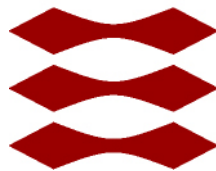


Formal Modelling and Verification of Railway Time Tables

Kristian Hede s062378

DTU



Kongens Lyngby 2012
IMM-MSc-2012-71

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-MSc-2012-71

Summary (English)

The goal of the thesis is to investigate how formal methods can be used to verify and create railway timetables.

First a formal model of railway networks and timetables in RSL is created. Based on the formal model of RSL, a model in UPPAAL is created, which is able to verify properties of existing timetables. A model in UPPAAL CORA is then created, which is able to generate timetables, which satisfy the same properties as those of the UPPAAL model.

Lastly a tool written in Java is created, which provides a graphical user interface for creating timetables. The tool uses UPPAAL CORA and the created model for creating timetables, and is able to visualize results. This tool is considered a prototype, and is an example of how formal methods can be used to create timetables.

Summary (Danish)

Målet med denne thesis er at undersøge hvordan formelle metoder kan bruges til at verificere og generere køreplaner for jernbanedrift.

Først bliver en formel model af jernbanenetværk og køreplaner i RSL lavet. Baseret på den formelle model i RSL, er en model i UPPAAL blevet lavet, som er i stand til at verificere egenskaber i eksisterende køreplaner. En model i UPPAAL CORA er herefter lavet, som er i stand til at generere køreplaner, som kan opfylde de samme egenskaber som dem i UPPAAL modellen.

Til sidst er et værktøj i Java lavet, som giver en grafisk brugergrænseflade til lave køreplaner. Værktøjet benytter sig en UPPAAL CORA, samt den tilhørende model til at generere køreplaner, og det er i stand til at visualisere resultaterne. Dette værktøj betragtes som en prototype, og er et eksempel på, hvordan formelle metoder kan bruges til at generere køreplaner.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Computer Science and Engineering, with the study line of Software Engineering.

This thesis has been written during the period of January 30th 2012 to July 16th 2012, under the supervision of associate professor Anne Elisabeth Haxthausen and associate professor Alex Landex and is worth 30 ECTS credits.

The thesis consists of the following written report, with an attached CD, containing the created models, the created tool and the report and figures of the report.

All of the figures in this thesis have been created by the author Kristian Hede, unless otherwise noted.

Lyngby, 16-July-2012

Kristian Hede s062378

Acknowledgements

I would like to thank my supervisor **Anne E. Haxthausen** for choosing me for this project, and being a great supervisor who always found the time to guide me when a needed it. Your high motivation during the development of this thesis had a very positive effect on me. You were always good at providing constructive feedback and you were very structured in your role as a supervisor. Having you as a supervisor has certainly been one of my best work experiences at DTU.

I would also like to thank my second supervisor **Alex Landex** for being a part of this thesis. You have provided a very solid foundation for knowledge regarding railway operations, and you have skillfully provided feedback on the railway domain.

Finally, I would like to thank my family and friends, especially my father **Henrik O. Hede** for helping by proofreading the report, and **Diana J. Sommer** for being supportive during the development of this thesis.

Contents

Summary (English)	i
Summary (Danish)	iii
Preface	v
Acknowledgements	vii
1 Introduction	1
1.1 Goals	2
1.2 Thesis Overview	3
2 Domain Description	5
2.1 Basic Terms	6
2.1.1 Station	6
2.1.2 Open Line	7
2.1.3 Train	7
2.1.4 Route	8
2.2 Railway Network	8
2.2.1 Headway Time	9
2.2.2 The Open Lines	11
2.2.3 The Stations	13
2.3 Timetable	13
2.3.1 Passenger Timetable	13
2.3.2 Working Timetable	14
2.4 Trains Running According to a Timetable	16
2.4.1 Delay	16
2.4.2 Handling Delay	17

3	Formal Model in RSL	19
3.1	Utilities of RSL	20
3.1.1	Types	20
3.1.2	Functions - Predicates and Auxilliary Functions	21
3.1.3	Test Cases	21
3.2	Model Overview	22
3.3	Model of Railway Network	22
3.3.1	Railway Network Types	24
3.3.2	Sample Railway Network	25
3.3.3	Considering a Railway Network Valid	27
3.3.4	Railway Network Auxilliary Functions	28
3.3.5	Railway Network Predicates	30
3.4	Model of Timetable	31
3.4.1	Timetable Types	31
3.4.2	Sample Timetables	32
3.4.3	Considering Timetables Valid	34
3.4.4	Timetable Auxilliary Functions	35
3.4.5	Timetable Predicates	38
3.5	Using Test Cases to Validate	46
4	Using UPPAAL To Verify Timetables	49
4.1	Utilities of UPPAAL	50
4.1.1	Clocks/Time in UPPAAL	50
4.1.2	UPPAAL Description Language	50
4.1.3	UPPAAL Model-checker	53
4.2	UPPAAL Model	54
4.2.1	Validating Timetables - The Input	56
4.2.2	Global Declarations	56
4.2.3	The Hurry Template	69
4.2.4	The Train Template	69
4.2.5	System Declarations	80
4.2.6	Optimizations	84
4.3	Getting Results Using The Model-checker	85
5	Using UPPAAL CORA To Generate Timetables	89
5.1	Cost and Remaining of UPPAAL CORA	90
5.2	Optimality in Timetables	91
5.3	UPPAAL CORA Model	93
5.3.1	Timetable Request - The Input	94
5.3.2	Global Declarations	94
5.3.3	Train Template	97
5.3.4	Optimizations	107
5.4	Getting Results Using The Model-checker - The Output	110

6	The Tool	113
6.1	Analysis	113
6.1.1	The Scope of the Tool	114
6.1.2	Create a Railway Network	115
6.1.3	Create Timetables Requests	115
6.1.4	Visualizing Output	116
6.1.5	Using the model-checker of UPPAAL CORA	116
6.1.6	Limiting the UPPAAL CORA Model	116
6.1.7	Regular Features	117
6.1.8	Limiting the Complexity of the Tool	117
6.2	Design	117
6.2.1	Design Pattern	118
6.2.2	Generating Timetables	119
6.3	Implementation	123
6.3.1	Technology	124
6.3.2	The Structure of the Tool	124
6.3.3	The Final Look of the Tool	125
7	Evaluation	131
7.1	Running Time	131
7.1.1	Running Time When Verifying Timetables	132
7.1.2	Running Time When Generating Timetables	132
7.2	The Verifications and the Generated Timetables	133
7.2.1	The Resulting Verifications	133
7.2.2	The Resulting Generated Timetables	133
7.3	Creating a Tool Which Utilizes Formal Methods	134
8	Conclusion	137
8.1	Further Work	138
A	RSL files	141
A.1	RailwayNetwork.rsl	141
A.2	Timetable.rsl	145
A.3	TestCases.rsl	155
B	The UPPAAL CORA models used by the final tool	161
B.1	The full UPPAAL CORA model, used by the tool	161
B.2	The UPPAAL CORA model, used by the tool, excluding station headway times	169
B.3	The UPPAAL CORA model, used by the tool, excluding open line headway times	169
B.4	The UPPAAL CORA model, used by the tool, excluding both headway times	172

C Running Times	175
C.1 Verifier	175
C.2 Generator	179
D Tool User Guide	187
D.1 The Railway Network	187
D.2 Creating the Timetable Requests	189
D.3 Generating Timetables	194
Bibliography	199

Introduction

This thesis is a part of the larger research project RobustRailS¹, including DTU Transport, DTU Management, DTU Fotonik, DTU Informatics, Banedanmark (Rail Net Denmark), Bremen University, Trafikstyrelsen (The Danish Transport Authority), DSB S-tog and DSB. RobustRailS is charged with the purposes of investigating methods, which can be applied to improve the reliability and sustainability of railway operation. This thesis is written for DTU Informatics.

When planning the traffic of a railway network, timetables for every train running on the network are created. The resulting timetables specify how the trains should perform in the railway network. Hence they have a great influence in the reliability and sustainability of railway operations. It is therefore desirable to devote resources towards investigating new methods of verifying and creating these timetables.

Banedanmark is currently experiencing problems with having too many train delays on a daily basis. As a result people are dissatisfied with the train operation, and may consequently stop using public train transportation. It is therefore a high priority of Banedanmark to create timetables, which can decrease the delay on daily train operations. A challenging aspect of timetabling, is to create the timetables, in such a manner that they will become robust against delays, by

¹RobustRailS stands for Robustness in Railway Operations.

being able to eliminate the delay as fast as possible. In order to increase the robustness of timetables, the regularity of the trains can be decreased. This trade off between robustness and regularity should be addressed and incorporated whenever attempting to create timetables.

Currently, there are several different tools on the market for creating and managing timetables, such as RailSys [Lan10] and Train Planning System (TPS) [TPS11]. Neither of these use formal methods in creating timetables, nor is it common practice to apply formal methods when creating timetables for railway networks in other tools.

The purpose of this thesis is to investigate how formal methods can be used to verify and create timetables, and evaluate if formal methods is a practical approach - both with regards to speed and the quality of the results. In order to investigate this, several timetabling models are created, which can be used to verify and create timetables, utilizing formal methods.

1.1 Goals

In order to investigate the use of formal methods in timetabling, the following four goals were given, ending in a tool, which is able to create timetables using formal methods.

1. Create a formal model of a railway network and a collection of timetables in RSL[Gro92], hereby formalising the domain of the problem, and defining the level of details involved in the model.
2. Based of the formal model created in RSL, create a UPPAAL[BDL] model for verifying existing timetables in an existing railway network. The model should take the timetables and railway network as input parameters, such that the same model can be used to verify different timetables in different railway networks.
3. Based on the experiences of step 2, create a model in UPPAAL CORA² for creating a collection of timetables based on a railway network and some requirements of the desired timetables. This model should find the optimal collection of timetables, which also introduces the need to create a definition of optimality in timetables.

²UPPAAL CORA is a branch of UPPAAL, the official description is found at <http://people.cs.aau.dk/adavid/cora/index.html>

4. Finally, create a tool, which is able to take a railway network and some requirements for the timetables as input. Using the model created in step 3, the tool should create the optimal timetables and produce human readable output displaying the timetables.

After these four steps have been completed, an evaluation of the resulting models and the final tool will be presented, discussing the advantages and disadvantages of using formal methods in timetabling, as well as whether or not it is a practical approach, and can be used on real systems.

1.2 Thesis Overview

In chapter 2 the basic domain of timetabling will be presented - explaining the different terms of the domain and their technical meaning.

Chapter 3 will describe the creation of the RSL model of the domain (step 1).

Chapter 4 will describe the creation of the UPPAAL model (step 2).

Chapter 5 will describe the creation of the UPPAAL CORA model, including what is considered an optimal collection of timetables (step 3).

Chapter 6 will describe the creation of the final tool, and how it has used the UPPAAL CORA model, in order to create timetables (step 4).

Chapter 7 will present an evaluation of the use of formal methods in timetabling, based on the experiences of the previous chapters.

Chapter 8 will present the conclusion of the thesis and provide suggestions for future work.

Appendix A contains a print of the created RSL files.

Appendix B contains a print of the UPPAAL CORA model files used by the final tool.

Appendix C contains a presentation of the running times of the final tool.

Appendix D contains a user guide of the final tool.

This thesis is written on the assumptions that the reader has the following prerequisites:

- Knowledge of formal methods.
- A basic understanding of finite state-machines.
- A basic understanding of programming.

CHAPTER 2

Domain Description

Terms in timetabling and railway theory in general, vary from place to place. In Europe, some definitions are used, whereas in North America, other definitions are used [HP08]. The definitions of the terms used in this thesis are mainly based on [HP08] and [Lan11], which use the European definitions.

Section 2.1 introduces the common terms basic to railway timetabling.

Section 2.2 describes the abstraction level taken when representing a railway network.

Section 2.3 Gives a definition of what a timetable actually consists of, and the two different kinds of timetables which are utilized.

Section 2.4 Gives a definition of what it means for a train to run according to a timetable. This includes a definition of when a train is delayed, and how a timetable can be constructed, in order to be more robust against delays.

Section 3.4.3 introduces what it takes for a timetable to be considered valid.

2.1 Basic Terms

Many of the terms used when discussing theory of railway timetables, are also used in everyday speech. This can be confusing, as people might have a different understanding of the technical terms used in this thesis. This section will present the definitions of the basic terms of timetabling, which are used in this thesis.

2.1.1 Station

A *station* is an area where the trains are allowed to make a scheduled stop. The purpose of such a stop can vary between the following:

- Embarking and disembarking passengers or freight.
- Waiting for another train to overtake¹.
- Waiting for another train to pass crossing tracks.
- To reverse directions

Due to the fact that passengers should not be made aware of stops based on one of the last three items, there are two different types of stations:

Technical Station is an area at which a train can stop and wait, be overtaken, or reverse - passengers are not allowed to embark or disembark here. This type of station is always accompanied by signals. A technical station cannot also be a passenger station.

Passenger Station is an area at which a train can stop and wait, be overtaken, or reverse - passengers are able to embark and disembark at this type of station. A passenger station cannot also be a technical station.

Stations of both types are given a name in order to identify it. In this thesis the term station will be used when the context does not distinguish between a technical station and a passenger station, otherwise the specific station type will be used.

A station has a set of platform tracks, which identifies the tracks available at a station where a train can stop or pass through. In this thesis, only one train is allowed at a platform track at a time.

¹This requires the station to have more than one track

2.1.2 Open Line

In order to connect stations to the same railway network, tracks are laid between them. A connection of tracks between two stations is called an open line. An *open line* is defined by the two stations, which represent the two end points of the open line.

It should be noted that an open line may consist of several tracks - hereby causing the open line to be able to handle more trains, than if the open line mere contained a single track.

2.1.3 Train

In Europe, a train is defined as a vehicle scheduled to run on the open lines of a railway network[HP08].

A train is either selfpropelled or it has a locomotive. A selfpropelled train is also called a train unit, and can be attached to other train units. A locomotive contains the engine which drives a train which is not selfpropelled. A locomotive is considered a train on its own, but it will most likely have several carts attached - be they passenger coaches or freight wagons. The term train covers both the locomotive, the train units and the carts².

The purpose of a train is to carry passengers or freight, from one station to another. In order to do so, they utilize the open lines between stations, which are their means of transportation.

When generating and verifying timetables, the different train types available is a factor. A train type dictates the maximum speed, acceleration, capacity (for passengers or freight) and price for a train. It is out of scope for this thesis to take the full impact of these values into account. Therefore this thesis will only consider a single train type, which travels at a constant speed³ and the train capacity and train cost are considered irrelevant.

²Trains are also referred to as rolling stock[Lan11]

³This means no acceleration takes place, which is a simplification, as in this case, trains will not slow down before stopping, and will not speed up when starting, it will simply start and stop instantly.

2.1.4 Route

A route defines the open lines a train should traverse, along with the stations where the train will stop during the journey through a route. In effect, a route is an ordered list of stations, which a train is assigned to. Figure 2.1 shows Nærumbanen[Lok] displayed as a route (the x's mark conditional stops⁴).

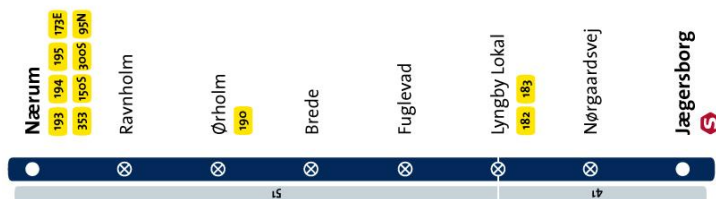


Figure 2.1: Nærumbanen displayed as a route. The image is from <http://www.lokalbanen.dk>, March 13, 2012

Nærumbanen has a single technical station between Jægersborg and Nørgaardsvej, where the trains go for repairs or when they are idle. This station is called *Remisen*. The route displayed in figure 2.1 is meant for passengers to read, hence technical stations are omitted.

2.2 Railway Network

The Danish railway network consists of several smaller railway networks, varying in size and usage. A company called Lokalbanen⁵ administrates several small railway networks, including the local railway network Nærumbanen. Nærumbanen consists of eight passenger stations and one technical station. These stations are connected by eight open lines - this railway network is a real working network, and has therefore acted as a sample railway network during this thesis. In order to test more elaborate railway networks, more complicated ones have been created and used.

A railway network consists of a set of stations, which are connected by open lines. Both the stations and open lines utilize a value called *headway time*.

⁴A train will not stop at a conditional stop if no passengers have marked they will exit there, and no passengers are waiting to get on.

⁵<http://www.lokalbanen.dk>

2.2.1 Headway Time

A station and an open line needs to have time to prepare for an incoming train, therefore it is required that a certain amount of time passes between trains entering a station, or entering an open line. Furthermore, it is important for trains to be separated by a certain time, due to the fact that they can have a braking distance of several kilometers - which they cannot always see ahead. This minimum time required to be between trains at all time is referred to as the *minimum headway time*.

It is possible to add a buffer time to the minimum headway time, resulting in a total headway time, simply referred to as headway time. The main reason for adding a buffer time, is to increase the punctuality of a timetable.

Headway times can be added to either stations or open lines, and the precise definitions can vary from country to country. The definitions presented here are based on what is deemed as relevant within the scope of this thesis to verify. More complex definitions exist, and are explained in [HP08].

2.2.1.1 Headway Time for Stations

The headway time of a station defines the amount of time required to pass between any two trains arriving at the station. The motivation for this headway time is to allow the station to prepare for incoming trains.

Figure 2.2⁶ shows an open line from station A to station B, where two trains travel from station A to station B. The difference in the two arrival times at station B, is called the *arrival-arrival* time.

When looking at a plan on a graph like figure 2.2, the arrival-arrival time must always be greater than or equal to the headway time.

2.2.1.2 Headway Time for Open Lines

The headway time of an open line defines the minimum amount of time required to pass between any two trains arriving at an open line, and any two trains leaving the open line.

⁶In a graph like these, the vertical line below a station, represents time, and the horizontal lines connecting the vertical lines, are trains travelling between the stations. When the horizontal line connect with the vertical lines, it means that the train is at a station.

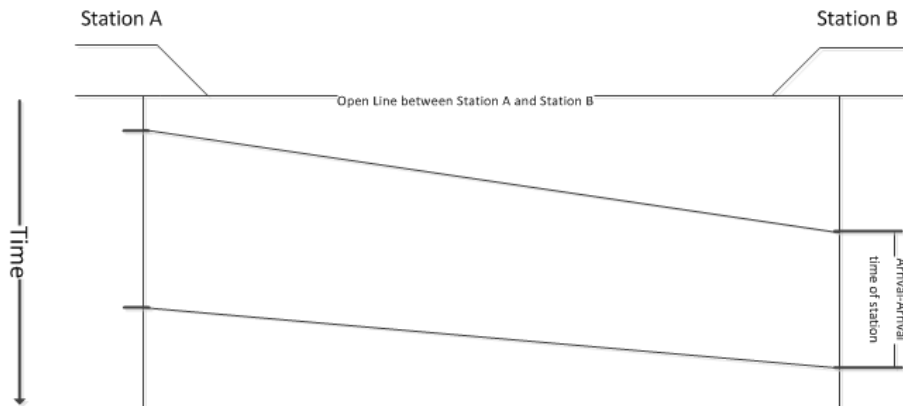


Figure 2.2: The Arrival-Arrival time at a station must be greater than or equal to the headway time of that station.

As with the station, these two times have a name - the *arrival-arrival* time and the *exit-exit*⁷ time of an open line. There are multiple motivational factors for taking such a value into account:

- As with a station, the headway time of an open line makes sure that an open line has time to prepare for a train to enter. This reason is guided more towards the motivation for adding the arrival-arrival time.
- For this thesis, trains are assumed to run at a constant speed between stations. This means that for a train going from station A to B, the line representing it in a graph like figure 2.3, will never curve, and it is therefore possible to use the arrival-arrival times and exit-exit times, to state that the closest two trains have been to each other on an open line (measured in time) - is the smallest of these two times. Based on this, the headway time of an open line, also guarantees a minimum distance between two trains, and is therefore a safety property in this aspect as well.

Figure 2.3 shows the arrival-arrival time and exit-exit time on an open line. Both of these times must always be greater than or equal to the headway time of the open line.

⁷It can also be called Departure-Departure time

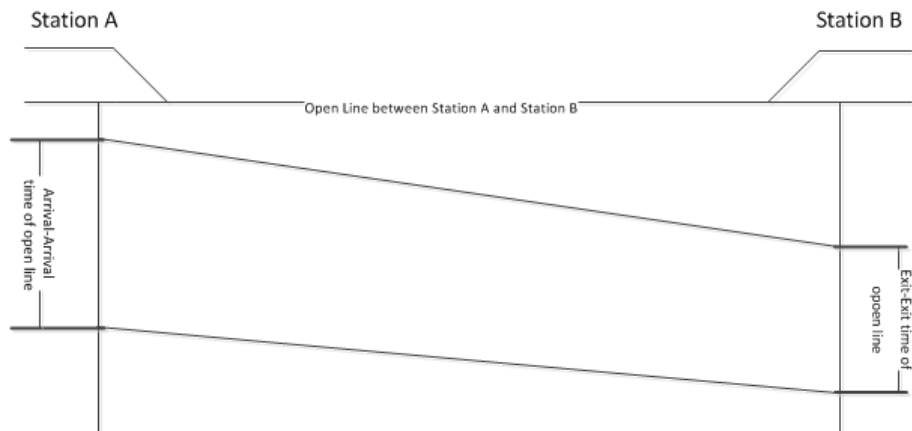


Figure 2.3: The Arrival-Arrival time and Exit-Exit time of an open line must be greater than or equal to the headway time of that open line.

2.2.2 The Open Lines

The open lines of a railway network represent the formation in which the tracks are laid - this can be represented in many different ways. In this thesis, an abstraction level has been chosen, such that the actual physical tracks are irrelevant - it is the fact that two stations are connected, which is considered - these are the open lines of the railway network. These open lines will then contain information required to represent such a connection properly. As a result, a track layout consists of a set of open lines, where each open line contains a value defining whether or not it is a *double track*, the *minimum running time*, the *headway time* and the *capacity* of the open line.

2.2.2.1 Double Tracks and Single Tracks

A track being represented in an open line of this thesis, is always either a double track or a single track. In reality, an open line may have more tracks, but in Denmark open lines with more than two tracks are very rare, hence open lines in this thesis are limited to being either single tracked or double tracked.

The purpose of double tracks is to increase the flexibility of a railway network, by allowing more trains to traverse the open line - hence a double track is most often seen in the busiest open lines of a railway network.

A double track allows for trains to travel both directions of an open line simultaneously, whereas a single track can only be traversed in one direction at a time. In some railway networks, it is possible for both tracks of a double track to be used in the same direction, effectively doubling the capacity of an open line.

For this thesis, it has been chosen that a train is only allowed to travel in the right track of a double track - such that both tracks cannot be used in the same direction. This has been chosen in order to simplify the model of the open lines, as this is not the focus of this thesis.

2.2.2.2 Running Time

The minimum running time of an open line denotes the least amount of time required by a train type to traverse the open line. This value is needed for all types of trains on all open lines.

The minimum running time of an open line, is calculated from the actual length of the track and the top speed at which a train is able and allowed to traverse this track. The minimum running time therefore depends the length of the track, the allowed speed on the track (which can vary from section to section), and the maximum speed of a train type.

It is possible to extend the minimum running time, with a running time supplement - resulting in a total running time. The total running time is simply referred to as running time. The main reason for adding a running time supplement, is to make timetables based on the running times more robust against delays[Lan11].

Due to the fact that this thesis limits itself to a single train type, a single minimum running time for each open line will be defined.

2.2.2.3 Capacity

The capacity of an open line is a value representing how many trains are allowed to be on the open line at the same time.

An open line can be split into smaller segments called *block segments*. This is done by adding signals along the open line. The capacity is based on the amount

of block segments an open line contains - with room for one train in every block segment.

For this thesis, it has been chosen that the block segments and signals along an open line, are represented by the capacity of the open line. In effect, this means that the amount of block sections of an open line is equal to the capacity.

2.2.3 The Stations

The stations of railway networks often contain a signalling system, and a set of platform tracks, where the trains are allowed to stop and take on passengers. At large stations in Denmark⁸, there are possibly dependencies between platform tracks, such as certain platform tracks are only accessible from certain open lines and there may be several different headway times for certain sets of platform tracks at the station.

For this thesis, a station has been simplified, such that each platform track has room for exactly one train, each platform track is accessible from all open lines of the station, and each station has one headway time for the entire station.

2.3 Timetable

A timetable is one of two types, it is either a *passenger timetable*, meant to be read by passengers, or it is a *working timetable*, meant to be read by e.g. train drivers. Both types of timetables describe the details of a route, where the detail level is higher on working timetables than on passenger timetables.

2.3.1 Passenger Timetable

The purpose of a passenger timetable is to inform the passengers of where and when a train will stop for embarking and disembarking on a route. It is common to find passenger timetables either online or at a station. They typically display the following information of a route:

- The name of the route.

⁸Such as 'Hovedbanegården'.

- Crosstrains - At each station, a list of identifiers of other working timetables is written. The identifiers denotes which other working timetables, are planning for a train to be present at the specific station at the same time. This column in the working timetables is disregarded in this thesis⁹.

A unique working timetable is defined for each journey through a route from the first station to the last. This means that for every time a route is initiated, a unique working timetable is specified for the entire journey through that route. Figure 2.5 shows an actual working timetable for NærumBanen.

Jægersborg-Nærum				
Km	Station	2350-1		
		An.	Af.	X-tog
0,5	Remisen			
0	Jægersborg		23:50	2337-2
1,2	Nørgaardsvej	X	23:51	
1,9	Lyngby Lokal	X	23:53	
3,3	Fuglevad	23:55	23:55	
4,5	Brede	X	23:57	
5,5	Ørholm	23:59	00:00	2357-2
6,1	Ravnholm	X	00:01	
7,8	Nærum	00:03		
Kører		Hv undt. lø		

Figure 2.5: Working timetable for NærumBanen. 'An.' is the arrival time, where 'X' marks a conditional stop, 'Af.' is the departure time, 'X-tog' is the crosstrains, '2350-1' is the identifier and 'Hv undt. lø' states that this is for weekdays excluding Saturday. The image is from [Lok]

In working timetables for S-tog, there is an additional value at each stop in the working timetable called *dwell time*. For this thesis, the working timetables will include the scheduled dwell times at each stop.

2.3.2.1 Dwell Time

When trains stop at a station, it is for a reason stated in section 2.1.3. The least amount of time required by a train to hold at a station is referred to as the *minimum dwell time*.

In addition to the minimum dwell time, there is also a dwell time supplement. Dwell time supplement can be added to the minimum dwell time, in order to

⁹Crosstrains are not defined in either [TKO11] or [TKS11].

extend the total dwell time. The total dwell time is simply referred to as dwell time. The main reason for wanting to add a dwell time supplement at a stop is for a train to wait for passengers from a connecting train[Lan11].

2.4 Trains Running According to a Timetable

In order to validate or generate a timetable, it is important to understand what it means for a train to be driven according to a timetable. The basic definition is, that a train arrives and departs at a station, at the given times of the corresponding timetable. Practically, it is necessary to define certain allowed deviations, before classifying a train as being delayed. For example, it is not possible to foresee the exact times, passengers are done embarking and disembarking at a station - this varies from time to time. Therefore creating timetables with a precision level down to seconds, and expecting a train to run according such timetables precisely, is not possible. It should be noted that practically, a train is only classified as delayed, if it departs from a station or arrives at a station, later than a certain time limit after the timetable has stated it to arrive or depart.

For the scope of this thesis, it has been chosen that the verification of a timetable, is based upon the fact that trains run exactly according the timetables. In this thesis, when generating timetables, a certain level of robustness versus delays is introduced, hereby allowing the collection of timetables, to be able to allow for a certain degree of delay.

The following sections will explain the concept of delay, and the different types of delays, and factors regarding delay. Furthermore, it will be presented how timetables can allow for delays.

2.4.1 Delay

A train is considered to be delayed, when it either arrives at a station or departs from a station too much later than scheduled¹⁰. Different types of delays exist, they are divided into two different types of delay[Lan11]:

Initial delay is when a train is delayed for a reason not involved in any other train delays. This type of delay can be caused by passengers embarking

¹⁰The actual limit which defines when a train is delayed varies.

and disembarking, a technical error in the railway network or weather conditions[Lan11]. Figure 2.6 displays an initial delay of a train.

Consecutive delay is a delay caused by the delay of a different train in the network. This sort of delay can happen if a train is delayed on an open line or a platform track, occupying the open line or platform track for longer than scheduled, forcing the next train to wait, and hereby be delayed. Figure 2.7 displays two delayed trains, where the delay of the blue train is a consecutive delay, caused by the initial delay of the red train.

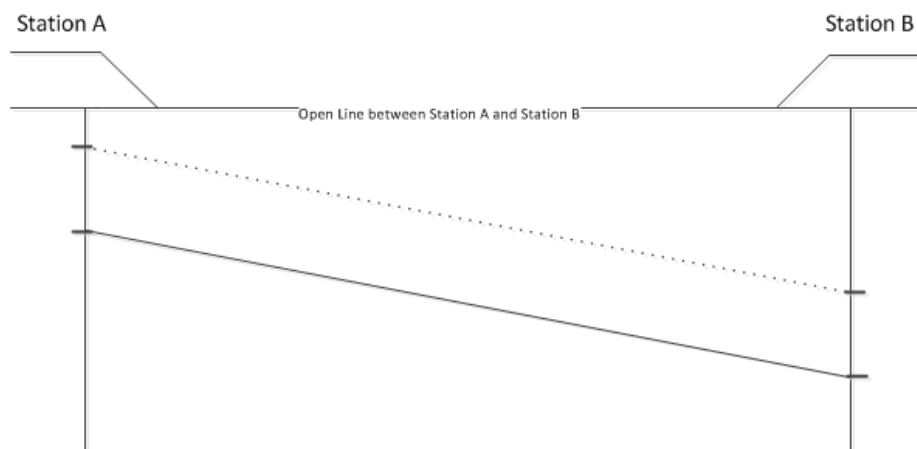


Figure 2.6: The train is delayed for any number of reason not involving any other trains in the railway network, this is called Initial Delay. The dashed line is the scheduled plan for the train, the full line is the performed plan of the train.

In this thesis, the term delay is used when the type of delay is irrelevant, else the type will be stated.

2.4.2 Handling Delay

It is possible to create timetables which take delays into account, thus making timetables more robust against potential delays. This robustness is achieved by adding the values previously mentioned:

Dwell time supplement which can be added to the minimum dwell time.

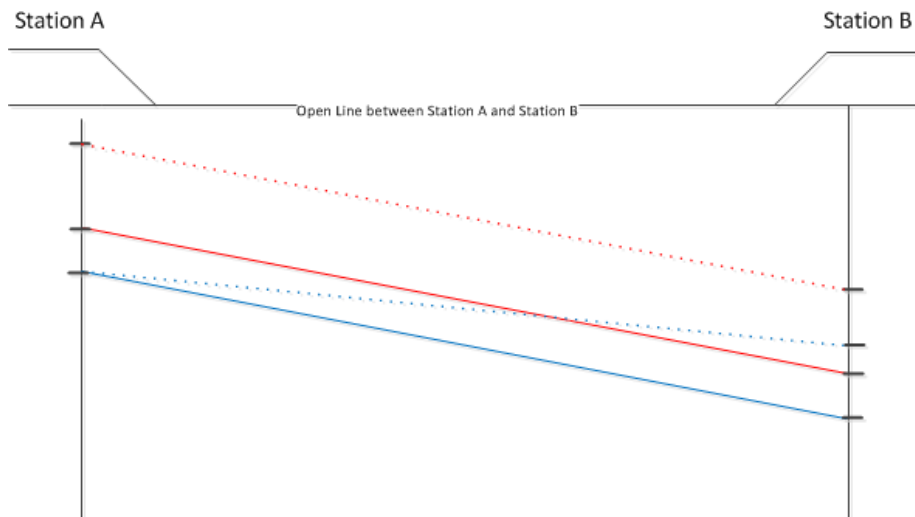


Figure 2.7: The red train is the cause of the initial delay. The blue train is then delayed to the initial delay of the red train. The delay of the blue train is called Consecutive Delay.

Running time supplement which can be added to the minimum running time.

Buffer time which can be added to the minimum headway time.

Increasing either of these three values, allows a train to catch up on delay. The more the values are increased, the more a train can compensate for a delay, hence lowering the effect of consecutive delay on other trains. When increasing these values however, the trains will become less frequent and have longer running times, as the trains will be slowed down by increasing the values.

The process of determining the optimal value for the three parameters, is a weighing of how robust the timetables should be against delays versus the frequency of the trains.

For this thesis, when generating timetables, it is possible to specify the running time supplement to be used in the timetables. The evaluation of the value of the running time supplement has been discussed in [Tho12], and is concluded to be optimal at 7% of the minimum running time of the open line.

Formal Model in RSL

When creating a formal model of railway networks and timetables, it is possible to take different levels of details and specific information into account, e.g. physics in railways (such as torque in turns and abrasion of the tracks or trains), or a country's laws regarding the domain (such as getting permits and upholding legal standards). In order for a model to serve best, a level of abstraction should be chosen, such that the model will be of a size and complexity, reflecting the purpose of the project.

A formal model for the purposes of this thesis, will reflect the railway network and the timetables affiliated with said network. An introduction to the modelling language is first given, a presentation of the model of the railway network is then presented, followed by the model of the timetables, extending the model of the railway network.

In this chapter, section 3.1 will present the utilities of RSL, and their connection with the created model.

Section 3.2 will briefly present a model overview of the final model.

Section 3.3 will present the final railway network model.

Section 3.4 will present the final timetable model.

Finally section 3.5 will explain how the models are validated in RSL.

3.1 Utilities of RSL

This formal model has been developed using RSL[Gro92]¹. RSL contains *types*, which reflect the terms explained in the domain description (section 2). In RSL, it is also possible to create *functions*. In this thesis, functions are used to create *predicates* of the values, and *auxilliary functions* used by the predicates. Predicates are the functions used to state the conditions under which values of the types are considered valid. The formal model in RSL can be seen in appendixes A.2 and A.1.

After the types and predicates were created, *test cases* were created in RSL, in order to test the predicates. A test case can instantiate a value and call a function, with said value. The return value of the function is then calculated, where it is then possible to see if a function returns the expected value. The test cases of the formal model can be seen in appendix A.3.

3.1.1 Types

A type is defined as a collection of logically related values[Gro92], where basic types, such as integers and texts, already exists. Based on the existing basic types, as well as the type operators (set, list, etc.), it is possible to declare new types. These new types can in turn be utilized by other new types, hereby providing the foundation of defining the set of logically related values of a formal model.

When defining types, it is important to have a well defined domain, as the purpose of the types is to reflect the terms of a domain description as much as possible - within the scope of the project. As a result, the types will become more intuitive, often making the formal model less complex and more understandable in general.

¹RAISE Specification Language

3.1.2 Functions - Predicates and Auxilliary Functions

A function in RSL, is defined as a mapping from values of one type to values of another type[Gro92] - meaning that a function will take certain types as parameters, and will return certain types, independent of the parameter types. This thesis uses functions for two different purposes - to state a predicate and to create an auxilliary function.

To create properties of values in the formal model, functions referred to as predicates are used. Stating predicates is a method of stating requirements for the values of the defined types[Gro92]. An example predicate could be *The capacity of a station must be greater than zero*. When creating predicates, it is often an effective approach to make a statement of the domain in plain English², and then proceed to converting said statement into RSL³. In this thesis, a predicate function will always return a boolean value, indicating whether or not the predicate is satisfied.

In order to limit the complexity of the predicate functions, auxilliary functions can be created. These functions serve to split complex functions, into more, less complex functions, effectively making all of the functions easier to understand.

In the RSL descriptions of a railway network and a timetable (appendixes A.1 and A.2), the predicates are prepended with *pred*, whereas the auxilliary functions are not.

3.1.3 Test Cases

Once the types and functions have been created, it is possible to instantiate values to test the predicates by invoking them and checking whether or not they return true. Such an invocation is called a test case. In this thesis, test cases are provided for each of the predicates as well as for the auxilliary functions.

The main motivation for having test cases for auxilliary functions is to ease the debugging of the auxilliary functions during development. In order to get a quick overview of the results of the test cases, it is a possibility to have all the functions return a boolean value, where false would indicate an error. The predicates are defined to always return a boolean, but the auxilliary functions can return any type. Therefore, when a test case is defined for an auxilliary function, an expected result is defined. If the returned value is equal to the

²Or any other spoken language

³Or any other modelling language

expected result, it will show as true, else false. This gives the viewer a quick overview of any detected errors or unexpected results.

When running a test case, it is done by translating the model of the test case into SML[HR99]⁴, and then invoking the functions in SML. Further technical investigation of the process of translating to SML, is not a purpose of this thesis.

3.2 Model Overview

This section will give an overview of the final model in RSL.

The final model consists of two main types and a set of test cases. The two main types are:

RailwayNetwork which represents the railway network, the timetables are operating within. This type is defined in the file `RailwayNetwork.rsl`, and is explained in section 3.3.

TimetableSet which represents the set of timetables to be verified. This type is defined in the file `Timetable.rsl`, and is explained in section 3.4.

The predicates and auxiliary functions for the `TimetableSet`, utilize the `RailwayNetwork` type, which result in the fact that the file containing the `TimetableSet` type, extends the file of `RailwayNetwork`.

The test cases are used to instantiate the predicates, in order to validate timetables in RSL. The RSL file containing the test cases extends the `TimetableSet` (which also provides it with the `RailwayNetwork` type). The figure 3.1 shows the general structure of the model.

3.3 Model of Railway Network

The development of a formal model of a railway network, was based on the necessity to verify and generate timetables. In order for a collection of timetables to be verified or generated, information regarding the open lines, the stations, and the connection of these is required. Based on the domain description, the following items are required to be known of each open line:

⁴Standard ML

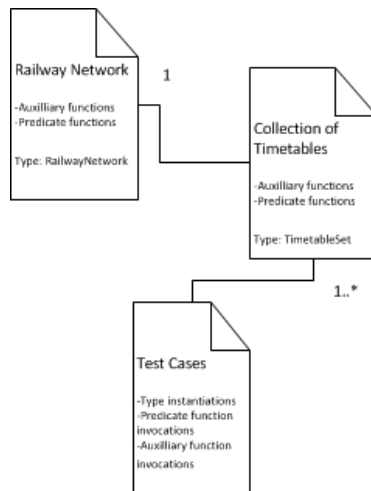


Figure 3.1: The general overview of the RSL model. The functions of the collection of timetables use the railway network, and the test cases uses the collection of timetables (which also provides the test cases with the RailwayNetwork type).

- The two stations which it connects, which can also act as an identifier.
- The minimum running time.
- The capacity.
- The headway time.
- Whether or not it is double or single tracked.

The following items of a station is required:

- An identifier - preferably a name.
- The capacity of the station, which is a result of the amount of platform tracks present at the station.

Based on these values, it is possible to see which stations are connected through which open lines, and it is possible to verify properties regarding the different information associated with the open lines and stations. It should be noted that the two end stations of an open line is used as an identifier - this results in the fact that it is not possible to have several different open lines between the same

two stations. This is justifiable, due to the fact that to have two independent open lines between the same two stations, is very rarely seen, and is therefore disregarded for the purposes of this thesis.

The platform tracks of a station, are replaced by a capacity value, which represents the amount of platform tracks in the station. With this abstraction level, a station can be considered a single entity, where all the platform tracks can be accessed from all of the open lines of the station. This abstraction level has been chosen, due to the fact that the internal structure of a station, and movement inside of a station, is out of scope for this thesis.

The following types are used to define a railway network and is also written in appendix [A.1](#).

3.3.1 Railway Network Types

The types of the railway network are defined based on the domain description (section 2). The following is the RSL specifications of the types of a railway network:

```

type
  Time = Nat,
  Name = Text,
  Capacity = Nat,
  DoubleTrack = Bool,
  MinimumRunningTime = Time,
  HeadwayTime = Time,
  Station = Name,
  OpenLine = Station  $\times$  Station,
  StationTable =
    Station  $\overrightarrow{m}$  (Capacity  $\times$  HeadwayTime),
  OpenLineTable =
    OpenLine  $\overrightarrow{m}$ 
      (DoubleTrack  $\times$  MinimumRunningTime  $\times$  Capacity  $\times$ 
        HeadwayTime),
  RailwayNetwork = StationTable  $\times$  OpenLineTable

```

The following is a description of each of the types:

Name is used as identification. A name consists of a text.

Time is a necessary concept in this abstraction level, as both the length of the open lines and the speed of the train types are implicitly defined in the time it takes to traverse an open line in minutes. Time is a natural number.

Capacity is the capacity of an open line. Capacity is a natural number.

