let route_elements = first_element :: vacancies

            return Route (route_elements, direction) }

    let extractRouteFragmentsFromXML (path : string) (train_len : int) (route_ids : string list)
                                                : Result<Route, string> =
            let extractRouteFragment : string -> Result<Route, string> = fun route_id ->
                extractRouteFragmentFromXML path train_len route_id
            route_ids
            |> Seq.map extractRouteFragment
            |> Result<_,_>.reduce stitchRoutePair

[<AutoOpen>]
module ModelGenerationFromXML =
    type ModelGenerationParameters =
            { modelGeneratorFunction : ModelGeneratorFunction
              xml_file_path : string
              routes : string list list }
    let generateModelFromXML : ModelGenerationParameters -> Result<string,string> =
            fun parameters -> resultFlow {
            let default_train_length = 2
            let extractRouteFragments : string list -> Result<Route,string> =
                extractRouteFragmentsFromXML parameters.xml_file_path default_train_length
            let! (routes : Route list) =
                parameters.routes
                |> Result<_,_>.traverse extractRouteFragments

            let! (layout : RailwayNetworkLayout) = createLayoutFromXML parameters.xml_file_path
            let validateAndGenerate = validateAndGenerateModel parameters.modelGeneratorFunction

            let train_ids = [0 .. Seq.length routes - 1]
                            |> List.map (sprintf "%i" >> TrainId)

            let! model =
                {   xml_file_path = parameters.xml_file_path
                    train_ids = train_ids
                    train_length = default_train_length
                    routes = routes }
                |> extractBasicModelObjectsFromXML
                |> validateAndGenerate layout

            return model }
```

E.3.8 ScriptTools.fs

```
namespace ScriptingTools

open InterlockingModel
open XMLExtraction.LayoutExtraction
open UMC
open Utils
open System.IO

[<AutoOpen>]
module SimpleTypes =
    type SimpleTrackSegment = LLinear of name : string
                            | LPointFork of name : string * PointPosition
                            | LPointStem of name : string
    let (<+>) (el1 : SimpleTrackSegment) (el2 : SimpleTrackSegment) = el1, el2
    type SimpleLayout = (SimpleTrackSegment * SimpleTrackSegment) list

    type SimpleRouteElement =
        | RLinear of name : string * length : int
        | RPoint of name : string * position : PointPosition
    type SimpleRoute = SimpleRouteElement list

    type SimpleTrain =
        { id : string
        ; length : int
        ; route : SimpleRoute
        ; route_direction : RouteDirection }
    type SimpleTrains = SimpleTrain list

    type LayoutType = CustomLayout of SimpleLayout
                    | XMLLayout of path : string

    type SimpleModelArgs =
        { trains : SimpleTrain list
        ; layout : LayoutType
        ; show_stats : bool
        ; output_file : string option }

    type Stats =
        { num_of_trains : int
        ; train_lengths : int list
        ; route_lengths : int list
        ; total_route_sub_segments : int
        ; total_linears : int
        ; total_points : int
        ; shared_points : int
        ; shared_linears : int}

    type SimpleRouteElement with
        static member Length (element : SimpleRouteElement) =
            match element with
```

```
            | RLinear(_,len) -> len
            | _              -> 1

/// Functions for converting script model representation to
/// to a validated internal representation
[<AutoOpen>]
module ScriptTools =
    let private toLayoutSegment : SimpleTrackSegment -> LayoutSegment =
        function
        | LPointFork(n, Plus) -> PointForkLayoutSegment (n, Plus)
        | LPointFork(n, Minus) -> PointForkLayoutSegment (n, Minus)
        | LPointStem n -> PointStemLayoutSegment n
        | LLinear n -> LinearLayoutSegment n

    type SimpleRouteElement with
        static member toRouteSegment : SimpleRouteElement -> RouteSegment =
            function
            | RLinear(n,len) -> LinearRouteSegment (LinearId n, len)
            | RPoint(n, Plus) -> PointRouteSegment (PointId n, Plus)
            | RPoint(n, Minus) -> PointRouteSegment (PointId n, Minus)

    let private getRailwayLayoutFromCustom (custom_layout : SimpleLayout) : RailwayNetworkLayout =
        custom_layout
        |> List.map (fun (el1,el2) -> toLayoutSegment el1, toLayoutSegment el2)
        |> Map.ofList

    let private getRoute (train : SimpleTrain) : Route =
        let route_elements =
            train.route
            |> List.map SimpleRouteElement.toRouteSegment
        Route (route_elements, train.route_direction)

    let private getTrains (trains : SimpleTrain list) : Trains =
        let toTrain : SimpleTrain -> Train = fun t ->
            { id = TrainId t.id; length = t.length; route = getRoute t }
        trains |> List.map toTrain

    /// Extract linears from layout
    let private getLinears (layout : RailwayNetworkLayout) : Linears =
        Map.toList layout
        |> List.unzip
        ||> List.append
        |> List.choose
            (function
             | LinearLayoutSegment n -> Some({id = LinearId n; train = None})
```

```
                | _ -> None)
            |> Set.ofList
            |> Set.toList

    /// Extract points from layout
    let private getPoints (layout : RailwayNetworkLayout) : Points =
        Map.toList layout
        |> List.unzip
        ||> List.append
        |> List.choose
            (function
                | PointForkLayoutSegment(n, _)
                | PointStemLayoutSegment n -> Some({id = PointId n; position = Plus})
                | _ -> None)
        |> Set.ofList
        |> Set.toList

    let private getObjects (trains : SimpleTrains) (layout : RailwayNetworkLayout)
                           : ModelObjects =
        let train_ids = trains |> List.map (fun t -> TrainId t.id)
        let trains_map : Map<TrainId,Train> =
            List.zip train_ids (getTrains trains)
            |> Map.ofList

        let linears = getLinears layout
        let linear_ids = linears |> List.map Linear.id
        let linears_map : Map<LinearId,Linear> =
            List.zip linear_ids linears
            |> Map.ofList

        let points = getPoints layout
        let point_ids = points |> List.map Point.id
        let points_map : Map<PointId,Point> =
            List.zip point_ids points
            |> Map.ofList
        { trains = trains_map
          linears = linears_map
          points = points_map }