DoubleTrack indicates whether or not an open line is double tracked. It is a Bool value - true if the open line is a double track, false otherwise.

MinimumRunningTime is the minimum running time of an open line, meaning the least amount of time it takes for a train type to traverse the open line. The minimum running time is a Time.

HeadwayTime is the headway time of an open line or a station. A headway time is a Time.

Station is a station of the railway network, both a passenger station and a technical station. A station is a Name.

OpenLine is the identifier of an open line in the railway network. An OpenLine is defined as two stations - the order of which is insignificant.

StationTable is the collection of the stations, their headway times and their capacity. The StationTable maps a Station to a Capacity and a Headway-Time.

OpenLineTable is the collection of the open lines present in the railway network. The OpenLineTable maps an OpenLine to a DoubleTrack, a MinimumRunningTime, a Capacity and a HeadwayTime.

RailwayNetwork is the representation of a railway network. It consists of a StationTable and an OpenLineTable.

The dependency of the types are depicted in figure 3.2.

3.3.2 Sample Railway Network

Figure 3.3 depicts an example railway network, consisting of three passenger stations (A, B and C) and one technical station (D). Stations A and D and stations D and B are connected by a double tracked open line, and stations D and C are connected by a single tracked open line. Stations A, B and C are not directly connected. Each station has two platforms.

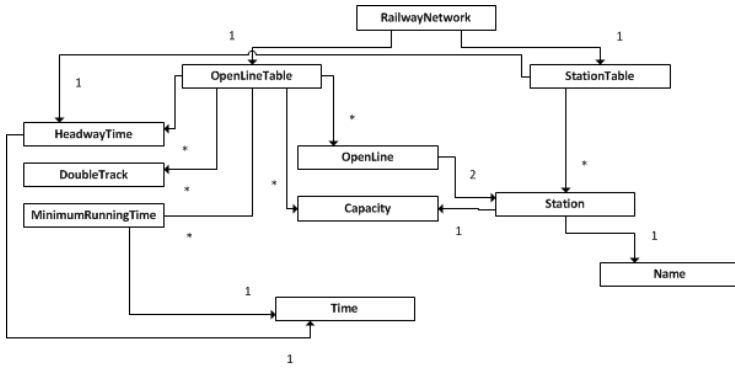


Figure 3.2: The types of the RailwayNetwork

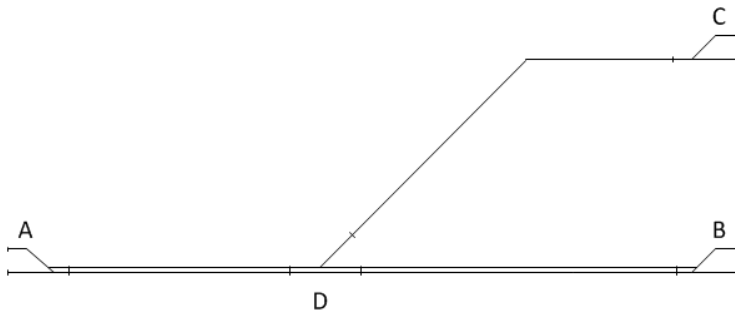


Figure 3.3: A sample railway network of three connected passenger stations (A, B and C) and a technical station (D)

The railway network presented in figure 3.3 is equivalent to the open lines of table 3.1 combined with the station table 3.2

	Capacity	Double Tracks?	Minimum Running Time	Headway Time
A → D	1	true	3	2
D → A	1	true	3	2
B → D	1	true	4	2
D → B	1	true	4	2
C → D	1	false	3	2
D → C	1	false	3	2

Table 3.1: Table containing the open lines of figure 3.3 - their capacity, whether or not it is a double track, their minimum running time and their headway time.

Station	Capacity	Headway Time
A	2	2
B	2	2
C	2	2
D	2	2

Table 3.2: Table containing stations of the railway network from 3.3

The following RSL specifications is the railway network presented in figure 3.3, modelled as the values of the RSL model:

```

test_StationTable : StationTable =
  [ "A" ↦ (2, 2),
    "B" ↦ (2, 2),
    "C" ↦ (2, 2),
    "D" ↦ (2, 2) ]
openLineTable : OpenLineTable =
  [ ("A", "D") ↦ (true, 3, 1, 2),
    ("B", "D") ↦ (true, 4, 1, 2),
    ("C", "D") ↦ (false, 3, 1, 2) ]
railwayNetwork : RailwayNetwork =
  (stationTable, openLineTable)

```

3.3.3 Considering a Railway Network Valid

Now that the types are in order, it is time to consider how to formalize, whether or not a railway network is considered valid. For this thesis, a railway network

is considered valid, if the following two properties hold:

- All of the stations used to define the open lines in the open lines table, must also be defined in the station table.
- All of the stations defined in the station table, must be connected in the same railway network, i.e. it has to be possible to reach any station from any other station, by utilizing the open lines.

If these properties hold, a railway network is considered valid. It should be noted that the example in section 3.3.2 is valid based on these two properties.

Now that the predicates have been expressed in a natural language, they should be translated into RSL, by creating auxilliary functions and predicate functions.

3.3.4 Railway Network Auxilliary Functions

When defining the predicates of the RSL model, several auxilliary functions were developed. The purpose of these functions is to make the predicates more intuitive and smaller, hence most of the auxilliary functions are used by the predicates.

The auxiliary functions not used by the predicates, are used by other auxiliary functions. The purpose for such a function can vary from having to iterate through a collection⁵ and having to prepare a variable for use (often a list or set). The following is a list of the auxilliary functions affiliated with the RSL model of a railway network.

get_OpenLine This is used to retrieve the open line between two stations from the railway network.

- *Station station1* - One station in the desired open line.
- *Station station2* - The other station in the desired open line.
- *RailwayNetwork (stationTable, openLineTable)* - The railway network.

This function checks whether or not (*station1, station2*) is in the domain of the *openLineTable*, if so, this is returned as the open line, else (*station2,*

⁵This is done by recursive functions

station1) is returned. It should be noted that this function is dependant on *pred4_all_routes_of_timetables_can_be_traversed*, because it relies on the fact that the open line between station1 and station2 actually exists. This is reflected in the precondition stating that either (station1, station2) or (station2, station1) is in the domain of the openLineTable.

connect_one_station Connects a station from a set of unconnected stations to a set of connected stations if possible. It has the following parameters:

- *Station-set unconnected* - Is the set of unconnected stations.
- *Station-set connected* - Is the set of connected stations.
- *RailwayNetwork (stationTable, openLineTable)* - The railway network.

The function takes the head of the unconnected stations, tries to connect to the connected stations, based on the railway network. If a station could be connected, it adds the station to the connected set of stations and returns the new connected stations. If no station could be connected, it returns the empty set.

are_all_stations_connected Checks whether or not a set of stations is connected in a railway network. It has the following parameters:

- *Station-set unconnected* - Is the set of stations yet to be connected. Initiated with every station except one.
- *Station-set connected* - Is the set of stations successfully connected so far. Initiated with the station not in unconnected initially.
- *RailwayNetwork (stationTable, openLineTable)* - The railway network.

The function uses *connect_one_station* to connect a station from unconnected to connected one at a time, until the set of connected stations is equal to the domain⁶ of the StationTable of the RailwayNetwork. It returns the boolean value true, if all stations could be connected successfully, else false.

It should be noted that the model for the railway network also contains functions, which return information regarding the station or the open lines. A function exists to get each of the capacity, the headway time, the double track value, and the two stations of an open line. A function for each of the station capacity and station headway time also exists. These functions are trivial and are therefore not explained here, they can be seen in appendix A.1.

⁶The domain of a map in RSL is equivalent to the set of keys of a map

3.3.5 Railway Network Predicates

The following predicates are taken from section 3.3.3, and are translated into RSL in the following predicate functions:

pred All_stations_are_defined The stations used to define the OpenLines must also be defined as a Station.

In the RSL model, this is done by saying that all of the OpenLines in the domain of the OpenLineTable consists of two stations - both being present in the domain of StationTable.

```

pred All_stations_are_defined :
  RailwayNetwork → Bool
pred All_stations_are_defined(
  (stationTable, openLineTable)) ≡
  (∀ (station1, station2) : OpenLine •
    (station1, station2) ∈ dom (openLineTable) ⇒
      station1 ∈ dom (stationTable) ∧
      station2 ∈ dom (stationTable)),

```

pred All_stations_are_connected It must be possible to reach any station in the network, from any other station in the network. This means that all of the stations must be connected in the same network.

In the RSL model, this means there must be OpenLines connecting all of the stations defined in the StationTable. This is checked by using the auxilliary function `are_all_stations_connected`.

```

pred All_stations_are_connected :
  RailwayNetwork → Bool
pred All_stations_are_connected(
  (stationTable, openLineTable)) ≡
  let initial_station = hd (dom (stationTable)) in
  are_all_stations_connected(
    dom (stationTable) \ {initial_station},
    {initial_station}, (stationTable, openLineTable)

```

A consequence of this predicate is also that all stations have to be part of at least one open line.

If these predicates are upheld, the railway network in this formal model is considered to be valid.

3.4 Model of Timetable

The development of a formal model for a timetable, was based on the necessity to verify and generate timetables. In this case, it is a priority to make the timetables reflect as many of the details as possible, which is needed when a train driver has to utilize a timetable. As a result, the abstraction level of the formal model developed of a timetable is quite low, and the model almost stores the same level of details as [TKS11].

The model stores a name of the timetable, which can be set to the name of the train assigned to the timetable. This has value does not present any theoretical restrictions within the scope of this thesis, other than it must be unique.

The specific platform track which the train is supposed to arrive at, at the station, is not represented in this model. As explained in the domain description (section 2), the platform tracks of a station has been simplified, to be represented by a capacity of a station instead, hence the platform tracks in the timetable is omitted.

3.4.1 Timetable Types

The types of a timetable are based on the terms presented in the domain description (section 2). The following is the RSL specifications of the types of a Timetable:

type

```

ArrivalTime = Time,
DepartureTime = Time,
DwellTime = Time,
Route = Station*,
Stop =
    Station × ArrivalTime × DepartureTime × DwellTime,
Timetable = Name × Stop*,
TimetableSet = Timetable-set

```

The following is a list describing each of the types:

ArrivalTime is the time a train arrives at a station.

DepartureTime is the time a train departs from a station.

DwellTime is the scheduled dwell time at a station.

Route is a route through a railway network, denoting the stations a train will pass. A route is an ordered list of Stations.

Stop is the information regarding a particular entry in a timetable. A stop consists of a Station, an ArrivalTime, a DepartureTime and a DwellTime.

Timetable is a representation of a timetable, denoting the name of the timetable along with the information regarding the stops related to the timetable - it is currently the full representation of a working timetable and not a passenger timetable. A Timetable is a Name and an ordered list of Stops.

TimetableSet is the collection of all the timetables for each train affiliated with the railway network. This type is created to ease the development of predicates concerning multiple timetables.

The dependency of the types are depicted in figure 3.4.

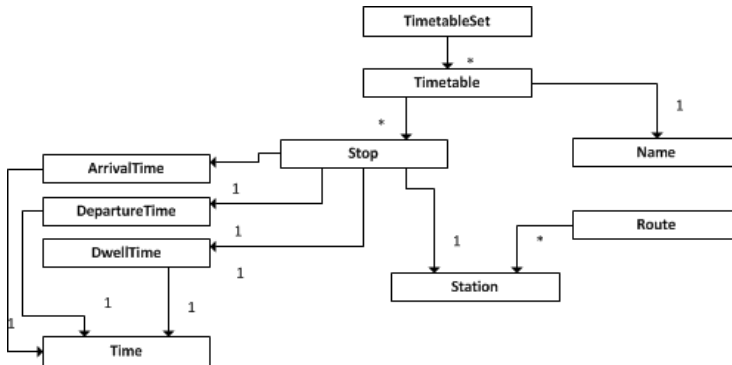


Figure 3.4: The types of the Timetable

3.4.2 Sample Timetables

Tables 3.3 and 3.4, depicts two example timetables, based on figure 3.3.

A-B	Arrival Time	DepartureTime	Dwell Time
A	0	1	0
D	4	4	0
B	14	17	0
D	21	21	0
A	24	24	0

Table 3.3: Sample timetable going from station A to B and back to A, passing through station D, made for the railway network of figure 3.3

C-A	Arrival Time	DepartureTime	Dwell Time
C	0	1	0
D	4	4	0
A	7	10	0
D	13	13	0
C	16	16	0

Table 3.4: Sample timetable going from station C to A and back to C, passing through station D, made for the railway network of figure 3.3

The following is the RSL specifications of the two sample timetables of tables 3.3 and 3.4, which run on the sample railway network of section 3.3.

```
test_TimetableList : TimetableSet =
  {("A-B",
    <<("A", 0, 1, 0),
      ("D", 4, 4, 0),
      ("B", 14, 17, 0),
      ("D", 21, 21, 0),
      ("A", 24, 24, 0)>>),
    ("A-C",
    <<("C", 0, 1, 0),
      ("D", 4, 4, 0),
      ("A", 7, 10, 0),
      ("D", 13, 13, 0),
      ("C", 16, 16, 0)>>)}
}
```

3.4.3 Considering Timetables Valid

For this thesis, when dealing with safety in timetables, aspects such as signal handling, abrasion and human error are not considered. When a collection of timetables is considered valid, it is done based on the fact that trains are running according to the timetables. This means that the validated safety properties considered when generating or verifying a timetable, can only be considered valid, if the trains actually run according to the scheduled times. If for some reason they do not run according to schedule, it will be the safety properties of the signal system, or human influence, which will guarantee the safety of the trains.

There are many different possible properties, which can contribute to the quality of a validation of a collection of timetables. For this thesis, the following eight properties all have to hold, in order for a collection of timetables to be considered valid:

- Trains cannot overtake another train on an open line.
- Trains have to satisfy the minimum running times of the open lines.
- Trains have to satisfy the dwell times at the stations.
- The headway times of stations must be upheld.
- The capacity of stations must never be exceeded.
- The headway times of open lines must be upheld.
- The capacity of the open lines must never be exceeded.
- Single track open lines cannot be utilized in both directions simultaneously.

It should be noted that breaking any of these properties does not necessarily cause an accident to happen. But if any of these properties are broken, it cannot be guaranteed that an accident will not occur. The lack of this guarantee is therefore cause to classify a collection of timetables as invalid.

The sample timetables of section [3.4.2](#) are valid, based on these properties.

Now that the predicates have been expressed in a natural language, they should be translated into RSL, by creating auxiliary functions and predicate functions.

3.4.4 Timetable Auxilliary Functions

The motivations and purposes of the auxilliary functions of the timetable model, are the same as described in the beginning of section 3.3.4. The following is a list of the auxilliary functions developed for the formal model of a timetable in RSL:

get_Route_from_Timetable Constructs the route of a Timetable in an ordered list of Stations. It has one parameter:

- *Timetable (timeTableName, stops)* - Is the Timetable from which a route should be constructed.

The function returns the concatenated list containing the head of the stops in the timetable, with a recursive call to itself - where the tail of stops is the stops of the new timetable parameter - hereby removing the head from the recursive call. The function terminates when there are no more stops left. It returns an ordered list of stations.

get_movements_of_Timetable Constructs the set of every consecutive pair of stops in a Timetable. It has one parameter:

- *Timetable (timeTableName, stops)* - Is the Timetable from which the consecutive pair of stops should be constructed.

The function returns the union between the head of the stops in the timetable and the head of the tail of the stops in the timetable, with a recursive call to itself. The timetable of the recursive call is given the tail of stops of the old timetable - hereby removing the head from the recursive call. The function terminates when there the tail of stops is empty. It returns a set of pair of stops.

is_Route_possible Checks whether or not a route is connected in a railway network. It has the following parameters:

- *Route route* - The route to validate.
- *RailwayNetwork (stationTable, openLineTable)* - The railway network.

This function goes through each consecutive pairs of stations in the route, and checks whether or not the station pair is defined as an open line in the railway network. It returns false if a check fails, and true if all checks succeed.

are_travel_times_possible Checks whether or not the scheduled running times of the open lines of a timetable are too small. It has the following parameters:

- *Timetable* (*timetableName*, *stops*) - The timetable to validate.
- *RailwayNetwork* *railwayNetwork* - The railway network.

This function goes through each consecutive pairs of stations in the timetable. For each pair, it checks whether or not the difference between the departure time of the first stop, and the arrival time of the second stop exceeds the minimum running time of the open line. If all of the pairs exceed the minimum running time, the function returns true, else false.

does_timetable_occupy_open_line_in_time_period_DIRECTED Determines whether or not at a train following a certain timetable will occupy a specific open line in the stated direction in a certain time period. It has the following parameters:

- *Timetable* *timetable* - The timetable to potentially occupy the open line.
- *OpenLine* *openLine* - The open line to check if occupied.
- *Time from* - The beginning of the time period.
- *Time to* - The end of the time period.

This function goes through each pair of consecutive stops of the timetable, and checks whether or not the open line of the two stops is equal to the *openLine* parameter (seeing if the two stops utilize the open line in question). If it is the relevant open line, then it checks whether or not the departure time of the first stop, or the arrival time of the second stop is between the *from* and *to* parameters - meaning that it either departed or arrived in the relevant time period. It also checks if the departure of the first stop is before the *from* parameter, and the arrival time of the second stop is later than the *to* parameter. If any of these checks are true for any of the consecutive pairs of stops - it means that the open line is occupied and it will return true, else false.

is_open_line_occupied_in_time_period_DIRECTED Determines whether or not at least one train occupies an open line in a certain time period. It has the following parameters:

- *OpenLine* *openLine* - The open line to check if occupied.
- *Time from* - The beginning of the time period.
- *Time to* - The end of the time period.
- *TimetableList* *timetableList* - The list of timetables to potentially occupy the open line.

This function goes through each Timetable in the timetableList recursively, and invokes *does_timetable_occupy_open_line_in_time_period_DIRECTED*. It returns the intersection of the return values to the invokes of *does_timetable_occupy_open_line_in_time_period_DIRECTED*.

add_timetable_to_trains_at_station_count Checks whether or not a timetable will schedule a train to be present at a station at a point in time, if so, it adds one to a counter, else it does not add anything. It has the following parameters:

- *Station station* - The station to check for a train at.
- *Time time* - The time to check for a train.
- *Timetable timetable* - The timetable to check for a train.
- *Int count* - The variable keeping track on the amount of trains.

This function goes through each stop in the timetable recursively, and adds 1 to count and returns count if the train is present at the relevant station at the given point in time. If the train is not present at the given station at the given point in time, it returns count unchanged.

all_trains_at_station_count Gets the amount of trains present in a station at a certain time. It has the following parameters:

- *Station station* - The station to count the trains at.
- *Time time* - The time to count trains.
- *TimetableList timetableList* - All of the timetables to count the trains from.
- *Int count* - The variable keeping track on the amount of trains.

This function goes through each Timetable in the timetableList recursively, and returns the summation of invoking *add_timetable_to_trains_at_station_count* for each timetable, and the same station and time.

add_timetable_to_trains_at_open_line_count Checks whether or not a timetable will schedule a train to be present at an open line at a point in time, if so, it adds one to a counter, else it does not add anything. It has the following parameters:

- *OpenLine openLine* - The open line to check for a train at.
- *Time time* - The time to check for a train.
- *Timetable timetable* - The timetable to check for a train.
- *Int count* - The variable keeping track on the amount of trains.

This function goes through each consecutive pairs of stops in the timetable recursively, and adds 1 to count and returns count if the train is present at the relevant open line at the given point in time. If the train is not present at the given open line at the given point in time, it returns count unchanged.

all_trains_at_open_line_count Gets the amount of trains present in an open line at a certain time. It has the following parameters:

- *OpenLine openLine* - The open line to count the trains at.
- *Time time* - The time to count trains.
- *TimetableList timetableList* - All of the timetables to count the trains from.
- *Int count* - The variable keeping track on the amount of trains.

This function goes through each Timetable in the timetableList recursively, and returns the summation of invoking *add_timetable_to_trains_at_open_line_count* for each timetable, and the same open line and time.

3.4.5 Timetable Predicates

When defining the timetables of a railway network, this model considers two different types of predicates - namely the predicates concerning the conditions of a single timetable, and the predicates concerning the conditions of multiple timetables (when the timetables are interleaving). The items of the list of properties required to be satisfied in section 3.4.3 is included in these predicates - along with the additional predicates:

- The journey of a timetable should be possible in the railway network.
- All timetable names must be unique.

These two items are only considered for the model in RSL, and are disregarded for the remainder of the thesis.

3.4.5.1 Predicates for a Single Timetable

When taking a single timetable into consideration, the following predicates are considered:

pred_all_routes_of_timetables_can_be_traversed A timetable is based on a route through multiple stations in the railway network. This predicate states that the route of a timetable has to exist in the railway network.

In the RSL model, this means that all of the stations in the route of the timetable, must be connected in the railway network the correct order. For all the timetables, the route is found with the auxilliary function *get_Route_from_Timetable*, and the auxilliary function *is_Route_possible* has to return true.

```

pred_all_routes_of_timetables_can_be_traversed :
  TimetableSet × RailwayNetwork → Bool
pred_all_routes_of_timetables_can_be_traversed(
  timetableSet, railwayNetwork) ≡
  (∀ (timetableName, stops) : Timetable •
    (timetableName, stops) ∈ timetableSet ⇒
      if (stops = ⟨⟩) then true
      else
        is_Route_possible(
          get_Route_from_Timetable(
            (timetableName, stops)), railwayNetwork)
      end),

```

pred_minimum_running_times_upheld When a train travels from one station to another based on a timetable, the departure station has a scheduled departure time, and the destination station has a scheduled arrival time. Furthermore, each open line has a time it takes to traverse the open line - the running time. This predicate states that the running time of an open line in a timetable must not exceed the scheduled running time (*ArrivalTime* – *DepartureTime*) of two consecutive stops in a timetable.

In the RSL model, this is done by stating that the auxilliary function *are_travel_times_possible* has to return true for all timetables.

```

pred_minimum_running_times_upheld :
  TimetableSet × RailwayNetwork → Bool
pred_minimum_running_times_upheld(
  timetableSet, railwayNetwork) ≡
  (∀ (timetableName, stops) : Timetable •
    (timetableName, stops) ∈ timetableSet ⇒
      if (stops = ⟨⟩) then true
      else
        are_travel_times_possible(

```

```

      (timetableName, stops), railwayNetwork)
end),

```

pred_dwell_times_upheld This predicate states that the dwell times defined in the stops of a timetable are upheld. This means that the defined dwell time of a stop, must not exceed the scheduled dwell time (*DepartureTime* – *ArrivalTime*) of a stop in a timetable.

In the RSL model, this is done by stating that for all stops of all timetables (*DepartureTime* – *ArrivalTime*) \geq *DwellTime*.

```

pred_dwell_times_upheld : TimetableSet → Bool
pred_dwell_times_upheld(timetableSet) ≡
  (∀ (timetableName, stops) : Timetable •
    (timetableName, stops) ∈ timetableSet ⇒
      (∀ (station, at, dt, dwt) : Stop •
        (station, at, dt, dwt) ∈ stops ⇒
          (dt – at) ≥ dwt)),

```

3.4.5.2 Predicates for a Collection of Timetables

When taking a collection of timetables into consideration, the following predicates are considered:

pred_stations_never_exceed_capacity All stations are defined with a set of platform tracks. This predicate states there can never be more trains present at a station, than there are platform tracks.

In the RSL model, this is actually done by stating that for all arrival times of all stops of all timetables, the amount of trains scheduled to be present (found by using the auxiliary function *all_trains_at_station_count*) cannot exceed the number of platform tracks of the station.

```

pred_stations_never_exceed_capacity :
  TimetableSet × RailwayNetwork → Bool
pred_stations_never_exceed_capacity(
  timetableSet, railwayNetwork) ≡
  (∀ (timetableName, stops) : Timetable •
    (timetableName, stops) ∈ timetableSet ⇒

```



```

in
  all_trains_at_open_line_count(
    get_OpenLine(
      station1, station2, railwayNetwork),
    dt1, timetableSet, 0) ≤ capacity
end)),

```

pred_stations_headway_times_upheld This predicate states that the headway time has to be upheld for all stations at all times. This means that the time of arrival at a station and the time of arrival of the next train at the same station, must, at least, be separated by the headway time of said station.

In the RSL model, this is done by stating that for all pairs of consecutive stops, originating from different timetables, if they arrive at the same station, the difference between the arrival time of the first train and the arrival time of the second train, must be greater than or equal to the headway time.

```

pred_stations_headway_times_upheld :
  TimetableSet × RailwayNetwork → Bool
pred_stations_headway_times_upheld(
  timetableSet, railwayNetwork) ≡
  (∀ (timetableName1, stops1) : Timetable •
    (timetableName1, stops1) ∈ timetableSet ⇒
    (∀ (timetableName2, stops2) : Timetable •
      (timetableName2, stops2) ∈ timetableSet ⇒
      (∀ (station1, at1, dt1, dwt1) : Stop •
        (station1, at1, dt1, dwt1) ∈ stops1 ⇒
        (∀
          (station2, at2, dt2, dwt2) : Stop
          •
            (station2, at2, dt2, dwt2) ∈
            stops2 ⇒
            ((station1, at1, dt1, dwt1) ≠
              (station2, at2, dt2, dwt2) ∧
              station1 = station2) ⇒
            let
              headwayTime =
                get_Station_HeadwayTime(
                  station1, railwayNetwork
                )
            in

```


$$\begin{aligned} & (\mathbf{abs} \ (at1 - at2)) \geq \\ & \quad \mathbf{headwayTime} \\ & \mathbf{end}))))), \end{aligned}$$

pred_open_lines_headway_times_upheld This predicate states that the headway time has to be upheld for all open lines at all times. This means that the time any two trains enter the same open line, must be separated by at least the headway time. The time two trains exit the same open line, must also at least be separated by the headway time of said open line.

In the RSL model, this is done by stating that for all two pairs of consecutive stops, originating from different timetables and using the same open line, the difference between the departure times of each of the first stops and the arrival times of each the second stops, must be greater than or equal to the headway time.

```

pred_open_lines_headway_times_upheld :
  TimetableSet × RailwayNetwork → Bool
pred_open_lines_headway_times_upheld(
  timetableSet, railwayNetwork) ≡
  (∀ timetable1 : Timetable •
    timetable1 ∈ timetableSet ⇒
      (∀ timetable2 : Timetable •
        timetable2 ∈ timetableSet ⇒
          (∀
            ((station1, at1, dt1, dwt1),
             (station2, at2, dt2, dwt2)) :
              (Stop × Stop)
            •
            ((station1, at1, dt1, dwt1),
             (station2, at2, dt2, dwt2)) ∈
              get_movements_of_Timetable(timetable1) ⇒
                (∀
                  ((station3, at3, dt3, dwt3),
                   (station4, at4, dt4, dwt4)) :
                    (Stop × Stop)
                  •
                  ((station3, at3, dt3, dwt3),
                   (station4, at4, dt4, dwt4)) ∈
                    get_movements_of_Timetable(
                      timetable2) ⇒
                      (timetable1 ≠ timetable2 ∧
                       (station1, station2) =

```

```

(station3, station4)) ⇒
let
  headwayTime =
    get_OpenLine_HeadwayTime(
      get_OpenLine(
        station1, station2,
        railwayNetwork),
      railwayNetwork)
in
  (abs (dt1 - dt3)) ≥
    headwayTime ∧
  (abs (at2 - at4)) ≥
    headwayTime
end))))),

```

pred_trains_do_not_attempt_to_overtake This predicate states that trains are not scheduled to overtake each other on the open lines. This means that if two trains depart from the same station, and arrives at the same station, the train departing first, must also be the first train to arrive.

In the RSL model, this is done by stating that for all two pairs of consecutive stops, originating from different timetables and using the same open line with the same departure and destination station (same direction), the train with the first departure time, must also have the first arrival time. If the two trains are running in opposite direction (has opposite departure and destination stations), they will not attempt to overtake - hence in such a situation this predicate does not fail.

```

pred_trains_do_not_attempt_to_overtake :
  TimetableSet → Bool
pred_trains_do_not_attempt_to_overtake(timetableSet) ≡
  (∀ timetable1 : Timetable •
    timetable1 ∈ timetableSet ⇒
      (∀ timetable2 : Timetable •
        timetable2 ∈ timetableSet ⇒
          (∀
            ((station1, at1, dt1, dwt1),
             (station2, at2, dt2, dwt2)) :
              Stop × Stop
            •
            ((station1, at1, dt1, dwt1),
             (station2, at2, dt2, dwt2)) ∈
              get_movements_of_Timetable(timetable1) ⇒
                (∀

```

$$\begin{aligned}
& ((\text{station3}, \text{at3}, \text{dt3}, \text{dwt3}), \\
& (\text{station4}, \text{at4}, \text{dt4}, \text{dwt4})) : \\
& \quad \text{Stop} \times \text{Stop} \\
& \bullet \\
& ((\text{station3}, \text{at3}, \text{dt3}, \text{dwt3}), \\
& (\text{station4}, \text{at4}, \text{dt4}, \text{dwt4})) \in \\
& \quad \text{get_movements_of_Timetable}(\\
& \quad \text{timetable2}) \Rightarrow \\
& \quad ((\text{timetable1} \neq \text{timetable2} \wedge \\
& \quad (\text{station1}, \text{station2}) = \\
& \quad (\text{station3}, \text{station4})) \Rightarrow \\
& \quad ((\text{dt1} < \text{dt3} \wedge \text{at2} < \text{at4}) \vee \\
& \quad (\text{dt3} < \text{dt1} \wedge \text{at4} < \text{at2})))))) \\
&),
\end{aligned}$$

pred_no_single_track_open_lines_utilized_in_both_directions_simultaneously

This predicate states that an open line defined as a single track, cannot be used by two trains going in opposite directions at the same time.

In the RSL model, this is done by stating that for all pairs of consecutive stops of all timetables, the function *is_open_line_occupied_in_time_period_DIRECTED* has to return true, in the time period between the departure time of the first stop and the arrival time of the second stop, and the open line is defined as station of the second stop to the station of the first stop.

```

pred_no_single_track_open_lines_utilized_in_both_directions_simultaneously :
  TimetableSet × RailwayNetwork → Bool
pred_no_single_track_open_lines_utilized_in_both_directions_simultaneously(
  timetableSet, railwayNetwork) ≡
  (∀ timetable : Timetable •
    timetable ∈ timetableSet ⇒
      (∀
        ((station1, at1, dt1, dwt1),
         (station2, at2, dt2, dwt2)) : Stop × Stop
          •
            ((station1, at1, dt1, dwt1),
             (station2, at2, dt2, dwt2)) ∈
              get_movements_of_Timetable(timetable) ⇒
                (let
                  doubleTrack =
                    get_OpenLine_DoubleTrack(
                      (station1, station2),
                      railwayNetwork)

```

```

in
  doubleTrack  $\vee$ 
   $\sim$  is_open_line_occupied_in_time_period_DIRECTED(
    station2, station1, dt1, at2,
    timetableSet)
end)),

```

pred_timetable_names_are_unique This predicate states that there cannot be two timetables with the same name.

In the RSL model, this is done by stating that for all pairs of timetables, their names are different.

```

pred_timetable_names_are_unique :
  TimetableSet  $\rightarrow$  Bool
pred_timetable_names_are_unique(timetableSet)  $\equiv$ 
  ( $\forall$  (timetableName1, stops1) : Timetable  $\bullet$ 
    (timetableName1, stops1)  $\in$  timetableSet  $\Rightarrow$ 
    ( $\forall$  (timetableName2, stops2) : Timetable  $\bullet$ 
      (timetableName2, stops2)  $\in$  timetableSet  $\Rightarrow$ 
      (timetableName1, stops1)  $\neq$ 
      (timetableName2, stops2)  $\Rightarrow$ 
      timetableName1  $\neq$  timetableName2))

```

3.5 Using Test Cases to Validate

This section will describe how test cases of RSL are used to test the predicates of railway networks and timetables.

In order to specify test cases in RSL, some instantiations of the types are required. As an example, the sample railway network of section 3.3.2, and the sample timetables of section 3.4.2 can be created as values in RSL.

Test cases are then defined in RSL, which are comprised of a single invocation of a function. Using SML, it is then possible to determine the return value of each of the test cases. When the predicates of either a railway network or a set of timetables are tested, they all need to return true for either the railway network or the set of timetables to be considered valid.

All of the predicates of the railway network and the set of timetables are tested in the RSL file of appendix A.3. The following is a single test case, testing the

predicate of a railway network, that all stations must be connected. This sample uses the railway network of figure 3.3.

```
pred_All_stations_are_connected(test_RailwayNetwork)
```

When this test case is run in SML, the following output is printed:

```
[pred_All_stations_are_connected] true
```

Similar test cases exist for the remaining predicates, which all return true for the stated railway network and timetable instantiations.

Using UPPAAL To Verify Timetables

UPPAAL[[BDL](#)] is a tool able to model, simulate and verify real-time systems, which can be expressed as a network of timed finite-state machines. With this in mind, UPPAAL seems well suited to model trains running on a railway network according to a pre-defined collection of timetables, as this can be considered a real-time system, and it is possible to model such a system using timed finite-state machines. Previous applications of UPPAAL include verifying communication protocols and multimedia applications[[BDL](#)]. No work has been found to suggest that UPPAAL has been used to verify timetables prior to this thesis.

In this chapter, section [4.1](#) will introduce UPPAAL, by presenting the utilities available in UPPAAL, and how it works in general.

Section [4.2](#) will present the model created in UPPAAL, which can be used to verify timetables.

Section [4.3](#) will end the chapter by presenting how results can be extracted, using the created model.

4.1 Utilities of UPPAAL

UPPAAL is divided into three main parts - a description language, a simulator and a model-checker. For this thesis the simulator is solely used for debugging purposes, and is therefore not discussed in this thesis. The following sections provides a short introduction to the time aspect, the description language and the model-checker of UPPAAL.

4.1.1 Clocks/Time in UPPAAL

A unique feature of UPPAAL, is that it handles *timed* finite-state machines. The fact that time is included, is one of the main motivation points for using UPPAAL in this thesis. Time is included in UPPAAL as clock variables, which allow for the finite-state machines to spend time in a state, and create conditions based on time. When time passes in the system, all of the clocks will know of this and they will all perform the same increase in time.

Time in UPPAAL is represented as a real number, and clocks can be set at any time. When a clock is defined, they are initialized to zero by default - and it is possible to assign any clock any positive value at any time. Setting a clock, will not have any direct effect on any other clocks in the system.

Clocks in UPPAAL are also able to show intervals of time. This happens if the system is in a valid state, and it is possible to remain in that state for a period of time, or pass on to another valid state in that period of time (before being forced to leave or deadlock the system). UPPAAL will detect this possibility, and state that the clock is in a time interval, rather than at a specific point in time, i.e. if no restrictions of time exist in the system, all of the clocks will simply show as being greater than zero.

4.1.2 UPPAAL Description Language

The UPPAAL description language is used to describe the model. The model consists of three different parts:

- A collection of *templates*, which are used to describe the finite-state machines of the model, as well as functions and variables, local to each template.

- A collection of *global declarations*, which are used to define global functions and variables.
- The *system declarations*, which is where the templates are instantiated.

Section 4.1.2.1 gives a description of how templates work in UPPAAL.

Section 4.1.2.2 gives a description of how the global declarations are used.

Finally, section 4.1.2.3 presents how the system declarations instantiate the templates.

4.1.2.1 Templates

Templates are the finite-state machines of the model, including local declarations only accessible from the template. A finite-state machine consists of a set of states, and a set of directed edges, connecting the states¹. In order to express a model as a finite-state machine in UPPAAL, four different types of labels can be added to edges of the finite-state machines:

Selections A selection can bind a non-deterministic value, in a given range, to an identifier, which can be accessed by the other three types of labels of that edge. Selections are not used in the final models created for this thesis. Some intermediate models contained selections, but as they were quite expensive (increased running time), they were removed when optimizing the models.

Guards A guard is a boolean expression, which must be evaluated to true, if the edge is to be traversed at a given time. If no guard is present, or if the guard evaluates to true - the edge is said to be *enabled*.

Synchronisation Different processes can synchronize with each other using channels, meaning that two edges in different templates can be dependant on each other, and can be traversed in the same action. Synchronisations are only used rarely in the models of this thesis, in the form of an urgent channel. The effect of an urgent channel is, that whenever an enabled edge is able to synchronize over an urgent channel - this edge must be traversed without delay. The reason non-urgent channels are not used in the final models of this thesis, is that the model consist of a single main template, and it is not possible to synchronize the same template.

¹The reader is assumed to have knowledge of the basic theory of finite-state machines.

Updates An update on an edge, is an expression with side effects. Whenever an edge with an update is traversed, the side effects of the expression change the state of the system.

It is also possible to add an invariant to a state in UPPAAL, which reflects the condition which must be satisfied for the system to be that state. The states of a finite state-machine of a template in UPPAAL, can be declared as being a *committed* state. When a system is in a committed state, the next edge taken, must be an outgoing edge of the committed state, and no time is allowed to pass. If no outgoing enabled edges exists from a committed state after it is entered, the system is deadlocked.

4.1.2.2 Declarations

Declarations of UPPAAL have similarities to Java, C and C++ [BDL], with the functionality of creating variables, constants, datatypes and functions. All of these different types of declarations are meant to be used by the templates of the model. They could for example be used in state invariants or as one of the labels on an edge. Many of the labels of the edges in the final models of this thesis, use functions in order to keep the finite-state machines as human readable and small as possible. The guard labels use functions which return a boolean value, and the update labels use functions which are able to mutate the system.

It should be noted that when defining an integer, it is possible to define the range, which the integer is will never exceed. The point of defining such a range, is to decrease the number of states, needed to be searched by the model-checker of UPPAAL, when attempting to verify conditions. An integer in the range of zero to 10 is defined as follows:

```
int[0,10] example;
```

4.1.2.3 System Declarations

When creating a model to be verified or simulated in UPPAAL, the final step is to instantiate templates. The instantiations of templates are called *processes*, and they run in parallel. A single template can be instantiated into any number of processes, where each process gets their own set of local declarations from the local declarations of the template. It is also possible to add parameters to the templates - which are then included in the local scope of said process.

4.1.3 UPPAAL Model-checker

The model-checker of UPPAAL is used to specify and verify conditions for a model. In order to specify conditions, a query language exists, which is a subset of timed computation tree logic (TCTL)[BDL]. The query language is used for simple conditions, hence for a detailed description of the language, the reader is referred to [BDL].

The model-checker is capable of the following set of actions:

- Verify a single condition.

If successfully verified, and the condition allows it², a diagnostic trace can be produced, showing an example of a diagnostic trace where the condition is true.

If the condition failed to verify, a diagnostic trace can be produced, to show an example of the condition failing.

- Verify a set of conditions, no diagnostic trace can be produced when verifying more than one condition.
- When producing a diagnostic trace, the fastest (with least time spent) can be produced.
- When producing a diagnostic trace, the shortest (with least steps taken) can be produced.
- When producing a diagnostic trace, a random trace can be produced.

Besides choosing what kind of diagnostic trace UPPAAL should generate, it is possible to set certain options, determining how the model-checker should proceed when checking conditions, for example what type of algorithm to use when verifying - breadth first, depth first etc. The possibilities and details of these options are explained in [BDL], and the best options for the created model for verifying timetables, are discussed in section 4.3.

²If it is a condition stating that a certain state is never reached, a diagnostic trace is not able to show that, however, if the condition states a certain state is reached at some time, a diagnostic trace, showing how to reach the state, will be possible.

4.2 UPPAAL Model

This section will present the model created for verifying timetables. When creating the UPPAAL model for verifying timetables, many of the desired properties have already been defined in the formal model in RSL, section 3. The properties used to define when a collection of timetables is valid in RSL, is carried over to the model in UPPAAL. The methods used to check these properties in RSL, cannot be directly translated into this model, but they have been used as guidelines towards how it could be done.

The purpose of this model is not to be used to validate railway networks. It should therefore be noted that the model created here, assumes that the provided railway networks, are valid according to the RSL specifications of section 3.3.

The RSL model has also acted as inspiration when defining the datatypes, which should hold the information of the railway network and timetables. The types defined in RSL can more or less be carried over to this model, hence they share similarities.

During the development of the model, it was a high priority for the model to be reconfigurable, meaning it should be able to be used to verify any collection of timetables for any railway network, without having to change the templates of the model, but only the data representing the railway network and the timetables.

Another priority of the model, is in case of a property failing during validation, you should be able to see which property has failed, where it failed and when it failed.

The final model for verifying timetables consists of the following three parts:

- The global declarations, which store the global data, including the input of a railway network and a collection of timetables.
- Two templates, a Train template and a Hurry template. A process of the Train template represents a single train, running according to one of the timetables in the global declarations. The Hurry template is there to provide an urgent channel, and initialize the system.
- The system declarations, which creates a single Hurry process and a number of Train processes, equal to the number of timetables to be verified.

Figure 4.1, shows a sketch of the contents of the model.

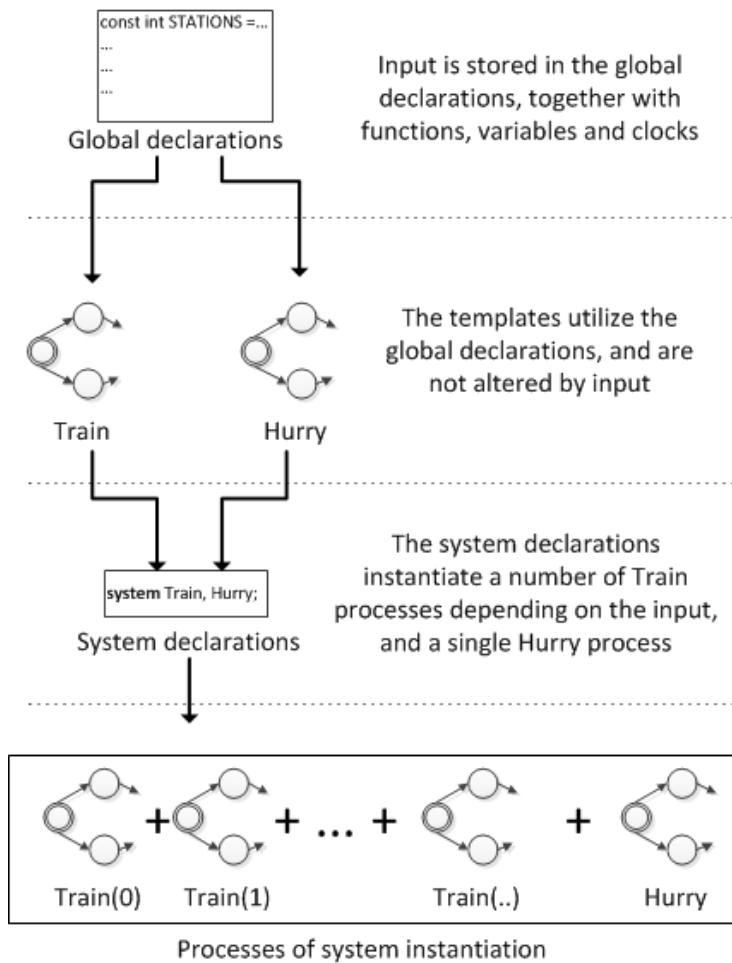


Figure 4.1: The general structure of the final model.

In this section, section 4.2.1 describes the input required, in order to use the model for verifying timetables.

Section 4.2.2 introduces the datatypes, clocks, variables, constants and functions of the global declarations. It should be noted that the usage for some of these declarations will first appear in the next sections of 4.2.3, 4.2.4 and 4.2.5.

Section 4.2.3 provides a short description of the Hurry template.

Section 4.2.4 will explain the Train template in detail.

Section 4.2.5 will explain the system declarations.

Section 4.2.6 will end the model description, by presenting the different aspects of optimizations performed, during the development of the model.

4.2.1 Validating Timetables - The Input

In order to use the model for validating a collection of timetables, it requires some input from the user:

A railway network including an open line table and a station table is needed.

A collection of timetables which should be the collection of timetables to be verified. Each timetable should be an ordered list of stops, where a stop consists of a station, an arrival time, a departure time, and a desired dwell time.

All of these values are to be specified by the user of the model, and are declared in the global declarations.

4.2.2 Global Declarations

The global declarations are used by all of the Train processes, as well as the Hurry process. The global declarations are used to handle the following five aspects of the model:

- Datatypes are defined for two things; storing the input of section 4.2.1, and creating a queue.

- The input of section 4.2.1 is stored as constants.
- Variables, representing the states of the open lines and the stations are declared.
- Clocks and Channels used by the templates are declared.
- Functions are declared, including accessor functions and functions with sideeffects, altering the state of the system.

The following sections will describe these five types of declarations, in the listed order.

4.2.2.1 Datatypes

In order to store information as other datatypes than the primitive datatypes of UPPAAL, new datatypes are defined. These datatypes are used to express the following:

- A station table entry given as input. This datatype is converted from the RSL model, section 3.3.1.
- An open line table entry given as input. This datatype is also converted from the RSL model, section 3.3.1.
- A timetable given as input. This datatype is converted from the RSL model, section 3.4.1.
- A FIFO³ queue structure, for which the purpose is clarified in section 4.2.4.
- The id of a Train process is defined as a, integer between zero and the amount of trains defined by the amount of timetables. The purpose for this datatype is explained in section 4.2.5.

The following are the defined datatypes in UPPAAL. As the instantiations of these datatypes are actually constants, sample values are presented in section 4.2.2.2.

³'First In First Out'

```

//An open line
typedef struct{
    int station1;
    int station2;
} OpenLine;

//An open line in the open line table
typedef struct{
    OpenLine openLine;
    bool doubleTrack;
    int MRT; //Minimum Running Time
    int capacity;
    int HWT; //Headway Time
}OpenLineTableEntry;

//A station in the station table
typedef struct{
    int station;
    int capacity;
    int HWT; //Headway Time
}StationTableEntry;

//An entry in a timetables
typedef struct {
int stationId;
int AT; //Arrival Time
int DT; //Departure Time
int DWT; //Dwell Time
}StopEntry;

/*A queue for each open line denoting the order in which trains enter,
and when they are expected to leave the open line again*/
typedef struct{
    int train;
    int expectedExit;
} queueEntry;

//t_id is the datatype of the id parameter of the Train template
typedef int[0, TRAINS-1] t_id;

```

It should be noted that a difference between this datatype and the types in RSL, is that a station is represented by an integer in UPPAAL, and a text in RSL. The reason for this is that strings or texts does not exist in UPPAAL. Due to the

fact that reading station names as integers, can lessen the readability notably, a workaround for this is presented in section 4.2.2.2.

4.2.2.2 Constants

The constants declared in this model, all originate from the input of the railway network and collection of timetables. The constants here, are defined with sample values of the railway network of Lokalbaneln. Two timetables are given to be verified in these sample values:

```
//Number of Stations
const int STATIONS = 9;

//Number of Open Lines
const int OPENLINES = 8;

//Number of Trains
const int TRAINS = 2;

//Number of stops on the routes for each train
const int TRAINSTOPS[TRAINS] = {8,9};

/*The largest amount of stops in a timetable.
  This is used to create the timetables array*/
const int MAXLENGTH = 9;

/*Lokalbanen stations, these constants are created to increase
  the readability of the other constants*/
const int remisn = 0;
const int jagersborg = 1;
const int norgaardsvej = 2;
const int lyngbylokal = 3;
const int fuglevad = 4;
const int brede = 5;
const int orholm = 6;
const int ravnholm = 7;
const int narum = 8;

//Open line table for Lokalbaneln
const OpenLineTableEntry openLineTable[OPENLINES] =
  {{{jagersborg,norgaardsvej},false,1,1,0},
```

```

    {{norgaardsvej,lyngbylokal},false,1,1,0},
    {{lyngbylokal,fuglevad},false,1,1,0},
    {{fuglevad,brede},false,2,1,0},
    {{brede,orholm},false,2,1,0},
    {{orholm,ravnholm},false,1,1,0},
    {{ravnholm,narum},false,2,1,0},
    {{jagersborg,remisen},false,2,1,0}};

//Station table for Lokalbanen
const StationTableEntry stationTable[STATIONS] =
    {{remisen, 6, 1},
     {jagersborg, 2, 1},
     {norgaardsvej, 1, 1},
     {lyngbylokal, 1, 1},
     {fuglevad, 2, 1},
     {brede, 1, 1},
     {orholm, 2, 1},
     {ravnholm, 1, 1},
     {narum, 2, 1}};

//Timetables
const StopEntry stops[TRAINS][9] = {
    //Train 630-1 of Lokalbanen
    {{jagersborg,30,30,0},
     {norgaardsvej,31,31,0},
     {lyngbylokal,33,33,0},
     {fuglevad,35,35,0},
     {brede,37,37,0},
     {orholm,39,40,0},
     {ravnholm,41,41,0},
     {narum,43,43,0},
     {-1,-1,-1,-1}},

    //Train 640-1 of Lokalbanen
    {{remisen, 34, 34,0},
     {jagersborg,36,40,0},
     {norgaardsvej,41,41,0},
     {lyngbylokal,43,43,0},
     {fuglevad,45,45,0},
     {brede,47,47,0},
     {orholm,49,50,0},
     {ravnholm,51,51,0},
     {narum,53,53,0}}};

```

4.2.2.3 Variables

The variables of the model, are used to represent the stations and the open lines of the railway network, as well as keeping track on how far each train has travelled at any given time.

When defining a queue for each open line, and the trains present at each open line, it is done for each direction in the open line. The result, being an additional dimension of size two, in the relevant arrays. This dimension represents the two different directions available to traverse on the open line - namely the direction of value 0 and of value 1. In order to distinguish the two directions, the datatype of the open line is used. Even though an open line is not directed, it is defined by two stations - station1 and station2 (see section 4.2.2.1). The direction of value 0, represents the direction going towards station1 from station2, and the direction of value 1 is the direction going towards station2 from station1.

The following are the variable declarations of the model:

```

/*
The FIFO queue, a queue exists for each open line(first dimension of array)
in both directions (the second dimension of the array), with room for
all of the trains (the third dimension of the array).
*/
queueEntry queue[OPENLINES][2][TRAINS];

//The current position of each train, 0 is the first stop in the timetable
int currentStop[TRAINS] = {0, 0};

//The amount of trains present in each direction on each open line
int[0, TRAINS] trainsAtOpenLine[OPENLINES][2];

//The amount of trains present at each station
int[0, TRAINS] trainsAtStation[STATIONS];

```

As an example `trainsAtOpenLine[3][0]`, would be the number of trains currently going from fuglevad to brede, and `trainsAtOpenLine[3][1]` would be the number of trains going from brede to fuglevad, according the sample constants of section 4.2.2.2.

It should be noted that the amount of trains present at an open line, could be derived from the queue, hereby eliminating the need for the variable `trainsAtOpenLine`. Both arrays have been kept, in order to keep a simulation of

the model as readable as possible. Keeping and maintaining both arrays has a negligible impact on the running time.

4.2.2.4 Clocks and Channels

The clocks in the model are used to incorporate the time aspect of the timetables into the model. The clock *time* is never reset, and is used to read the time of the system.

The following are the declarations of the clocks and the urgent channel of the model:

```
//Global time, this clock is never reset
clock time;

/*Trainclocks, used by each train to determine
how much time has been spent in different states*/
clock TrainClock[TRAINS];

//The time elapsed since a train last entered each open line
clock openLineLastEntered[OPENLINES];

//The time elapsed since a train last exited each open line
clock openLineLastExit[OPENLINES];

//The time elapsed since a train last entered each station
clock stationLastEntered[STATIONS];

//The urgent channel
urgent chan hurry;
```

4.2.2.5 Functions

The functions of the model can be split up into three different categories:

- Accessor functions, which retrieve information from the constants, and does not have cause sideeffects.
- Updating functions, which mutate the variables of the model, causing the state of the system to change.

- Initializers, which are used to initialize certain variables.

The functions of the model are now described, in the order of the stated categories. Descriptions of each individual function is given as comments in the code snippets.

Accessor Functions

The following function is an accessor function for getting the headway time of a station:

```
//Get headway time of a station
int GetStationHWT(int stationId)
{
    return stationTable[stationId].HWT;
}
```

Similar functions exist for open line headway time, minimum running time, capacity, the double track value and a station capacity.

Some of the remaining accessor functions dealing with open lines, use a parameter *dir*. This parameter represents the relevant direction, for which the function is called. *dir* is a station in the open line, and is interpreted as described in section 4.2.2.3 - meaning that if *dir* equals station1 of the open line, the direction value of 0 is used, and if *dir* equals station2 of the open line, the direction value of 1 is used.

The following functions are the remaining accessor functions of the model:

```
/*Get an open line
   openLineId is the id of the open line.*/
OpenLine GetOpenLine(int openLineId)
{
    return openLineTable[openLineId].openLine;
}

/*Get the id of the open line between the two station parameters.
   station1 of the parameters, is not necessarily station1 of the open line,
   and station2 of the parameters is not necessarily station2 of the open line
   station1 is one of the stations of the open line.
   station2 is the other of the stations of the open line*/
```

```

int GetOpenLineId(int station1, int station2)
{
    for (i : int[0,OPENLINES-1])
    {
        if((openLineTable[i].openLine.station1 == station1 &&
            openLineTable[i].openLine.station2 == station2) ||
            (openLineTable[i].openLine.station1 == station2 &&
            openLineTable[i].openLine.station2 == station1))
            return i;
        }
    }
    return -1;
}

/*Get the total amount of trains present at an open line - regardless of direction.
Used when checking for the capacity of a single tracked open line, in which one
of the directions will always be 0
openLineId is the id of the open line.*/
int GetTrainsAtOpenLine(int openLineId)
{
    return trainsAtOpenLine[openLineId][0] + trainsAtOpenLine[openLineId][1];
}

/*Get the amount of trains present in a direction on an open line
openLineId is the id of the open line.
dir is the relevant direction of the open line.*/
int GetTrainsAtOpenLineDir(int openLineId, int dir)
{
    if(openLineTable[openLineId].openLine.station1 == dir)
        return trainsAtOpenLine[openLineId][0];
    else
        return trainsAtOpenLine[openLineId][1];
}

/*Determines whether or not an open line
is occupied in the opposite direction of dir
openLineId is the id of the open line.
dir is the relevant direction of the open line.*/
bool IsOpenLineOccupiedOppositeDirection(int openLineId, int dir) {
    OpenLine openLine = openLineTable[openLineId].openLine;
    if(openLine.station1 == dir)
        return GetTrainsAtOpenLineDir(openLineId, openLine.station2) > 0;
    else
        return GetTrainsAtOpenLineDir(openLineId, openLine.station1) > 0;
}

```

```

/*Get the expected exit of the train at the head of the queue of an open line
  openLineId is the id of the open line.
  dir is the relevant queue of the open line.*/
int GetQueueExpectedExit(int openLineId, int dir)
{
  if(IsOpenLineDoubleTrack(openLineId))
    if(openLineTable[openLineId].openLine.station1 == dir)
      return queue[openLineId][0][0].expectedExit;
    else
      return queue[openLineId][1][0].expectedExit;
  else
    return queue[openLineId][0][0].expectedExit;
}

/*Get the train at the head of the queue of an open line
  openLineId is the id of the open line.
  dir is the relevant queue of the open line.*/
int GetQueueLatestTrain(int openLineId, int dir)
{
  if(IsOpenLineDoubleTrack(openLineId))
    if(openLineTable[openLineId].openLine.station1 == dir)
      return queue[openLineId][0][0].train;
    else
      return queue[openLineId][1][0].train;
  else
    return queue[openLineId][0][0].train;
}

```

Updating Functions

The updating functions of the model, are the functions increasing and decreasing the amount of trains in a direction of an open line, and entering and leaving the queues of the open lines. It should be noted that no functions exist to increase or decrease the amount of trains present at each station, as this action is trivial, because no direction is included. Therefore increasing and decreasing the number of trains at stations is done directly in the template.

The following declarations are the updating functions of the model, with corresponding comments to describe each function:

```

/*Increase the amount of trains present in a direction on an open line.

```

```

    openLineId is the id of the open line.
    dir is the relevant direction of the open line.*/
void IncreaseTrainsAtOpenLineDir(int openLineId, int dir)
{
    if(openLineTable[openLineId].openLine.station1 == dir)
        trainsAtOpenLine[openLineId][0]++;
    else
        trainsAtOpenLine[openLineId][1]++;
}

/*Decrease the amount of trains present in a direction on an open line.
    openLineId is the id of the open line.
    dir is the relevant direction of the open line.*/
void DecreaseTrainsAtOpenLineDir(int openLineId, int dir)
{
    if(openLineTable[openLineId].openLine.station1 == dir)
        trainsAtOpenLine[openLineId][0]--;
    else
        trainsAtOpenLine[openLineId][1]--;
}

/*Enter the queue of an open line.
    openLineId is the id of the open line.
    trainId is the train entering the queue.
    latestExit is the time the train is expected to leave the open line.
    dir is the direction the train traverses the open line*/
void EnterQueue(int openLineId, int trainId, int latestExit, int dir)
{
    queueEntry entry = {trainId, latestExit};
    if(IsOpenLineDoubleTrack(openLineId))
        if(openLineTable[openLineId].openLine.station1 == dir)
            queue[openLineId][0][GetTrainsAtOpenLineDir(openLineId, dir)] = entry;
        else
            queue[openLineId][1][GetTrainsAtOpenLineDir(openLineId, dir)] = entry;
    else
        queue[openLineId][0][GetTrainsAtOpenLine(openLineId)] = entry;
}

/*Exit the queue of an open line.
    openLineId is the id of the open line.
    dir is the direction the train traverses the open line*/
void ExitQueue(int openLineId, int dir)
{
    for (i : int[0,TRAINS-1])

```



```

{
  //If open line is a double track
  if(IsOpenLineDoubleTrack(openLineId))
    //Find the correct direction
    if(openLineTable[openLineId].openLine.station1 == dir)
    {
      queueEntry empty = {-1, -1};
      if(i != TRAINS-1)
      {
        queue[openLineId][0][i] = queue[openLineId][0][i+1];
        queue[openLineId][0][i+1] = empty;
      }
      else
        queue[openLineId][0][i] = empty;
    }
  else
  {
    queueEntry empty = {-1, -1};
    if(i != TRAINS-1)
    {
      queue[openLineId][1][i] = queue[openLineId][1][i+1];
      queue[openLineId][1][i+1] = empty;
    }
    else
      queue[openLineId][1][i] = empty;
  }
  //If open line is single track
  else
  {
    queueEntry empty = {-1, -1};
    if(i != TRAINS-1)
    {
      queue[openLineId][0][i] = queue[openLineId][0][i+1];
      queue[openLineId][0][i+1] = empty;
    }
    else
      queue[openLineId][0][i] = empty;
  }
}
}
}

```

Initializers

The last functions remaining are the initializers, which initializes the queues of the open lines, and the clocks related to the stations and open lines of the system. The clocks are initialized to 1000000000, as they represent the time elapsed since a train last entered each station and open line, and a train last left an open line. At the beginning of a simulation - this should theoretically be infinity, but this is not possible in UPPAAL, and the largest value possible is chosen instead.

The following declarations are the initializers of the model:

```
//Initialize the queue
void initQueue()
{
  for (i : int[0, OPENLINES-1])
  {
    for (j : int[0, TRAINS-1])
    {
      int empty = -1;
      queue[i][0][j] = empty;
      queue[i][1][j] = empty;
    }
  }
}
```

```
//Initialize the station clocks and open line clocks used for headway times
void initEnterExitClocks()
{
  //Station headway clocks
  for (i :int[0, STATIONS-1])
  {
    stationLastEntered[i] = 1000000000;
  }
  for (i :int[0, OPENLINES-1])
  {
    openLineLastEntered[i] = 1000000000;
    openLineLastExit[i] = 1000000000;
  }
}
```

All of the global declarations have now been presented, and the Hurry template will be introduced.

4.2.3 The Hurry Template

The Hurry template initializes the global declarations which needs to be initialized, and it provides an urgent channel. The initial state is committed, making the outgoing edge of the initial state, the first action taken in the system. This edge invokes the functions to initialize the queues and the clocks of the stations and open lines.

The resulting state has one outgoing edge, which awaits a synchronization on an urgent channel.

The template can be seen in figure 4.2.

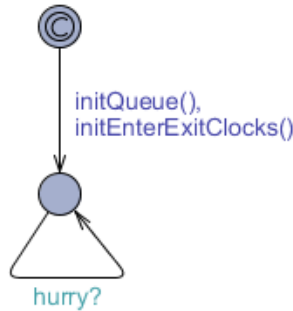


Figure 4.2: The Hurry template, which initializes the system, and provides an urgent channel

4.2.4 The Train Template

The Train template represents a single train running according to one of the timetables given as input on the railway network also given as input (as explained in section 4.2.1).

It has a single *id* parameter, which is used to access the variables and constants regarding the variables associated with the specific instantiation of the template (such as the *currentStop* variable). The timetable assigned to an instantiation of the Train template, is the timetable at the index of the id of the Train, i.e. the train with id 1, will be associated with the timetable of stops[1][.], where the second dimension of the array, is the stops in the timetable.

The template is able to handle any railway networks, which can be defined as defined in the railway network of the formal model in RSL, section 3.3. Figure 4.3 shows a simplified version of the Train template, where some states and edges are left out in order to clarify the concept of the model. For the model to be able to handle an arbitrary railway network, four states are introduced, where they are split into three different categories, as stated by their color:

Inactive - Gray When a train is inactive, it is not yet time for it to stop at the first designated stop in its timetable.

Active - Blue When a train is in one of the active states, it means that it is currently journeying through a timetable expressed in the input. When a train is in `AtStation`, it reflects the fact that according to the timetable, it is currently waiting at a station. When the train has departed from the station, and is travelling on to the next stop, it enters the `EnRoute` state. When the train is stated to arrive at the next stop, it will enter the `AtStation` state again. This will continue until every stop has been traversed, and the train will then enter the `Complete` state.

Complete - Green When a train is complete, it means that it has successfully journeyed through its given timetable - if all trains are completed, it means a collection of timetables has been successfully validated.

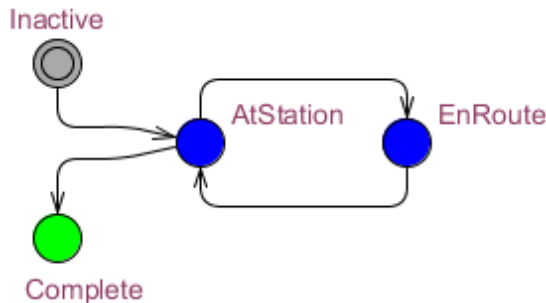


Figure 4.3: The basic concept of how the Train template is able to handle a dynamic railway network.

The properties to validate the collection of timetables against, are now introduced. Looking at the formal model in RSL, there are the following eight properties which should hold:

1. Trains cannot overtake another train on an open line.

2. Trains have to satisfy the minimum running times of the open lines.
3. Trains have to satisfy the dwell times at the stations.
4. The headway times of stations must be upheld.
5. The capacity of stations must never be exceeded.
6. The headway times of open lines must be upheld.
7. The capacity of the open lines must never be exceeded.
8. Single track open lines cannot be utilized in both directions simultaneously.

All of these properties can be checked for in the transitions between entering an open line, or entering a station. Two additional committed states are introduced as evaluating states, one as an intermediate state on the edge `AtStation-EnRoute`, and the other as an intermediate state on the edge `EnRoute-AtStation`. Furthermore nine error states are introduced - one for each property and one extra for the headway time of open lines. These error states can be reached either from the two new evaluating states, and some can also be reached from the inactive state. An error state can be entered from the inactive state, if a train is stated to enter a station, which will cause a property to fail, such as station capacity or station headway time.

Figure 4.4 shows all of the states and edges in the final model, the edge labels and state invariants have been removed to ease the readability of the basic structure in the template.

For the purpose of presenting the final model, the edge labels and state invariants will be introduced to the template of figure 4.4 in small steps, until the final model is reached.

The next thing introduced, is edge labels and state invariants responsible for making sure that the trains travel through the railway network, based on the associated timetable, in the *stops* constant. In order for this to happen, invariants and edge guards are placed, forcing the system to enter the `AtStation` state, when the arrival time has been reached, and to enter the `EnRoute` state when the departure time has been reached. Furthermore guards are added, such that whenever `AtStation` is entered and the train has reached its final stop, it will go to the complete state. Finally, when leaving `AtStation` and the final stop is not yet reached, the variable `currentStop` for the Train is incremented. Figure 4.5 shows the model, with the added invariants and edge labels.

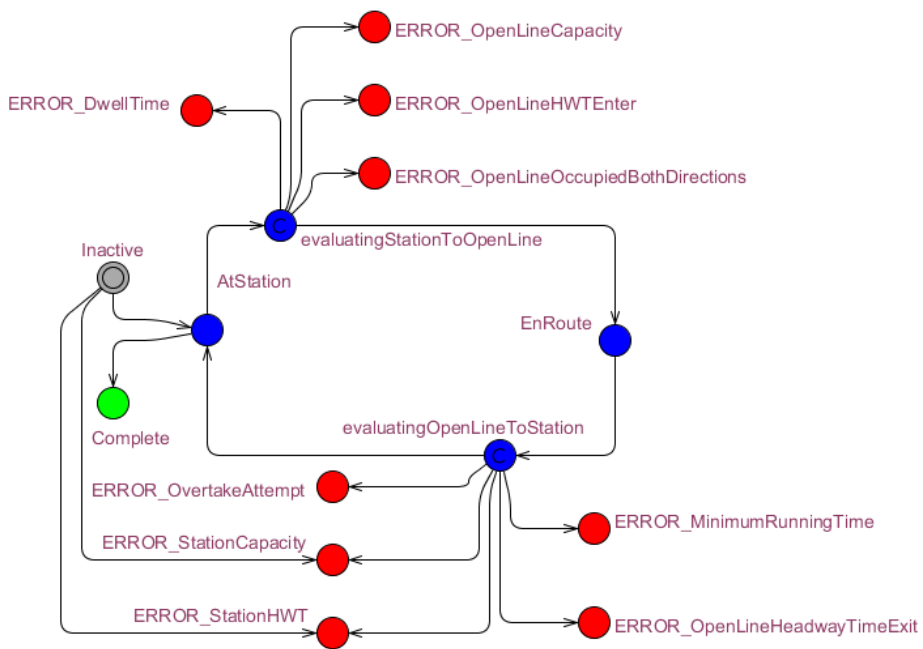


Figure 4.4: The Train template of the UPPAAL model, with all edge labels and state invariants removed

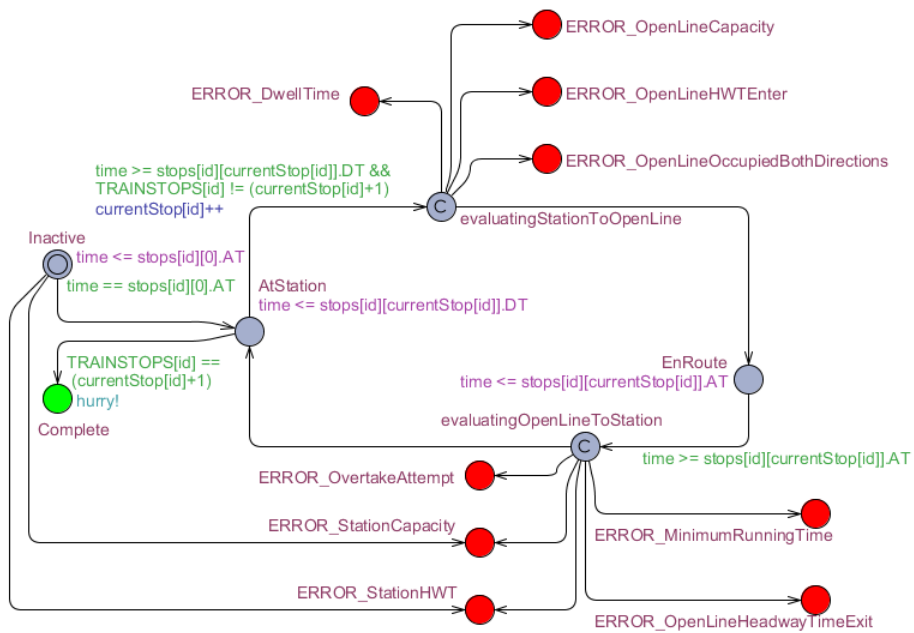


Figure 4.5: The Train template, with the added edge labels (showing green) and state invariants (showing red), guiding the train through a timetable correctly

When introducing the check, to validate whether or not a train attempts to overtake another train on an open line⁴, the *queue* variable is used. Upon entering an open line, the train enters the queue, and when the train leaves the open line to enter a station, the train must be at the front of the queue - if not, another train was scheduled to be overtaken, and an error has occurred. Figure 4.6 has added the property of checking whether or not trains are scheduled to overtake one another. The functions `AttemptedOvertake()` and `GetCurrentOpenLine()` are declared in the template of `Train`, and consists of:

```
/*Determines whether or not a train is currently attempting
to overtake another train, by advancing in the queue*/
bool AttemptedOvertake() {
    return GetQueueLatestTrain(GetCurrentOpenLine(),
        stops[id][currentStop[id]].stationId) != id;
}

//Gets the id of the open line in the openlinetable
int GetCurrentOpenLine() {
return GetOpenLineId(
    stops[id][currentStop[id]-1].stationId,
    stops[id][currentStop[id]].stationId);
}
```

When adding the checks for the minimum running time property⁵, and the dwell time property⁶, invariants and edge guards were added, stating the train must have spent at least the dwell in the `AtStation` state, and must have at least spent the minimum running time of the open line in the `EnRoute` state. In order to achieve this, a the *TrainClock* is used to check how much time was spent in the relevant states. Figure 4.7 has added the properties of minimum running times and dwell times.

When adding the check for the property of capacity for stations⁷, the *trainsAtStation* variable is used. The variable is incremented when a train enters the station, and decremented when a trains leaves the station. A set of guards is added on two different sets of edges, the edges between `Inactive` and `AtStation`, and `Inactive` and `ERROR_StationCapacity`, and the edges between `evaluatingOpenLineToStation` and `AtStation`, and `evaluatingOpenLineToStation` and `ERROR_StationCapacity`. The guards state that if the amount of trains present

⁴Property 1

⁵Property 3

⁶Property 2

⁷Property 5

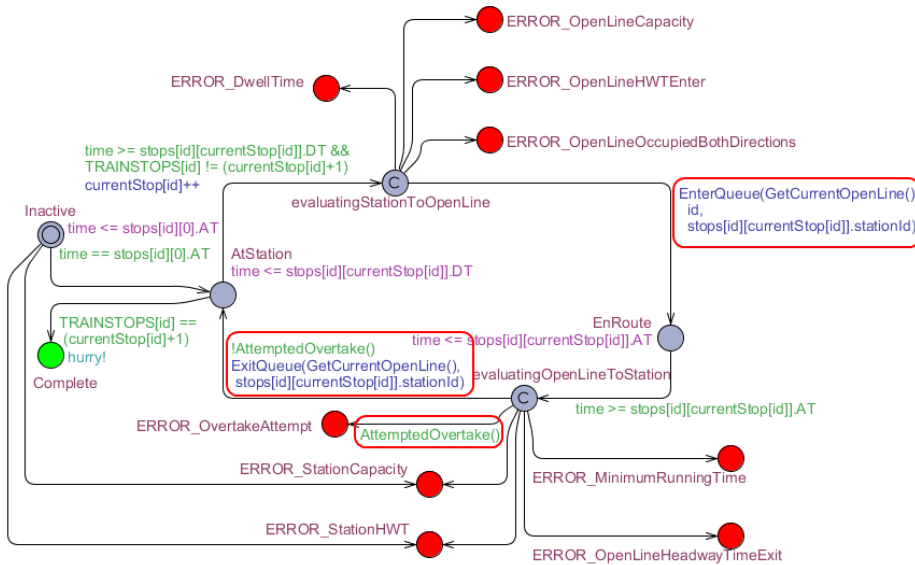


Figure 4.6: The train template, with the added check for overtaking. The red circles represent the additions made to the model

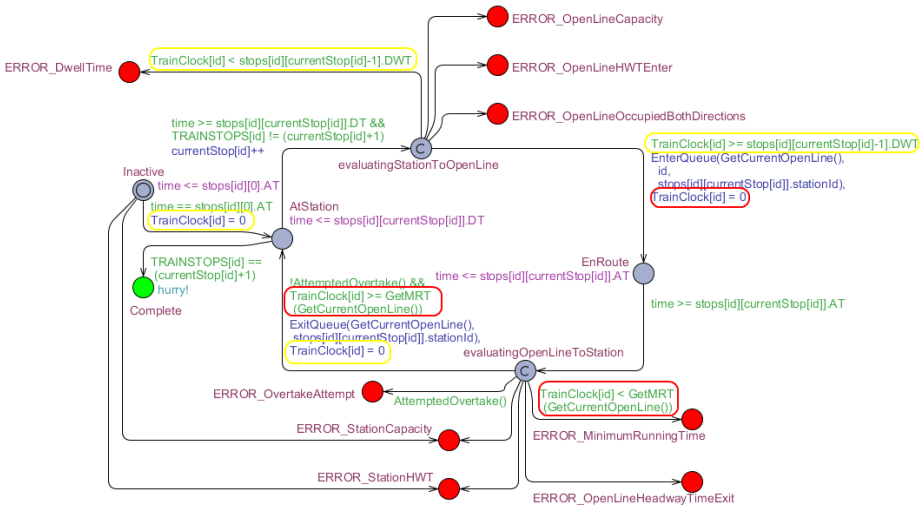


Figure 4.7: The Train template, with the added properties of validating dwell times and minimum running times. The red circles represent the addition for the minimum running time property, and the yellow circles represent the addition for the dwell time property.

at the station would exceed the capacity of the station upon entering, the error state should be entered.

When adding the check for the property of headway times for a station⁸, the *stationLastEntered* clock is used, representing the time elapsed since a train last entered a station. This clock is reset to zero, each time a train enters the station. A set of guards is added on the same two sets of edges as explained with the station capacity property, only this time, the error state is `ERROR_StationHWT`. The guards state that if the time elapsed since the last train entered a station is less than the headway time of the station, the error state should be entered. Figure 4.8 has added the two station properties, where the `StationCapacityError()` guard label, the `EnterStation(int stationId)` and `LeaveStation(int stationId)` update labels have been created as the following functions:

```
//Determine whether or not a station capacity error has incurred
bool StationCapacityError()
{
    return trainsAtStation[stops[id][currentStop[id]].stationId] >=
        GetStationCapacity(stops[id][currentStop[id]].stationId);
}

/*Make the appropriate updates when entering a station
   stationId is the entered station.*/
void EnterStation(int stationId) {
    trainsAtStation[stationId]++;
    stationLastEntered[stationId] = 0;
}

/*Make the appropriate updates when leaving a station
   stationId is the station to leave from*/
void LeaveStation(int stationId)
{
    trainsAtStation[stationId]--;
}
```

It should be noted that UPPAAL is not able to create the guard

```
stationLastEntered[stops[id][currentStop[id]].stationId] <
    GetStationHWT(stops[id][currentStop[id]].stationId)}
```

⁸Property 4

as a bool function - when attempting to do so, an error reporting incompatible types occur. If it was possible, this would also have been incorporated in the `StationCapacityError()` function, to lessen the complexity of reading the finite-state machine.

When adding the check for the property of capacity for open lines⁹, the variable *trainsAtOpenLine* is used. It is incremented when a train enters the open line, and decremented when a train leaves. For a single tracked open line, it is not necessary with a counter in each direction for this property, but the last property can utilize this, which is the reason for counters in both directions of an open line being utilized - regardless of the open line being single or double tracked. In addition to the counters, guards are added on the two outgoing edges of `evaluatingStationToOpenLine` reaching `EnRoute` and `ERROR_OpenLineCapacity`, stating that there must be room for at least one more at that open line, and in that direction, if not, go to the error state.

When adding the check for the property of headway time for open lines¹⁰, the two clocks *openLineLastEntered* and *openLineLastLeft* are used. The first represents the time elapsed since a train last entered the open line, and the second represents the time elapsed since a train last left the open line. These clocks are set to zero when a train enters the open line and when a train leaves the open line, respectively. Two sets of guards are added, one set of guards on the outgoing edges from `evaluatingStationToOpenLine` reaching `EnRoute` and `ERROR_OpenLineHWTEnter`, stating that if the time elapsed since a train last entered the open line, is less then the headway time of the open line - then enter the error state. The other set of guards are added on the outgoing edges of `evaluatingOpenLineToStation` reaching `ERROR_OpenLineHWTExit` and `AtStation`, stating that if the time elapsed since a train last exited the open line, is less then the headway time of the open line - then enter the error state.

When adding the check for the final property of not allowing a single tracked open line to be utilized in both directions simultaneously¹¹, the variable *trainsAtOpenLine* used for the capacity property of an open line is enough. A set of guards is added on the outgoing edges from `evaluatingStationToOpenLine` reaching `ERROR_OpenLineOccupiedBothDirections` and `EnRoute`, stating that if the open line is single tracked, *trainsAtOpenLine* of the opposite direction must be zero.

Figure 4.9 has added the three open line properties. It should be noted that some of the prior updates and guards have been moved to some of the following new functions:

⁹Property 6

¹⁰Property 7

¹¹Property 8


```

//Determine whether or not the capacity of the current open line is exceeded
bool OpenLineCapacityError()
{
    int openLineId = GetCurrentOpenLine();
    int dir = stops[id][currentStop[id]].stationId;
    //If doubletrack and the capacity in the direction has room for no more
    if(IsOpenLineDoubleTrack(openLineId))
        return GetTrainsAtOpenLineDir(openLineId, dir) >=
            GetOpenLineCapacity(openLineId)
    //If singletrack and the capacity has room for no more
    else
        return GetTrainsAtOpenLine(openLineId) >=
            GetOpenLineCapacity(openLineId));
}

/*Determine whether or not the open line is occupied
in both directions and is single tracked*/
bool OccupiedOppositeError()
{
    return
        (IsOpenLineOccupiedOppositeDirection
         (GetCurrentOpenLine(), stops[id][currentStop[id]].stationId) &&
         !IsOpenLineDoubleTrack(GetCurrentOpenLine()));
}

/*Make the appropriate updates when leaving an open line
openLineId is the open line to leave.
dir is the direction which the trains leaves.*/
void LeaveOpenLine(int openLineId, int dir) {
    ExitQueue(openLineId, stops[id][currentStop[id]].stationId);
    DecreaseTrainsAtOpenLineDir(openLineId, dir);
    openLineLastExit[openLineId] = 0;
}

/*Make the appropriate updates when entering an open line
openLineId is the open line to enter.
dir is the direction which the trains enters.*/
void EnterOpenLine(int openLineId, int dir) {
    EnterQueue(GetCurrentOpenLine(), id, stops[id][currentStop[id]].AT,
        stops[id][currentStop[id]].stationId);
    IncreaseTrainsAtOpenLineDir(openLineId, dir);
    openLineLastEntered[openLineId] = 0;
}

```

As with the guard regarding the headway time of stations, the guards of the headway times of open lines cannot be created as boolean functions, otherwise they would have.

All of the properties are now included, and the model is close to the final model. The last thing missing, is that currently, if one train is set to leave a station, and another train is set to enter the same station at the same time - two different diagnostic traces are checked, as illustrated in figures 4.10 and 4.11

In figure 4.10, the number of trains at station x reaches three (assuming station x already has two trains present), while the number of trains at station x only reaches two in figure 4.11. As the model-checker checks all traces, the trace of figure 4.10 will cause an error. This in itself can be a correctly defined error, however this is not desirable, due to the fact that the actual working timetables of Nærumbanen[Lok] would be invalid, if this was the case. As the working timetables of Nærumbanen are real practical working timetables, this thesis works towards a validation tool, which is able to validate these timetables as well.

To solve this, a priority is added, where the train entering the station is forced to act first, hereby enforcing the course shown in figure 4.11. This is why the *expectedExit* value of a *queueEntry* exists. When a train is designated to leave a station, a guard is added, stating that no other train should be designated to enter this station at the same time, without already having entered. This guard is placed on the edge between *AtStation* and *evaluatingStationToOpenLine*, and the *ExitQueue* and *EnterQueue* functions have an added parameter, namely the expected time of exit, which is the time of arrival at the station. Figure 4.12 shows the final Train template.

4.2.5 System Declarations

The system declarations of the UPPAAL model has the following single line written in it:

```
system Train, Hurry;
```

This creates a single Hurry process, and a number of Train processes equal to the *TRAINS* constant in the global declaration. All of these processes are created in parallel.