    /// Pretty print the raw layout from an XML file
    let printRawLayout (path : string) =
        let layoutSegmentToSimple segment =
            match segment with
            | LinearLayoutSegment id -> LLinear id
            | PointForkLayoutSegment(id, pos) -> LPointFork(id, pos)
            | PointStemLayoutSegment id -> LPointStem id
            |> sprintf "%A"

        let layout = resultFlow {
            let! layout = createLayoutFromXML path
            let output_layout =
```

```
            layout
            |> Map.toList
            |> List.map (fun (x,y) ->
                          let x_simple = layoutSegmentToSimple x
                          let y_simple = layoutSegmentToSimple y
                          sprintf "%s <+> %s" (x_simple.PadRight 24) y_simple)
        return output_layout |> String.concat "\n" }
    match layout with
    | Ok layout_string -> printfn "%s" layout_string
    | Error msg -> printfn "ERROR: %s" msg

let private generateStats (trains : SimpleTrains) (layout : RailwayNetworkLayout) : Stats =
    let num_of_trains = trains |> List.length
    let train_lengths = trains |> List.map (fun {length=len} -> len)
    let route_lengths = trains |> List.map (fun {route=r} -> List.length r)
    let total_route_sub_segments =
        trains
        |> Seq.map
            (fun {route=r} ->
             Seq.map (SimpleRouteElement.Length) r |> Seq.sum)
        |> Seq.sum
    let total_linears = getLinears layout |> List.length
    let total_points = getPoints layout |> List.length

    let getPointNames = function RPoint(n,_) -> Some n | _ -> None
                         |> Seq.choose
    let shared_points =
        uniqueProducts trains
        |> Seq.map (fun ({route=r1}, {route=r2}) ->
                    let r1_points = r1 |> getPointNames |> Set.ofSeq
                    let r2_points = r2 |> getPointNames |> Set.ofSeq
                    Set.intersect r1_points r2_points)
        |> Seq.map Set.count
        |> Seq.sum
    let getLinearNames = function RLinear(n,_) -> Some n | _ -> None
                          |> Seq.choose
    let shared_linears =
        uniqueProducts trains
        |> Seq.map (fun ({route=r1}, {route=r2}) ->
                    let r1_linears = r1 |> getLinearNames |> Set.ofSeq
                    let r2_linears = r2 |> getLinearNames |> Set.ofSeq
                    Set.intersect r1_linears r2_linears)
        |> Seq.map Set.count
        |> Seq.sum
    { num_of_trains = num_of_trains
```

```
        ; train_lengths = train_lengths
        ; route_lengths = route_lengths
        ; total_route_sub_segments = total_route_sub_segments
        ; total_linears = total_linears
        ; total_points = total_points
        ; shared_points = shared_points
        ; shared_linears = shared_linears }


    let generateUMCModel (model_args : SimpleModelArgs) : unit =
        let validateAndGenerate =
            UMCModelConstruction.composeModel
            |> validateAndGenerateModel
        let output = resultFlow {
            let! layout =
                match model_args.layout with
                | CustomLayout custom_layout ->
                    Ok(getRailwayLayoutFromCustom custom_layout)
                | XMLLayout path -> createLayoutFromXML path
            let trains = model_args.trains
            let objects = getObjects trains layout
            let! model = validateAndGenerate layout objects
            let stats = generateStats trains layout
            let output =
                if model_args.show_stats then
                    sprintf "STATS:\n%A\n\nMODEL:\n%s" stats model
                else sprintf "MODEL:\n%s" model
            return output }
        match output, model_args.output_file with
        | Ok output, None -> printfn "%s" output
        | Ok output, Some file_path ->
            File.WriteAllText(file_path, output)
            printfn "model written to file %s" file_path
        | Error msg, _ -> printfn "ERROR:\n%s" msg
```

E.3.9 MiniModelGenerator.fs

```
module MiniModelGenerator

open System.Text.RegularExpressions
open InterlockingModel
open UMC
open XMLExtraction
open Utils
open System
open System.IO

type ModelOutput = UMC
                 | Raw // representing the raw F# objects

type ModelOutput with
    static member fromString (s : string) : Result<ModelOutput,
        string> =
        match s.ToLower() with
```

```
        | "umc" -> Ok UMC
        | "raw" -> Ok Raw
        | _ -> Error "wrong model type"

let parseItin arg =
    match Regex("[^\[\]]+").Match arg with
    | m when m.Success && m.Value <> "" ->
        let parsed =
            m.Value.Split ','
            |> Array.filter ((<>)"")
            |> List.ofArray
        if Seq.length parsed > 0
        then Ok parsed
        else Error "error in itinerary argument no routes provided"
    | _ -> Error "error in itinerary argument"

let parseInt arg =
    match Int32.TryParse arg with
    | true, x -> Ok x
    | _ -> Error (sprintf "not an integer: %A" arg)

let getPath curr_dir path_arg =
    if File.Exists path_arg
    then Ok path_arg
    else
        let full_path = sprintf "%s/%s" curr_dir path_arg
        if File.Exists full_path
        then Ok full_path
        else Error (sprintf "no file with path %s" path_arg)

[<EntryPoint>]
let main argv =
    let curr_dir = Directory.GetCurrentDirectory()
    if Array.length argv < 4 then
        [ "Following arguments must be provided (in same order):"
          "1. file-path of xml file"
          "2. model type (umc|raw)"
          "3. itinerary for train 1 in the form [r_1,r_2,...]"
          "4. itinerary for train 2 in the form [r_3,r_4,...]"
          "5. ... "]
        |> String.concat "\n"
        |> printfn "%s"
    else
      let model = resultFlow {
          let! path = getPath curr_dir argv.[0]
          let! model_output_type = ModelOutput.fromString argv.[1]
          let! routes =
              [2 .. Array.length argv - 1]
              |> Result<_,_>.traverse
                  (fun arg_id -> resultFlow {
                    let! route = parseItin argv.[arg_id]
                    return route })
          let parameters =
              { modelGeneratorFunction = generateRawModel
                xml_file_path = path
```

```
                    routes = routes }
            let! model =
                match model_output_type with
                | Raw -> generateModelFromXML parameters
                | UMC ->
                    { parameters with modelGeneratorFunction =
                        UMCModelConstruction.composeModel }
                    |> generateModelFromXML
            return model }
        match model with
        | Ok model -> printfn "%s" model
        | Error err -> printfn "ERROR: \n%s" err
    0 // return an integer exit code
```

E.3.10 Prelude.fsx

```
(*
    Loading the required files and assemblies for the scripts
*)

// required assembly for parsing xml files

#r "../../packages/FSharp.Data/lib/net40/FSharp.Data.dll"

(* loading required files *)
#load "Utils.fs"
#load "InterlockingModel.fs"
#load "UMCTrainClass.fs"
#load "UMCLinearClass.fs"
#load "UMCPointClass.fs"
#load "UMC.fs"
#load "XMLExtraction.fs"
#load "ScriptTools.fs"

open ScriptingTools
```

E.3.11 Script.fsx

A sample script for defining a model using the DSL scripting tools library.

```
(*
    Example script for generating a UMC model with a custom layout
*)

#load "Prelude.fsx"

(* importing required modules *)
open InterlockingModel
open ScriptingTools

(* Define the trains and routes to be used in the model *)
```

```
let trains : SimpleTrains =
    [ { id = "1"
        length = 3
        route = [ RLinear(name = "1", length = 3)
                ; RPoint(name = "1", position = Plus)
                ; RLinear(name = "2", length = 2)
                ; RPoint(name = "2", position = Plus)
                ; RLinear(name = "4", length = 3) ]
        route_direction = Up }
      { id = "2"
        length = 3
        route = [ RLinear(name = "4", length = 3)
                ; RPoint(name = "2", position = Minus)
                ; RLinear(name = "3", length = 3) ]
        route_direction = Down } ]

(* Define the network layout in a 'left-to-right' fashion,
   using '<+>' to indicate connection between elements *)
let network_layout : SimpleLayout =
    [ LLinear "1"               <+> LPointStem "1"
    ; LPointFork("1", Plus)    <+> LLinear "2"
    ; LPointFork("1", Minus)   <+> LLinear "3"
    ; LLinear "2"               <+> LPointFork("2",Plus)
    ; LLinear "3"               <+> LPointFork("2",Minus)
    ; LPointStem "2"            <+> LLinear "4" ]

(* Generate model based on the definitions above *)
generateUMCModel { trains = trains
                 ; layout = CustomLayout(network_layout)
                 ; show_stats = true
                 ; output_file = Some "mymodel.txt" }
```

E.4 Tests

This section defines a few unit tests and property based tests that all evaluates to true for the current source code.

E.4.1 Tests.Utils.fs

Testing selected functions from the Utils module.

```
namespace Tests.Utils
open Utils

module crossProductOfLists =
    open Xunit
    open FsUnit.Xunit
```

```
    open FsCheck.Xunit

    [<Fact>]
    let ``simple test1`` () =
        let result = crossProductOfLists [1;2] ["a";"b"] |> Set.ofSeq
        let expected = Set.ofList [1,"a"; 1,"b"; 2,"a"; 2,"b"]
        result |> should equal expected

    [<Property>]
    let ``length of cross product list is n**2, where n is the length of one of the input lists``
        (xs : int list) =
        let result =
            crossProductOfLists xs xs
            |> List.ofSeq
            |> List.length
        let expected = float(List.length xs)**2.0 |> int
        result = expected


module uniqueProducts =
    open FsCheck
    open FsCheck.Xunit

    let sets_bigger_than_two : Arbitrary<NonEmptySet<int>> =
        Arb.Default.NonEmptySet<int>()
        |> Arb.filter (fun x -> Set.count x.Get >= 2)
        |> Arb.filter (fun x -> Set.count x.Get <= 12)

    let n_choose_k n k =
        let rec factorial n =
            match n with
            | n when n = 0 -> 1
            | n when n = 1 -> 1
            | n -> factorial(n - 1) * n
        (factorial n) / ((factorial k)*(factorial (n - k)))

    [<Property>]
    let ``product elements are unique in each product``() =
        Prop.forAll sets_bigger_than_two (fun xs ->
            List.ofSeq xs.Get
            |> uniqueProducts
            |> Seq.map (fun (e1,e2) -> e1 <> e2)
            |> Seq.reduce (&&))

    //[<Property(Verbose = true)>]
    [<Property>]
    let ``n choose k products produced``() =
        Prop.forAll sets_bigger_than_two (fun xs ->
            let input = xs.Get
            let n = Set.count input
            let k = 2
            let expected = n_choose_k n k
            let result =
```