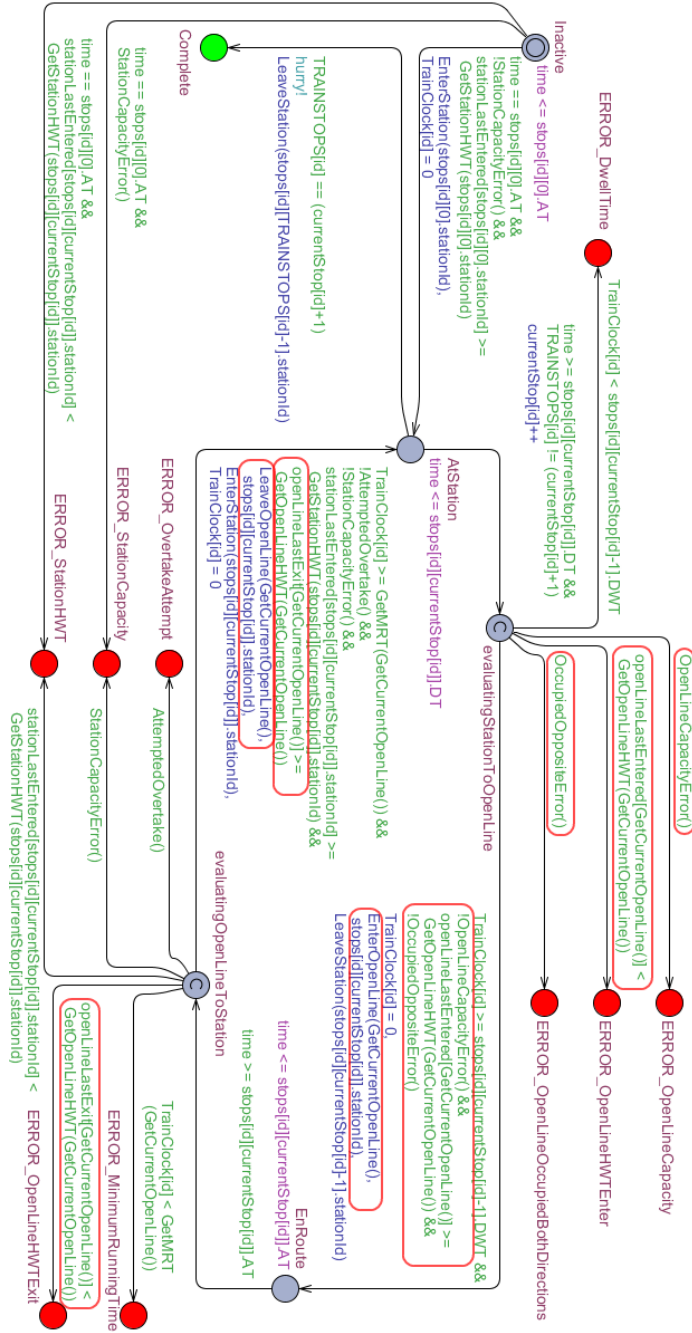


Figure 4.9: The Train template with all of the properties added, the new additions are the open lines headway time, capacity and the fact that single tracked open lines cannot be utilized in both directions simultaneously. The red circles represent the additions.

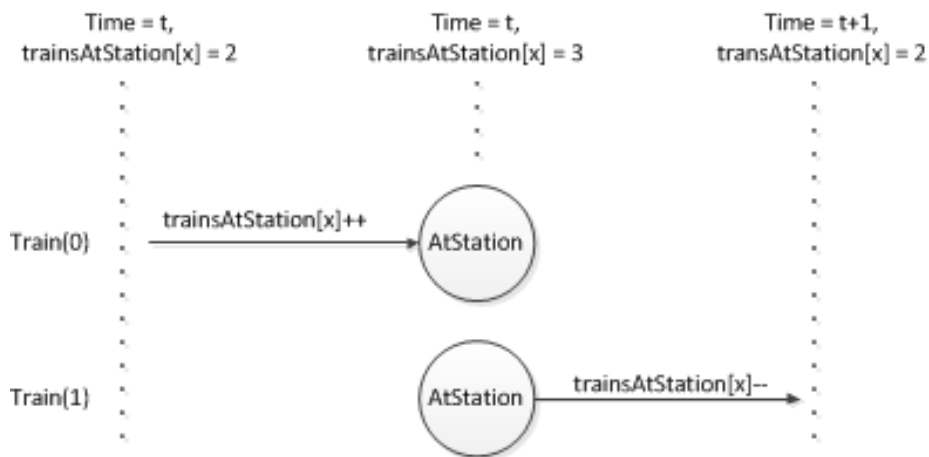


Figure 4.10: Train(0) first enters station x , where there are already two present - resulting in three trains present at station x during a diagnostic trace.

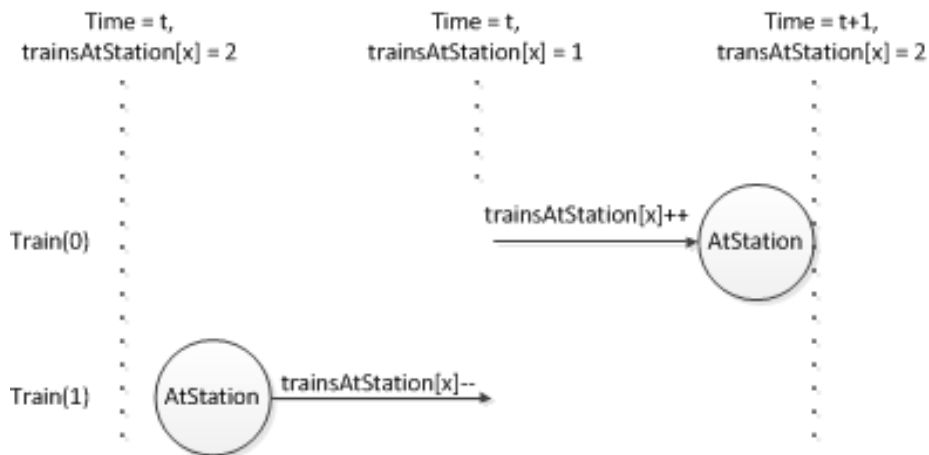


Figure 4.11: Train(1) first leaves station x , where there are already two present - resulting in station x never exceeding two trains present during a diagnostic trace.

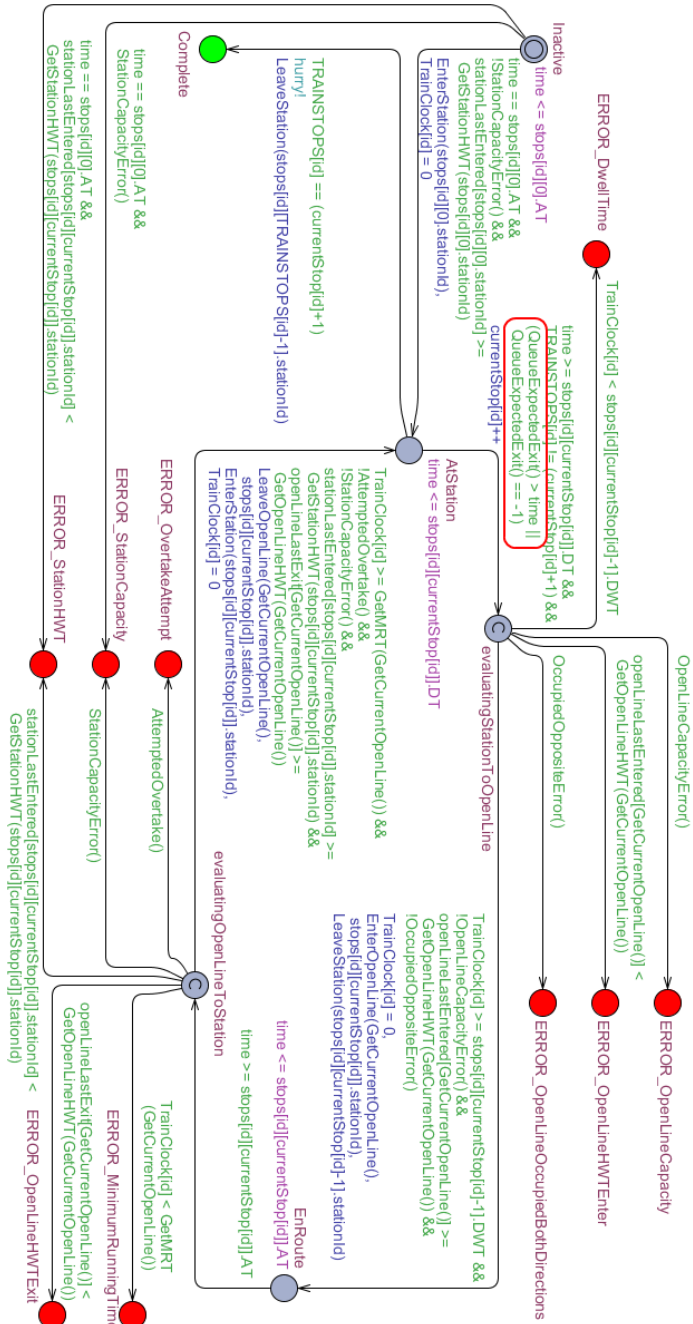


Figure 4.12: The final version of the Train template. The red circle represents the additions.

Multiple Train processes are created this way, as a result of the parameter of the Train template being a t_id , and a t_id is an integer in the range between zero and $TRAINS-1$.

The motivation for creating the Train processes in this manner, is to avoid having to alter the system declarations, when different input is provided to the model. This way, the correct amount of Train processes will be created automatically, based on the $TRAINS$ constant, which is provided as input.

4.2.6 Optimizations

During the development of this model, it was discovered that running time could be an issue, for validating large railway networks with many simultaneous trains. Therefore an effort was put into optimizing the running time of the model, and certain guidelines have been followed in order to secure a fast and still correct model:

Use as few clocks as possible Whenever a single clock can be used for different purposes, while the model is still correct, this should be done. The state space needed to be searched will increase with the amount of clocks added to the system - hence fewer clocks mean faster running time.

Avoid large select statements if possible Large select statements will cause the evaluation of an edge to increase in complexity, and because edges might need to be evaluated a great number of times, when using the verification tool of UPPAAL, large select statements can increase the running time notably.

Define variables within a range Whenever a variable is defined, it should be determined whether or not a range can be defined with it. When defining a range for a variable, it reduces the state space needed to be searched when using the verification tool of UPPAAL, thus increasing the running time.

When following these guidelines, the final model is able to verify an acceptable number of trains in a fairly complex railway network - the running time limitations will be presented further in section 7.

4.3 Getting Results Using The Model-checker

When using the created model to verify timetables, the model-checker of UPPAAL is used. In order to use the model-checker, a set of queries are created, one query for each of the error, and one query for each timetable to be verified. Figure 4.13 shows the nine queries of the error states, and four queries for the timetables.

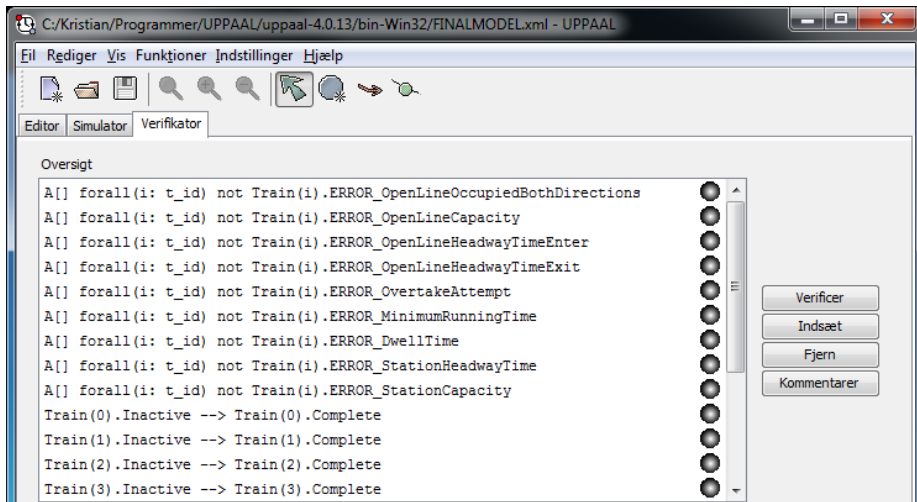


Figure 4.13: The queries, used by the model-checker to verify four timetables.

The top nine queries of figure 4.13, regarding the error states can be read as follows: 'For all states of all Train processes Train(i), Train(i) is not in the state ERROR_...'. This effectively means that no Train process ever enters an error state. The remaining queries can be read as follows: 'Once Train(n) enters the state InActive, it will eventually enter the state Complete'. Due to the fact that InActive is the initial state, these conditions state that each Train process will eventually enter the Complete state. It should be noted that a query of this kind should be made for each timetable to be verified.

In order to verify these, it is important to choose the correct options for verifications in UPPAAL. The options is set under 'Indstillinger' (Settings), and the following options should be chosen¹²:

Søgeorden (Search Order) Bredde først (Breadth First) - The model-checker

¹²These decisions are based on the explanations of the options, in the Help section of UPPAAL.

needs to search all states, to verify that the error states are never entered, and breadth first is the most efficient option when the complete state space must be searched.

Tilstandsrumreduktion (State Space Reduction) Ingen (None) - The memory size used by the model-checker is not considered a problem, therefore in order to obtain the greatest speed, no state space reduction should be performed. Choosing None has a notable effect on the running time.

Tilstandsrumrepræsentation (State Space Representation) DBM - Again the memory used by the model-checker is not considered a problem, and this option is stated to work fast, but may require a lot of memory.

Diagnostisk spor (Diagnostic Trace) This choice of this option varies, depending on the use of the model-checker. When wanting to validate all of the queries, the option 'Ingen' (None) should be chosen, as this allows the user to select and verify all of the queries simultaneously. If a condition was found to be invalid, this option should be 'En eller anden' (Some), allowing the user to select the a single property, and generate a diagnostic trace which shows the state invalidating the condition.

Ekstrapolation (Extrapolation) Automatisk (Automatic) - This options deals with the regard of termination, and UPPAAL has the possibility of determining the best way of doing this automatically.

Størrelse på hashtabel (Hash table Size) Is irrelevant, as it is stated to have no effect unless under 'Underapproximering' (Under Approximation) has been selected as the option for extrapolation.

Genbrug (Reuse) Should be checked, as this allows the model-checker to reuse a generated portion of the state space, when several queries are checked.

In order to check the conditions of figure 4.13, they are all marked and the button 'Verificer' (Verify) is pressed. Once the model-checker has completed, a successful will be shown as seen in figure 4.14, where all the conditions are validated, which can be seen by the green lamp next to each query.

If a condition fails, it will be marked by the lamp next to the query appearing as red. In figure 4.15, the capacity of a station has been exceeded in the provided collection of timetables, and it can be seen to be the process Train(2) which has caused it. Once this error has been detected, it is possible to identify how, where and when it occurred by choosing the 'Some' option of the Diagnostic Trace. Then the station capacity query should be chosen and verified by itself, resulting in a diagnostic trace, which will provide the entire diagnostic trace up until the state of the error.

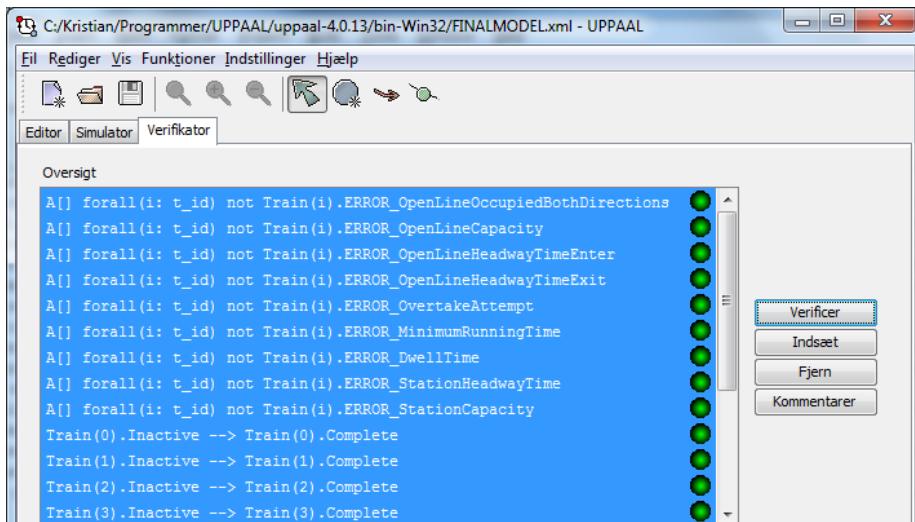


Figure 4.14: All of the queries are successfully checked, represented by a green lamp next to each query. In this case, the collection of timetables has been validated.

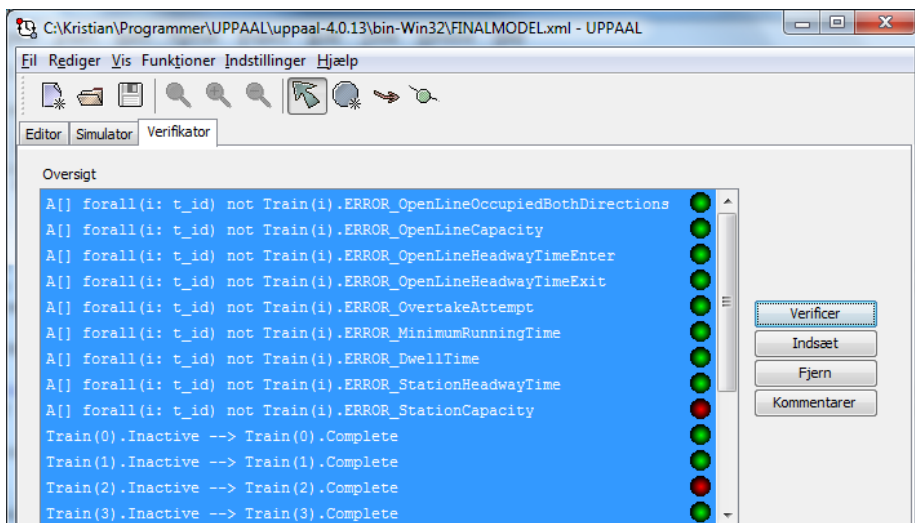


Figure 4.15: A station capacity error has been detected in the collection of timetables.

CHAPTER 5

Using UPPAAL CORA To Generate Timetables

This chapter will introduce UPPAAL CORA, present the optimality definition used in this thesis, it will present the model created in UPPAAL CORA, and lastly the chapter will present how output is created.

Regular UPPAAL can search for the shortest (least steps) or the fastest (least time) diagnostic trace - but it does not have any regular means of searching for an optimal diagnostic trace based on custom parameters. For this, a branch of UPPAAL exists, called UPPAAL CORA¹ (**C**ost **O**ptimal **R**eachability **A**nalysis).

In this chapter, section 5.1 will introduce the extra features available in UPPAAL CORA.

Section 5.2 will explain the considerations taken, when defining optimality in timetables in this thesis.

Section 5.3 will present the model created in UPPAAL CORA, which can be used to create a set of timetables.

Section 5.4 will end the chapter, by presenting how results can be extracted, using the created model.

¹<http://people.cs.aau.dk/~adavid/cora/index.html>

5.1 Cost and Remaining of UPPAAL CORA

UPPAAL CORA is an extension of UPPAAL. Therefore, any valid models created in UPPAAL, are also valid in UPPAAL CORA. UPPAAL CORA introduces the notion of *cost* to the templates, by introducing a predefined variable *cost*.

In order to utilize the costs defined in a model, the model-checker of UPPAAL CORA has been extended with the possibility of getting the best diagnostic trace based on the cost value, i.e. the diagnostic trace with least cost, see figure 5.1.

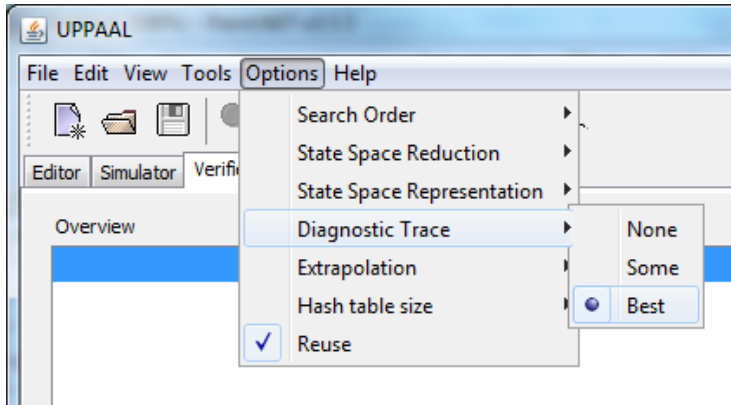


Figure 5.1: The best diagnostic trace option of UPPAAL CORA

Based on the cost, it is therefore possible to guide the diagnostic trace towards an optimal trace, where the optimality is expressed through cost in a template. Cost can either be included in the invariant of a state, or in the update label of an edge:

- When a cost is declared as an invariant in a state, the cost defines the rate at which the cost variable grows, for each time unit spend in said state. If several different processes are in a state, which has a cost, the rate is the sum of these costs.
- When a cost is declared as an update on an edge, it acts as an incrementation of the cost variable, when that edge is taken.

As a small example, figure 5.2, shows a template with two states A and B and a clock x . It costs nothing to spend time in state A, it has a cost 1 for each

time unit spent in B, and the selfloop in A increments the cost with 3. Table 5.1 shows an example simulation, where seven edges are taken, and the cost is shown for each transition.

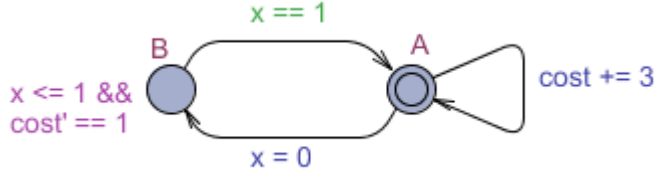


Figure 5.2: Cost defined in a state invariant, and as an update on an edge, x is a clock.

Transition Taken	Current State	Cost
-	A	0
A \rightarrow A	A	3
A \rightarrow A	A	6
A \rightarrow B	B	6
B \rightarrow A	A	7
A \rightarrow B	B	7
B \rightarrow A	A	8
A \rightarrow A	A	11

Table 5.1: Sample simulation trace of figure 5.2

UPPAAL CORA also introduces the notion of *remaining*. The purpose of the remaining variable of UPPAAL CORA, is to improve the performance of the model-checker. It is used by declaring remaining as a meta integer variable. The value stored in remaining should be an estimate of the remaining cost to reach a goal in the model-checker. It must always be *admissible*, meaning that it should always be equal to, or less then the actual minimum cost for reaching the goal.

5.2 Optimality in Timetables

This section will briefly present the considerations taken, when deciding which factors to take into account in determining whether or not a collection of timetables is optimal.

In order to determine whether or not a collection of timetables is optimal, several

factors are relevant to take in consideration. The following list presents the four factors used in this thesis:

- Short travel times between stations.
- Short dwell time at stations.
- Robustness against delays.
- Timetables should be prioritized equally.

The first two items present factors which aims at timetables to schedule for trains to run as fast as possible all of the time. If these were the only criteria in determining an optimal collection of timetables, the resulting timetables would handle delay poorly, as they would have no chance of eliminating delay without rescheduling. The third criteria represents the fact that an optimal collection of timetables should also have a degree of robustness against delays. Methods for increasing robustness against delays were introduced in section [2.4.2](#).

The methods presented in section [2.4.2](#), will all decrease the regularity of a collection of timetables, hence the two first items and the third item counteract each other. In order to establish the technical criteria for an optimal collection of timetables, a trade-off must be established.

In this thesis, robustness is introduced, by adding a running time supplement to each open line. As a result, the summation of the following values are kept as close to zero as possible in an optimal collection of timetables:

- The dwell time at stations exceeding the desired dwell time (additional dwell time).
- The travel time between stations exceeding the minimum running time + the running time supplement (additional running time).

It is important to note that optimality is defined for a *collection* of timetables. If the first three items were to be the final definition of optimality, a collection of timetables could have all of the additional dwell time and additional running time collected on a single timetable, severely reducing the quality of a single timetable compared to the rest of the timetables.

In order to avoid this, the fact that timetables should be prioritized equally is introduced. This represents the fact that the additional dwell time and the

additional running time of a collection of timetables, should be equally distributed out on all of the timetables whenever possible, hereby avoiding the scenario where a few timetables are of severely poorer quality than the rest of the timetables.

5.3 UPPAAL CORA Model

When creating the UPPAAL CORA model for generating timetables, inspiration was mainly drawn from two related models; The UPPAAL model for verifying timetables of chapter 4, and a model presented on the UPPAAL CORA website², called *Aircraft Landing Problem* (ALP).

The UPPAAL model for verifying timetables, which is described in section 4, was used as a starting point on how the different information should be stored in datastructures, as much of the same type of information is required for this model. The basic idea of the generic template was also reused, making it possible to create a custom railway network, and any amount of trains. Both of these attributes contribute to the fact that the UPPAAL model for verifying timetables, and the UPPAAL CORA model for generating timetables, share certain similarities.

The case study presented on the website of UPPAAL CORA, called *Aircraft Landing Problem* (ALP), simulates a number of aircrafts needing to land on a number of runways at an airport. The aircrafts are designated to land at a certain time. If the aircraft lands earlier, a penalty is issued by the airport - hereby increasing the total cost of the landing sequence. If the aircraft lands later, the cost is increased both by a penalty as well as cost of fuel. The goal of the ALP model, is then to present the cheapest sequence of landing the planes, by utilizing the *cost* feature of UPPAAL CORA. The concept of aircrafts being required to land at runways at a certain time, as well as the notion of introducing penalties when arriving earlier or later than a designated time, both have certain similarities to the process of generating timetables, as trains are set to arrive at stations at certain time, and passenger dissatisfaction, and errors following delays can be considered penalties - hence inspiration in utilizing the cost feature of UPPAAL CORA, was drawn from this model.

The model for generating timetables consists of one main Train template, which is a template, capable of generating timetables, based on the railway network information, as well as the timetable requests stated in the global declarations.

²<http://people.cs.aau.dk/~adavid/cora/casestudies.html>

It should be noted that a second template exists, called Hurry, which is similar to the Hurry template of the model for verifying timetables (seen in figure 4.2).

5.3.1 Timetable Request - The Input

In order for the model to generate a collection of timetables, it requires some input from the user:

A railway network including an open line table and a station table is needed³.

A running time supplement (RTS) for each of the open lines of the railway network. This is based on a percentage, and should be calculated for each open line in the network, based on their minimum running times. This value exists to increase robustness against delays in the created collection of timetables.

A route for each train in the desired collection of timetables. This should be an ordered list of stations, representing the stops of the desired timetable. For each stop, a desired dwell time should be indicated.

A start time interval for each route stating a time period for the arrival time of the first stop of the route to be placed in.

Four cost rates related to the cost utility of UPPAAL CORA. Their usage is presented in section 5.3.3.

Two time threshold values used to distribute the additional running time and additional dwell time in the created timetables. Their usage is also presented in section 5.3.3.

All of these values are to be specified by the user of the model, and are declared in the global declarations.

5.3.2 Global Declarations

The global declarations of the UPPAAL CORA model, are almost identical to the global declarations of the global declarations in the model for verifying timetables, which are presented in section 4.2.2. The following list introduces the changes in the global declarations.

³Same as for the model in chapter 4

- The *stopEntry* datatype has been altered.
- More constants were added in order to accomodate new input associated with this model.
- The need for *expectedExit* in *queueEntry* has been removed, hence the datatype *queueEntry* has been removed.
- The *queue* variable has been altered.

It should be noted that no new clocks, channels or functions have been added.

5.3.2.1 New Datatypes

The *queueEntry* datatype has been removed, and the *stopEntry* datatype has been altered to accomodate the request input, storing a station of a desired stop and a desired dwell time:

```
typedef struct {
int[-1,STATIONS-1] stationId;
int DWT; //Dwell Time
}StopEntry;
```

5.3.2.2 New Constants

All of the existing constants of the global declarations of the UPPAAL Model for verifying timetables are still present, and can be seen in section 4.2.2.2.

The new constants of this model, store the new input of the model, as described in section 5.3.1, for which the purposes are clarified in section 5.3.3.

The *stopEntry* datatype has also been altered, hence the constant stops have been altered. It is displayed below with with an example request for two timetables in the railway network Lokalbanel[Lok]. The following are the new constants with sample values, and the new *stops* constant, with sample values for two timetable request in the railway network of Lokalbanel:

```
//The costs for waiting in the non-expensive states
const int WaitingAtStation = 1;
```

```

const int WaitingAtOpenLine = 1;

//The costs for waiting in the expensive states
const int WaitingAtStationExp = 3;
const int WaitingAtOpenLineExp = 3;

//Threshold for cost increase, expressed as time
const int StationCostThreshold = 3;
const int OpenLineCostThreshold = 3;

//Start time intervals for each train (2 trains)
const int intervals[TRAINS][2] = {{0,3}, {4,4}};

//The running time supplement for each open line
const int RTS[OPENLINES] = {0,0,0,0,0,0,0,0};

//The timetable requests
const StopEntry stops[TRAINS][9] = {
  {{jagersborg,0},
   {norgaardsvej,0},
   {lyngbylokal,0},
   {fuglevad,0},
   {brede,0},
   {orholm,0},
   {ravnholm,0},
   {narum,0},
   {-1,-1}},

  {{remisen,0},
   {jagersborg,0},
   {norgaardsvej,0},
   {lyngbylokal,0},
   {fuglevad,0},
   {brede,0},
   {orholm,0},
   {ravnholm,0},
   {narum,0}}};

```

5.3.2.3 New Variables

The only change to the variables of the model, has been to the queue, which no longer stores the expectedExit value, hence the queue variable stores the order,

in which the trains entered an open line:

```
/*A queue for each direction of an open
line denoting the order in which trains enter.*/
int queue[OPENLINES][2][TRAINS];
```

5.3.3 Train Template

The Train template represents a train given one of the routes of the request explained in section 5.3.1. The Train template of this model, has the same parameter as the Train template for the model of section 4.2.4 - an id of datatype *t_id*.

In order to gain an overview of the general structure of the template, figure 5.3 displays the template without guards, updates, synchronizations and state invariants (selects are never used). A train has three types of states, which are similar to the corresponding types of states in the model for verifying timetables (section 4.2.4):

Inactive - Gray When a train is inactive, it has not yet been placed in the railway network. This can be for one of the three following reasons:

- It is earlier than the earliest time in the start time interval, for that train.
- The current time is within the start time interval, but it is not yet possible to enter the first station.
- It is passed the latest time in the start time interval, and the train will never be allowed to enter the system. If this is the reason, the current diagnostic trace is not valid.

Active - Blue When a train is in one of the active states, it means that it is currently trying to complete a route expressed in the request.

Complete - Green When a train is complete, it means that it has successfully journeyed through its given route - if all trains are completed, it means a collection of timetables has been successfully created, although not necessarily an optimal collection of timetables.

When comparing with the template used to verify timetables (seen in figure 4.4), it can be seen that there are four similar states:

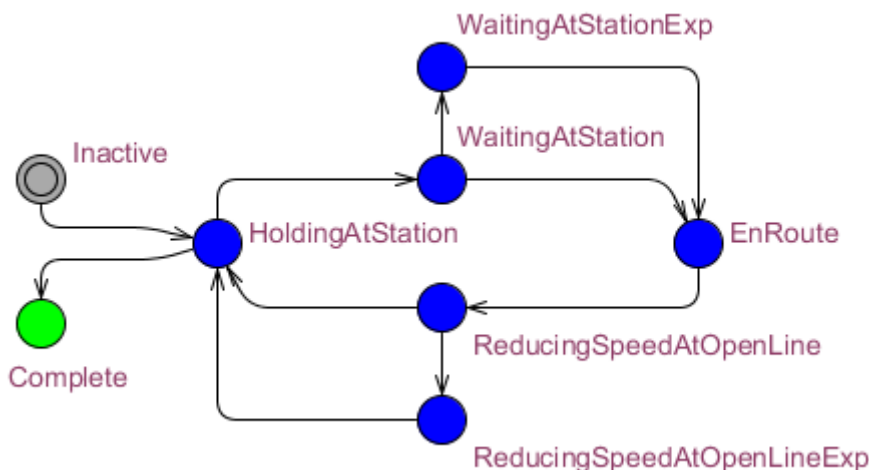


Figure 5.3: The Train template, stripped of everything except state names and edges. The color coding is explained in section 5.3.3

Inactive The train has not yet entered the railway network.

Complete The train has completed its route.

HoldingAtStation The train is currently holding at a station.

EnRoute The train is currently travelling on an open line.

All of these states have the same meaning as the corresponding states of the verification model. In addition to these four states, the model for generating timetables contains four other states. These four states are all related to the cost functionality, and their purpose is to construct diagnostic traces, optimal in regards to section 5.2 - these four states are:

WaitingAtStation When a train spend time in this state, it means that the train is holding at a station for longer time than is defined as necessary by the dwell time of the request. The reason for spending time in this state, is that the open line has restrictions, which prohibit the train from entering the open line at the moment.

The motivation for introducing this state, is the fact that spending additional time at a station, should be avoided if possible. Therefore spending time in this state increases the cost of the diagnostic trace - hence it is

desirable to pass through this state and on to the *EnRoute* state as fast as possible.

WaitingAtStationExp When a train has spent a certain amount of time in *WaitingAtStation*, it is forced to enter this state. Being in this state is more expensive than being in *WaitingAtStation*.

The motivation for this state, is to distribute the additional dwell time for trains. This is achieved because after a certain amount of time has passed, it will become more expensive to wait at the station. Therefore it is cheaper to have two trains in *WaitingAtStation* for x amount of time, than to have one train in this state for $2x$ amount of time - assuming that $2x$ causes the Train process to enter this state.

ReducingSpeedAtOpenLine When a train spend time in this state, it is because the train is not able to enter the destination station. The reason for spending time in this state, is that the station has restrictions, which prohibit the train from entering the station at the moment.

The motivation for introducing this state, is much the same as for *WaitingAtStation*. The fact that spending additional time at an open line, should be avoided if possible. Therefore spending time in this state increases the cost of the diagnostic trace - hence it is desirable to pass through this state and on to the station as fast as possible.

ReducingSpeedAtOpenLineExp Is the same to *ReducingSpeedAtOpenLine*, as *WaitingAtStationExp* is to *WaitingAtStation*. When a certain amount of time has been spent at *ReducingSpeedAtOpenLine*, the train goes to this state, with a higher cost - effectively spreading out the additional running time between stations in the collection of timetables.

As a result of this cost structure, the cost rate of a diagnostic trace, will increase whenever a train enters either *WaitingAtStation* or *ReducingSpeedAtOpenLine*. The cost rate will increase even further, if a train enters one of the expensive states - *WaitingAtStationExp* or *ReducingSpeedAtOpenLineExp*, figure 5.4 shows the basic idea of the cost model.

The general differences between this model, and the model used to verify timetables of section 4 can be summarized in the following list:

- There are no evaluating states.
- There are no arrival times or departure times.
- There is a start interval for each requested timetable, which must be complied with.

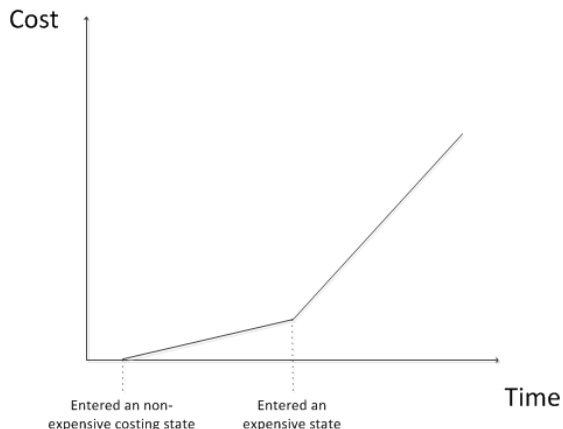


Figure 5.4: The cost of a diagnostic trace increases when entering a costing state, and the cost rate increases even further when entering an expensive state.

- The situation illustrated in figures 4.10 and 4.11 is disregarded for this model.

As with the model for verifying timetables, and as dictated by the formal model in RSL, there are the following eight properties to hold for a collection of timetables to be valid - the first three properties are affiliated with trains, the next two are affiliated with stations, and the last three are affiliated with open lines:

1. Trains cannot overtake another train on an open line.
2. Trains have to satisfy the minimum running times of the open lines.
3. Trains have to satisfy the dwell times at the stations.
4. The headway times of stations must be upheld.
5. The capacity of stations must never be exceeded.
6. The headway times of open lines must be upheld.
7. The capacity of the open lines must never be exceeded.
8. Single track open lines cannot be utilized in both directions simultaneously.

In order to introduce these properties into the model, a different approach is taken, than the approach of the UPPAAL model of section 4. In the previous model evaluation states exist, which lead to error states, in case a property has been broken due to the running of the trains. In this model, there are no evaluating states leading to error states. In this model, the conditions leading to error states in the model of section 4, are used to prevent the system from entering an erroneous state at all. This effectively ensures that the system only allows diagnostic traces, in which all of the properties hold.

The presentation will follow the same pattern as section 4.2.4. The edge labels and state invariants will be introduced to the template of figure 5.3 in small steps, until the final model is reached.

The method used to prevent trains from overtaking one another on an open line⁴ is much the same as in the verification model - the *queue* variable is used. Upon entering *EnRoute*, the train enters the queue, and when attempting to enter *AtStation*, the train must be at the front of the queue. Figure 5.5 has added the property of trains not being able to overtake one another. The methods *AttemptedOvertake()* and *GetCurrentOpenLine* are declared in the template of *Train*, and they are identical to those of section 4.2.4.

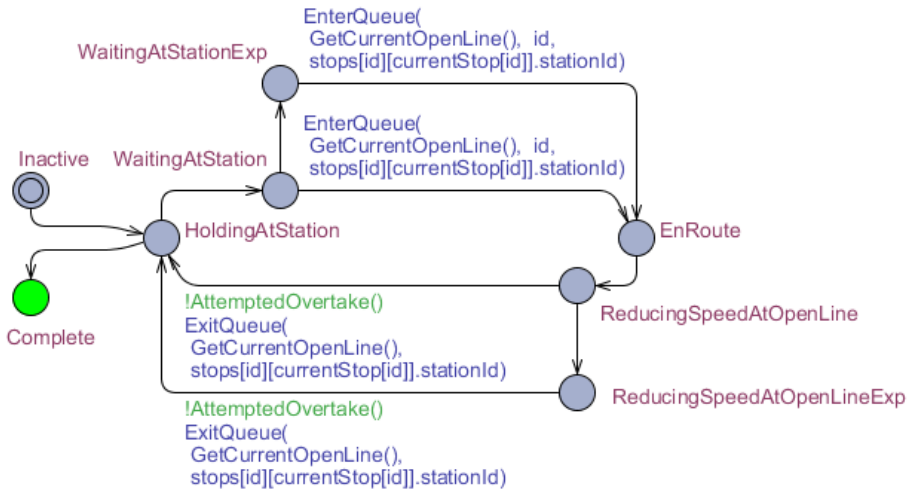


Figure 5.5: Trains cannot overtake has been incorporated into the model

In order to use the model to create timetables where trains satisfy the minimum

⁴Property 1

running times of open lines⁵ and the dwell times at stations⁶ - the same approach as section 4.2.4 is used; *TrainClock* is reset to zero upon entering *AtStation* or *EnRoute*, the invariants of these states and the guards of the outgoing edges prevent the Train process from leaving the state, before the dwell time has passed for stations, and the minimum running time has passed for open lines. For the open line, an addition has been made of *RTS*⁷, in order to incorporate robustness versus delays in timetables, as discussed in section 5.2. Figure 5.6 has added the properties of satisfying dwell times at stations and minimum running times of open lines. The red circles are the additions regarding the dwell times, the yellow circles are the additions regarding the minimum running time.

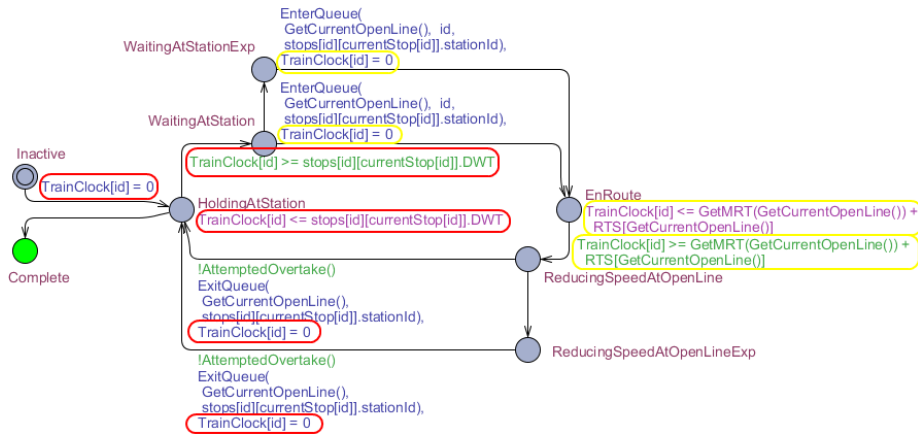


Figure 5.6: The three train properties are included in this figure, the red circles represent the additions for satisfying dwell time at stations, and the yellow circles represent the additions for satisfying minimum running times of open lines

When adding the property of capacity for stations⁸, the *trainsAtStation* variable is used in the same manner as section 4.2.4. It is incremented when a train enters *AtStation*, and decremented when a train enters *EnRoute*. A guard stating that there must be at least room for one more at the station, for a train to be able to enter, is then added on the edges between *ReducingSpeedAtOpenLine/ReducingSpeedAtOpenLineExp* and *HoldingAtStation*.