```
                List.ofSeq input
                |> uniqueProducts
                |> Seq.length
            expected = result)


module Result =
    open Xunit
    open FsUnit.Xunit

    [<Fact>]
    let ``simple success test of flow`` () =
        let result : Result<int,string> = resultFlow {
            let! x = Ok 42
            let! y = Ok 3
            return x + y }
        let expected : Result<int,string> = Ok 45
        result |> should equal expected

    [<Fact>]
    let ``simple fail test of flow`` () =
        let result = resultFlow {
            let! x = Ok 42
            let! y = Error "wrong"
            return x + y }
        let expected : Result<int,string> = Error "wrong"
        result |> should equal expected

    [<Fact>]
    let ``simple success test of sequence`` () =
        let input = [Ok 1; Ok 2; Ok 3]
        let expected = Ok [1;2;3]
        let result = Result<_,_>.sequence input
        result |> should equal expected

    [<Fact>]
    let ``simple fail test of sequence`` () =
        let input = [Ok 1; Error "wrong"; Ok 3]
        let expected : Result<int list,string> = Error "wrong"
        let result = Result<_,_>.sequence input
        result |> should equal expected

    [<Fact>]
    let ``simple success test of traverse`` () =
        let input = [1 .. 5]
        let expected : Result<int list, string> = Ok [1 .. 5]
        let result =
            input
            |> Result<_,_>.traverse
                (function
                 | n when n = 0 -> Error "wrong"
                 | n -> Ok n)
        result |> should equal expected

    [<Fact>]
```

```
    let ``simple fail test of traverse`` () =
        let input = [0 .. 5]
        let expected : Result<int list, string> = Error "wrong"
        let result =
            input
            |> Result<_,_>.traverse
                (function
                    | n when n = 0 -> Error "wrong"
                    | n -> Ok n)
        result |> should equal expected

module maybe =
    open Xunit
    open FsUnit.Xunit

    [<Fact>]
    let ``simple success test of maybe flow`` () =
        let result : int option = maybe {
            let! x = Some 42
            let! y = Some 3
            return x + y }
        let expected : int option = Some 45
        result |> should equal expected

    [<Fact>]
    let ``simple fail test of maybe flow`` () =
        let result : int option = maybe {
            let! x = Some 42
            let! y = None
            return x + y }
        let expected : int option = None
        result |> should equal expected
```

E.4.2 Tests.InterlockingModel.fs

Testing selected functions from the InterlockingModel module.

```
namespace Tests.InterlockingModel
open Utils
open InterlockingModel

module RouteConstruction =
    open FsCheck
    open FsCheck.Xunit

    /// Generate integers bigger than zero
    let int_bigger_than_zero =
        Arb.generate<NonNegativeInt>
        |> Gen.where (fun i -> i.Get > 0)

    /// Generator of random routes
```

```
    let routeGen : Gen<Route> =
        let route s =
            let direction = Arb.generate<RouteDirection>
            let linear_segment id (l : NonNegativeInt) =
                LinearRouteSegment(LinearId id, l.Get)
            let point_segment id p =
                PointRouteSegment(PointId id, p)
            let segment id =
                [ Gen.map (linear_segment id) int_bigger_than_zero
                ; Gen.map (point_segment id) Arb.generate<PointPosition> ]
                |> Gen.oneof
            let segments =
                [for id in [1..s] |> Seq.map string -> segment id]
                |> Gen.sequence
            Gen.map2 (fun s d -> Route(s,d)) segments direction
        Gen.sized route
    let validRouteArb : Arbitrary<Route> =
        Arb.fromGen routeGen
        |> Arb.filter
            (fun (Route(segments,_)) ->
             let first_is_linear =
                 match Seq.head segments with
                 | LinearRouteSegment _ -> true
                 | _ -> false
             let last_is_linear =
                 match Seq.last segments with
                 | LinearRouteSegment _ -> true
                 | _ -> false
             first_is_linear
             && last_is_linear)

    [<Property>]
    let ``route split in half and stitched with stitchRoutePair yields same route`` () =
        Prop.forAll validRouteArb
            (fun route ->
             let (Route(all_segments, dir)) = route
             let half = (List.length all_segments) / 2
             let segments1 = List.take half all_segments
             let segments2 = List.skip (half-1) all_segments
             let route1 = Route(segments1, dir)
             let route2 = Route(segments2, dir)
             let result = stitchRoutePair route1 route2
             match result with
             | Ok result_route -> result_route = route
             | Error msg -> raise (System.ArgumentException(msg)))

    // gen route from layout and verifyRoutes
    // gen length constrained routes and checkLengthConstraints
```

Appendix F

Experiment Scripts

This chapter contains the scripts used to generate the models used as experiments in the Experiments chapter.

F.1 SimpleTwoTrains.fsx

Generates 10 models with an increasing number of stations, where the first model have one station and the last have ten stations.

```
#load "Prelude.fsx"

open InterlockingModel
open ScriptingTools

let basic_layout (names : string list) =
    let [l1; p2; l3; l4; p5; l6] = names
    [ LLinear l1              <+> LPointStem p2
    ; LPointFork(p2, Plus)   <+> LLinear l3
    ; LPointFork(p2, Minus)  <+> LLinear l4
    ; LLinear l3              <+> LPointFork(p5,Plus)
    ; LLinear l4              <+> LPointFork(p5,Minus)
    ; LPointStem p5           <+> LLinear l6 ]

let generateLayout (N : int) : SimpleLayout =
```

```
    let layouts = seq {
        for n in [0..N-1] do
            let start = n * 5 + 1
            let end' = start + 5
            let layout = [start .. end']
                         |> List.map string
                         |> basic_layout
            yield layout }
    List.ofSeq layouts
    |> List.concat

let generateRoute layout_map (segment : SimpleTrackSegment) (pos :
    PointPosition) =
    let rec loop segment = seq {
        yield segment
        let next_segment = layout_map |> Map.tryFind segment
        match next_segment with
        | Some segment' ->
            match segment' with
            | LPointStem n -> yield! loop (LPointFork(n, pos))
            | LPointFork(n,_) -> yield! loop (LPointStem n)
            | _ -> yield! loop segment'
        | None -> () }
    loop segment

let generateTrains (t1_length : int) (t2_length : int) (layout :
    SimpleLayout) : SimpleTrains =
    let layout_map = Map.ofList layout
    let route1 =
        generateRoute layout_map (LLinear "1") Plus
        |> List.ofSeq
        |> List.map (function
                     | LLinear n -> RLinear(n, t1_length)
                     | LPointStem n | LPointFork(n,_) -> RPoint(n,
                         Plus))
    let route2 =
        generateRoute layout_map (LLinear "4") Minus
        |> List.ofSeq
        |> List.map (function
                     | LLinear n -> RLinear(n, t2_length)
                     | LPointStem n | LPointFork(n,_) -> RPoint(n,
                         Minus))
        |> List.rev
    [ {id = "1"; length = t1_length; route = route1;
        route_direction = Up}
    ; {id = "2"; length = t2_length; route = route2;
        route_direction = Down} ]

for i in [1..10] do
    let layout = generateLayout i
    let trains = generateTrains 2 2 layout
    let output_file = sprintf
        "../../UMCModels/SimpleTwoTrains/model%i.umc" i
    generateUMCModel { trains = trains
                     ; layout = CustomLayout layout
```

```
                        ; show_stats = true
                        ; output_file = Some output_file }
```

F.2 BranchManyTrains.fsx

Generates four models with an increasing number of trains, where the first model have two trains and the last have four trains.
(The script can easily be adjusted to generate more models with more trains. However, as already mentioned in the Experiments chapter, a model even with just four trains can take many hours to model check)

```
#load "Prelude.fsx"

open InterlockingModel
open ScriptingTools

let branchLayout xs =
    let rec loop xs  = seq {
        match xs with
        | [x;y] ->
            let prev = x + y
            let prev_id = string prev
            yield LPointFork(prev_id, Plus) , LLinear (string x)
            yield LPointFork(prev_id, Minus), LLinear (string y)
        | xs when List.length xs > 2 ->
            let current_id = xs |> Seq.sum |> string
            let half = (List.length xs) / 2
            let xs_left = xs |> List.take half
            let xs_right = xs |> List.skip half
            let stem_left_id = xs_left |> Seq.sum |> string
            let stem_right_id = xs_right |> Seq.sum |> string

            yield LPointFork(current_id, Plus), LPointStem
                stem_left_id
            yield! loop xs_left
            yield LPointFork(current_id, Minus), LPointStem
                stem_right_id
            yield! loop xs_right
        | _ -> () }
    let tail = xs |> loop |> List.ofSeq
    let stem_id = xs |> Seq.sum |> string
    let head = LLinear "0", LPointStem stem_id
    head :: tail

let getRouteTrace layout_map start end' =
    let rec loop prev route =
        match layout_map |> Map.tryFind prev with
        | Some(LLinear n) when (LLinear n) = (LLinear end') ->
            RLinear(n, 2) :: route
        | Some(LLinear n) ->
```

```
            (RLinear(n,2) :: route)
            |> loop (LLinear n)
        | Some(LPointStem n) ->
            let result_left =
                //(LPointFork(n, Plus) :: route) |> loop
                (RPoint(n, Plus) :: route)
                |> loop (LPointFork(n, Plus))
            let result_right =
                //(LPointFork(n, Minus) :: route) |> loop
                (RPoint(n, Minus) :: route)
                |> loop (LPointFork(n, Minus))
            match result_left, result_right with
            | x::xs, [] -> x::xs
            | [], y::ys -> y::ys
            | _ -> []
        | Some(LPointFork(n,pos)) ->
            //(LPointStem n :: route) |> loop
            (RPoint(n,pos) :: route)
            |> loop (LPointStem n)
        | None -> []
    loop (LLinear start) [RLinear(start, 2)]

let dualBranch (levels : int) (num_trains : int) =
    let size = int(2.0**(float levels))
    let [xs; ys] = [1..2*size] |> List.chunkBySize size
    let right = branchLayout ys
    let left = branchLayout xs |> List.map (fun (x,y) -> y,x)
    let layout = [left; right] |> List.concat
    let layout_map = Map.ofList layout
    let getRouteInLayout = getRouteTrace layout_map
    let trains =
        List.zip xs ys
        |> List.map
            (fun (l1,l2) ->
             let lin1 = string l1
             let lin2 = string l2
             let route = getRouteInLayout lin1 lin2
             {id=lin1; length=2; route=route;
                 route_direction=Down})
        |> List.take num_trains
        |> List.ofSeq
    layout, trains