When adding the property of headway times for a station⁹, the clock *stationLastEntered* is used, representing the time elapsed since a train last en-

⁵Property 2

⁶Property 3

⁷Running Time Supplement

⁸Property 5

⁹Property 4

tered a station. This clock is reset to zero, each time a train enters `AtStation`. A guard is then added on the edge between `ReducingSpeedAtOpenLine/ReducingSpeedAtOpenLineExp` and `HoldingAtStation` and on the edge between `Inactive` and `Station`, stating that the value of said clock is equal to or greater than the headway time for the station. Figure 5.7 has added the two station properties, where the updating functions `EnterStation(int stationId)` and `LeaveStation(int stationId)` are identical to the same functions in section 4.2.4. The guard `StationHasRoom()` is declared as follows:

```
//Determine whether or not station has room for more
bool StationHasRoom()
{
    return trainsAtStation[stops[id][currentStop[id]].stationId] <
        GetStationCapacity(stops[id][currentStop[id]].stationId);
}
```

It should be noted that in the function `EnterStation`, a variable called `currentStation` is updated - this variable only used to parse the diagnostic trace in the tool presented in section 6, and does therefore not have any theoretical influence in the model.

When adding the property of capacity for open lines¹⁰, the variable `trainsAtOpenLine` is used, as in section 4.2.4. It is incremented when a train enters the open line, and decremented when a train leaves. For a single tracked open line, it is not necessary with a counter in each direction for this property, but the last property can utilize this, which is the reason for counters in both directions of an open line being utilized - regardless of the open line being single or double tracked. In addition to the counters, a guard is added on the edge between `WaitingAtStation/WaitingAtStationExp` and `EnRoute`, stating that there must be room for at least one more at that open line, and in that direction. This function is called `OpenLineHasRoom()` and is declared as follows:

```
// Determines whether or not an open line has room for one more
bool OpenLineHasRoom() {
    int openLineId = GetCurrentOpenLine();
    int dir = stops[id][currentStop[id]].stationId;
    //If doubletrack and the capacity in the direction has room for more
    if(IsOpenLineDoubleTrack(openLineId))
        return GetTrainsAtOpenLineDir(openLineId, dir) <
            GetOpenLineCapacity(openLineId);
}
```

¹⁰Property 6

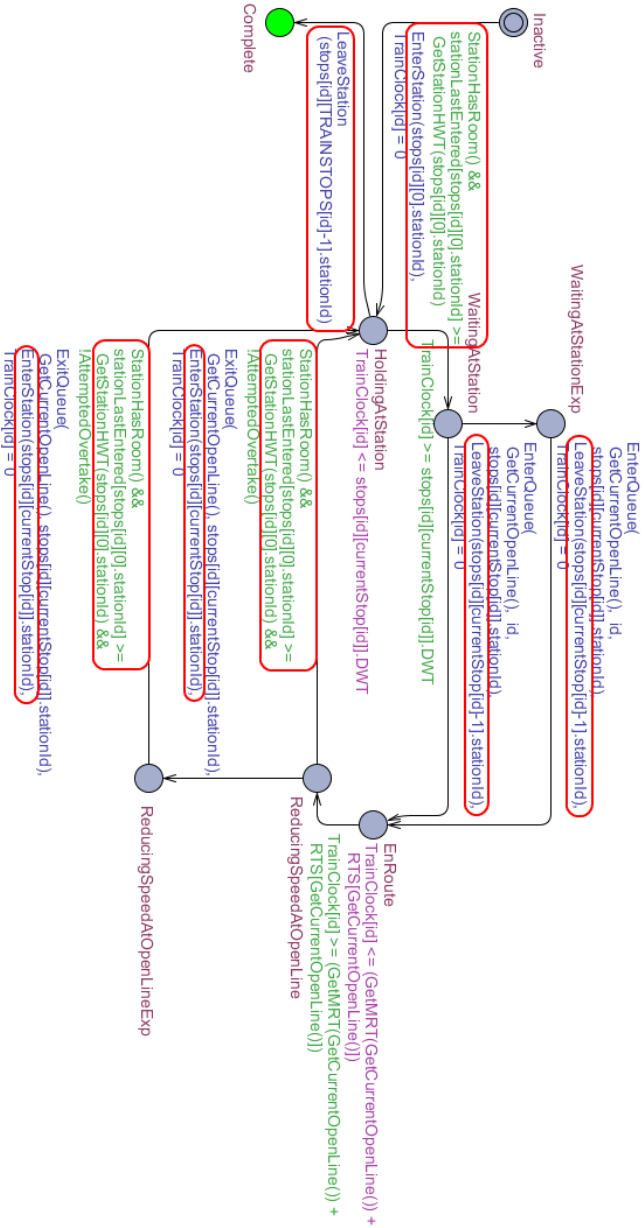


Figure 5.7: The three train properties and the two station properties are included in this figure. The red circles represent the additions for the two new station properties.

```

//If singletrack and the capacity has room for more
else
    return GetTrainsAtOpenLine(openLineId) < GetOpenLineCapacity(openLineId);
}

```

When adding the property of headway time for open lines¹¹, the two clocks `openLineLastEntered` and `openLineLastLeft` are used. The first one representing the time elapsed since a train last entered the open line, and the second one representing the time elapsed since a train last left the open line. These clocks are set to zero when a train enters `EnRoute` and when a train enters `AtStation`, respectively. In addition to these clocks, two guards are added, the first is added on the edge between `WaitingAtStation/WaitingAtStationExp` and `EnRoute`, which states that the time since the last train entered the open line, must be equal to or greater than the headway time of the open line. The second guard is added on the edge between `ReducingSpeedAtOpenLine/ReducingSpeedAtOpenLineExp` and `HoldingAtStation`, stating that the time since the last train left the open line, must be equal to or greater than the headway time of the open line.

When adding the final property of not allowing a single tracked open line to be utilized in both directions simultaneously¹², no new variable is added - the variable `trainsAtOpenLine` is sufficient. A guard is added on the edge between `WaitingAtStation/WaitingAtStationExp` and `EnRoute`, stating that if the open line is single tracked, the capacity counter of the opposite direction must be zero. Figure 5.8 has added the three open line properties, and the functions `LeaveOpenLine()` and `EnterOpenLine()` are identical to those of section 4.2.4 and `OpenLineNotOccupiedOpposite()` is declared as follows:

```

/*Determine whether or not the open line is occupied
in both directions and is single tracked*/
bool OpenLineNotOccupiedOpposite()
{
    return
        !(IsOpenLineOccupiedOppositeDirection
          (GetCurrentOpenLine(), stops[id][currentStop[id]].stationId) &&
          !IsOpenLineDoubleTrack(GetCurrentOpenLine()));
}

```

Now all the properties are in place, and it is time to add the cost functionality of the model. A cost is added to each of the four states `WaitingAtStation`, `Re-`

¹¹Property 7

¹²Property 8

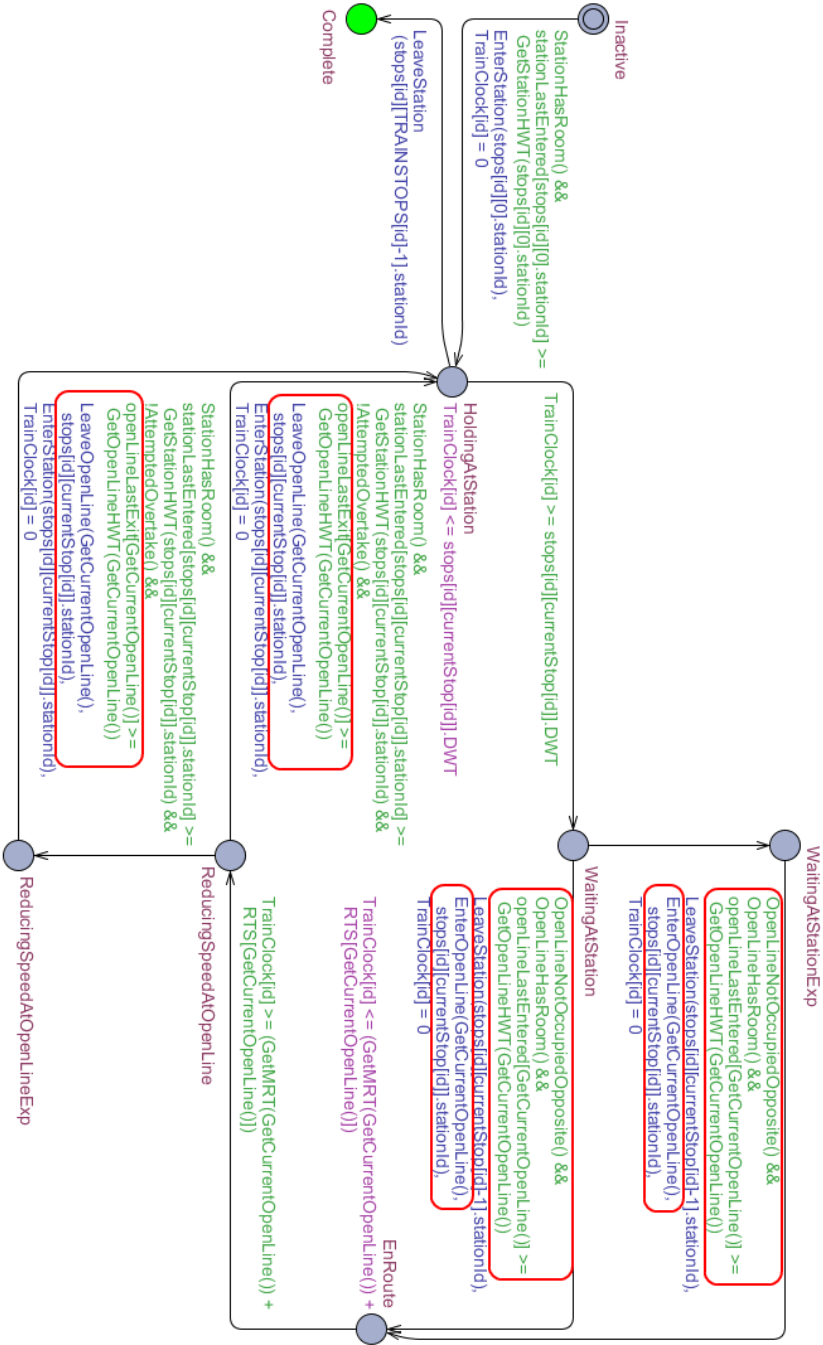


Figure 5.8: The three train properties, the two station properties and the three open line properties are included in this figure. The red circles represent the additions for the three new open line properties.

ducingSpeedAtOpenLine, WaitingAtStationExp and ReducingSpeedAtOpenLineExp - the cost of the two latter should be greater than the first two. In order to force the model to enter the expensive states, the TrainClock is reset to zero when entering these states, and when a period of time has elapsed, and they are still not able to advance (either to AtStation or to EnRoute), the model is forced into the expensive state due to an invariant in the least expensive state, and a guard between the two states. Figure 5.9 has added the cost properties, where the cost values as well as the timelimits in the least expensive states are declared as variables.

Two things remain to be added in order to achieve the finished template. The first is the same pattern of increasing the currentStop of the trains whenever they travel from one station to the other, and when the last stop is reached, the train will enter the Complete state - this is similar to the model for verification of timetables.

The second thing required is the fact that trains need to be inserted into the system in a certain time interval. This is achieved by adding a guard on the edge between Inactive and HoldingAtStation, stating that the current time must be between to two edges of the interval. The final template can be seen in figure 5.10.

5.3.4 Optimizations

All of the optimizations considered in the model for verifying timetables using regular UPPAAL (as explained in section 4.2.6), also applies to this model created in UPPAAL CORA. As a result, this model was created following the same guidelines as section 4.2.6 - reusing clocks, avoiding large select statements on edges and applying ranges to variables. With this in mind, the most effective optimization performed during the development of this model, has been reducing the number of templates, edges and states. Previous versions contained several templates, with more states and more edges. The evaluation complexity of the guards, has roughly remained the same - only spread out on more edges. Reducing these templates into a single template (disregarding the Hurry template), reduced both the state space and the size of a stored state - which effectively reduced the search time for the best diagnostic trace.

An optimization available only to UPPAAL CORA, is the remaining feature discussed in section 5.1. With this model however, it is difficult to provide an admissible evaluation of the remaining variable during a simulation. This is due to the fact that it is not easily derived whether or not a request of timetables, will actually result in any incrementation of the cost value at all.

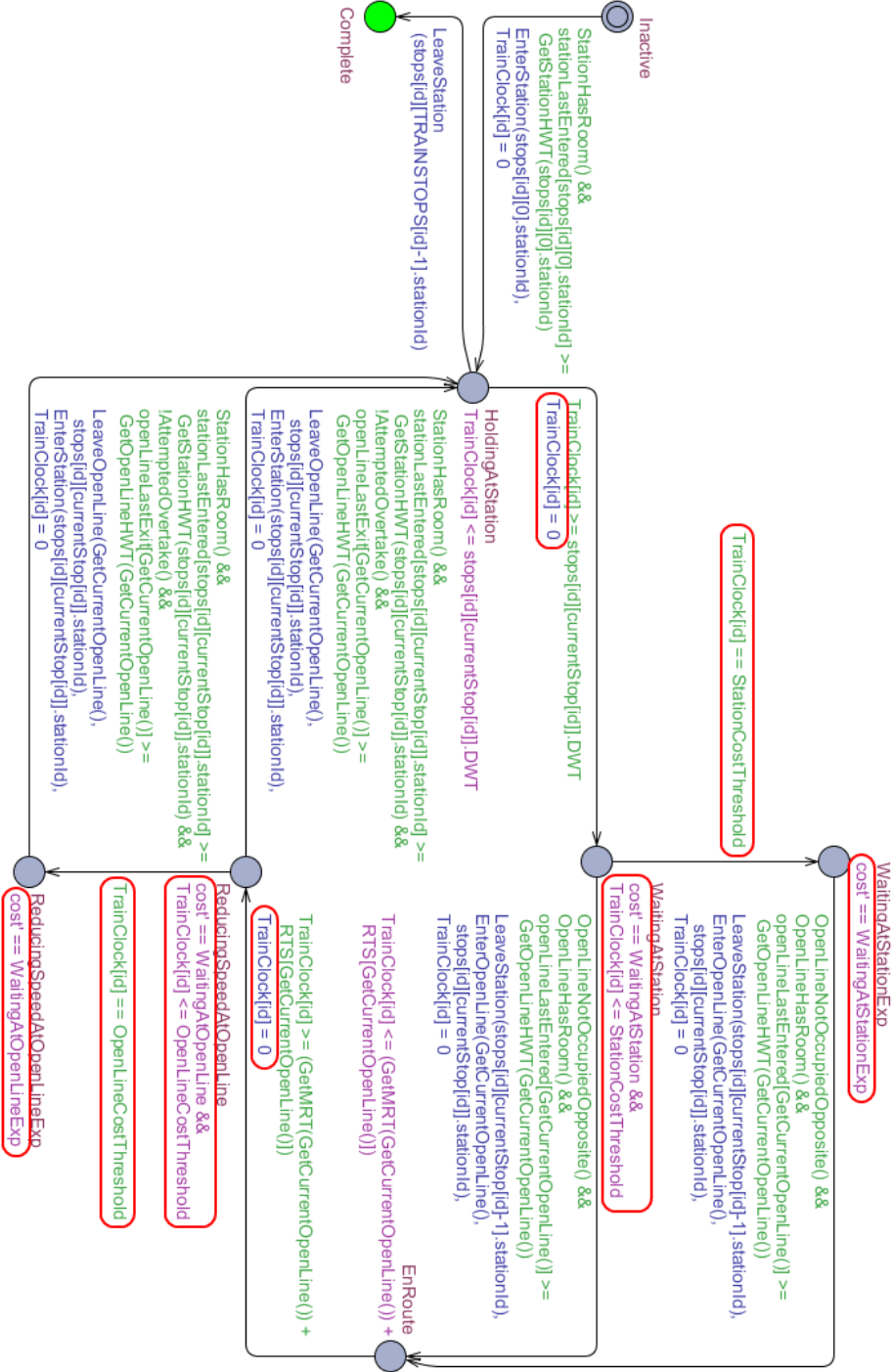


Figure 5.9: All of the properties as well as the cost functionality of the model as been added. The red circles represent the additions for the cost functionality.

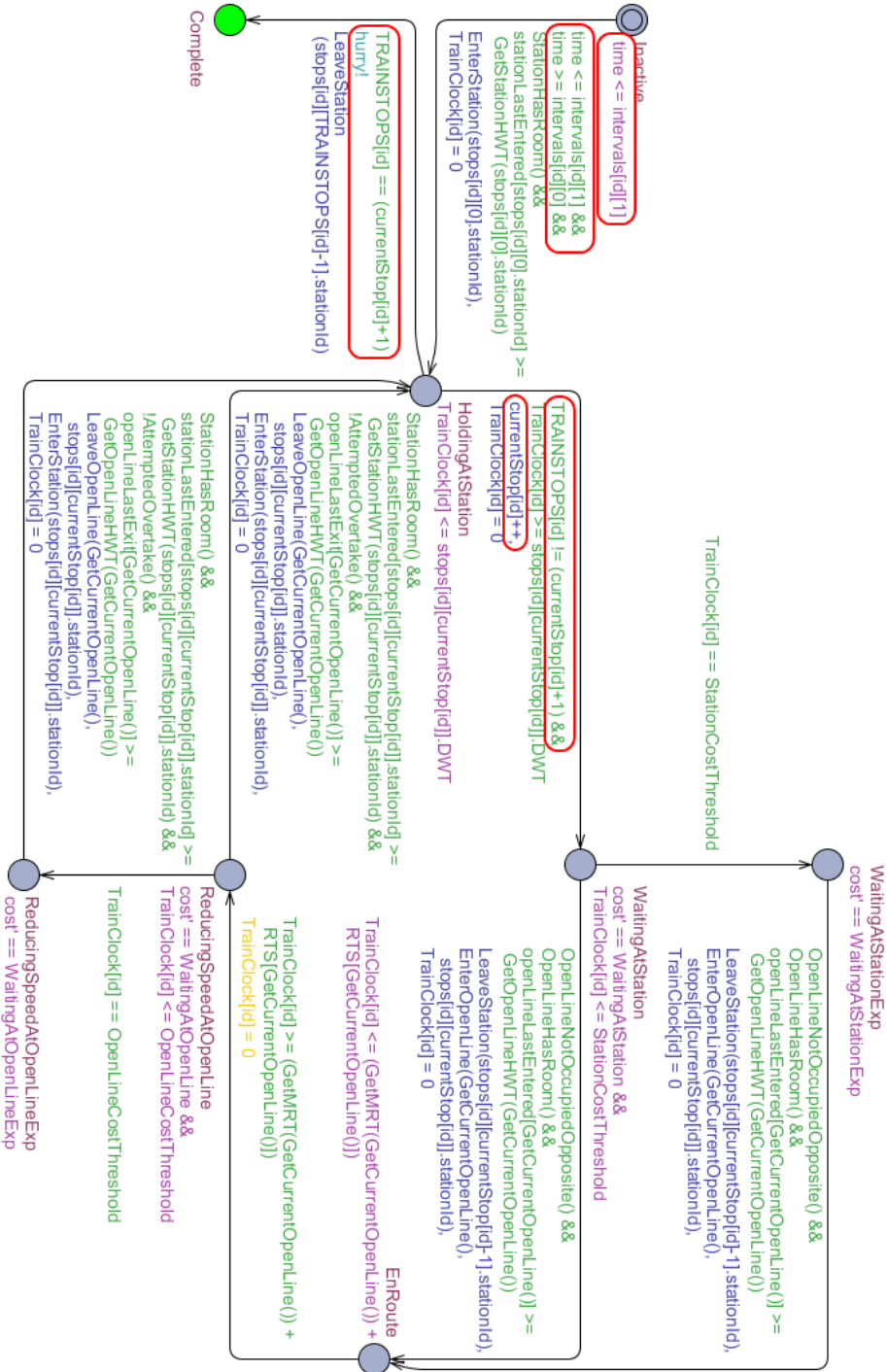


Figure 5.10: The final Train template of the UPPAAL CORA model

This would most likely require a custom analysis of the request, based on the railway network. Such an evaluation could in itself cause the model to become even slower, because it would have to be done very often. In further attempts to optimize this model, it could be very beneficial to investigate the possibilities of evaluating the remaining value during a simulation of the model.

5.4 Getting Results Using The Model-checker - The Output

When getting output from the model, there are two different ways of getting it - both ways require a query to be satisfied, opposed to the verification model, which contained several queries. The single query required for this model is stating that a state exists, in which all of the trains are in the complete state. In a model requiring to generate three timetables, the query would look like this:

```
E<>(Train(0).Complete && Train(1).Complete && Train(2).Complete)
```

One way of getting a result, is using the 'verifier' tab of UPPAAL CORA, which is somewhat similar to the 'verifikator' tab of regular UPPAAL - it simply contains different options. The options settings providing the fastest way to generate the best diagnostic for this thesis, is the following:

Search Order Best first - Due to the fact that it is looking for the best diagnostic trace. This option is important, and does provide significant improvement in the evaluation time.

State Space Reduction None or Conservative - One of these should be chosen, as memory limitations is not considered a problem, due to the fact that input large enough for the memory to become an issue, will most likely not be able to finish within a reasonable time anyway. The last option (Aggressive) slows the evaluation time noticeably. Whether None or Conservative is chosen, has not had any measurable effect during tests.

State Space Representation DBM - The other option (Compact Data Structure) is targeted for models with a large amount of clocks¹³, in order to reduce memory consumption. Again - memory consumption is not considered an issue, therefore DBM is chosen, which is faster, but requires more memory. During testing, it did have a noticeable impact choosing DBM.

¹³Stated in the 'Help' section of UPPAAL CORA

Diagnostic Trace Best - This is one of the main points for using UPPAAL CORA.

Extrapolation None or Automatic - This option is to help the model guarantee termination. As this current model can never continue indefinitely, no extrapolation is needed, and it is stated that extrapolation is 'relatively' expensive in the help section of UPPAAL CORA. In tests, it made no measurable difference between choosing None or Automatic.

Hash table size This option is stated to have no effect unless *under approximation* has been chosen in the state space representation option - as this particular option is not available in UPPAAL CORA, the hash table size option is apparently irrelevant. During tests, no measurable difference was noted between any of the available hash table sizes.

Reuse This option enables the verifier to reuse some of the state space, when verifying several properties - as only one property is verified in this case, this option is irrelevant. In tests, no measurable difference was noted when turning this on or off.

With these options selected, the user clicks the 'Check' button and, if possible, a diagnostic trace will appear in the simulator.

The second way, and the one used by the final tool of this thesis, is to use the command line:

```
verifyta -t3 -C model.xml query.q 2> diagnostictrace.txt
```

This command line produces a diagnostic trace in a text file. The options have the following meaning:

-t3 This option is equivalent to choosing the option Diagnostic Trace - Best, the option Search Order - Best First and disables reuse.

-C This option disables most memory reduction techniques.¹⁴

model.xml This is the model file.

query.q This is the query file.

2> diagnostictrace.txt This prints the diagnostic trace from standard error output to the file diagnostictrace.txt.

¹⁴This is quoted from the 'Help' of the command line option, the precise meaning of the word 'most' is unknown.

As the most crucial options of the model-checker are included in these options - this produces the best diagnostic trace, the fastest way possible. The final tool of this thesis parses the produced text into the final output of the tool.

The Tool

This chapter will describe the tool created in Java, for using the model of chapter 5 to generate timetables.

The final tool is stored in the physical CD, attached to the thesis, along with the source code.

Section 6.1 will provide an analysis of the requirements of the tool.

Section 6.2 will present the design decisions of the tool.

Section 6.3 will briefly present the technology used to create the tool, the structure of the tool, and the resulting look of the tool.

6.1 Analysis

The formal model of chapter 5 can be used to generate timetables, by using the model-checker of UPPAAL CORA. However, the process of providing input to the model, and the process of interpreting the output produced by UPPAAL CORA is quite complex.

In order to provide a greater foundation, to base an evaluation of using formal methods for timetabling on - a tool to use the model of chapter 5 to create timetables, is created. As a result, the complexity of providing input and interpreting output is reduced.

In this chapter, section 6.1.1 will present the general scope of the tool.

Section 6.1.2 will present the requirements of the tool, when creating a railway network.

Section 6.1.3 will present the requirements of the tool, when creating timetable requests.

Section 6.1.4 will present how the tool should visualize results.

Section 6.1.5 will present the requirements of the tool, in order to use the model-checker of UPPAAL CORA.

Section 6.1.6 will present the desired feature to decrease the running time of the tool, by limiting the complexity of the UPPAAL CORA model.

Section 6.1.7 will present required features of the tool, which are not necessarily specific for timetabling.

Section 6.1.8 will present limitations chosen to keep the development of the tool simple.

6.1.1 The Scope of the Tool

The purpose of this tool, is to act as the final step, in using formal methods to generate timetables. It has been created to present an example of how a formal model created in UPPAAL CORA, can be used to generate timetables, which are presented in a human readable fashion.

The scope of this tool is to act as a prototype, meaning that the basic desired functionality of the tool should be present, but it will not be able to detect or handle all types of errors, which can be introduced by users.

The existing tools TPS[TPS11] and RailSys[Lan10] both provide several features, which are out of scope for this tool, such as editing existing timetables, and visualizing railway networks. This tool focuses mainly on the creation of

timetables and visualizing the results, additional features of timetabling tools can be added in further work with the tool.

6.1.2 Create a Railway Network

The user should be able to create a railway network, using the tool. For the scope of this tool, this should be incorporated as two different lists:

- One list containing the open lines of the railway network, where each open line contains the same information as that of the model of chapter 5 (name, capacity and headway time).
- One list containing the stations of the railway network, where each station contains the same information as that of the model of chapter 5 (two end stations, capacity, headway time, minimum running time and whether or not it is a double track).

It should be possible to add, remove and edit open lines and stations in their respective lists. Furthermore it should only be possible to create open lines between stations which have already been created in the list of stations.

The graphical representation of the lists should be textual to keep it simple.

6.1.3 Create Timetables Requests

The user should be able to create timetable requests, using the tool. Each request should contain the same information as a request in chapter 5 (name, list of stops and start interval). These timetable requests should be in a list, and it should be possible to add, remove and edit timetable requests in the list. It should be noted that this also means editing the stops in the timetable request.

In a timetable request, it should only be possible to create stops at stations, which are already defined in the list of stations.

The graphical representation of the list should be textual to keep it simple.

6.1.4 Visualizing Output

In order to visualize the generated timetables, the tool should be able to print a textual representation of the generated timetables, stating the station, arrival time and departure time of each stop.

The tool should also be able to make graphs, as the graph in figure 2.2.

6.1.5 Using the model-checker of UPPAAL CORA

One of the points of this tool, is to provide an example of how a model created in UPPAAL CORA, can be used to generate timetables. This should be done by utilizing the model-checker of UPPAAL. As a result, the need for the following features are introduced:

- The tool should be able to create a model file and a query file of UPPAAL CORA, similar to those of chapter 5, where the constants of the model file are based on the user input of the tool.
- The tool should be able to interpret the output of the model-checker.
- The tool needs a reference to an installation of UPPAAL CORA.

6.1.6 Limiting the UPPAAL CORA Model

The time used by the model-checker of UPPAAL CORA to generate a diagnostic trace can be very long. In order to generate a greater amount of timetables, the tool should be able to remove certain properties of the model. This will decrease the complexity of the model, and therefore also decrease the running time. By adding this functionality to the tool, it is possible to create timetables, which interleave more, but are not guaranteed to preserve all of the properties defined in chapter 5.

The main motivation of being able to limit the UPPAAL CORA model, is to generate graphs containing more timetables, hereby also showing more of what the visualization part of the tool is capable of.

6.1.7 Regular Features

The tool should incorporate the following general features:

- It should be possible to save and load a railway network and timetable requests created in the tool.
- Due to the fact that generating timetables, by using the model-checker of UPPAAL CORA might take a long time, a cancel functionality should be incorporated, allowing the user to cancel the process of creating timetables.
- Once the reference to the install directory of UPPAAL CORA has been provided, the tool should be able to restore this after the tool has been closed.

6.1.8 Limiting the Complexity of the Tool

In order to keep the tool simple, and within the scope of the thesis, the following limitations are introduced:

- When attempting to generate timetables, the railway network must be valid, according to the RSL specifications of section 3.3.
- When attempting to generate timetables, the stops in the timetable requests must be traversable on the railway network.

Basically, this means that the tool will not check whether or not the railway network and timetable requests created by the user are valid.

6.2 Design

This section will present the design decisions made, when developing the tool.

When designing the structure of the tool, there are two main aspects to consider:

- The design pattern to use for the tool.

- How the tool should utilize the model-checker of UPPAAL CORA, to generate timetables.

The following sections will present these two items.

6.2.1 Design Pattern

The tool should be able to represent a model of a railway network, and a model of timetable requests. These models should be visualized in a graphical user interface (GUI), and the user should be able to edit these models through the GUI.

For a program with such a structure, the design pattern of model-view-controller (MVC)[KP88] is generally used a lot, and it is therefore chosen as the main design pattern of this tool.

The basic idea of MVC, is to separate the GUI (the view) from the data and business rules of the data model (the model). In order to have this separation, there needs to be an element which mediates the input of the GUI, to the model, and in turn updates the view to display the model correctly. Basically the three following components form the MVC design pattern:

Model This is where all of the data and business logic is stored, it should have no knowledge of the view.

View This is the GUI part of the program, providing the user with the possibility of providing input, and viewing the state of the model. The view should not have a direct reference to the model, but instead have the controller perform the necessary actions based on the input of the user.

Controller This is the link between the view and the model. The controller should provide methods to the view for retrieving information from the model and alter the model. The controller needs knowledge of both the view and the model.

There are many different variations of the MVC design pattern. Figure 6.1 shows the chosen MVC design pattern for this thesis. As figure 6.1 displays, the controller has a view and a model, and the model and the view each have a reference back to the controller. The only direct connection between the view and the model, is that the view may some times use the data types of the model

to display the elements, but in order to access the actual data, the view goes through the controller.

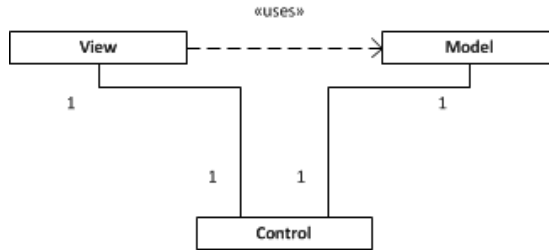


Figure 6.1: The MVC design pattern chosen for this thesis

6.2.2 Generating Timetables

In order to generate timetables, the tool will have to use the model-checker of UPPAAL CORA. As explained in section 6.1.5, a model file and a query file should be generated based on the user input, and the output should be parsed.

When generating a model file and a query file for UPPAAL CORA, the information of the data model of the tool is used. As both the model and the controller have direct access to the model, this functionality could be placed in either. Due to the fact that the model and query files of UPPAAL CORA are not actually a part of the data model of the tool, it has been placed in the controller.

When the controller has created the model and query files of UPPAAL CORA, based on the data model, the controller uses them as a parameter for the command line options of the model-checker of UPPAAL CORA (as explained in section 5.4, and stores the resulting diagnostic trace. In order to parse the diagnostic trace, the package TraceParser has been created, and the controller uses this package to parse the trace, after which the parsed trace is given to the view, which then visualizes it.

Figure 6.2 shows the steps taken by the tool when a user prompts the tool to generate timetables. It should be noted that the view (GUI) is blocked throughout the entire sequence. To accommodate this, a cancel box appears, providing the opportunity of cancelling the sequence.

The following two sections will describe how the model file and the query file of UPPAAL CORA are created for each specific request to generate timetables, and how the resulting diagnostic trace is interpreted.

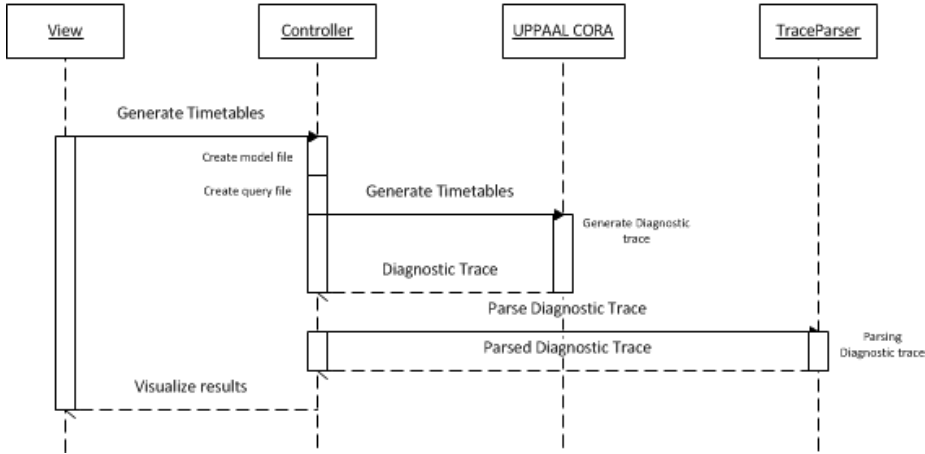


Figure 6.2: A sequence diagram showing the process of happening when a user asks the tool to generate timetables.

6.2.2.1 Creating the model and query file of UPPAAL CORA

As mentioned earlier, the model file and the query file of UPPAAL CORA are created for each request for timetables, are created based on the data stored in the tool. In order to do this, it is important to remember that the constants of the UPPAAL CORA model, represents the input of the model.

The tool does not create a new UPPAAL CORA model from scratch, instead, the existing model of UPPAAL CORA from chapter 5 is copied into a *resources* package in the tool, where the input constants of the UPPAAL CORA model are replaced by unique strings, which can be identified by the tool. For example the declaration of the *TRAINS* constant, is replaced by the string:

```
const int TRAINS = ##TRAINS##;
```

and the declaration of the constant *stops*, is replaced by the string:

```
const Stops stops[TRAINS][##MOSTSTOPS##] = ##STOPS##;
```

The tool identifies all of these replacement strings, and based on the data model of the tool, the correct values are inserted, such as the number of trains (replaces *##TRAINS##*), the most amount of stops in the timetable requests

(replaces `##MOSTSTOPS##`), and the stops of each timetable request (replaces `##STOPS##`).

The query file is created in the same manner. The query file of the UPPAAL CORA model of chapter 5 is copied into the *resources* package. The single query in the query file of the tool looks like this:

```
E<>(##TRAINS##)
```

Where `##TRAINS##` is then replaced by:

```
Train(0).Complete && Train(1).Complete && ... && Train(n).Complete
```

Where n is the number of timetable requests.

The UPPAAL CORA model files used in the tool, can be seen in appendices [B.1](#), [B.2](#), [B.3](#) and [B.4](#).

6.2.2.2 The Trace Parser

In order to understand the parsing of a diagnostic trace, the simple template of figure 6.3 is used to generate a diagnostic trace, validating the condition that the process of the template can enter state C:

```
E<>(Process.C)
```

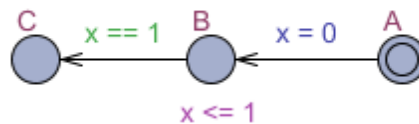


Figure 6.3: A simple template to generate a small diagnostic trace.

The diagnostic trace is as follows:

```
State:
( Process.A )
```

```
Process.x=0 rate=0 cost=0

Transitions:
  Process.A->Process.B { 1, tau, x := 0 }

State:
( Process.B )
Process.x=0 rate=0 cost=0

Delay: 1

State:
( Process.B )
Process.x=1 rate=0 cost=0

Transitions:
  Process.B->Process.C { x == 1, tau, 1 }

State:
( Process.C )
Process.x=1 rate=0 cost=0
```

From this diagnostic trace, one is able to read each state of the diagnostic trace. For each state, the state of each process is provided, as well as the values of all of the variables and clocks.

It is also possible to read each transition and time delay of the diagnostic trace. For each transition, the original state of the firing process and the destination state of the firing process is shown.

The Trace Parser utilizes this knowledges and uses regular expressions to identify each state and each transition or delay. For each state the values of the variables and clocks are stored. For each transition, the process firing, the original state and the destination state are stored.

Figure 6.4 shows the UML structure of the Trace Parser, showing that a State has a list of templates and a list of declarations. The templates list represents each process in the system and their current state, while the declarations list represents each variable and clock, associated with their values in the given state.

Based on this information, the Trace Parser is able to generate the timetable structure also shown in figure 6.4 (in this case the timetable is called a train).

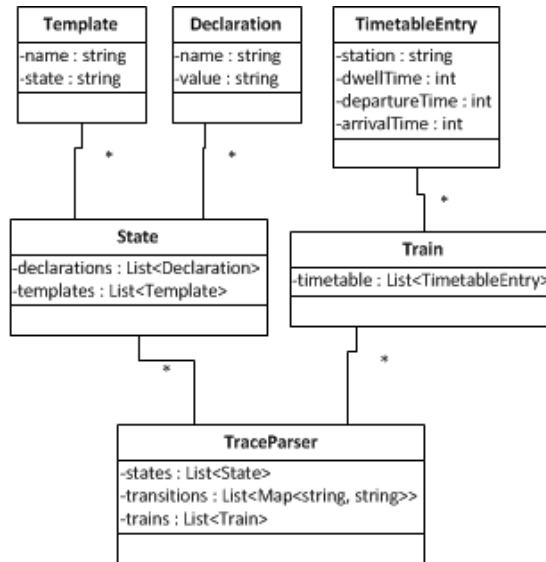


Figure 6.4: The structure of the TraceParser

This is done by going through all transitions and all states, and noting whenever a train enters or leaves station.

For further details, the source code is located on the attached CD.

6.3 Implementation

This section will present the technology used to create the tool, it will present the final structure of the tool, and it will briefly show the final look of the tool.

Technicalities of the Java programming language is not a priority in this thesis, therefore, detailed implementation issues regarding details of Java specific implementations are omitted. The source code of the tool is available on the attached CD.

6.3.1 Technology

In order to create the tool, Java is used with the NetBeans platform[Bi2]. NetBeans has a graphical editor for creating user interfaces in Java using Swing¹. As much of the work of this tool lies with the user interface, the graphical editor for creating user interfaces was the main reason for choosing NetBeans.

In order to visualize the output of the tool in a graph, the external library JFreeChart² is used. This library allows the tool to create charts similar to those of figure 2.2.

6.3.2 The Structure of the Tool

The UML diagram in figure 6.5, depicts the structure of the tool. It is possible to see the model of the tool, the view of the tool and the controller of the tool:

The model of the tool is grouped in a dashed box marked *Model*, and the class `TimetablesToolData` is the main class of the model. The model should be able to represent a railway network and timetable requests, as stated in the previous models of chapters 3, 4 and 5. The railway network can be seen in the class `TimetablesToolData`, in the list of `Stations` and a set of `OpenLines`, which together form the railway network. As for the timetable requests, it can also be seen that `TimetablesToolData` has a list of `TimetableRequests`.

The view consists of the single class `TimetablesToolGUI`, which presents the user interface. It uses the `JFreeChart` library to create a graph for visualizing results. The view also uses some of the data types of the model to present the GUI.

The controller is also a single class, it has a reference to both the view and the model. The controller also uses the *resources* package, which contains the model files and query file explained in section 6.2.2.1. Furthermore the controller uses the parser to parse the diagnostic trace, created by the model-checker of UPPAAL CORA, and passes the parsed diagnostic trace on to the view.

¹A Java library used to create user interfaces.

²JFreeChart is a free Java chart library, used to display charts. It is distributed under the terms of the GNU Lesser General Public License (LGPL), which allows for its use in this thesis. <http://www.jfree.org/jfreechart/>

The class containing the main function of the tool is `TimetablesTool`, which is also where and object of the model (`TimetablesToolData`), the view (`TimetablesToolGUI`) and the controller (`Controller`) is created.

It should be noted that the structure of the program, enables a quick and easy way of implementing saving and loading. All of the data is stored in the `TimetablesToolData` object, and as a result, the classes used by `TimetablesToolData` and the class itself, simply needs to implement the interface *Serializable* of Java. Then the `TimetablesToolData` object is the only object which needs to be saved to a file or loaded from a file.

6.3.3 The Final Look of the Tool

This section will show the visual results of the requirements stated in section 6.1. It should be noted that appendix D provides a complete guide to the tool.

Figure 6.6 shows the tool, where the railway network of Lokalbansen has been created. This part of the tool is the result of the requirements stated in section 6.1.2

Figure 6.7 shows the tool, where four timetable requests have been created for the railway network of Lokalbansen, and the tool is currently working on generating timetables. This part of the tool is the result of section 6.1.3 and partly the results of section 6.1.7.

Figures 6.8 and 6.9 show the results of the generated timetables. Figure 6.8 shows the generated graph and figure 6.9 shows the text representations of the timetables. It should be noted that time is increased going up the vertical axis (not down as in figure 2.2. This part of the tool is the result of section 6.1.4.

For a full tour of the tool, the user is referred to appendix D.

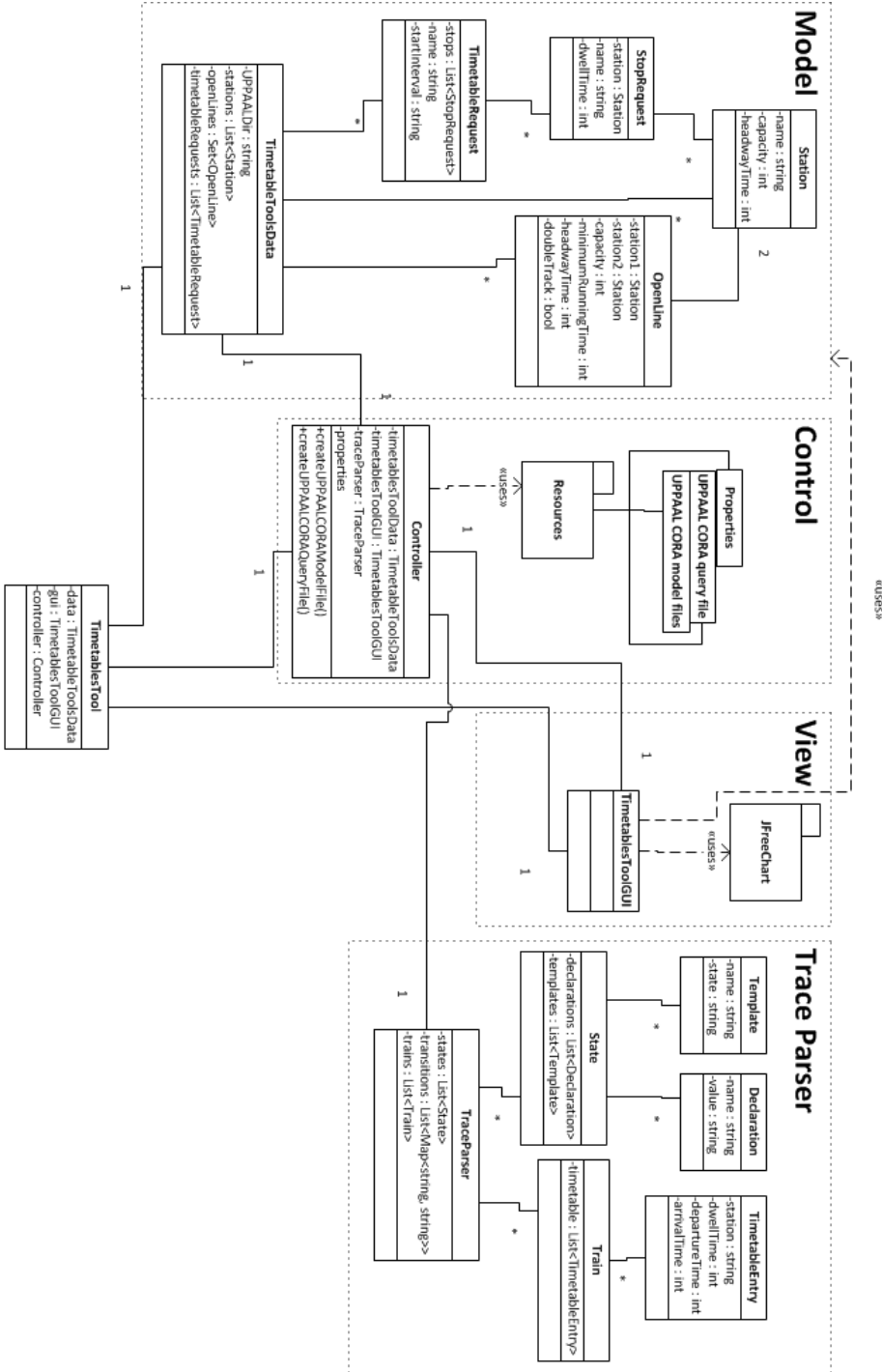


Figure 6.5: A UML diagram, depicting the structure of the tool, which follows the MVC design pattern.

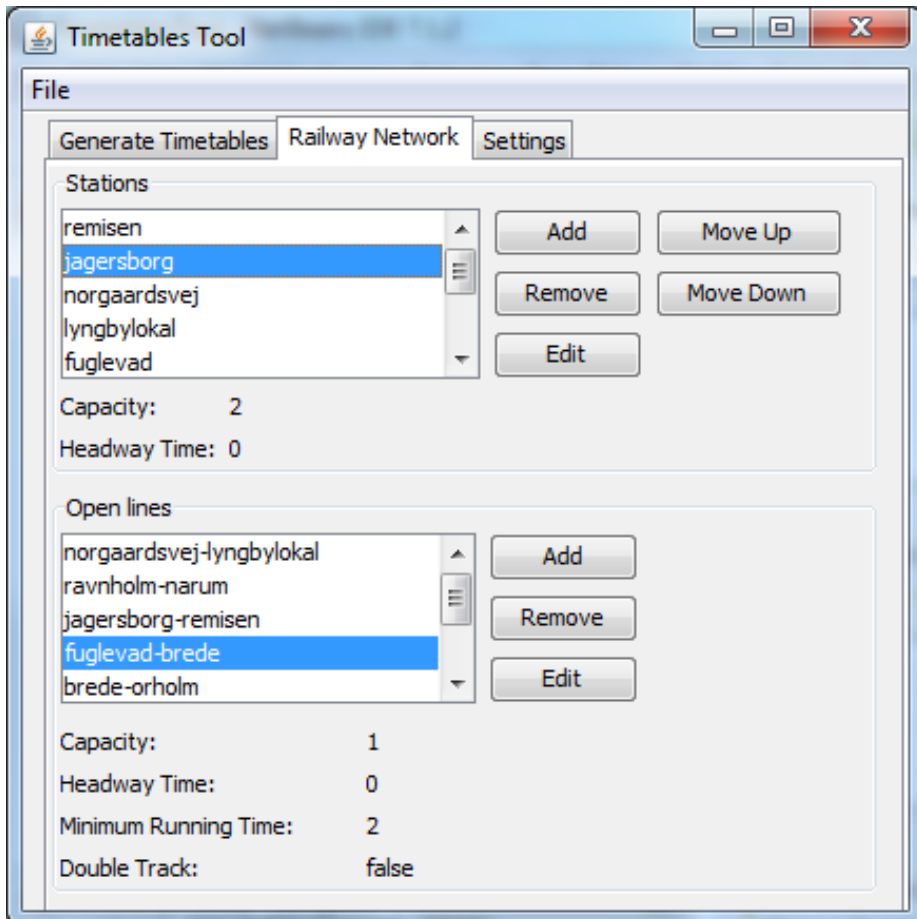


Figure 6.6: Using the tool to create a railway network.

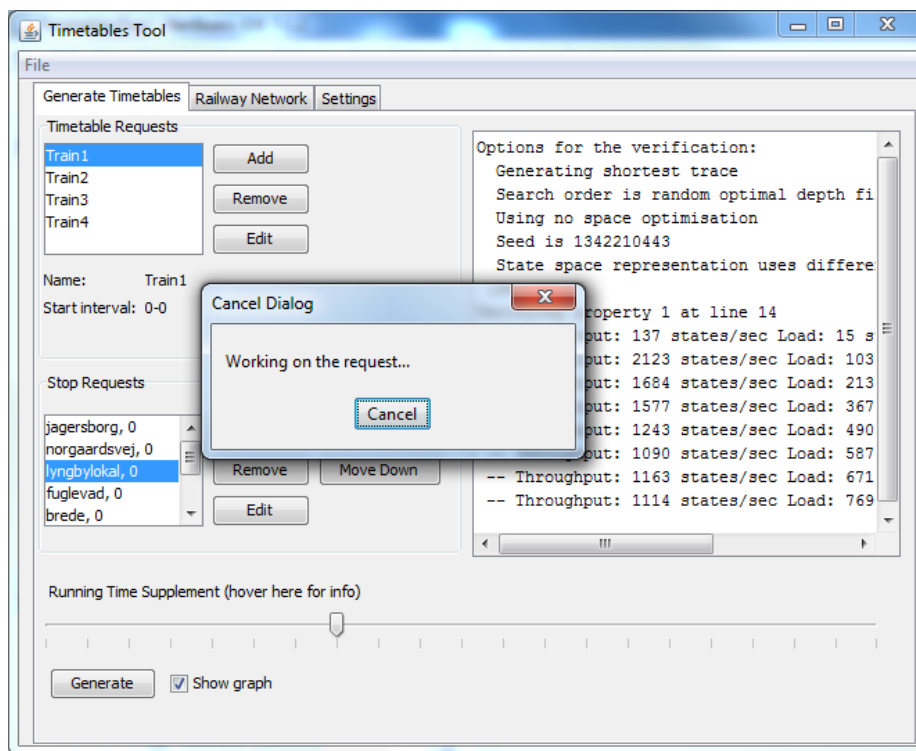


Figure 6.7: Four timetable requests have been created and the tool is currently generating the requested timetables.

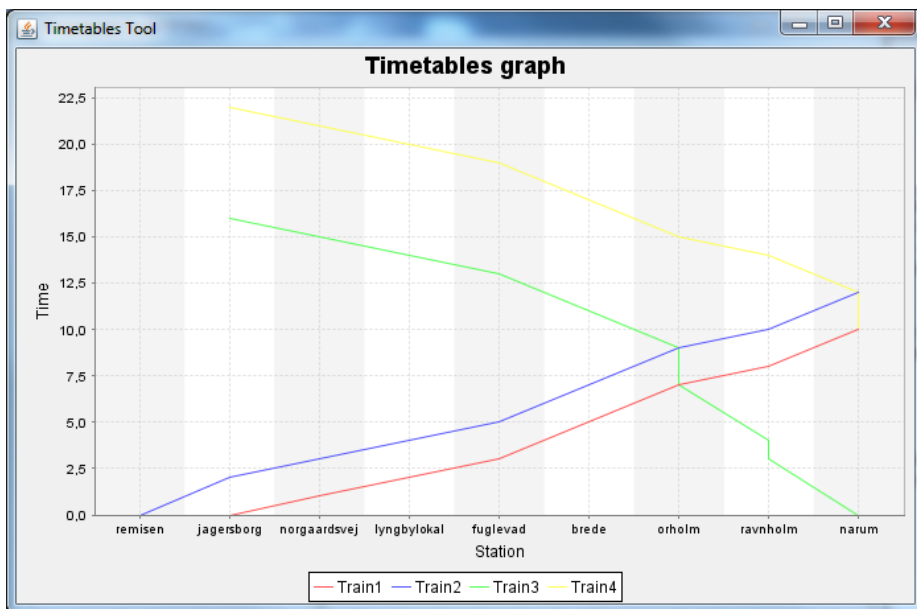


Figure 6.8: The finished graph of the generated timetables.

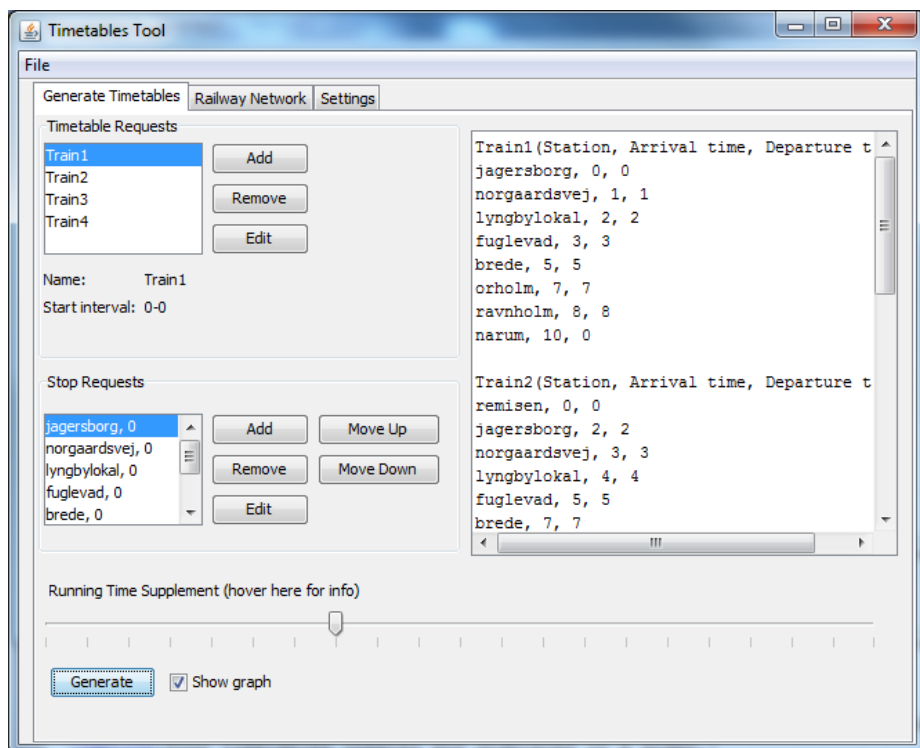


Figure 6.9: The text representation of the finished timetables.

Evaluation

This section will discuss relevant points when evaluating the use of formal methods in timetabling.

Section [7.1](#) will discuss the running times of the processes of verifying and generating timetables.

Section [7.2](#) will discuss how useful the resulting verifications and timetables are.

Section [7.3](#) will discuss the process of creating a tool, to utilize formal models when generating timetables.

7.1 Running Time

In order to evaluate whether or not the use of formal methods is practical in timetabling, it is important to consider the running time of validating and creating collections of timetables.

The following two sections will present the practical running time when verifying and generating timetables, using the models of [chapter 4](#) and [5](#) respectively.

7.1.1 Running Time When Verifying Timetables

The amount of time it takes to verify timetables is highly dependent on how many timetables are active at the same time. The timetables of Lokalbansen have at most four different timetables active at the same point in time. Increasing the amount of active trains will increase the running time.

The tests performed in appendix C.1 show that using the model-checker of UPPAAL to verify existing timetables, can be done for the actual working timetables of Lokalbansen [Lok] - and that it can be done within an acceptable time limit. When looking at the graph of figure C.4, the time spent on verifying timetables for Lokalbansen can be expressed in a polynomial curve.

The graph in figure C.5 shows the continued tendency curve of the running times when verifying timetables. Based on this graph, it would take approximately 12 minutes to verify all of the 72 working timetables in [Lok], using the model of chapter 4 with the model-checker of UPPAAL.

Using 12 minutes to validate the entire set of timetables representing the railway operations of Lokalbansen in 2003, is not a long time. Therefore using the model-checker of UPPAAL CORA to validate timetables, is highly applicable, when considering the running times.

7.1.2 Running Time When Generating Timetables

The tests performed in appendix C.2 have shown that the final tool for creating timetables, is severely hindered by the running times. As when verifying timetables, the amount of active timetables at the same time is crucial to the running time.

Figure C.6 shows a graph of the generated timetables for 12 different trains, which was created almost instantly. In this collection of timetables however, there is only a single timetable active at a time.

When having several timetables active at the same time, the running time is greatly increased. Appendix C.2 shows, if two timetable requests have opposite journeys in Lokalbansen, and have identical start intervals, the tool will take longer than 10 minutes to generate the two timetables¹. However, if the start intervals of the timetable requests are displaced, the tool is able to generate up

¹The test request was cancelled after 10 minutes.

to four interleaving timetables (as seen in figure C.12) within a little under two minutes.

Based on these results, using the model-checker of UPPAAL CORA to generate timetables, is not advisable. The running times are too long, even when there are only two interleaving timetables.

7.2 The Verifications and the Generated Timetables

In order to evaluate whether or not the use of formal methods is practical in timetabling, it is important to consider the quality of the results, when validating and creating collections of timetables.

The following two sections will discuss the quality of the resulting verifications and generated timetables, using the models of chapter 4 and 5 respectively.

7.2.1 The Resulting Verifications

When using the model-checker of UPPAAL to verify timetables (as explained in section 4.3, the results will first and foremost display whether or not the collection of timetables can be considered valid.

When using this approach however, it is also possible to identify exactly where, when and how something went wrong, allowing the user to quickly identify the problem. This feature of identifying problems, is very usable, and is considered to be an important point when verifying timetables, using the model-checker of UPPAAL. This is also an important reason, for choosing the model-checker of UPPAAL to verify timetables, rather than the test cases of RSL, which are not able to be as specific as UPPAAL in showing errors.

7.2.2 The Resulting Generated Timetables

When using the model-checker of UPPAAL CORA to generate timetables (as explained in section 5.4), the resulting timetables are guaranteed to satisfy the desired properties in collections of timetables.

The existing tools of TPS[TPS11] and RailSys[Lan10], do not create timetables in the same manner. When timetables are created in these tools, they are manually created, in the sense that timetables are created by the user, and then the tools will point out errors, which the user must then manually fix. The created tool in this thesis automatically creates timetables, which are guaranteed to be valid (based on the provided properties).

The resulting timetables, when using the model-checker of UPPAAL CORA to generate timetables, can therefore be said to be easier applied to railway operations, as there will be little need of adjusting the timetables after they have been created (assuming the model of UPPAAL CORA has the desired properties incorporated). It should be noted that the existing tools of RailSys and TPS do provide a far greater level of detail in planning, such as rolling stock restrictions (speed, capacity, cost etc.).

7.3 Creating a Tool Which Utilizes Formal Methods

The last step in using formal models for verifying and generating timetables, is to create a tool, which is usable by people with no knowledge of the formal models.

It should be noted that the tool for this thesis cannot verify existing timetables, it can only generate timetables. A tool for verifying timetables could be created in much the same manner as the tool created in this thesis for creating timetables.

When creating a tool to generate timetables based on a formal model, the type of formal model, is highly influential. In this case the formal model is created in UPPAAL CORA, and the tool should therefore utilize the model-checker of UPPAAL CORA, which introduces the need for a connection between the tool and the model-checker of UPPAAL CORA.

The final tool of this thesis is an example of how a custom tool could be created, such that it generates timetables, using formal methods. Once the formal model of chapter 5 was created, the process of creating this tool has mostly been limited to two following items:

- Creating a GUI and a data model.
- Having the tool utilize the model-checker of UPPAAL CORA.

Due to the fact that neither of these subjects are complex, and the need for programmatically generating timetables has been removed - the process of creating a custom tool, utilizing formal methods in timetabling is concluded to be a viable approach.

Conclusion

The main goal of this thesis has been to investigate how formal methods can be used to verify and generate timetables. This investigation has been based on four steps: (1) A formal model of the timetabling domain in RSL was created. (2) A formal model in UPPAAL, to be used for verifying timetables was created. (3) A formal model in UPPAAL CORA, to be used for generating timetables was created (4) A tool which is able to generate timetables, by utilizing the model created in UPPAAL CORA was created.

The created formal model in RSL, provides the necessary technical definitions of the terms used in timetabling. Many of the terms used in timetabling have varying definitions, depending on the country. Therefore this type of model is a practical way of formalizing and specifying terms, in a manner and language which can be read by anyone with knowledge of formal models.

The formal model created in UPPAAL, can be used to verify timetables provided by the user, in a railway network provided by the user. When using the model-checker of UPPAAL with the created model, it has been argued in section 7.1.1 that it is possible to validate the 72 timetables of Lokalbaneln (provided in [Lok]), in approximately 12 minutes¹. Furthermore, in case of an invalid collection of timetables, it is possible to identify where, when and how the error occurs.

¹It has been tested up to 12 timetables, and based on those results, 72 timetables would take 12 minutes.

The formal model created in UPPAAL CORA, can be used to generate timetables in a railway network provided by the user. A tool has been created, which allows a user to generate timetables. The tool generates timetables by utilizing the created model in UPPAAL CORA along with the model-checker of UPPAAL CORA, effectively providing an example of a tool using formal methods in generating timetables.

The timetables created by the tool, are guaranteed to be valid according to the properties stated in the formal model of UPPAAL CORA. The existing tools of TPS[TPS11] and RailSys[Lan10], do not provide a similar guarantee. The timetables generated by the created tool can therefore be said to be more complete than the timetables created by the existing tools, as no manual adjustment is needed once they are generated. The time it takes for the tool to generate the timetables however, is far too long. It cannot create timetables for more than four interleaving trains in under 10 minutes², and if the requested timetables are too intertwined, it cannot even do two timetables in 10 minutes³. In an attempt to use the tool to reproduce 4 timetables of Lokalbansen, the tool was stopped after five hours, which suggests that in order for the tool to generate the 72 timetables of lokalbansen, the formal model of UPPAAL CORA should be improved significantly or another type of model should be investigated.

Based on the experiences gathered in in this thesis, it must be concluded that although using formal methods for generating timetables, can produce high quality timetables, it simply takes too long to be a viable approach.

Using formal methods for verifying timetables seems to show promise. The verification results have been shown to be generated within a viable time limit, and they are of a quality, which enables the user to pinpoint the source of a potential error. The conclusion for using formal methods for verifying timetables must therefore be it is a viable approach, and should be investigated further.

8.1 Further Work

This section will present possibilities for advancing the work done by this thesis.

The most pressing matter of the work done in this thesis, is the fact that using the formal model of UPPAAL CORA to generate timetables is slow. A way of increasing the speed of generating timetables, is to improve the model of UPPAAL CORA, in such a manner that the model-checker can work faster. A

²The test was cancelled after 10 minutes.

³Again the test was cancelled after 10 minutes.

number of optimizations have already been performed (as explained in sections 5.3.4 and 4.2.6), but there is still an unexplored point of optimization - the *remaining* variable of UPPAAL CORA. The most promising way of optimizing the model of UPPAAL CORA, is most likely to investigate a method of evaluating the remaining value. An explanation of the remaining variable is given in section 5.1, or it can be found on the website of UPPAAL CORA⁴.

Another relevant work item to improve the thesis, would be to increase the detail level of the formal models. Including more aspects of timetabling, will provide a more complete foundation, both for verifying and generating timetables. An improvement could be made to the formal model of a railway network, where stations are probably one of the most simplified elements of the formal model. The stations could have platform tracks included, along with which platform tracks are connected to which open lines. Rolling stock are also one of the more simplified elements. The rolling stock could be included more in the model, by adding information such as maximum speed, acceleration and cost.

Finally more work can be put into the created tool, in order to make it a more complete tool for generating timetables. The main point of improvement to the tool at the moment would be to add user input validation. Currently, there is nothing to validate the railway network or timetable requests, which is provided by the user. Inspiration for this, could be drawn from the formal model in RSL (chapter 3), which already contains some conditions for existing railway networks.

⁴<http://people.cs.aau.dk/~adavid/cora/language.html>

APPENDIX A

RSL files

A.1 RailwayNetwork.rsl

```
scheme RailwayNetwork =  
  class  
    type  
      Time = Nat,  
      Name = Text,  
      Capacity = Nat,  
      DoubleTrack = Bool,  
      MinimumRunningTime = Time,  
      HeadwayTime = Time,  
      Station = Name,  
      OpenLine = Station × Station,  
      StationTable = Station  $\overrightarrow{m}$  (Capacity × HeadwayTime),  
      OpenLineTable =  
        OpenLine  $\overrightarrow{m}$   
        (DoubleTrack × MinimumRunningTime × Capacity ×  
         HeadwayTime),  
      RailwayNetwork = StationTable × OpenLineTable  
  
  value
```

```

get_OpenLine :
  Station × Station × RailwayNetwork → OpenLine
get_OpenLine(
  station1, station2, (stationTable, openLineTable)) ≡
  if ((station1, station2) ∈ dom (openLineTable))
  then (station1, station2)
  else (station2, station1)
  end
pre
  (station1, station2) ∈ dom (openLineTable) ∨
  (station2, station1) ∈ dom (openLineTable),

get_OpenLine_DoubleTrack :
  OpenLine × RailwayNetwork → DoubleTrack
get_OpenLine_DoubleTrack(
  (station1, station2), (stationTable, openLineTable)) ≡
  let
    (doubleTrack, mrt, cap, hwt) =
      openLineTable(
        get_OpenLine(
          station1, station2,
          (stationTable, openLineTable)))
  in
    doubleTrack
  end,

get_OpenLine_MinimumRunningTime :
  OpenLine × RailwayNetwork → MinimumRunningTime
get_OpenLine_MinimumRunningTime(
  (station1, station2), (stationTable, openLineTable)) ≡
  let
    (doubleTrack, mrt, cap, hwt) =
      openLineTable(
        get_OpenLine(
          station1, station2,
          (stationTable, openLineTable)))
  in
    mrt
  end,

get_OpenLine_Capacity :
  OpenLine × RailwayNetwork → Capacity
get_OpenLine_Capacity(
  (station1, station2), (stationTable, openLineTable)) ≡

```

```

let
  (doubleTrack, mrt, cap, hwt) =
    openLineTable(
      get_OpenLine(
        station1, station2,
        (stationTable, openLineTable)))
in
  cap
end,

get_OpenLine_HeadwayTime :
  OpenLine × RailwayNetwork → HeadwayTime
get_OpenLine_HeadwayTime(
  (station1, station2), (stationTable, openLineTable)) ≡
let
  (doubleTrack, mrt, cap, hwt) =
    openLineTable(
      get_OpenLine(
        station1, station2,
        (stationTable, openLineTable)))
in
  hwt
end,

get_Station_Capacity :
  Station × RailwayNetwork → Capacity
get_Station_Capacity(
  station, (stationTable, openLineTable)) ≡
  let (capacity, hwt) = stationTable(station) in
  capacity
end
pre station ∈ dom (stationTable),

get_Station_HeadwayTime :
  Station × RailwayNetwork → HeadwayTime
get_Station_HeadwayTime(
  station, (stationTable, openLineTable)) ≡
  let (platforms, hwt) = stationTable(station) in
  hwt
end
pre station ∈ dom (stationTable),

connect_one_station :
  Station-set × Station-set × RailwayNetwork →

```

```

    Station-set
connect_one_station(
  unconnected, connected,
  (stationTable, openLineTable)) ≡
if (unconnected = {}) then {}
else
  let unconnected_station = hd (unconnected) in
    if
      (∃ connected_station : Station •
        connected_station ∈ connected ∧
        ((unconnected_station, connected_station) ∈
          dom (openLineTable) ∨
          (connected_station, unconnected_station) ∈
          dom (openLineTable)))
    then {unconnected_station}
    else
      connect_one_station(
        unconnected \ {unconnected_station},
        connected, (stationTable, openLineTable))
    end
  end
end,

are_all_stations_connected :
  Station-set × Station-set × RailwayNetwork → Bool
are_all_stations_connected(
  unconnected, connected,
  (stationTable, openLineTable)) ≡
if (connected = dom (stationTable)) then true
else
  let
    connected_station =
      connect_one_station(
        unconnected, connected,
        (stationTable, openLineTable))
  in
    if (connected_station = {}) then false
    else
      are_all_stations_connected(
        unconnected \ connected_station,
        connected ∪ connected_station,
        (stationTable, openLineTable))
    end
  end
end

```

```

    end,

    pred _All_stations_are_defined : RailwayNetwork → Bool
    pred _All_stations_are_defined(
      (stationTable, openLineTable) ≡
      (∀ (station1, station2) : OpenLine •
        (station1, station2) ∈ dom (openLineTable) ⇒
          station1 ∈ dom (stationTable) ∧
          station2 ∈ dom (stationTable)),

    pred _All_stations_are_connected :
      RailwayNetwork → Bool
    pred _All_stations_are_connected(
      (stationTable, openLineTable) ≡
      let initial_station = hd (dom (stationTable)) in
        are_all_stations_connected(
          dom (stationTable) \ {initial_station},
          {initial_station}, (stationTable, openLineTable)
        )
      )
    end
  end
end

```

A.2 Timetable.rsl

```

context: RailwayNetwork
scheme Timetable =
  extend RailwayNetwork with
  class
    type
      ArrivalTime = Time,
      DepartureTime = Time,
      DwellTime = Time,
      Route = Station*,
      Stop =
        Station × ArrivalTime × DepartureTime × DwellTime,
      Timetable = Name × Stop*,
      TimetableSet = Timetable-set

  value
    get_Route_from_Timetable : Timetable → Route

```

```

get _Route_from _Timetable(timetableName, stops) ≡
  if (stops = ⟨⟩) then ⟨⟩
  else
    let (originStation, oat, odt, odwt) = hd (stops) in
      ⟨originStation⟩ ^
      get _Route_from _Timetable(
        (timetableName, tl (stops)))
    end
  end,

get _movements_of _Timetable :
  Timetable → (Stop × Stop)-set
get _movements_of _Timetable((timetableName, stops)) ≡
  if (tl (stops) = ⟨⟩) then {}
  else
    {(hd (stops), hd (tl (stops)))} ∪
    get _movements_of _Timetable(
      (timetableName, tl (stops)))
  end,

is _Route_possible : Route × RailwayNetwork → Bool
is _Route_possible(route, (stationTable, openLineTable)) ≡
  if (tl (route) = ⟨⟩) then true
  else
    let station1 = hd (route) in
      let station2 = hd (tl (route)) in
        if
          ((station1, station2) ∈
            dom (openLineTable) ∨
            (station2, station1) ∈
            dom (openLineTable))
        then
          is _Route_possible(
            tl (route), (stationTable, openLineTable))
        else false
        end
      end
    end
  end,

are_travel_times_possible :
  Timetable × RailwayNetwork → Bool
are_travel_times_possible(
  (timetableName, stops), railwayNetwork) ≡

```



```

if (tl (stops) = ⟨⟩) then true
else
  let (originStation, oat, odt, odwt) = hd (stops) in
    let
      (destinationStation, dat, ddt, ddwt) =
        hd (tl (stops))
    in
      if
        (get_OpenLine_MinimumRunningTime(
          (originStation, destinationStation),
          railwayNetwork) ≤ dat - odt)
      then
        are_travel_times_possible(
          (timetableName, tl (stops)),
          railwayNetwork)
      else false
      end
    end
  end
end,

does_timetable_occupy_open_line_in_time_period_DIRECTED :
  Timetable × Station × Station × Time × Time →
  Bool
does_timetable_occupy_open_line_in_time_period_DIRECTED(
  (timetableName, stops), departureStation,
  destinationStation, from, to) ≡
if (tl (stops) = ⟨⟩) then false
else
  let (station1, at1, dt1, dwt1) = hd (stops) in
    let
      (station2, at2, dt2, dwt2) = hd (tl (stops))
    in
      if
        ((station1, station2) =
          (departureStation, destinationStation) ∧
          ((from < dt1 ∧ dt1 < to) ∨
           (from < at2 ∧ at2 < to) ∨
           (dt1 < from ∧ to < at2)))
        then true
        else
          does_timetable_occupy_open_line_in_time_period_DIRECTED(
            (timetableName, tl (stops)),
            departureStation, destinationStation,

```

```

        from, to)
    end
end
end
end,

is_open_line_occupied_in_time_period_DIRECTED :
  Station × Station × Time × Time × TimetableSet →
  Bool
is_open_line_occupied_in_time_period_DIRECTED(
  departureStation, destinationStation, from, to,
  timetableSet) ≡
  if (timetableSet = {}) then false
  else
    let headTT = hd (timetableSet) in
      does_timetable_occupy_open_line_in_time_period_DIRECTED(
        hd (timetableSet), departureStation,
        destinationStation, from, to) ∨
      is_open_line_occupied_in_time_period_DIRECTED(
        departureStation, destinationStation, from,
        to, timetableSet \ {headTT})
    end
  end,

add_timetable_to_trains_at_station_count :
  Station × Time × Timetable × Int → Int
add_timetable_to_trains_at_station_count(
  station, time, (timetableName, stops), count) ≡
  if (stops = ⟨⟩) then count
  else
    let (check_station, at, dt, dwt) = hd (stops) in
      if
        (station = check_station ∧ at < time ∧
         time < dt)
      then count + 1
      else
        add_timetable_to_trains_at_station_count(
          station, time, (timetableName, tl (stops)),
          count)
        end
      end
    end,

all_trains_at_station_count :
```

```

    Station × Time × TimetableSet × Int → Int
all_trains_at_station_count(
    station, time, timetableSet, count) ≡
if (timetableSet = {}) then count
else
    let timetable = hd (timetableSet) in
        let
            count =
                add_timetable_to_trains_at_station_count(
                    station, time, timetable, count)
        in
            all_trains_at_station_count(
                station, time, timetableSet \ {timetable},
                count)
        end
    end
end,

add_timetable_to_trains_at_open_line_count :
    OpenLine × Time × Timetable × Int → Int
add_timetable_to_trains_at_open_line_count(
    openLine, time, (timetableName, stops), count) ≡
if (tl (stops) = ⟨⟩) then count
else
    let (station1, at1, dt1, dwt1) = hd (stops) in
        let
            (station2, at2, dt2, dwt2) = hd (tl (stops))
        in
            if
                ((openLine = (station1, station2) ∨
                 openLine = (station2, station1)) ∧
                 dt1 ≤ time ∧ time < at2)
            then count + 1
            else
                add_timetable_to_trains_at_open_line_count(
                    openLine, time,
                    (timetableName, tl (stops)), count)
            end
        end
    end
end,

all_trains_at_open_line_count :
    OpenLine × Time × TimetableSet × Int → Int

```

```

all_trains_at_open_line_count(
  openLine, time, timetableSet, count) ≡
  if (timetableSet = {}) then count
  else
    let timetable = hd (timetableSet) in
      let
        count =
          add_timetable_to_trains_at_open_line_count(
            openLine, time, timetable, count)
      in
        all_trains_at_open_line_count(
          openLine, time, timetableSet \ {timetable},
          count)
      end
    end
  end,
end,

```

```

pred_all_routes_of_timetables_can_be_traversed :
  TimetableSet × RailwayNetwork → Bool
pred_all_routes_of_timetables_can_be_traversed(
  timetableSet, railwayNetwork) ≡
  (∀ (timetableName, stops) : Timetable •
    (timetableName, stops) ∈ timetableSet ⇒
    if (stops = ⟨⟩) then true
    else
      is_Route_possible(
        get_Route_from_Timetable(
          (timetableName, stops)), railwayNetwork)
    end),
end),

```

```

pred_minimum_running_times_upheld :
  TimetableSet × RailwayNetwork → Bool
pred_minimum_running_times_upheld(
  timetableSet, railwayNetwork) ≡
  (∀ (timetableName, stops) : Timetable •
    (timetableName, stops) ∈ timetableSet ⇒
    if (stops = ⟨⟩) then true
    else
      are_travel_times_possible(
        (timetableName, stops), railwayNetwork)
    end),
end),

```

```

pred_dwelling_times_upheld : TimetableSet → Bool
pred_dwelling_times_upheld(timetableSet) ≡

```

```

(∀ (timetableName, stops) : Timetable •
  (timetableName, stops) ∈ timetableSet ⇒
    (∀ (station, at, dt, dwt) : Stop •
      (station, at, dt, dwt) ∈ stops ⇒
        (dt - at) ≥ dwt)),

pred_stations_never_exceed_capacity :
  TimetableSet × RailwayNetwork → Bool
pred_stations_never_exceed_capacity(
  timetableSet, railwayNetwork) ≡
  (∀ (timetableName, stops) : Timetable •
    (timetableName, stops) ∈ timetableSet ⇒
      (∀ (station, at, dt, dwt) : Stop •
        (station, at, dt, dwt) ∈ stops ⇒
          let
            capacity =
              get_Station_Capacity(
                station, railwayNetwork)
          in
            all_trains_at_station_count(
              station, at, timetableSet, 0) <
              capacity
          end)),

pred_open_lines_never_exceed_capacity :
  TimetableSet × RailwayNetwork → Bool
pred_open_lines_never_exceed_capacity(
  timetableSet, railwayNetwork) ≡
  (∀ timetable : Timetable •
    timetable ∈ timetableSet ⇒
      (∀
        ((station1, at1, dt1, dwt1),
         (station2, at2, dt2, dwt2)) : (Stop × Stop)
        •
        ((station1, at1, dt1, dwt1),
         (station2, at2, dt2, dwt2)) ∈
          get_movements_of_Timetable(timetable) ⇒
          let
            capacity =
              get_OpenLine_Capacity(
                get_OpenLine(
                  station1, station2,
                  railwayNetwork), railwayNetwork)
          in

```

```

    all_trains_at_open_line_count(
      get_OpenLine(
        station1, station2, railwayNetwork),
      dt1, timetableSet, 0) ≤ capacity
  end)),

pred_stations_headway_times_upheld :
  TimetableSet × RailwayNetwork → Bool
pred_stations_headway_times_upheld(
  timetableSet, railwayNetwork) ≡
  (∀ (timetableName1, stops1) : Timetable •
    (timetableName1, stops1) ∈ timetableSet ⇒
      (∀ (timetableName2, stops2) : Timetable •
        (timetableName2, stops2) ∈ timetableSet ⇒
          (∀ (station1, at1, dt1, dwt1) : Stop •
            (station1, at1, dt1, dwt1) ∈ stops1 ⇒
              (∀
                (station2, at2, dt2, dwt2) : Stop
                •
                  (station2, at2, dt2, dwt2) ∈
                    stops2 ⇒
                      ((station1, at1, dt1, dwt1) ≠
                        (station2, at2, dt2, dwt2) ∧
                        station1 = station2) ⇒
                        let
                          headwayTime =
                            get_Station_HeadwayTime(
                              station1, railwayNetwork
                            )
                        in
                          (abs (at1 - at2)) ≥
                            headwayTime
                        end))))),

pred_open_lines_headway_times_upheld :
  TimetableSet × RailwayNetwork → Bool
pred_open_lines_headway_times_upheld(
  timetableSet, railwayNetwork) ≡
  (∀ timetable1 : Timetable •
    timetable1 ∈ timetableSet ⇒
      (∀ timetable2 : Timetable •
        timetable2 ∈ timetableSet ⇒
          (∀
            ((station1, at1, dt1, dwt1),

```

```

(station2, at2, dt2, dwt2)) :
  (Stop × Stop)
•
((station1, at1, dt1, dwt1),
 (station2, at2, dt2, dwt2)) ∈
get_movements_of_Timetable(timetable1) ⇒
(∀
  ((station3, at3, dt3, dwt3),
   (station4, at4, dt4, dwt4)) :
   (Stop × Stop)
  •
  ((station3, at3, dt3, dwt3),
   (station4, at4, dt4, dwt4)) ∈
  get_movements_of_Timetable(
    timetable2) ⇒
  (timetable1 ≠ timetable2 ∧
   (station1, station2) =
    (station3, station4)) ⇒
  let
    headwayTime =
      get_OpenLine_HeadwayTime(
        get_OpenLine(
          station1, station2,
          railwayNetwork),
        railwayNetwork)
  in
    (abs (dt1 - dt3)) ≥
      headwayTime ∧
    (abs (at2 - at4)) ≥
      headwayTime
  end))))),

pred_trains_do_not_attempt_to_overtake :
  TimetableSet → Bool
pred_trains_do_not_attempt_to_overtake(timetableSet) ≡
(∀ timetable1 : Timetable •
  timetable1 ∈ timetableSet ⇒
  (∀ timetable2 : Timetable •
    timetable2 ∈ timetableSet ⇒
    (∀
      ((station1, at1, dt1, dwt1),
       (station2, at2, dt2, dwt2)) :
       Stop × Stop
      •

```

```

      ((station1, at1, dt1, dwt1),
       (station2, at2, dt2, dwt2)) ∈
       get_movements_of_Timetable(timetable1) ⇒
       (∀
        ((station3, at3, dt3, dwt3),
         (station4, at4, dt4, dwt4)) :
         Stop × Stop
        •
        ((station3, at3, dt3, dwt3),
         (station4, at4, dt4, dwt4)) ∈
         get_movements_of_Timetable(
          timetable2) ⇒
         ((timetable1 ≠ timetable2 ∧
          (station1, station2) =
           (station3, station4)) ⇒
          ((dt1 < dt3 ∧ at2 < at4) ∨
           (dt3 < dt1 ∧ at4 < at2))))))
    ),

pred_no_single_track_open_lines_utilized_in_both_directions_simultaneously :
  TimetableSet × RailwayNetwork → Bool
pred_no_single_track_open_lines_utilized_in_both_directions_simultaneously(
  timetableSet, railwayNetwork) ≡
  (∀ timetable : Timetable •
   timetable ∈ timetableSet ⇒
    (∀
     ((station1, at1, dt1, dwt1),
      (station2, at2, dt2, dwt2)) : Stop × Stop
     •
     ((station1, at1, dt1, dwt1),
      (station2, at2, dt2, dwt2)) ∈
      get_movements_of_Timetable(timetable) ⇒
      (let
       doubleTrack =
         get_OpenLine_DoubleTrack(
           (station1, station2),
           railwayNetwork)
       in
       doubleTrack ∨
       ~ is_open_line_occupied_in_time_period_DIRECTED(
         station2, station1, dt1, at2,
         timetableSet)
      end))),