for i in [2..4] do
    let layout, trains = dualBranch 2 i
    let output_file = sprintf
        "../../UMCModels/ManyTrains2/model%i.umc" i
    generateUMCModel { trains = trains
                     ; layout = CustomLayout layout
                     ; show_stats = true
                     ; output_file = Some output_file }
```

Appendix G

XML sample

This chapter contains an example of a concrete XML file which is used by the developed system.
The file is referred to as *sample.xml* because it is used by the F# type provider to bootstrap the knowledge about the types stemming from XML files with similar layouts.

The XML file was originally created by Linh H. Vu, and has been obtained from the RobustRailS research project[Col, Hax] repository.

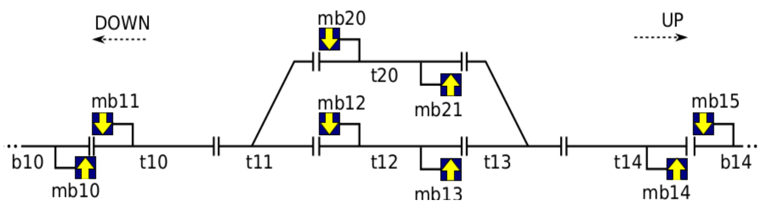

Figure G.1: The network represented in the XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.4.1"
    xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
<xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
```

```xml
<interlocking id="mini" version="0.1">
    <network id="mininetwork">
        <trackSection id="b10" length="100" type="linear">
            <neighbor ref="t10" side="up"/>
        </trackSection>
        <trackSection id="t10" length="87" type="linear">
            <neighbor ref="b10" side="down"/>
            <neighbor ref="t11" side="up"/>
        </trackSection>
        <trackSection id="t11" length="26" pointMachine="spskt11" type="point">
            <neighbor ref="t10" side="stem"/>
            <neighbor ref="t12" side="plus"/>
            <neighbor ref="t20" side="minus"/>
        </trackSection>
        <trackSection id="t12" length="3783" type="linear">
            <neighbor ref="t11" side="down"/>
            <neighbor ref="t13" side="up"/>
        </trackSection>
        <trackSection id="t13" length="81" pointMachine="spskt13" type="point">
            <neighbor ref="t12" side="plus"/>
            <neighbor ref="t20" side="minus"/>
            <neighbor ref="t14" side="stem"/>
        </trackSection>
        <trackSection id="t14" length="128" type="linear">
            <neighbor ref="t13" side="down"/>
            <neighbor ref="b14" side="up"/>
        </trackSection>
        <trackSection id="b14" length="128" type="linear">
            <neighbor ref="t14" side="down"/>
        </trackSection>
        <trackSection id="t20" length="76" type="linear">
            <neighbor ref="t11" side="down"/>
            <neighbor ref="t13" side="up"/>
        </trackSection>
        <markerboard distance="50" id="mb10" mounted="up" track="b10"/>
        <markerboard distance="50" id="mb11" mounted="down" track="t10"/>
        <markerboard distance="50" id="mb12" mounted="down" track="t12"/>
        <markerboard distance="50" id="mb13" mounted="up" track="t12"/>
        <markerboard distance="50" id="mb14" mounted="up" track="t14"/>
        <markerboard distance="50" id="mb15" mounted="down" track="b14"/>
        <markerboard distance="50" id="mb20" mounted="down" track="t20"/>
        <markerboard distance="50" id="mb21" mounted="up" track="t20"/>
    </network>
<routetable id="miniroutetable" network="mininetwork">
  <route id="r_1a" source="mb10" destination="mb13" dir="up">
```

```
    <condition type='point' val='plus' ref='t11'/>
    <condition type='point' val='minus' ref='t13'/>
    <condition type='signal' ref='mb11'/>
    <condition type='signal' ref='mb12'/>
    <condition type='signal' ref='mb20'/>
    <condition type='trackvacancy' ref='t10'/>
    <condition type='trackvacancy' ref='t11'/>
    <condition type='trackvacancy' ref='t12'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_7_'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_5a'/>
    <condition type='mutualblocking' ref='r_2a'/>
    <condition type='mutualblocking' ref='r_1b'/>
    <condition type='mutualblocking' ref='r_3_'/>
    <condition type='mutualblocking' ref='r_2b'/>
    <condition type='mutualblocking' ref='r_4_'/>
  </route>
  <route id="r_1b" source="mb10" destination="mb13" dir="up">
    <condition type='point' val='plus' ref='t11'/>
    <condition type='signal' ref='mb11'/>
    <condition type='signal' ref='mb12'/>
    <condition type='signal' ref='mb15'/>
    <condition type='signal' ref='mb20'/>
    <condition type='signal' ref='mb21'/>
    <condition type='trackvacancy' ref='t10'/>
    <condition type='trackvacancy' ref='t11'/>
    <condition type='trackvacancy' ref='t12'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_2b'/>
    <condition type='mutualblocking' ref='r_6a'/>
    <condition type='mutualblocking' ref='r_2a'/>
    <condition type='mutualblocking' ref='r_8_'/>
    <condition type='mutualblocking' ref='r_3_'/>
    <condition type='mutualblocking' ref='r_7_'/>
    <condition type='mutualblocking' ref='r_5a'/>
    <condition type='mutualblocking' ref='r_1a'/>
  </route>
  <route id="r_2a" source="mb10" destination="mb21" dir="up">
    <condition type='point' val='minus' ref='t11'/>
    <condition type='point' val='plus' ref='t13'/>
    <condition type='signal' ref='mb11'/>
    <condition type='signal' ref='mb12'/>
    <condition type='signal' ref='mb20'/>
    <condition type='trackvacancy' ref='t10'/>
    <condition type='trackvacancy' ref='t11'/>
    <condition type='trackvacancy' ref='t20'/>
    <condition type='mutualblocking' ref='r_6a'/>
    <condition type='mutualblocking' ref='r_7_'/>
    <condition type='mutualblocking' ref='r_3_'/>
    <condition type='mutualblocking' ref='r_2b'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_8_'/>
```

```
    <condition type='mutualblocking' ref='r_1a'/>
    <condition type='mutualblocking' ref='r_1b'/>
  </route>
  <route id="r_2b" source="mb10" destination="mb21" dir="up">
    <condition type='point' val='minus' ref='t11'/>
    <condition type='signal' ref='mb11'/>
    <condition type='signal' ref='mb12'/>
    <condition type='signal' ref='mb13'/>
    <condition type='signal' ref='mb15'/>
    <condition type='signal' ref='mb20'/>
    <condition type='trackvacancy' ref='t10'/>
    <condition type='trackvacancy' ref='t11'/>
    <condition type='trackvacancy' ref='t20'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_7_'/>
    <condition type='mutualblocking' ref='r_5a'/>