```



```

pred _timetable_names_are_unique : TimetableSet → Bool
pred _timetable_names_are_unique(timetableSet) ≡
  (∀ (timetableName1, stops1) : Timetable •
    (timetableName1, stops1) ∈ timetableSet ⇒
      (∀ (timetableName2, stops2) : Timetable •
        (timetableName2, stops2) ∈ timetableSet ⇒
          (timetableName1, stops1) ≠
            (timetableName2, stops2) ⇒
              timetableName1 ≠ timetableName2))
end

```

A.3 TestCases.rsl

Timetable

```

scheme TestCases =
  extend Timetable with
  class
    value
      test2_TimetableList : TimetableSet =
        {("a-b",
          ⟨("a", 0, 1, 0), ("b", 25, 32, 0),
            ("a", 57, 60, 0)⟩),
          ("b-a", ⟨("a", 2, 4, 0), ("b", 17, 65, 0)⟩)},
      test2_OpenLineTable : OpenLineTable =
        [("a", "b") ↦ (false, 12, 2, 1)],
      test2_StationTable : StationTable =
        ["a" ↦ (2, 2), "b" ↦ (2, 2)],
      test_TimetableList : TimetableSet =
        {("A-B",
          ⟨("A", 0, 1, 0), ("D", 4, 4, 0),
            ("B", 14, 17, 0), ("D", 21, 21, 0),
            ("A", 24, 24, 0)⟩),
          ("A-C",
          ⟨("C", 0, 1, 0), ("D", 4, 4, 0), ("A", 7, 10, 0),
            ("D", 13, 13, 0), ("C", 16, 16, 0)⟩)},
      test_OpenLineTable : OpenLineTable =
        [("A", "D") ↦ (true, 3, 1, 2),
          ("B", "D") ↦ (true, 4, 1, 2),
          ("C", "D") ↦ (false, 3, 1, 2)],

```

```

test_StationTable : StationTable =
  ["A" ↦ (2, 2), "B" ↦ (2, 2), "C" ↦ (2, 2),
   "D" ↦ (2, 2)],
test_RailwayNetwork : RailwayNetwork =
  (test_StationTable, test_OpenLineTable),
test_lokalbanen_TimetableList : TimetableSet =
  {"t1",
   ⟨("jagersborg", 30, 30, 0),
    ("norgaardsvej", 31, 31, 0),
    ("lyngbylokal", 33, 33, 0),
    ("fuglevad", 35, 35, 0), ("brede", 37, 37, 0),
    ("orholm", 39, 40, 0), ("ravnholm", 41, 41, 0),
    ("narum", 43, 43, 0))⟩,
   "t2",
   ⟨("remisen", 34, 34, 0),
    ("jagersborg", 36, 40, 0),
    ("norgaardsvej", 41, 41, 0),
    ("lyngbylokal", 43, 43, 0),
    ("fuglevad", 45, 45, 0), ("brede", 47, 47, 0),
    ("orholm", 49, 50, 0), ("ravnholm", 51, 51, 0),
    ("narum", 53, 53, 0))⟩,
   "t3",
   ⟨("narum", 47, 47, 0), ("ravnholm", 49, 49, 0),
    ("orholm", 50, 50, 0), ("brede", 52, 52, 0),
    ("fuglevad", 54, 55, 0),
    ("lyngbylokal", 56, 56, 0),
    ("norgaardsvej", 57, 57, 0),
    ("jagersborg", 59, 59, 0))⟩,
   "t4",
   ⟨("narum", 57, 57, 0), ("ravnholm", 59, 59, 0),
    ("orholm", 60, 60, 0), ("brede", 62, 62, 0),
    ("fuglevad", 64, 65, 0),
    ("lyngbylokal", 66, 66, 0),
    ("norgaardsvej", 67, 67, 0),
    ("jagersborg", 69, 69, 0))⟩},
test_lokalbanen_OpenLineTable : OpenLineTable =
  [("jagersborg", "norgaardsvej") ↦ (false, 1, 1, 0),
   ("norgaardsvej", "lyngbylokal") ↦ (false, 1, 1, 0),
   ("lyngbylokal", "fuglevad") ↦ (false, 1, 1, 0),
   ("fuglevad", "brede") ↦ (false, 2, 1, 0),
   ("brede", "orholm") ↦ (false, 2, 1, 0),
   ("orholm", "ravnholm") ↦ (false, 1, 1, 0),
   ("ravnholm", "narum") ↦ (false, 2, 1, 0),
   ("jagersborg", "remisen") ↦ (false, 2, 1, 0)],

```

```

test_lokalbanen_StationTable : StationTable =
  ["remisen" ↦ (6, 1), "jagersborg" ↦ (2, 1),
   "norgaardsvej" ↦ (1, 1), "lyngbylokal" ↦ (1, 1),
   "fuglevad" ↦ (2, 1), "brede" ↦ (1, 1),
   "orholm" ↦ (2, 1), "ravnholm" ↦ (1, 1),
   "narum" ↦ (2, 1)],
test_lokalbanen_RailwayNetwork : RailwayNetwork =
  (test_lokalbanen_StationTable,
   test_lokalbanen_OpenLineTable)

```

test_case

```

[pred_All_stations_are_defined]
  pred_All_stations_are_defined(test_RailwayNetwork),
[lokalbanen1]
  pred_All_stations_are_defined(
    test_lokalbanen_RailwayNetwork),
[pred_All_stations_are_connected]
  pred_All_stations_are_connected(test_RailwayNetwork),
[lokalbanen2]
  pred_All_stations_are_connected(
    test_lokalbanen_RailwayNetwork),
[pred_all_routes_of_timetables_can_be_traversed]
  pred_all_routes_of_timetables_can_be_traversed(
    test_TimetableList, test_RailwayNetwork),
[lokalbanen4]
  pred_all_routes_of_timetables_can_be_traversed(
    test_lokalbanen_TimetableList,
    test_lokalbanen_RailwayNetwork),
[pred_minimum_running_times_upheld]
  pred_minimum_running_times_upheld(
    test_TimetableList, test_RailwayNetwork),
[lokalbanen5]
  pred_minimum_running_times_upheld(
    test_lokalbanen_TimetableList,
    test_lokalbanen_RailwayNetwork),
[pred_dwell_times_upheld]
  pred_dwell_times_upheld(test_TimetableList),
[lokalbanen6]
  pred_dwell_times_upheld(
    test_lokalbanen_TimetableList),
[pred_stations_never_exceed_capacity]
  pred_stations_never_exceed_capacity(
    test_TimetableList, test_RailwayNetwork),
[lokalbanen7]

```

```

    pred_stations_never_exceed_capacity(
        test_lokalbanen_TimetableList,
        test_lokalbanen_RailwayNetwork),
[pred_open_lines_never_exceed_capacity]
    pred_open_lines_never_exceed_capacity(
        test_TimetableList, test_RailwayNetwork),
[lokalbanen8]
    pred_open_lines_never_exceed_capacity(
        test_lokalbanen_TimetableList,
        test_lokalbanen_RailwayNetwork),
[pred_stations_headway_times_upheld]
    pred_stations_headway_times_upheld(
        test_TimetableList, test_RailwayNetwork),
[lokalbanen9]
    pred_stations_headway_times_upheld(
        test_lokalbanen_TimetableList,
        test_lokalbanen_RailwayNetwork),
[pred_open_lines_headway_times_upheld]
    pred_open_lines_headway_times_upheld(
        test_TimetableList, test_RailwayNetwork),
[lokalbanen10]
    pred_open_lines_headway_times_upheld(
        test_lokalbanen_TimetableList,
        test_lokalbanen_RailwayNetwork),
[pred_trains_do_not_attempt_to_overtake]
    pred_trains_do_not_attempt_to_overtake(
        test_TimetableList),
[lokalbanen11]
    pred_trains_do_not_attempt_to_overtake(
        test_lokalbanen_TimetableList),
[pred_no_single_track_open_lines_utilized_in_both_directions_simultaneously]
    pred_no_single_track_open_lines_utilized_in_both_directions_simultaneously(
        test_TimetableList, test_RailwayNetwork),
[lokalbanen12]
    pred_no_single_track_open_lines_utilized_in_both_directions_simultaneously(
        test_lokalbanen_TimetableList,
        test_lokalbanen_RailwayNetwork),
[pred_timetable_names_are_unique]
    pred_timetable_names_are_unique(test_TimetableList),
[lokalbane13]
    pred_timetable_names_are_unique(
        test_lokalbanen_TimetableList),
[trains_at_station]
    all_trains_at_station_count(

```

```
    "a", 59, test_TimetableList, 0),
[all_trains_at_open_line]
  all_trains_at_open_line_count(
    ("a", "b"), 12, test_TimetableList, 0),
[get_Route_from_Timetable]
  get_Route_from_Timetable(
    hd (test_lokalbanen_TimetableList)),
[get_movements_of_Timetable]
  get_movements_of_Timetable(
    hd (test_lokalbanen_TimetableList)),
[is_Route_possible]
  is_Route_possible(
    ("jagersborg", "norgaardsvej", "lyngbylokal"),
    test_lokalbanen_RailwayNetwork),
[is_Route_possible_negative_test]
  is_Route_possible(
    ("jagersborg", "norgaardsvej", "fuglevad"),
    test_lokalbanen_RailwayNetwork) = false,
[is_open_line_occupied_in_time_period_DIRECTED]
  is_open_line_occupied_in_time_period_DIRECTED(
    "A", "D", 2, 5, test_TimetableList)
end
```


APPENDIX B

The UPPAAL CORA models used by the final tool

B.1 The full UPPAAL CORA model, used by the tool

The declarations stated here, and figure [B.1](#) shows the template of the model used in the final tool for creating the temporary model files. It should be noted that a lot of the edge labels and state invariants are marked as red. This happens because of syntax errors, caused by the string prepended and appended with `##`. The template is identical to that of figure [5.10](#).

```
//Global time
clock time;

meta int[-100000000000,100000000000] remaining;

//COSTS
const int WaitingAtStation = 1;
```

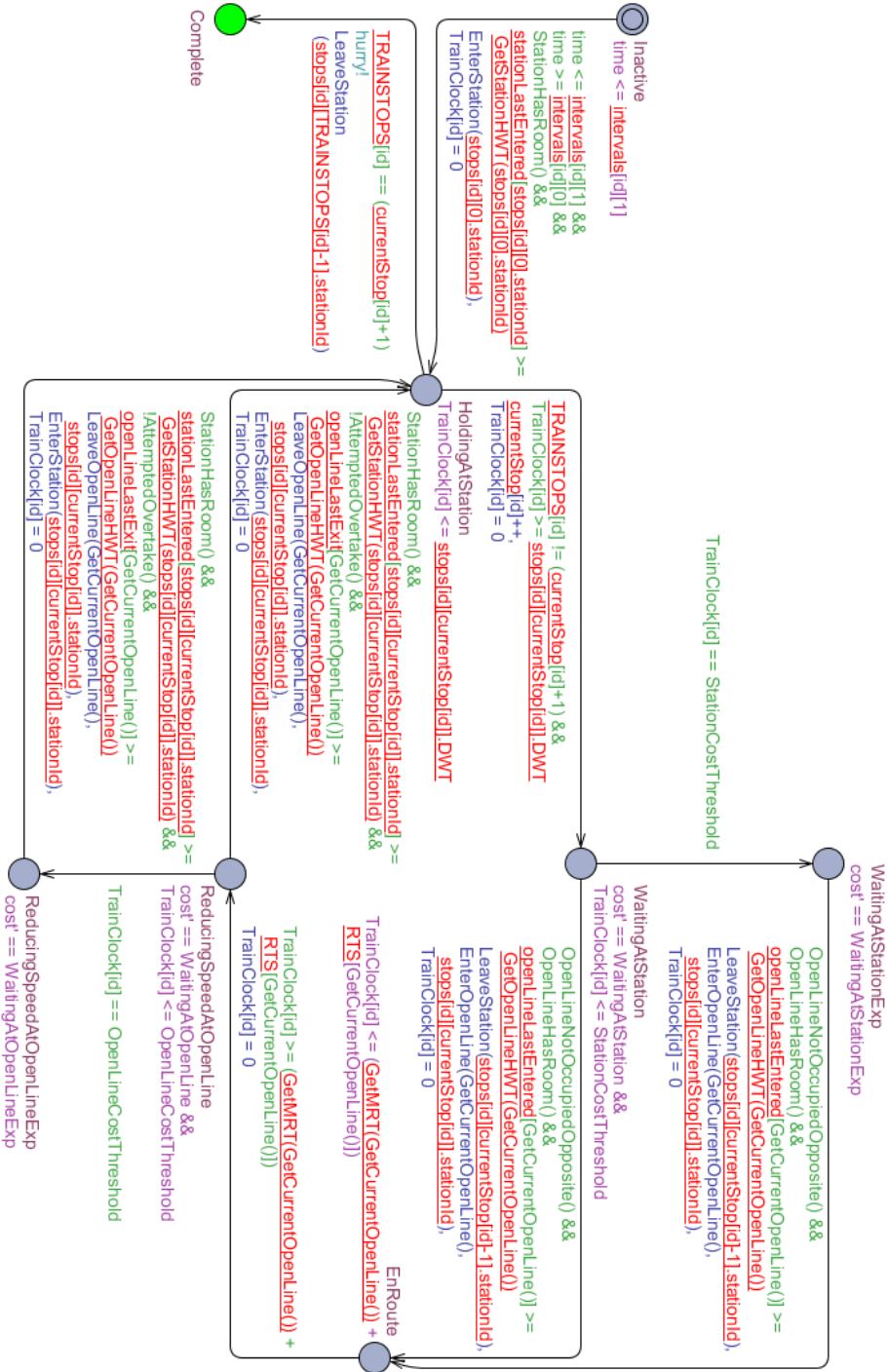


Figure B.1: The template used by the tool to create temporary model files. The red markings are caused by syntax errors in the global declarations.


```
const int WaitingAtOpenLine = 1;

const int WaitingAtStationExp = 3;
const int WaitingAtOpenLineExp = 3;

//Threshold for cost increase, expressed as time
const int StationCostThreshold = 3;
const int OpenLineCostThreshold = 3;

//Number of Stations
const int STATIONS = ##STATIONS##;

//Number of Open Lines
const int OPENLINES = ##OPENLINES##;

//Number of Trains
const int TRAINS = ##TRAINS##;
typedef int[0, TRAINS-1] t_id;

/*A queue for each direction of an open
line denoting the order in which trains enter.*/
int queue[OPENLINES][2][TRAINS];

//Trainclocks used by the trains to time various actions
clock TrainClock[TRAINS];

//Denotes the current stop of the trains
int[0, ##MOSTSTOPS##-1] currentStop[TRAINS];

//Channels
urgent chan hurry;

//Timetable request
//Start time intervals for each train
const int intervals[TRAINS][2] = ##STARTINTERVALS##;

//Amount of stops on the routes
const int TRAINSTOPS[TRAINS] = ##TRAINSTOPS##;

//stations
##STATIONCONSTS##

typedef struct {
int[-1,STATIONS-1] stationId;
```

```
int DWT; //Dwell Time
}Stops;

const Stops stops[TRAINS][##MOSTSTOPS##] =
##STOPS##;

typedef struct{
    int station1;
    int station2;
} OpenLine;

typedef struct{
    OpenLine openLine;
    bool doubleTrack;
    int MRT; //Minimum Running Time
    int capacity;
    int HWT; //Headway Time
}OpenLineTable;

//Open line table
const OpenLineTable openLineTable[OPENLINES] =
##OPENLINETABLE##;

//The running time supplement for each open line
const int RTS[OPENLINES] = ##RTS##;

//The amount of trains present in a direction on an open line
int[0, TRAINS] trainsAtOpenLine[OPENLINES][2];
//The time a train last entered an open line
clock openLineLastEntered[OPENLINES];
//The time a train last exited an open line
clock openLineLastExit[OPENLINES];

typedef struct{
    int[0,STATIONS-1] station;
    int capacity;
    int HWT; //Headway Time
}StationTable;

//Station table
const StationTable stationTable[STATIONS] =
##STATIONTABLE##;

//The amount of trains present at a station
```

```
int[0, TRAINS] trainsAtStation[STATIONS];
//The time a train last entered a station
clock stationLastEntered[STATIONS];

//Get minimum running time of open line
int GetMRT(int openLineId)
{
    return openLineTable[openLineId].MRT;
}

//Get capacity of open line
int GetOpenLineCapacity(int openLineId)
{
    return openLineTable[openLineId].capacity;
}

//Get headway time of open line
int GetOpenLineHWT(int openLineId)
{
    return openLineTable[openLineId].HWT;
}

//Get double track value of open line
bool IsOpenLineDoubleTrack(int openLineId)
{
    return openLineTable[openLineId].doubleTrack;
}

//Get headway time of a station
int GetStationHWT(int stationId)
{
    return stationTable[stationId].HWT;
}

//Get capacity of a station
int GetStationCapacity(int stationId)
{
    return stationTable[stationId].capacity;
}

//Get the id of an open line
int GetOpenLineId(int station1, int station2)
{
    for (i : int[0, OPENLINES-1])
```

```
{
    if((openLineTable[i].openLine.station1 == station1 &&
        openLineTable[i].openLine.station2 == station2) ||
        (openLineTable[i].openLine.station1 == station2 &&
        openLineTable[i].openLine.station2 == station1))
        return i;
    }
    return -1;
}

//Increase the amount of trains present in a direction on an open line
void IncreaseTrainsAtOpenLineDir(int openLineId, int dir)
{
    if(openLineTable[openLineId].openLine.station1 == dir)
        trainsAtOpenLine[openLineId][0]++;
    else
        trainsAtOpenLine[openLineId][1]++;
}

//Decrease the amount of trains present in a direction on an open line
void DecreaseTrainsAtOpenLineDir(int openLineId, int dir)
{
    if(openLineTable[openLineId].openLine.station1 == dir)
        trainsAtOpenLine[openLineId][0]--;
    else
        trainsAtOpenLine[openLineId][1]--;
}

//Get the total amount of trains present at an open line - regardless of direction
//Used when checking for the capacity of a single tracked open line, in which one of
int GetTrainsAtOpenLine(int openLineId)
{
    return trainsAtOpenLine[openLineId][0] + trainsAtOpenLine[openLineId][1];
}

//Get the amount of trains present in a direction on an open line
int GetTrainsAtOpenLineDir(int openLineId, int dir)
{
    if(openLineTable[openLineId].openLine.station1 == dir)
        return trainsAtOpenLine[openLineId][0];
    else
        return trainsAtOpenLine[openLineId][1];
}
```

```

//Determines whether or not an open line is occupied in the opposite direction of
bool IsOpenLineOccupiedOppositeDirection(int openLineId, int dir) {
    OpenLine openLine = openLineTable[openLineId].openLine;
    if(openLine.station1 == dir)
        return GetTrainsAtOpenLineDir(openLineId, openLine.station2) > 0;
    else
        return GetTrainsAtOpenLineDir(openLineId, openLine.station1) > 0;
}

//Get the train in front of the queue
int QueueGetFront(int openLineId, int dir)
{
    if(IsOpenLineDoubleTrack(openLineId))
        if(openLineTable[openLineId].openLine.station1 == dir)
            return queue[openLineId][0][0];
        else
            return queue[openLineId][1][0];
    else
        return queue[openLineId][0][0];
}

//Have a train enter the queue of a direction of an open line
void EnterQueue(int openLineId, int trainId, int dir)
{
    if(IsOpenLineDoubleTrack(openLineId))
        if(openLineTable[openLineId].openLine.station1 == dir)
            queue[openLineId][0][GetTrainsAtOpenLineDir(openLineId, dir)] = trainId;
        else
            queue[openLineId][1][GetTrainsAtOpenLineDir(openLineId, dir)] = trainId;
    else
        queue[openLineId][0][GetTrainsAtOpenLine(openLineId)] = trainId;
}

//Have a train exit the queue of a direction of an open line
void ExitQueue(int openLineId, int dir)
{
    int empty = -1;
    for (i : int[0, TRAINS-1])
    {
        if(IsOpenLineDoubleTrack(openLineId))
            if(openLineTable[openLineId].openLine.station1 == dir)
            {
                if(i != TRAINS-1)
                {

```

```

        queue[openLineId][0][i] = queue[openLineId][0][i+1];
        queue[openLineId][0][i+1] = empty;
    }
    else
        queue[openLineId][0][i] = empty;
}
else
{
    if(i != TRAINS-1)
    {
        queue[openLineId][1][i] = queue[openLineId][1][i+1];
        queue[openLineId][1][i+1] = empty;
    }
    else
        queue[openLineId][1][i] = empty;
}
else
{
    if(i != TRAINS-1)
    {
        queue[openLineId][0][i] = queue[openLineId][0][i+1];
        queue[openLineId][0][i+1] = empty;
    }
    else
        queue[openLineId][0][i] = empty;
}
}
}

//Initialize the queue
void initQueue()
{
    for (i : int[0, OPENLINES-1])
    {
        for (j : int[0, TRAINS-1])
        {
            int empty = -1;
            queue[i][0][j] = empty;
            queue[i][1][j] = empty;
        }
    }
}

//Initialize the station clocks and open line clocks used for headway times

```

```
void initEnterExitClocks()
{
    //Station headway clocks
    for (i :int[0, STATIONS-1])
    {
        stationLastEntered[i] = 1000000000;
    }
    for (i :int[0, OPENLINES-1])
    {
        openLineLastEntered[i] = 1000000000;
        openLineLastExit[i] = 1000000000;
    }
}
```

B.2 The UPPAAL CORA model, used by the tool, excluding station headway times

Figure B.2 shows the template of the model used in the final tool for creating the temporary model files without headway times for stations. The global declarations are identical to those of appendix B.1. It should be noted that a lot of the edge labels and state invariants are marked as red. This happens because of syntax errors, caused by the string prepended and appended with ##.

B.3 The UPPAAL CORA model, used by the tool, excluding open line headway times

Figure B.3 shows the template of the model used in the final tool for creating the temporary model files without headway times for open lines. The global declarations are identical to those of appendix B.1. It should be noted that a lot of the edge labels and state invariants are marked as red. This happens because of syntax errors, caused by the string prepended and appended with ##.

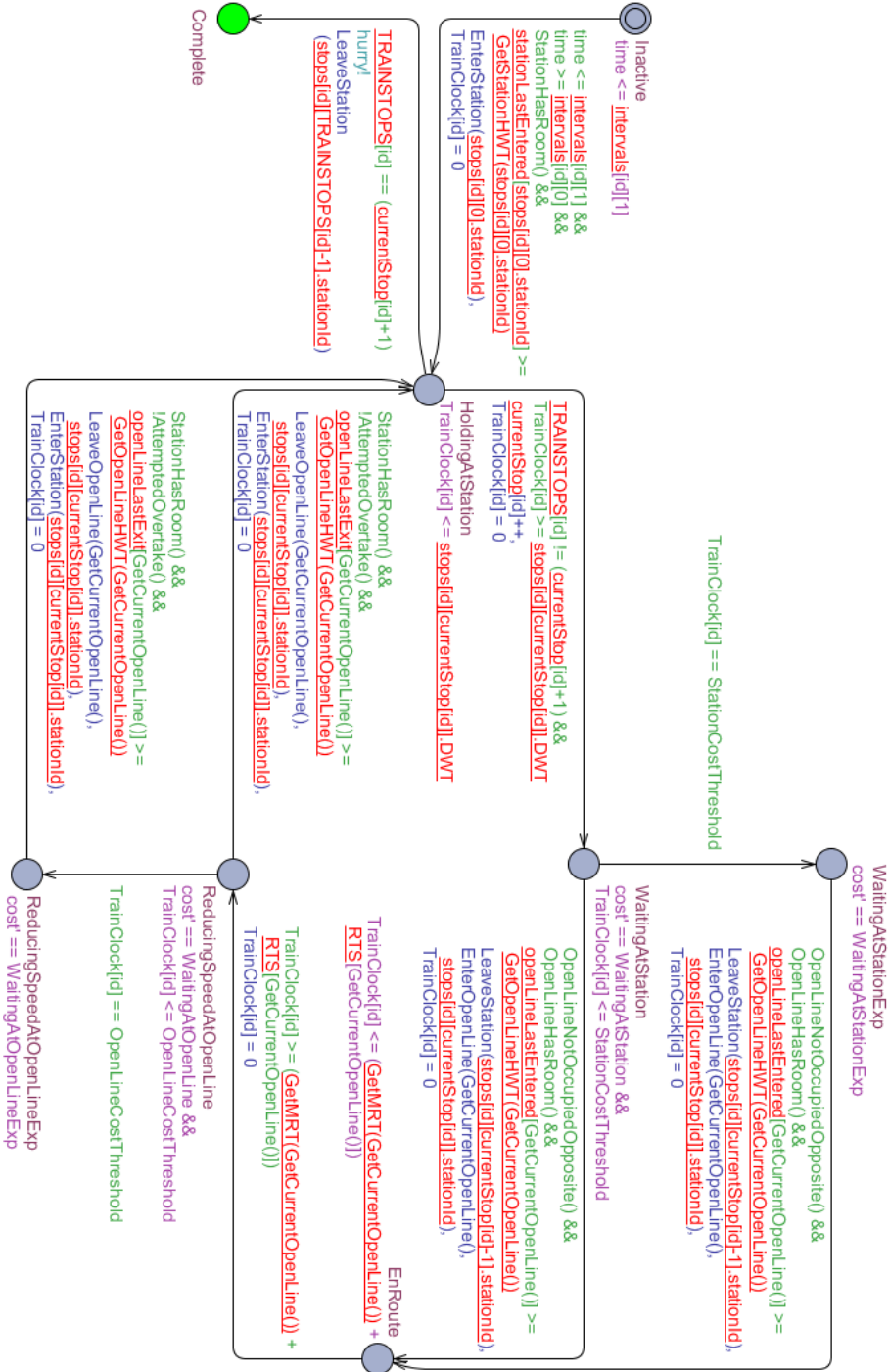


Figure B.2: The template used by the tool to create temporary model files, excluding station headway times. The red markings are caused by syntax errors in the global declarations.

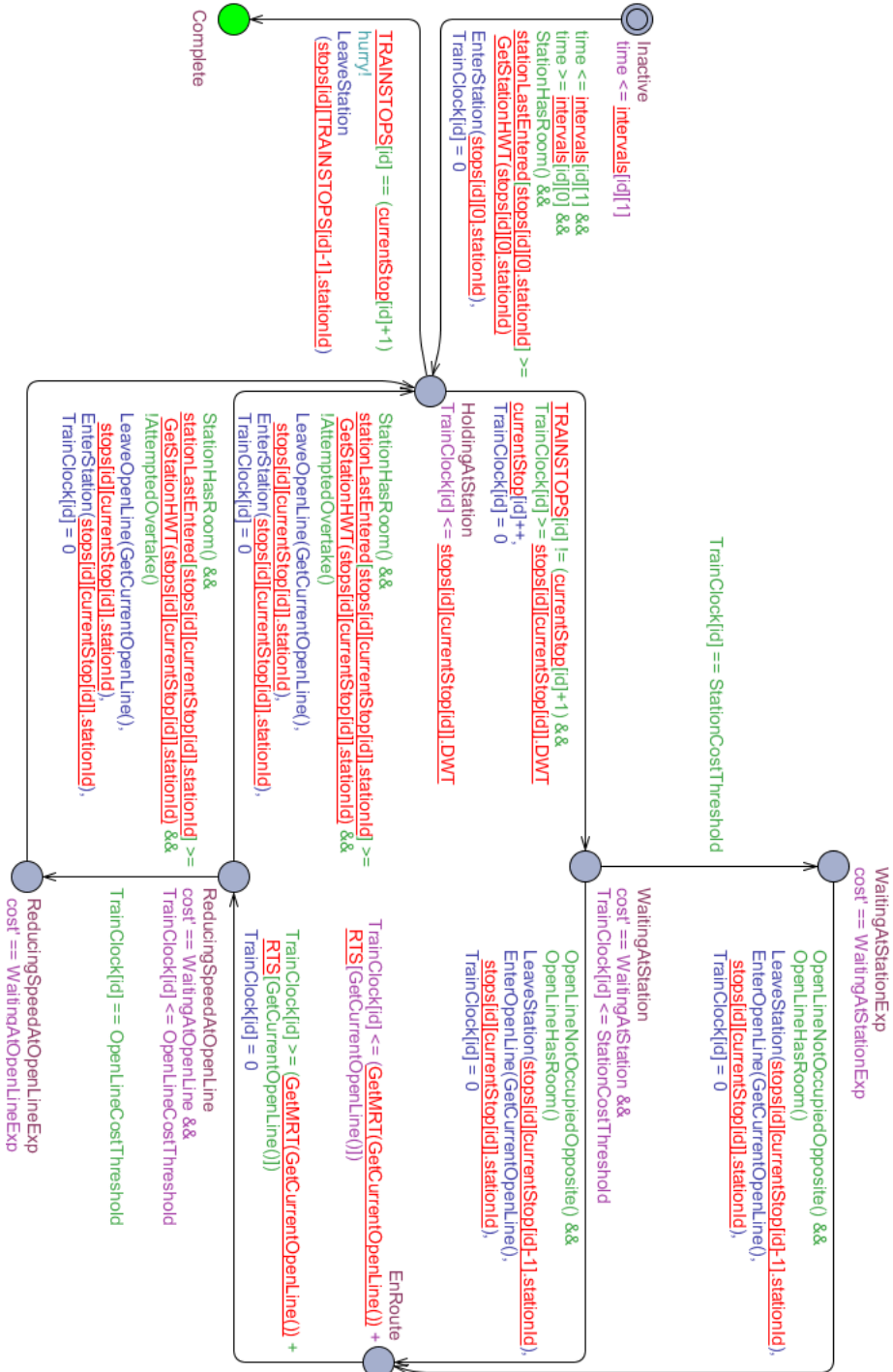


Figure B.3: The template used by the tool to create temporary model files, excluding open line headway times. The red markings are caused by syntax errors in the global declarations.

B.4 The UPPAAL CORA model, used by the tool, excluding both headway times

Figure [B.4](#) shows the template of the model used in the final tool for creating the temporary model files without headway times for either open lines or stations. The global declarations are identical to those of [appendix B.1](#). It should be noted that a lot of the edge labels and state invariants are marked as red. This happens because of syntax errors, caused by the string prepended and appended with `##`.

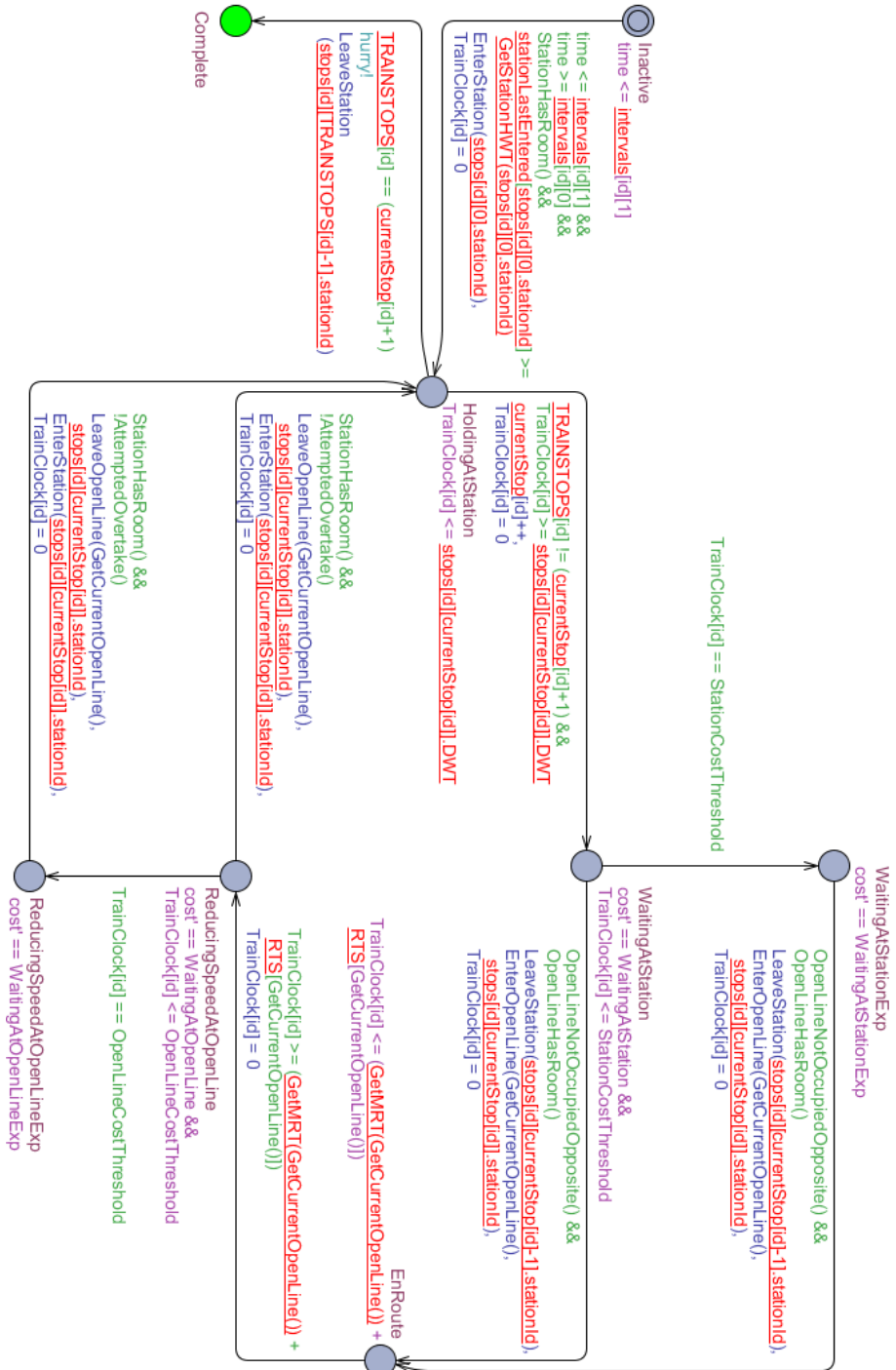


Figure B.4: The template used by the tool to create temporary model files, excluding all headway times. The red markings are caused by syntax errors in the global declarations.

Running Times

The tests were done on an Acer TravelMate 5530, with the following specifications:

- Processor: AMD Athlon(tm) X2 Dual-Core QL-64 2.10GHz
- Ram: 4,00GB (3,50 usable)
- Operating System: Windows 7 Professional (64-bit)

Time was measured manually by using a stopwatch.

C.1 Verifier

In order to test the running times of the model-checker of UPPAAL, using the model created in chapter 4, 12 timetables have been taken from the real working timetables of Lokalbane[Lok]¹, and put into the UPPAAL model.

¹The timetables 630-1, 640-1, 647-2, 650-1, 657-2, 700-1, 707-2, 710-1, 717-2 720-1, 727-2 and 737-2

The model-checker was then used to verify all of the required properties of the model, by adding one timetable at a time, measuring how long it took to verify, and how many states were searched during the verification.

Figure C.1 shows the running times of the UPPAAL model-checker when verifying between four and twelve timetables.

Figure C.2 shows the amount of states when verifying between one and twelve timetables.

Figure C.3 shows the amount of states searched per second (on average), when verifying between four and twelve timetables.

Figures C.4 and C.5 has added a polynomial tendency curve to the running time graph of figure C.1. Figure C.4 shows how well it fits at the measured values, and figure C.5 continues the tendency curve.

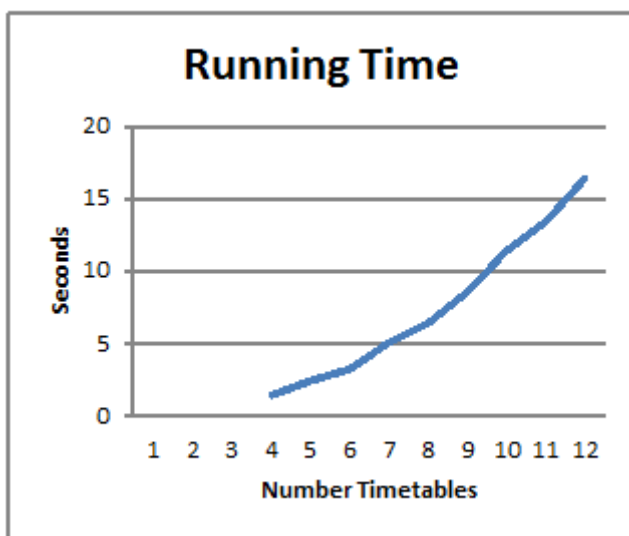


Figure C.1: The running times of the verifier, based on the amount of timetables. The running times under one second are omitted due to uncertainties.

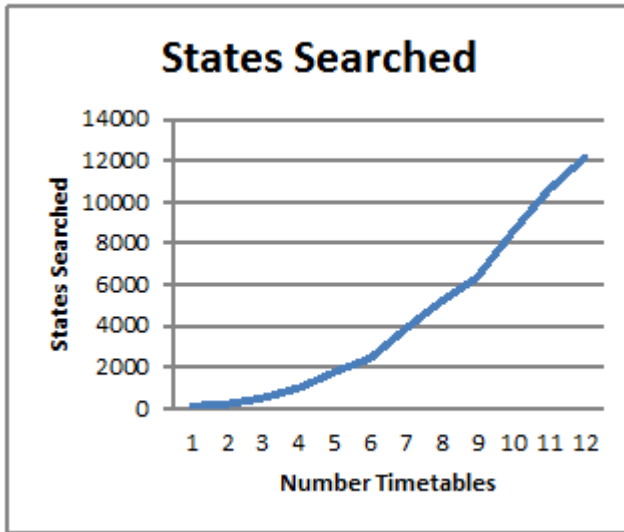


Figure C.2: The amount of states searched, based on the amount of timetables to verify.

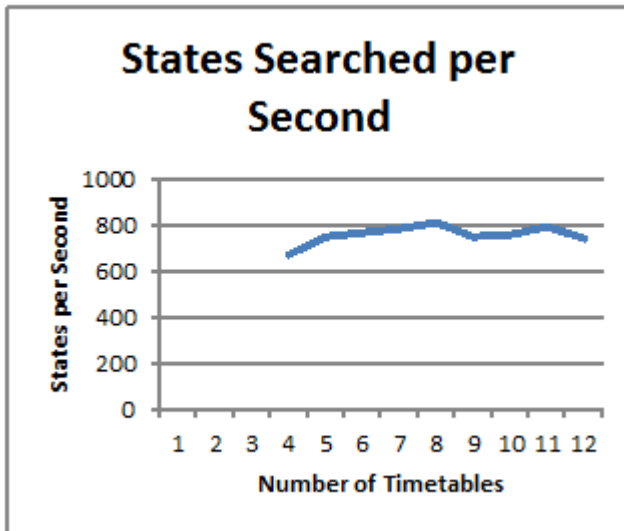


Figure C.3: The amount of states search per second, based on the amount of timetables to verify

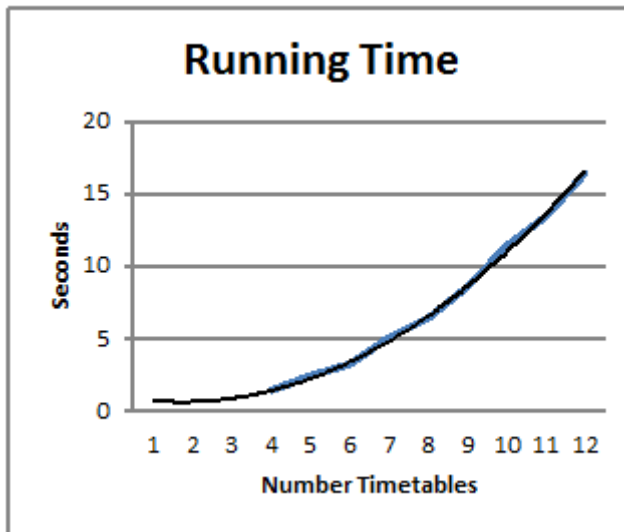


Figure C.4: A polynomial tendency curve of the running time graph (figure C.1), added to illustrate how well it fits the pattern.