    <condition type='mutualblocking' ref='r_3_'/>
    <condition type='mutualblocking' ref='r_4_'/>
    <condition type='mutualblocking' ref='r_6a'/>
    <condition type='mutualblocking' ref='r_1b'/>
    <condition type='mutualblocking' ref='r_2a'/>
    <condition type='mutualblocking' ref='r_1a'/>
  </route>
  <route id="r_3_" source="mb12" destination="mb11" dir="down">
    <condition type='point' val='plus' ref='t11'/>
    <condition type='signal' ref='mb10'/>
    <condition type='signal' ref='mb20'/>
    <condition type='trackvacancy' ref='t11'/>
    <condition type='trackvacancy' ref='t10'/>
    <condition type='mutualblocking' ref='r_5a'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_7_'/>
    <condition type='mutualblocking' ref='r_2a'/>
    <condition type='mutualblocking' ref='r_1b'/>
    <condition type='mutualblocking' ref='r_2b'/>
    <condition type='mutualblocking' ref='r_1a'/>
  </route>
    <route id="r_4_" source="mb13" destination="mb14" dir="up">
    <condition type='point' val='plus' ref='t13'/>
    <condition type='signal' ref='mb15'/>
    <condition type='signal' ref='mb21'/>
    <condition type='trackvacancy' ref='t13'/>
    <condition type='trackvacancy' ref='t14'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_5a'/>
    <condition type='mutualblocking' ref='r_6a'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_8_'/>
    <condition type='mutualblocking' ref='r_1a'/>
    <condition type='mutualblocking' ref='r_2b'/>
  </route>
  <route id="r_5a" source="mb15" destination="mb12" dir="down">
    <condition type='point' val='minus' ref='t11'/>
    <condition type='point' val='plus' ref='t13'/>
```

```
    <condition type='signal' ref='mb13'/>
    <condition type='signal' ref='mb14'/>
    <condition type='signal' ref='mb21'/>
    <condition type='trackvacancy' ref='t14'/>
    <condition type='trackvacancy' ref='t13'/>
    <condition type='trackvacancy' ref='t12'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_6a'/>
    <condition type='mutualblocking' ref='r_8_'/>
    <condition type='mutualblocking' ref='r_3_'/>
    <condition type='mutualblocking' ref='r_1a'/>
    <condition type='mutualblocking' ref='r_4_'/>
    <condition type='mutualblocking' ref='r_2b'/>
    <condition type='mutualblocking' ref='r_1b'/>
  </route>
  <route id="r_5b" source="mb15" destination="mb12" dir="down">
    <condition type='point' val='plus' ref='t13'/>
    <condition type='signal' ref='mb10'/>
    <condition type='signal' ref='mb13'/>
    <condition type='signal' ref='mb14'/>
    <condition type='signal' ref='mb20'/>
    <condition type='signal' ref='mb21'/>
    <condition type='trackvacancy' ref='t14'/>
    <condition type='trackvacancy' ref='t13'/>
    <condition type='trackvacancy' ref='t12'/>
    <condition type='mutualblocking' ref='r_7_'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_6a'/>
    <condition type='mutualblocking' ref='r_8_'/>
    <condition type='mutualblocking' ref='r_1a'/>
    <condition type='mutualblocking' ref='r_2b'/>
    <condition type='mutualblocking' ref='r_1b'/>
    <condition type='mutualblocking' ref='r_4_'/>
    <condition type='mutualblocking' ref='r_5a'/>
    <condition type='mutualblocking' ref='r_2a'/>
  </route>
  <route id="r_6a" source="mb15" destination="mb20" dir="down">
    <condition type='point' val='plus' ref='t11'/>
    <condition type='point' val='minus' ref='t13'/>
    <condition type='signal' ref='mb13'/>
    <condition type='signal' ref='mb14'/>
    <condition type='signal' ref='mb21'/>
    <condition type='trackvacancy' ref='t14'/>
    <condition type='trackvacancy' ref='t13'/>
    <condition type='trackvacancy' ref='t20'/>
    <condition type='mutualblocking' ref='r_8_'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_7_'/>
    <condition type='mutualblocking' ref='r_2a'/>
    <condition type='mutualblocking' ref='r_1b'/>
    <condition type='mutualblocking' ref='r_4_'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_5a'/>
    <condition type='mutualblocking' ref='r_2b'/>
```

```xml
  </route>
  <route id="r_6b" source="mb15" destination="mb20" dir="down">
    <condition type='point' val='minus' ref='t13'/>
    <condition type='signal' ref='mb10'/>
    <condition type='signal' ref='mb12'/>
    <condition type='signal' ref='mb13'/>
    <condition type='signal' ref='mb14'/>
    <condition type='signal' ref='mb21'/>
    <condition type='trackvacancy' ref='t14'/>
    <condition type='trackvacancy' ref='t13'/>
    <condition type='trackvacancy' ref='t20'/>
    <condition type='mutualblocking' ref='r_8_'/>
    <condition type='mutualblocking' ref='r_3_'/>
    <condition type='mutualblocking' ref='r_5a'/>
    <condition type='mutualblocking' ref='r_2b'/>
    <condition type='mutualblocking' ref='r_1b'/>
    <condition type='mutualblocking' ref='r_4_'/>
    <condition type='mutualblocking' ref='r_1a'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_6a'/>
    <condition type='mutualblocking' ref='r_2a'/>
  </route>
  <route id="r_7_" source="mb20" destination="mb11" dir="down">
    <condition type='point' val='minus' ref='t11'/>
    <condition type='signal' ref='mb10'/>
    <condition type='signal' ref='mb12'/>
    <condition type='trackvacancy' ref='t11'/>
    <condition type='trackvacancy' ref='t10'/>
    <condition type='mutualblocking' ref='r_2a'/>
    <condition type='mutualblocking' ref='r_1a'/>
    <condition type='mutualblocking' ref='r_2b'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_1b'/>
    <condition type='mutualblocking' ref='r_3_'/>
    <condition type='mutualblocking' ref='r_6a'/>
  </route>
  <route id="r_8_" source="mb21" destination="mb14" dir="up">
    <condition type='point' val='minus' ref='t13'/>
    <condition type='signal' ref='mb13'/>
    <condition type='signal' ref='mb15'/>
    <condition type='trackvacancy' ref='t13'/>
    <condition type='trackvacancy' ref='t14'/>
    <condition type='mutualblocking' ref='r_6a'/>
    <condition type='mutualblocking' ref='r_6b'/>
    <condition type='mutualblocking' ref='r_1b'/>
    <condition type='mutualblocking' ref='r_5b'/>
    <condition type='mutualblocking' ref='r_4_'/>
    <condition type='mutualblocking' ref='r_5a'/>
    <condition type='mutualblocking' ref='r_2a'/>
  </route>
</routetable>
</interlocking>
</xmi:XMI>
```

Bibliography

[BRA13] EDWIN BRADY. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[CEN11] CENELEC – European Committee for Electrotechnical Standardization. EN 50128:2011 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems, 2011.

[CH11] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm Sigplan Notices*, 46(4):53–64, 2011.

[Col] Multiple Collaborators. About RobustRailS. `http://www.robustrails.man.dtu.dk/About-the-project`. [Online; accessed November-2016].

[Fan12] Alessandro Fantechi. Distributing the Challenge of Model Checking Interlocking Control Tables. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, pages 276–289, 2012.

[Fol15] Andreas Foldager. A graphical domain-specific language for railway interlocking systems, et grafisk domænespecifikt sprog for jernbanesikringsanlæg, 2015.

[Hax] Anne E. Haxthausen. RobustRailS research project. `http://www.imm.dtu.dk/~aeha/RobustRailS/index/`. [Online; accessed November-2016].

[Hax14] Anne E. Haxthausen. An Institution for Imperative RSL Specifications. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software. Essays Dedicated to Kokichi Futatsugi*, number 8373 in Lecture Notes in Computer Science, pages 441–464. Springer, 2014.

[HP00] Anne E. Haxthausen and Jan Peleska. Formal Development and Verification of a Distributed Railway Control System. *IEEE Transaction on Software Engineering*, 26(8):687–701, 2000.

[maa] Open Source multiple authors. Fscheck. `https://fscheck.github.io/FsCheck/`. [Online; accessed 17-October-2016].

[mab] Open Source multiple authors. Fsharp.data. `http://fsharp.github.io/FSharp.Data`. [Online; accessed 17-October-2016].

[mac] Open Source multiple authors. Nuget. `https://www.nuget.org/`. [Online; accessed 17-October-2016].

[mad] Open Source multiple authors. Xunit - testing framework. `https://github.com/xunit/xunit`. [Online; accessed 17-October-2016].

[Maza] Franco Mazzanti. UMC 3.3 User Guide. `http://fmt.isti.cnr.it/umc/V4.2/umc.html`. [Online; accessed 10-October-2016].

[Mazb] Franco Mazzanti. Umc web tool. `http://fmt.isti.cnr.it/umc/V4.2/umc.html`.

[Maz09] Franco Mazzanti. Designing uml models with umc. Technical report, Technical report, Technical Report 2009-TR-43, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, 2009.

[mInB16] Danish magazine "Ingeniøren" and German newspaper "Bild". "forkert nødopkald fra togleder var skyld i fatal tysk togulykke", 2016. `http://ing.dk/artikel/forkert-noedopkald-fra-togleder-var-skyld-i-fatal-tysk-togulykke-183119` and `http://www.bild.de/news/inland/zugunglueck-bad-aibling/der-tragische-zweite-fehler-des-fahrdienstleiters-45` `bild.html`, accessed: March 2016.

[Pao10] Marco Paolieri. Modellazione di un sistema di interlocking distribuito tramite lo strumento umc. 2010.

[PGS16] Tomas Petricek, Gustavo Guerra, and Don Syme. Types from data: Making structured data first-class citizens in f#. 2016.

[PS14] Tomas Petricek and Don Syme. The f# computation expression zoo. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8324:33–48, 2014.

[TV09] Gregor. Theeg and Sergej. Vlasenko. *Railway signalling and interlocking : international compendium.* Eurailpress, 2009.

[VHP16] Linh Hong Vu, Anne E. Haxthausen, and Jan Peleska. Formal modelling and verification of interlocking systems featuring sequential release. *Science of Computer Programming*, pages –, 2016.

[Wika] Wikipedia. Mars climate orbiter. https://en.wikipedia.org/wiki/Mars_Climate_Orbiter.

[wikb] wikipedia. Monads. https://en.wikipedia.org/wiki/Monad_(functional_programming)#Continuation_monad.

[Wikc] Wikipedia. Therac-25. https://en.wikipedia.org/wiki/Therac-25.