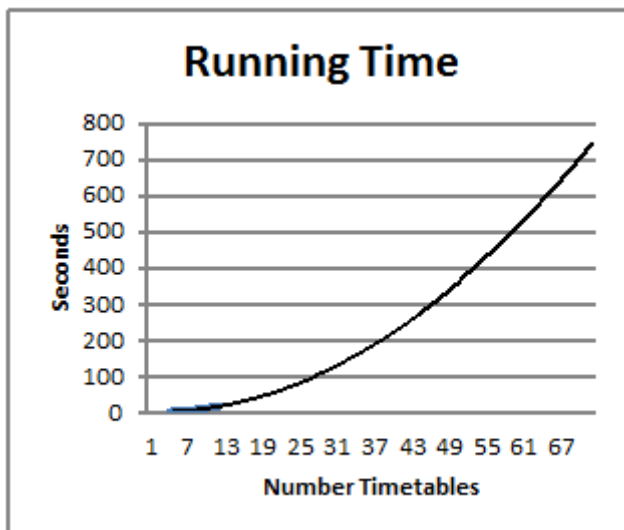


Figure C.5: The polynomial tendency curve of figure C.4 continued.

C.2 Generator

In order to test the running times of the tool, Lokalbansen is used as railway network, and the following different scenarios are measured:

1. Timetable requests with start intervals completely displacing them from each other.
2. Timetable requests running in the opposite direction, not displaced from each other.
3. Timetable requests running in opposite both directions, slightly displaced from each other.

(1) When the timetable requests run completely displayed of each other, the tool is able to create timetables for a large amount of trains in Lokalbansen. Figure C.6 shows the generated graph for 12 timetables, running in both directions. The tool created this plan instantly.

(2) When the timetable requests run opposite each other, and with start intervals, forcing them to interleave - the running time increases drastically. Figures C.7, C.8 and C.9, show the graphs of the generated timetables for two timetable requests going in opposite direction in the railway network.

- In figure C.7, the timetable requests were created with a fixed starting interval, separating the requests by 4 time units. This was created almost instantly.
- In figure C.8, the timetable requests were created with a fixed starting interval, separating the requests by 3 time units. This took approximately 4 seconds.
- In figure C.9, the timetable requests were created with a fixed starting interval, separating the requests by 2 time units. This took approximately 150 seconds.

The running time is increased drastically for each time unit the requests close in on each other. A test was also performed when they both had the same fixed start interval - this was cancelled after 10 minutes.

(3) When the timetable requests are just slightly displaced from each other, the tool is able to create a few more timetables than in the situation of (2). Figures

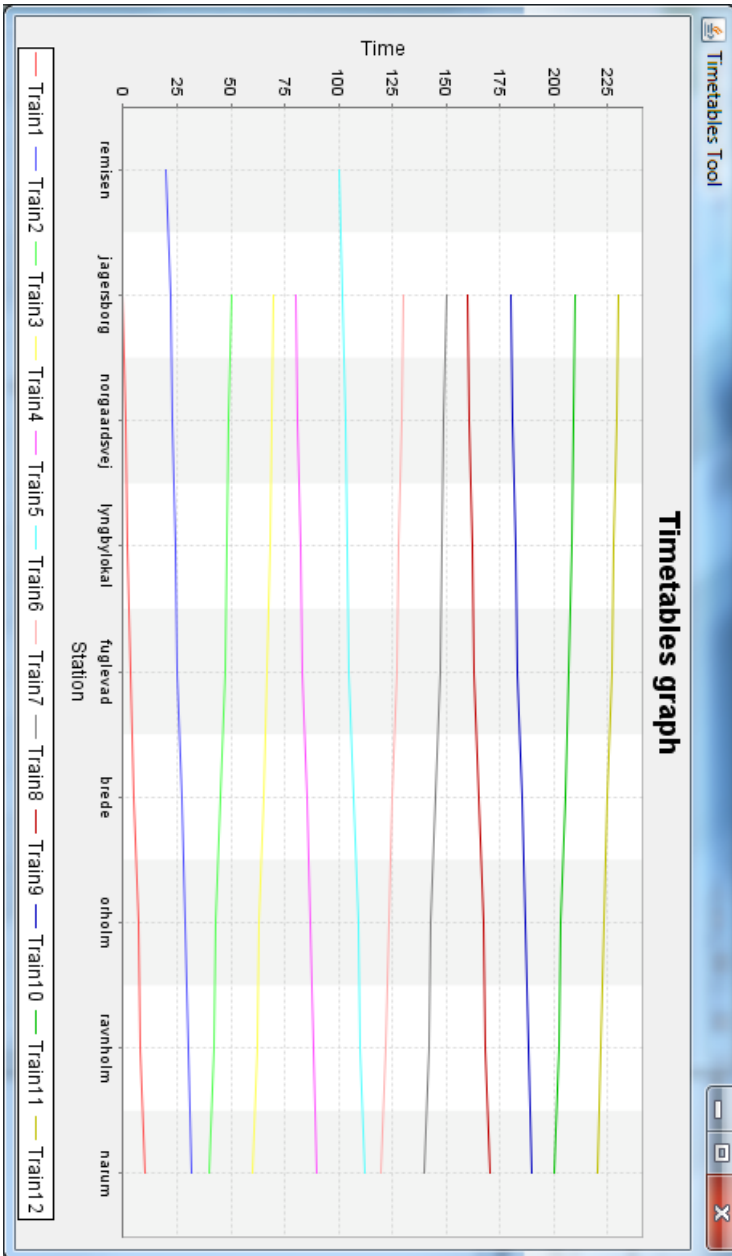


Figure C.6: The timetables for 12 trains, with start intervals completely displacing them of each other. The tool created this instantly.

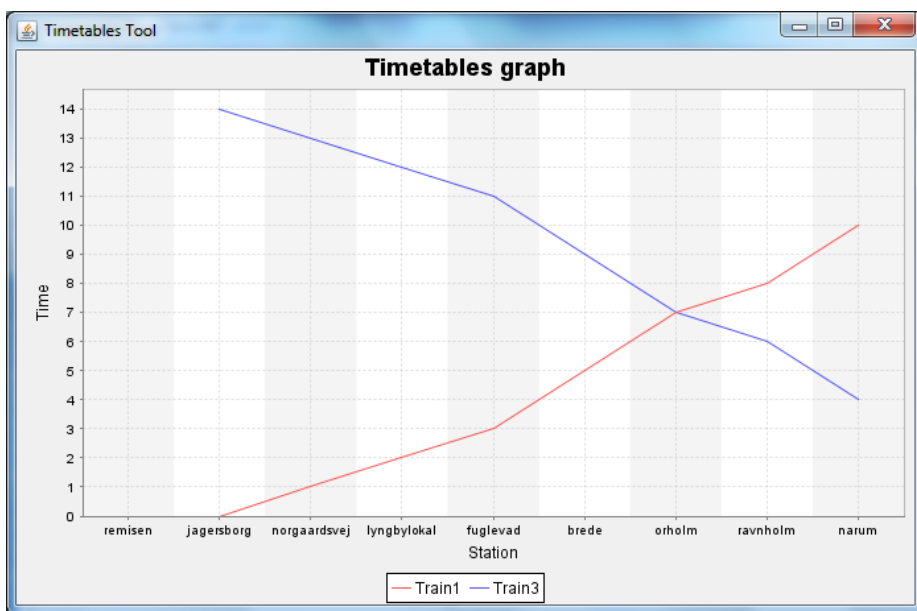


Figure C.7: Generated timetables for two trains running opposite each other. Train1 has a start interval of 0-0, and Train1 has a start interval of 4-4. This was generated instantly.

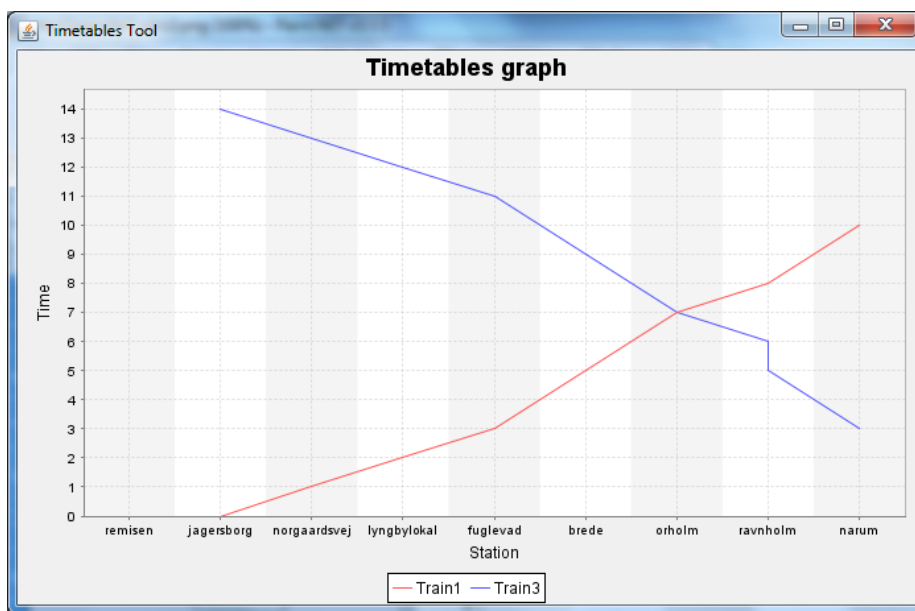


Figure C.8: Generated timetables for two trains running opposite each other. Train1 has a start interval of 0-0, and Train1 has a start interval of 3-3. This took approximately 4 seconds.

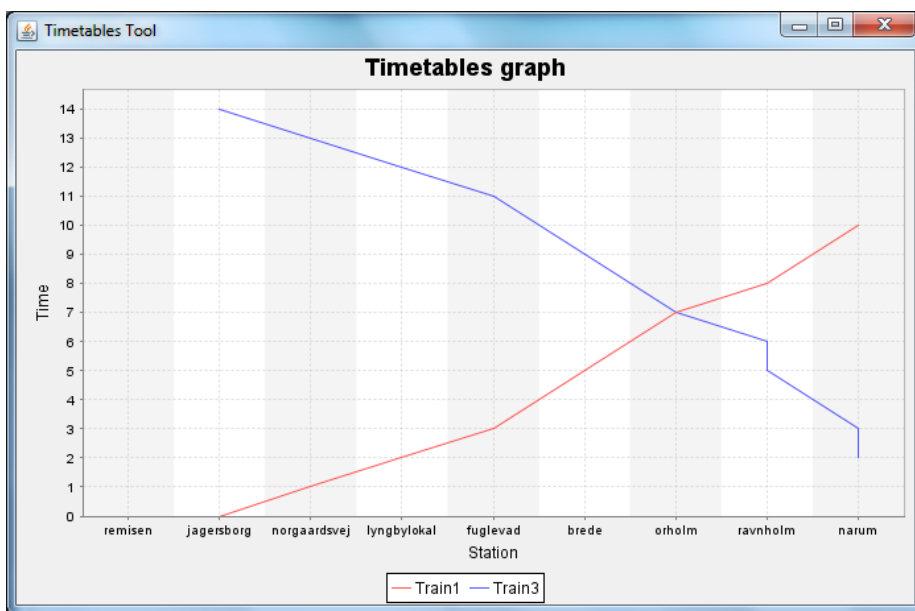


Figure C.9: Generated timetables for two trains running opposite each other. Train1 has a start interval of 0-0, and Train1 has a start interval of 2-2. This took approximately 150 seconds.

C.10, C.11 and C.12 show the graphs of the generated timetables for two, three and four timetables timetable requests interleaving in opposite directions, with a slight displacement - causing only two trains to be active in the railway network simultaneously. The start intervals of the timetable requests of figures C.10, C.11 and C.12 are fixed and displaced by 8 time units. The following list shows the time it took to generate these:

- In figure C.10 was generated almost instantly.
- In figure C.11 took approximately 80 seconds to generate.
- In figure C.12 took approximately 100 seconds to generate.

A test with five timetable requests was also done with this pattern - this was cancelled after 10 minutes. This also shows a sudden drastic increase in running time.

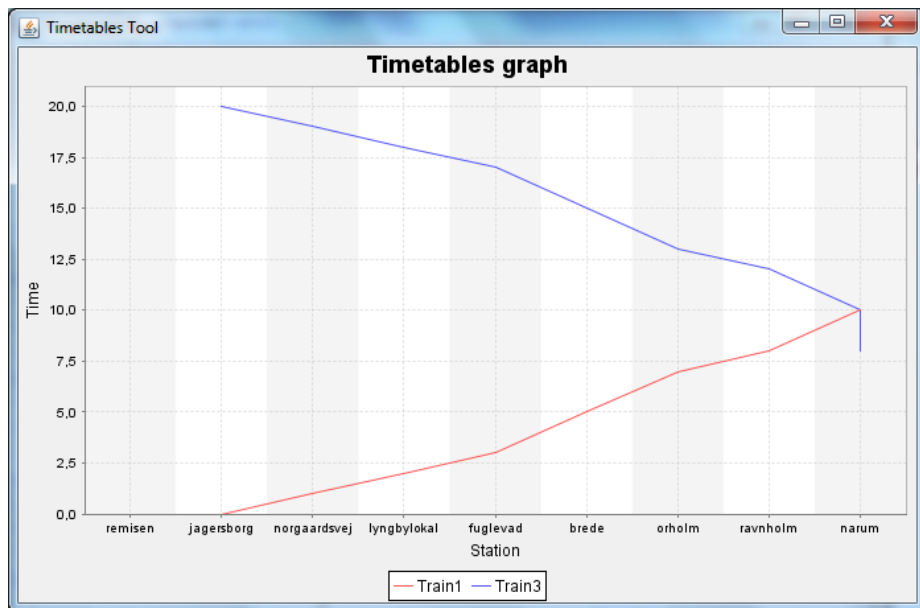


Figure C.10: Two trains interleaving, the start intervals are fixed and they are displaced by 8 time units. This was generated almost instantly.

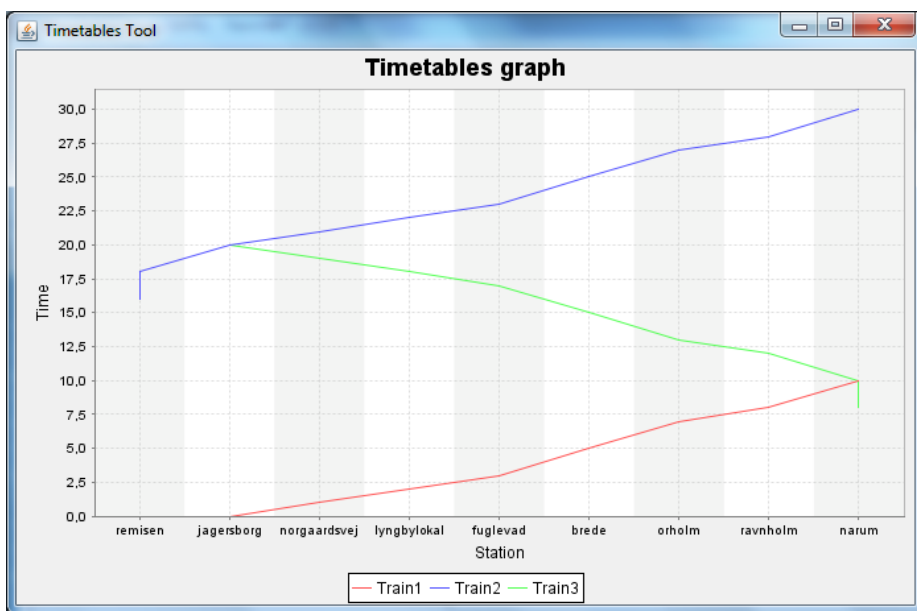


Figure C.11: Three trains interleaving, the start intervals are fixed and they are displaced by 8 time units. This took approximately 80 seconds to generate.

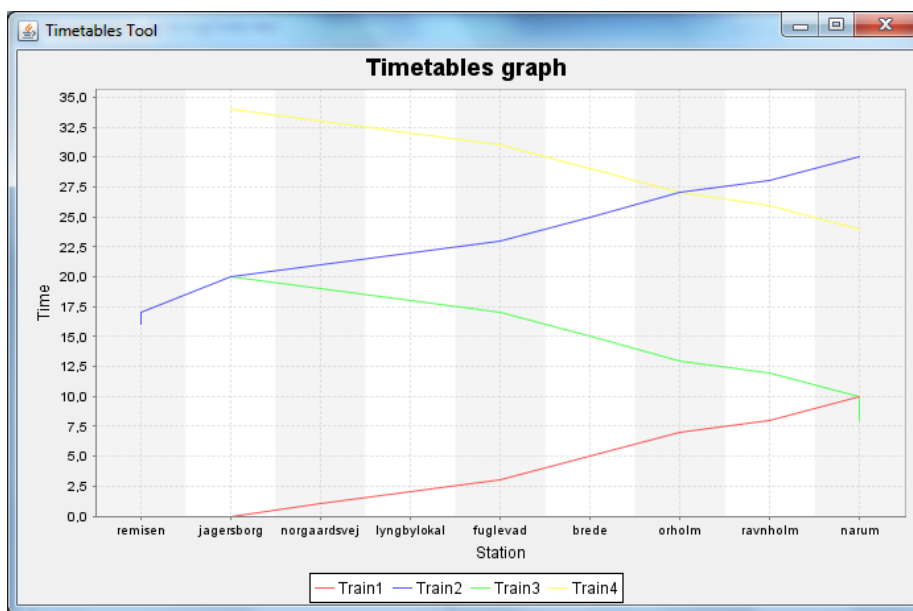


Figure C.12: Four trains interleaving, the start intervals are fixed and they are displaced by 8 time units. This took approximately 100 seconds to generate.

Tool User Guide

This chapter will provide screenshots, present the functionality of the tool and act as a guide showing how to use the tool. A file called 'Lokalbanen.ttt' is available on the CD, and can be loaded into the model, and act as an example. Figure [D.1](#) shows the tool immediately after it has been started.

Section [D.1](#) will explain how to create a railway network in the tool.

Section [D.2](#) will explain how to create timetable requests in the tool.

Section [D.3](#) will end the chapter, by explaining how to generate timetables in the tool.

D.1 The Railway Network

The first thing to do when starting the tool, is to create a railway network. This is done by choosing the 'Railway Network' tab.

Here it is possible to add stations and open lines, by pressing one of the 'Add' buttons, once they are pressed, a dialog appears, prompting the user to input

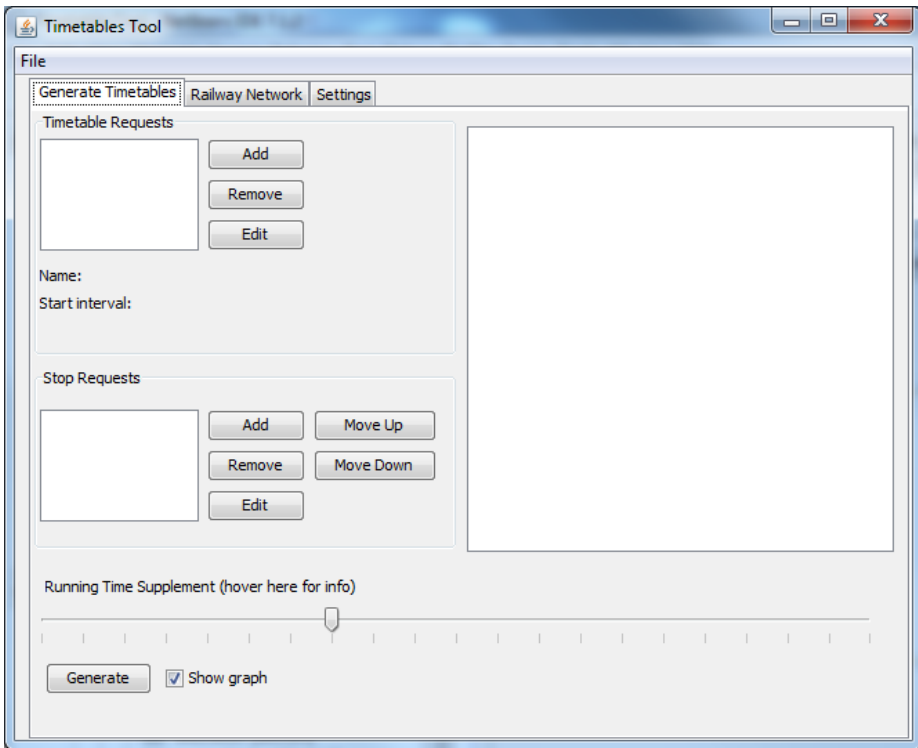


Figure D.1: The tool as it looks when it has just been started.

the required information. The dialogs can be seen in figures D.2 and D.3, and figure D.4 shows the tool after the railway network of Lokalbansen has been created.

It should be noted that it is possible to sort the list of stations, by moving them up or down, using the 'Move Up' and 'Move Down' buttons. The motivation for this sorting is presented in section D.3.

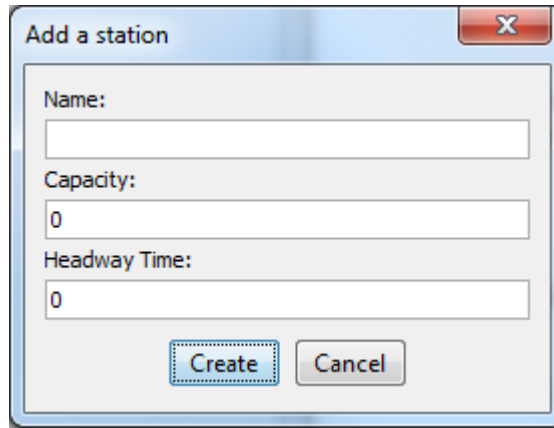


Figure D.2: When pressing the 'Add' button at the stations, this dialog appears.

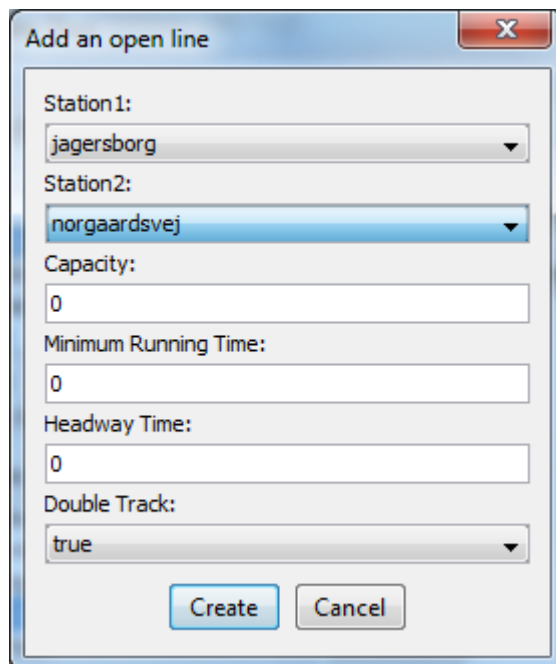
Both the open lines and the stations can be edited and deleted by selecting either a station or an open, and then press either the corresponding 'Edit' or corresponding 'Delete' button.

D.2 Creating the Timetable Requests

When wanting to create the timetable requests in the tool, the 'Generate Timetables' tab should be chosen.

Here a list of timetable requests and a list of stop requests are shown. The list of stop requests reflects the stops of the selected timetable request, if no timetable request is selected, this list will be empty.

In order to add a timetable request, the 'Add' button next to the timetable requests list should be pushed, which will bring up a dialog. When wanting to add a stop to a timetable request, a timetable request needs to be selected, and



The image shows a dialog box titled "Add an open line" with a close button (X) in the top right corner. The dialog contains the following fields:

- Station1: A dropdown menu with "jagersborg" selected.
- Station2: A dropdown menu with "norgaardsvej" selected.
- Capacity: A text input field containing "0".
- Minimum Running Time: A text input field containing "0".
- Headway Time: A text input field containing "0".
- Double Track: A dropdown menu with "true" selected.

At the bottom of the dialog, there are two buttons: "Create" and "Cancel".

Figure D.3: When pressing the 'Add' button at the open lines, this dialog appears. The available stations in the station dropdown menus, are the stations created in the railway network

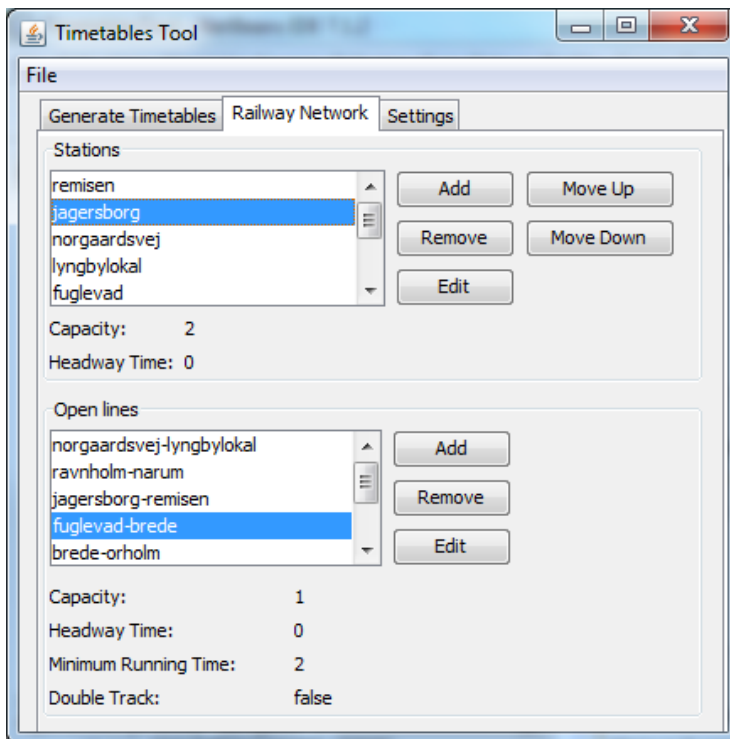


Figure D.4: The railway network of Lokalbane has been created.

then the 'Add' button next to the stop requests list should be pushed, which will also bring up a dialog (if a timetable request was selected). The dialogs can be seen in figures D.5 and D.6. and figure D.7 shows the tool after four timetable requests has been added, with the railway network of Lokalbansen.

It should be noted that the list of stop requests of the timetable requests is sorted, meaning that the displayed order, is the actual order in which the timetables requests should schedule a journey. This order can be rearranged by using the 'Move Up' and 'Move Down' buttons. The number next to the station name in the stop requests, is the desired dwell time.

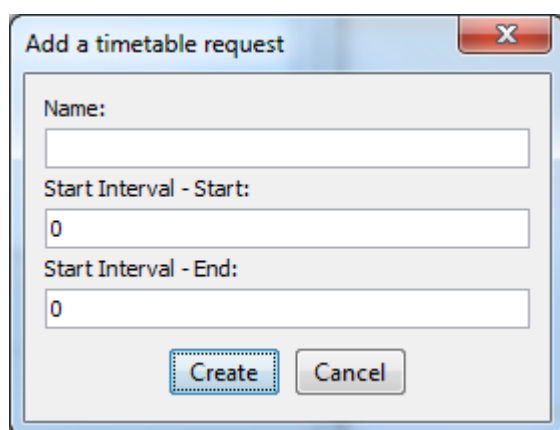
A screenshot of a dialog box titled "Add a timetable request". The dialog has a title bar with a close button (X). Inside, there are three text input fields: "Name:" (empty), "Start Interval - Start:" (containing "0"), and "Start Interval - End:" (containing "0"). At the bottom, there are two buttons: "Create" and "Cancel".

Figure D.5: When pressing the 'Add' button at the timetable requests, this dialog appears.

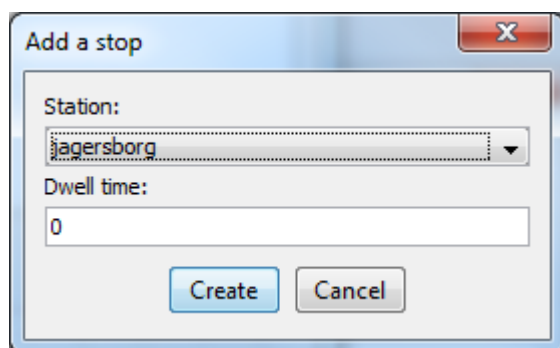
A screenshot of a dialog box titled "Add a stop". The dialog has a title bar with a close button (X). Inside, there is a dropdown menu labeled "Station:" with "Jagersborg" selected. Below it is a text input field labeled "Dwell time:" containing "0". At the bottom, there are two buttons: "Create" and "Cancel".

Figure D.6: When pressing the 'Add' button at the stop requests, and a timetable request is selected, this dialog appears. The available stations in the dropdown menu, are the stations created in the railway network.

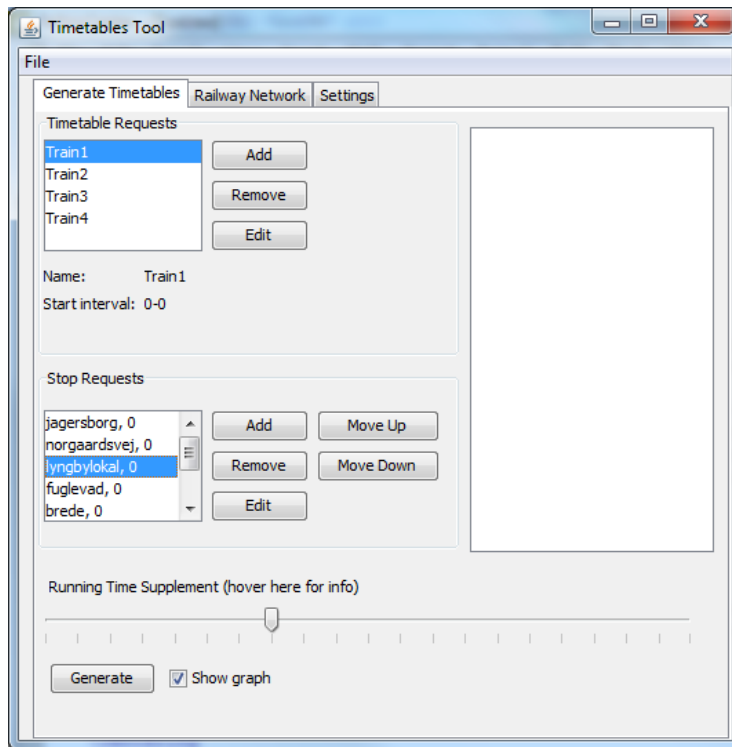


Figure D.7: Four timetables requests have been created, with the railway network of Lokalbansen.

Both the timetables requests and the stop requests can be edited or deleted, by selecting either a timetable request or a stop request, and then pressing either the corresponding 'Edit' or the corresponding 'Delete' button.

D.3 Generating Timetables

When wanting to generate timetables, the 'Settings' tab should be consulted first. Here it is necessary to specify the installation directory of UPPAAL CORA. Figure D.8 shows the settings, where the UPPAAL CORA directory has been chosen, and no properties have been excluded in the model.

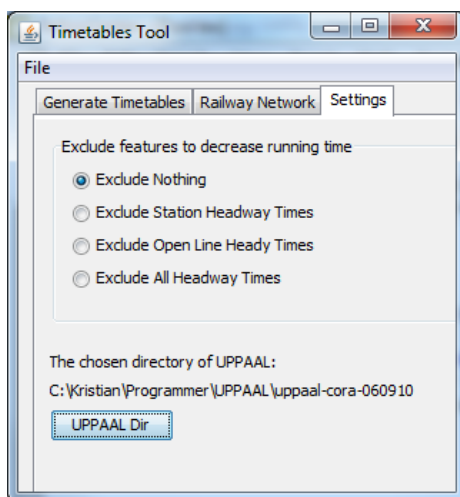


Figure D.8: The settings tab of the tool. The path of UPPAAL CORA has been chosen, and the full model is used.

Once this has been taken care of, the 'Generate Timetables' tab should be chosen again. If both a railway network and at least one timetable request (with at least one stop request) exists, a running time supplement should be chosen. The slider representing the running time supplement, reflects a percentage between 0 and 20, where 7% is the default value¹. If a graph should be displayed, the checkbox next to the 'Generate' button should be checked. Finally the 'Generate' button should be pushed in order to generate the timetables.

Figure D.9 shows what happens when the 'Generate' button is pushed. A 'Cancel' dialog appears, allowing the user to cancel the request, and the textbox in

¹This value is stated to be the optimal running time supplement in [Tho12]

the 'Generate Timetables' tab is showing the intermediate output of the model-checker of UPPAAL CORA while it is working.

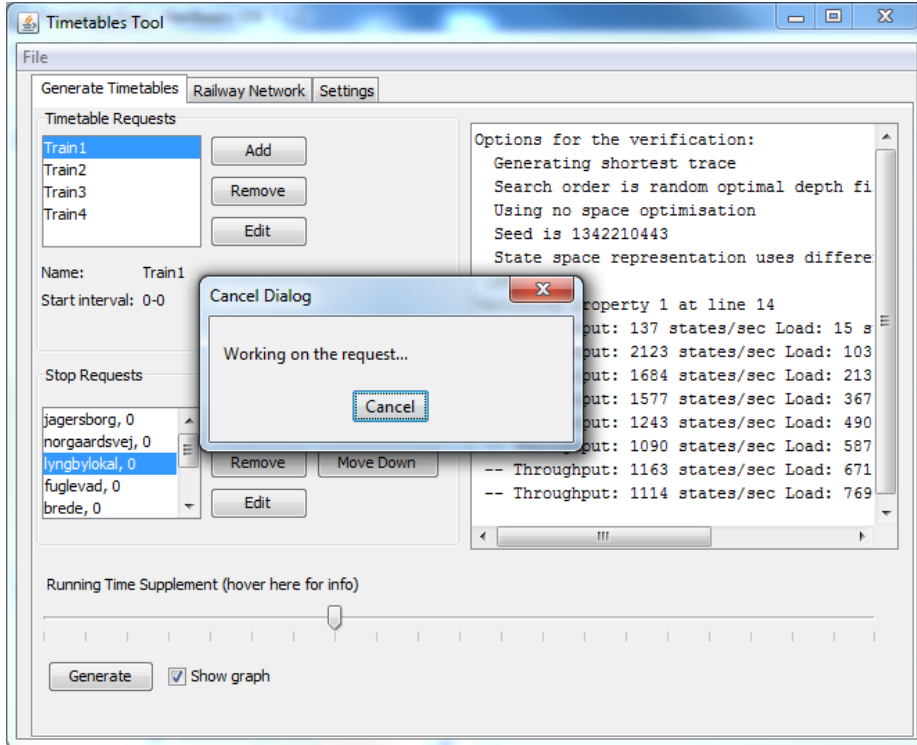


Figure D.9: The 'Generate' button was pressed, with four timetable requests in the railway network of Lokalbaneln.

Once it is finished a 'Finished!' dialog will appear, and the created timetables will be shown in a graph (if the checkbox was checked). Figure D.10 shows the generated graph of the example, where one can see that Train3 has to wait for Train1 and Train2 to leave Orholm station, before Train3 can continue on to Brede station². Furthermore it should be noted that these timetables are also printed in the textbox in tool, as shown in figure D.11.

If the timetable requests do not represent possible journeys of the railway network (for example if two consecutive are not connected), or some other peculiarity is introduced by the user, the tool will not be able to generate timetables.

The following items should be noted about the generated graph:

²These timetable were created by leaving the tool working overnight, more on the running times of the tool in appendix C.

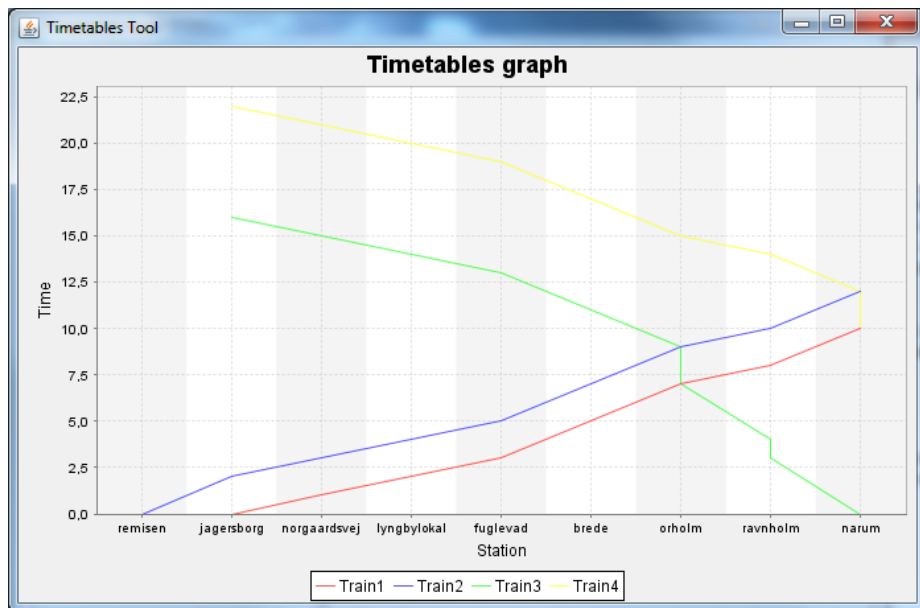


Figure D.10: Four timetables created and displayed in a graph.

- The names of the stations and trains in the generated graph, are taken from the railway network and the timetable requests, potentially allowing the user to make a graph with any amount of stations and stations names, and any amount of trains and train names.
- The order in which the stations appear in the bottom axis, is dependent on the order of the stations in the station list of the railway network in the tool (The top station of the list, will be furthest to the left of the bottom axis). This is the cause for the ability to rearrange the stations in the station list.
- The time increases moving *up* the vertical time axis, as opposed to the graph of figure 2.2, where the time is increasing going *down* the vertical time axis. Technical issues resulted in the inverse time axis.

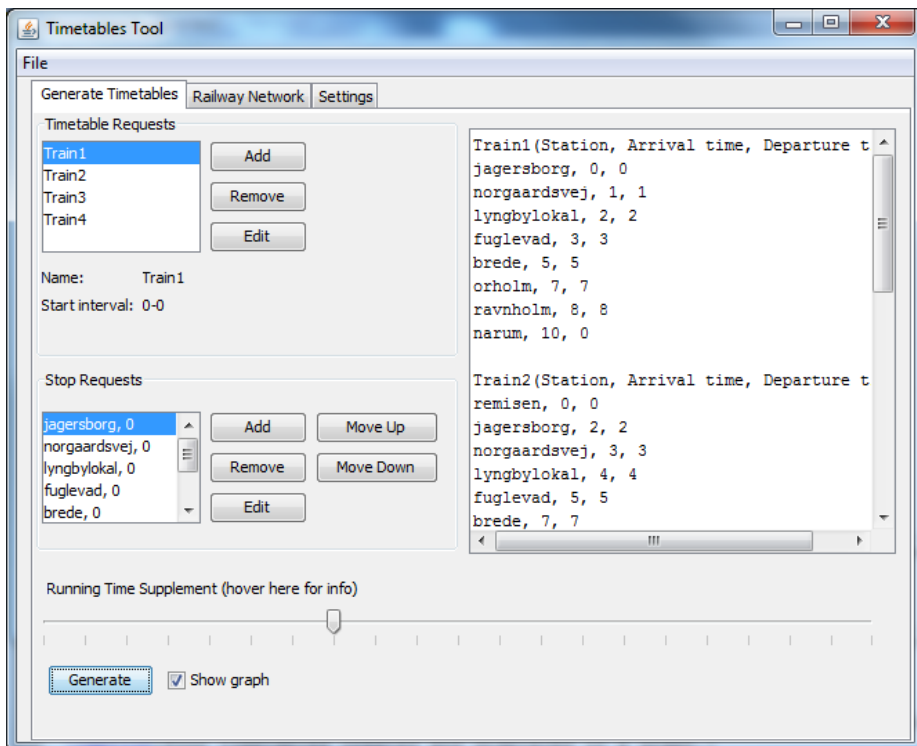


Figure D.11: The finished timetables presented as text.

Bibliography

- [BĪ2] Heiko Böck. *The Definitive Guide to NetBeans Platform 7*. Apress, 2012.
- [BDL] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on UPPAAL 4.0*. Department of Computer Science, Aalborg University, Denmark.
- [Gro92] The RAISE Language Group. *The RAISE SPECIFICATION LANGUAGE*. Prentice Hall International (UK) Ltd., 1992.
- [HP08] Ingo Arne Hansen and Jörn Pachl. *Railway Timetable & Traffic*. Eurailpress, 2008.
- [HR99] M. R. Hansen and H. Rischel. *Introduction to programming using SML*. Addison-Wesley, 1999.
- [KP88] Glenn E. Krasner and Stephen T. Pope. A description of the model-view-controller user interface paradigm in the smalltak-80 system. 1988.
- [Lan10] Alex Landex. *RailSys Tutorial*, 2010.
- [Lan11] Alex Landex. Rail traffic engineering. 2011.
- [Lok] *Tjenestekæreplanens Indledende Bemærkninger (TIB-LJN)*. Lokalbansen A/S.
- [Tho12] Mikkel Thorhauge. The usability of passenger delay models in socio-economic analysis. page 10, 2012.

- [TKO11] *Tjenestekøreplan Øst (TKØ2012)*. Bane Danmark, 2011.
- [TKS11] *Tjenestekøreplan S-tog (TKS2012)*. Bane Danmark, 2011.
- [TPS11] *Tutorial for TPS (Train Planning System)*, 2011.